LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

FAKULTÄT FÜR MATHEMATIK, INFORMATIK UND STATISTIK
INSTITUT FÜR INFORMATIK

Skriptum

zur Vorlesung

Algorithmen auf Sequenzen

gehalten im Wintersemester 2024/25

am Lehrstuhl für Bioinformatik

Volker Heun

29. Januar 2025

Version 10.28

Vorwort

Dieses Skript entstand parallel zur Vorlesung *Algorithmen auf Sequenzen* im Wintersemester 24/25 und baut auf dem vorherigen Skripten der Vorlesungen des Wintersemesters 03/04, des Wintersemesters 04/05, des Wintersemesters 06/07, des Wintersemesters 07/08, des Wintersemesters 09/10, des Sommersemesters 13, des Wintersemesters 15/16, des Wintersemesters 17/18, des Wintersemesters 18/19, des Wintersemesters 20/21 und des Wintersemesters 23/24 auf. Diese Vorlesung wurde an der Ludwig-Maximilians-Universität speziell für Studierende der Bioinformatik, aber auch für Studierende der Informatik, im Rahmen des gemeinsam von der Ludwig-Maximilians-Universität München und der Technischen Universität München veranstalteten Studiengangs Bioinformatik gehalten.

Das vorliegende Skript gibt den Inhalt aller Vorlesungen wieder, die sich jedoch inhaltlich ein wenig unterscheiden. Die Teile, die im Wintersemester 2024/25 nur teilweise behandelt wurden, sind mit einem + markiert und die Teile, die nicht Teil der Vorlesung waren, sind mit einem * markiert.

Diese Fassung ist weitestgehend überarbeitet worden, allerdings kann das Skript immer noch einige (Tipp)Fehler enthalten. Daher bin ich für jeden Hinweis darauf (an Volker.Heun@bio.ifi.lmu.de) dankbar.

An dieser Stelle möchte ich Sabine Spreer, die an der Erstellung des ersten Kapitels in LaTeX 2_ε maßgeblich beteiligt war, sowie Alois Huber und Hermann Klann, die an der Erstellung des zweiten mit sechsten Kapitels in LaTeX 2_ε maßgeblich beteiligt waren, danken. Außerdem bin ich folgenden Personen für Hinweise auf Tippfehler und Verbesserungsmöglichkeiten dankbar: Herrn Martin Bickeböller, Herrn Samuel Klein, Herrn Nick Lehner, Herrn Sebastian Strempel, Herrn Vladimir Viro, Herrn Stefan Weber und Herrn Jeremias Weihmann.

Weiterhin möchte ich insbesondere meinen Mitarbeitern Johannes Fischer, Simon W. Ginzinger, Benjamin Albrecht sowie Caroline Friedel und Marie-Sophie Friedl für Ihre Unterstützung bei den Veranstaltungen danken, die somit das vorliegende Skript erst möglich gemacht haben.

München, im Wintersemester 2024/25 Volker Heun

Inhaltsverzeichnis

1 Optimal Scoring Subsequences **1**

1.1 Maximal Scoring Subsequence 1

1.1.1 Problemstellung 1

1.1.2 Biologische Anwendungen 2

1.1.3 Naive Lösung 4

1.1.4 Lösen durch dynamische Programmierung 4

1.1.5 Divide-and-Conquer-Ansatz 5

1.1.6 Clevere Lösung 6

1.1.7 Zusammenfassung 7

1.2 All Maximal Scoring Subsequences 9

1.2.1 Problemstellung 9

1.2.2 Elementare Eigenschaften der strukturellen Definition 12

1.2.3 Ein Algorithmus zur Lösung 20

1.2.4 Zeitkomplexität 23

1.3 Bounded All Maximum Scoring Subsequences (*) 26

1.3.1 Problemstellung 26

1.3.2 Lösung mittels Dynamischer Programmierung 27

1.3.3 Effiziente Lösung mittels Dynamischer Programmierung 28

1.4 Bounded Maximal Scoring Subsequence (*) 29

1.4.1 Problemstellung 29

1.4.2 Links-Negativität 30

1.4.3 Algorithmus zur Lösung des BMSS-Problems 31

1.5 Maximal Average Scoring Subsequence (*) 34

1.5.1 Problemstellung 34
1.5.2 Rechtsschiefe Folgen und fallend rechtsschiefe Partitionen . . . 34
1.5.3 Algorithmus zur Konstruktion rechtsschiefer Zeiger 38
1.5.4 Elementare Eigenschaften von MASS 40
1.5.5 Ein Algorithmus für MASS 41
1.6 Weighted Maximal Average Scoring Subsequence (*) 45
1.6.1 Problemstellung 45
1.6.2 Elementare Eigenschaften 46
1.6.3 Generischer Algorithmus und seine Korrektheit 47
1.6.4 Linksschiefe Folgen und steigend linksschiefe Partitionen . . . 51

2 Suffix Trees Revisited 57

2.1 Definition von Suffix Tries und Suffix Trees 57
2.1.1 Σ-Bäume und (kompakte) Σ^+-Bäume 57
2.1.2 Grundlegende Notationen und elementare Eigenschaften . . . 58
2.1.3 Suffix Tries und Suffix Trees 59
2.2 Repräsentationen von Bäumen 62
2.2.1 Darstellung der Kinder mit Feldern 62
2.2.2 Darstellung der Kinder mit Listen 63
2.2.3 Darstellung der Kinder mit balancierten Bäumen 64
2.2.4 Darstellung des Baumes mit einer Hash-Tabelle 65
2.2.5 Speicherplatzeffiziente Feld-Darstellung 66
2.3 WOTD-Algorithmus 67
2.3.1 Die Konstruktion 67
2.3.2 Zeitbedarf 69
2.4 Der Algorithmus von Ukkonen 70
2.4.1 Suffix-Links 70

2.4.2 Verschachtelte Suffixe und verzweigende Teilwörter 72
2.4.3 Idee von Ukkonens Algorithmus 73
2.4.4 Ukkonens Online Algorithmus 76
2.4.5 Zeitanalyse 82

3 Repeats **85**

3.1 Exakte und maximale Repeats 85
3.1.1 Erkennung exakter Repeats 85
3.1.2 Charakterisierung maximaler Repeats 88
3.1.3 Erkennung maximaler Repeats 92
3.1.4 Revers-komplementäre Repeats 94
3.2 Tandem-Repeats und Suffix-Bäume 96
3.2.1 Was sind Tandem-Repeats 96
3.2.2 Eigenschaften von Tandem-Repeats 98
3.2.3 Algorithmus von Stoye und Gusfield 100
3.2.4 Laufzeitanalyse und Beschleunigung 103
3.2.5 Eine einfache Laufzeitanalyse (*) 104
3.2.6 Eine bessere Laufzeitanalyse 106
3.3 Tandem-Repeats mit Divide-&-Conquer 107
3.3.1 Algorithmus von Main und Lorentz 108
3.3.2 Longest Common Extensions 109
3.3.3 Conquer-Schritt 110
3.3.4 Algorithmus von Landau und Schmidt 112
3.4 Vokabulare von Tandem-Repeats 116
3.4.1 Vokabulare und Überdeckungen 116
3.4.2 Skizze des Algorithmus von Gusfield und Stoye 119
3.4.3 Tandem-Repeats und Lempel-Ziv-Zerlegungen 119

3.4.4 Phase I: Bestimmung einer linkesten Überdeckung 123

3.4.5 Phase II: Dekorierung einer Teilmenge (+) 129

3.4.6 Phase III: Vervollständigung der Dekorierung von $\mathcal{V}(t)$ (+) . . 135

4 Interludium: Lowest Common Ancestors 139

4.1 Algorithmus von Bender und Farach-Colton 139

4.1.1 Lowest Common Ancestor und Range Minimum Queries . . . 139

4.1.2 Euler-Kontur eines gewurzelten Baumes 140

4.1.3 Reduktion LCA auf RMQ 141

4.1.4 Ein quadratischer Algorithmus für RMQ 143

4.1.5 Eine verbesserte Variante 144

4.1.6 Incremental Range Minimum Query (*) 145

4.1.7 Ein optimaler Algorithmus für IRMQ (*) 146

4.1.8 Optimale Lösung für RMQ (*) 147

4.1.9 Eine einfachere optimale Variante nach Alstrup et al. 150

4.2 Algorithmus von Schieber und Vishkin (*) 154

4.2.1 LCA-Queries auf vollständigen binären Bäumen 155

4.2.2 LCA-Queries auf beliebigen Bäumen 158

4.2.3 Vorverarbeitung 163

4.2.4 Beziehung zwischen Vorfahren in T und B 164

4.2.5 Berechnung einer LCA-Query 166

4.2.6 LCA-Query-Algorithmus 169

5 Suffix-Arrays 173

5.1 Grundlegende Eigenschaften von Suffix-Arrays . . . 173

5.1.1 Was sind Suffix-Arrays? . . . 173

5.1.2 Konstruktion aus Suffix-Bäumen . . . 174

5.1.3 Algorithmus von Manber und Myers . . . 175

5.2 Algorithmus von Ko und Aluru (*) . . . 180

5.2.1 Typ S und Typ L Suffixe . . . 180

5.2.2 Sortieren der Suffixe mit sortierten Typ S/L Suffixen . . . 182

5.2.3 Sortierung der Typ S Suffixe . . . 186

5.2.4 Der Algorithmus von Ko und Aluru und Laufzeitanalyse . . . 191

5.3 Skew-Algorithmus von Kärkkäinen und Sanders . . . 193

5.3.1 Tripel und Verkürzung der Zeichenreihe . . . 193

5.3.2 Rekursives Sortieren der verkürzten Zeichenreihe . . . 193

5.3.3 Sortieren der restlichen Suffixe . . . 195

5.3.4 Mischen von A_{12} und A_0 . . . 196

5.3.5 Laufzeitanalyse . . . 197

5.3.6 Aktuelle Verfahren . . . 197

5.4 Suchen in Suffix-Arrays . . . 197

5.4.1 Binäre Suche . . . 197

5.4.2 Verbesserte binäre Suche . . . 198

5.4.3 Effiziente Berechnung der LCP-Tabelle . . . 202

5.5 Enhanced Suffix-Arrays (*) . . . 204

5.5.1 LCP-Intervalle . . . 205

5.5.2 Die Child-Tabelle und ihre Eigenschaften . . . 209

5.5.3 Optimale Suche in Enhanced Suffix-Arrays . . . 211

5.5.4 Berechnung der Child-Tabelle . . . 213

5.5.5 Komprimierte Darstellung der Child-Tabelle . . . 215

5.5.6 Simulation von Suffix-Baum-Algorithmen auf Suffix-Arrays . . 216
5.6 Extended Suffix Arrays 218
5.6.1 LCP-Intervalle 219
5.6.2 Navigation im Extended Suffix-Array 223
5.6.3 Optimale Suche in Extended Suffix-Arrays 224
5.6.4 Auffinden des Elters 224
5.6.5 Suffix-Links 227
5.6.6 LCA-Queries in Extended Suffix-Arrays 229
5.6.7 Speicherplatzbedarf 229
5.7 Burrows-Wheeler-Transformation und FM-Index 231
5.7.1 Burrows-Wheeler-Transformation 231
5.7.2 Inverse der Burrows-Wheeler-Transformation 235
5.7.3 Berechnung der LF-Funktion 238
5.7.4 FM-Index 240
5.7.5 Rank-Select-Datenstrukturen 244
5.7.6 Wavelet-Trees 248

6 Genome Rearrangements 251
6.1 Modellbildung 251
6.1.1 Rearrangements und zugehörige Mutationen 251
6.1.2 Permutationen 253
6.2 Sorting by Reversals 255
6.2.1 Komplexität von Min-SBR 256
6.2.2 2-Approximation für Min-SBR 256
6.2.3 Algorithmus und Laufzeitanalyse 262
6.3 Eine bessere untere Schranke für Min-SBR 264
6.3.1 Breakpoint-Graphen 264

6.3.2 Elementare Beobachtungen 267
6.3.3 Die untere Schranke 269
6.4 Sorting by Oriented Reversals 271
6.4.1 Orientierte Permutationen 271
6.4.2 Reality-Desire-Diagram 273
6.4.3 Der Overlap-Graph 275
6.4.4 Hurdles und Fortresses 280
6.4.5 Eine untere Schranke für Min-SOR 283
6.4.6 Sortierung orientierter Komponenten 285
6.4.7 Eliminierung von Hurdles (+) 293
6.4.8 Algorithmus für Min-SOR 296
6.5 Sorting by Transpositions (*) 298
6.5.1 Elementare Definitionen 298
6.5.2 Untere Schranken für die Transpositions-Distanz 299
6.5.3 Orientierte und Unorientierte Desire-Edges 300
6.5.4 Eine 2-Approximation 301
6.6 Sorting by Transpositions und Reversals (*) 306
6.6.1 Problemstellung 306
6.6.2 Untere Schranken für die Transversal-Distanz 308
6.6.3 Eine 2-Approximation 309
6.7 Sorting by Weighted Transversals (*) 315
6.7.1 Gewichtete Transversal-Distanz 316
6.7.2 Starke Orientierung und starke Hurdles 317
6.7.3 Eine untere Schranke für die Transversal-Distanz 318
6.7.4 Eine Approximation für Min-SWT 320
6.7.5 Approximationsgüte 322
6.8 Weitere Modelle (*) 324

6.8.1 Gewichtung der Operationen durch ihre Länge 324
6.8.2 Duplikationen 325
6.8.3 Multi-chromosomale Genome 325
6.8.4 Multiple Genome Rearrangements 326

A Literaturhinweise **329**

A.1 Lehrbücher zur Vorlesung 329
A.2 Skripten anderer Universitäten 329
A.3 Originalarbeiten 330
A.3.1 Optimal Scoring Subsequences 330
A.3.2 Suffix-Trees 331
A.3.3 Repeats 331
A.3.4 Lowest Common Ancestors and Range Minimum Queries . . . 332
A.3.5 Construction of Suffix-Arrays 333
A.3.6 Applications of Suffix-Arrays 334
A.3.7 Sorting by Reversals 335
A.3.8 Sorting by Oriented Reversals 336
A.3.9 Sorting by Transpositions 337
A.3.10 Sorting by Transversals 337
A.3.11 Erweiterungen zu Genome Rearrangements 338

B Index **341**

1 Optimal Scoring Subsequences

1.1 Maximal Scoring Subsequence

Ziel dieses Abschnittes ist es, (möglichst effiziente) Algorithmen für das Maximal Scoring Subsequence Problem vorzustellen. Dabei werden wir zunächst noch einmal kurz die wichtigsten Paradigmen zum Entwurf von Algorithmen wiederholen.

1.1.1 Problemstellung

MAXIMAL SCORING SUBSEQUENCE (MSS)

Eingabe: Eine Folge $(a_1, \ldots, a_n) \in \mathbb{R}^n$.
Gesucht: Eine (zusammenhängende) Teilfolge $(a_i, \ldots, a_j)$, die $\sigma(i, j)$ maximiert, wobei $\sigma(i, j) = \sum_{\ell=i}^{j} a_\ell$.

Bemerkung: Mit Teilfolgen sind im Folgenden immer (sofern nicht anders erwähnt) zusammenhängende (d.h. konsekutive) Teilfolgen einer Folge gemeint (also anders als beispielsweise in der Analysis).

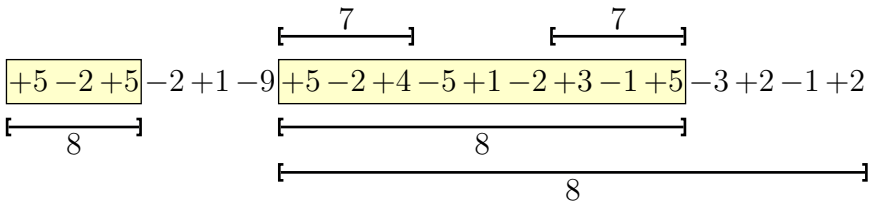


Abbildung 1.1: Beispiel: Maximal Scoring Subsequences

In Abbildung 1.1 ist ein Beispiel angegeben. Wie man dort sieht, kann es mehrere (und auch nicht-disjunkte) Lösungen geben.

Bemerkungen:

- Es sind mehrere Lösungen möglich.
- Die leere Folge mit Score 0 interpretieren wir auch als eine Lösung.

- Ist eine Lösung in der anderen enthalten, so wählen wir als Lösung immer eine kürzester Länge. Die anderen ergeben sich aus Anhängen von Teilfolgen mit dem Score Null. Darüber hinaus haben solche Lösungen noch eine schöne Eigenschaft, wie wir gleich sehen werden.

- Es gibt keine echt überlappenden Lösungen. Angenommen, es gäbe echt überlappende Lösungen a' und a'' einer gegebenen Folge a (siehe dazu auch Abbildung 1.2). Die Sequenz gebildet aus der Vereinigung beider Sequenzen (respektive ihrer Indices) müsste dann einen höheren Score haben, da der Score der Endstücke > 0 ist (sonst würde er in den betrachteten Teilfolgen nicht berücksichtigt werden).

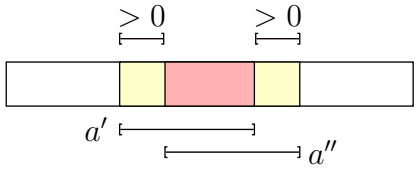


Abbildung 1.2: Skizze: Überlappende optimale Teilfolgen

1.1.2 Biologische Anwendungen

In diesem Abschnitt wollen wir kurz einige biologische Probleme vorstellen, die sich als Maximal Scoring Subsequence formulieren lassen.

Transmembranproteine: Bestimmung der transmembranen Regionen eines Proteins. Eingelagerte Proteine in der Membran sollten einen ähnlichen Aufbau wie die Membran selbst haben, da die Gesamtstruktur stabiler ist. Somit sollten transmembrane Regionen hydrophob sein.

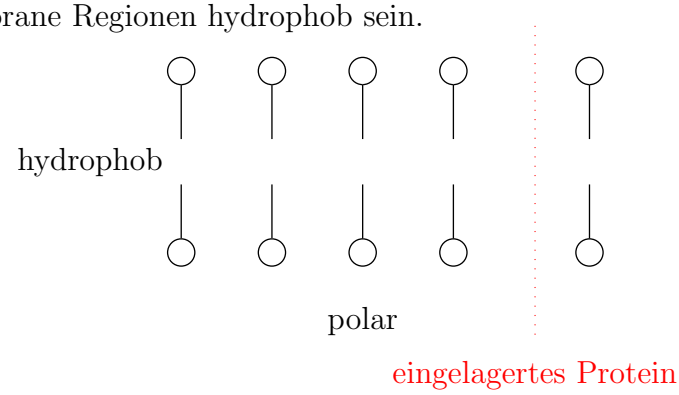


Abbildung 1.3: Beispiel: transmembrane Proteine

Mit einer geeigneten Gewichtung der Aminosäuren, können solche hydrophoben Regionen mit Hilfe der Lösung eines Maximal Scoring Subsequence Problems gefunden werden.

Für die einzelnen Aminosäuren werden die folgende Werte gemäß der Hydrophobizität der entsprechenden Aminosäure gewählt:

- hydrophobe Aminosäuren: ein Wert aus $\in [0:3]$;
- hydrophile Aminosäuren: ein Wert aus $\in [-5:0]$.

Lokalisierung GC-reicher DNS-Abschnitte: In GC-reichen Regionen der DNS finden sich häufig Gene. Das Auffinden solcher GC-reicher Regionen lässt sich als Maximal Scoring Subsequence Problem beschreiben:

- $C, G \to 1 - p$ für ein $p \in [0:1]$;
- $A, T \to -p$.

Zusätzlich können Längenbeschränkungen sinnvoll sein; obere, untere Schranke der Länge für z.B. Proteine, die man sucht.

Vergleichende Analyse von Genomen: Zwischen dem Mensch- und Maus-Genom liegen die Sequenz-Ähnlichkeiten für Exons bei 85% und für Introns bei 35%. Mit Hilfe eines lokalen Sequenz-Alignments (Smith-Waterman) lassen sich solche Übereinstimmungen gut auffinden. Jedoch kann es bei den gefundenen Lösungen den so genannten Mosaik-Effekt geben, d.h. sehr ähnliche Sequenzen sind immer wieder von sehr unähnlichen, jedoch relativ kurzen Stücken unterbrochen.

Mit Hilfe eines geeigneten Maximal Scoring Subsequences Problems können solche Mosaik-Effekte aufgedeckt werden. Hierzu wird eine Variante des Maximal Scoring Subsequence Problems verwendet. Man normiert die erzielten Scores mit der Länge der zugehörigen Teilfolge. Somit lassen sich so genannte *poor regions* (stark positive Teile, die mit kurzen stark negativen Fragmenten unterbrochen sind) ausschließen.

Konservierte Regionen: Gut konservierte Regionen eines mehrfachen Sequenzenalignments lassen sich durch Gewichtung der Spalten gemäß ihrer Ähnlichkeiten (beispielsweise SP-Maß einer Spalte) und einem anschließenden Auffinden von Maximal Scoring Subsequences bestimmen.

'Ungapped' local alignment: Auch lokales Alignment ohne Lücken (gaps) können aus der Dot-Matrix durch Anwenden von Algorithmen für das Maximal Scoring Subsequence Problem auf die Diagonalen effizient finden. Dieses Verfahren ist insbesondere dann effizient, wenn man mit Längenrestriktionen arbeiten will, oder den Score ebenfalls wieder mit der zugehörigen Länge der Folge normalisieren will.

1.1.3 Naive Lösung

Im Folgenden wollen wir eine Reihe von Algorithmen zur Lösung des Maximal Scoring Subsequence Problems vorstellen, die jeweils effizienter als die vorher vorgestellte Variante ist.

Die naive Methode bestimmt zuerst alle Werte $\sigma(i,j)$ für alle $i \leq j \in [1:n]$. Anschließend wird aus den Werten ein Maximum ermittelt, ggf. eines dessen zugehörige Sequenz die kürzeste Länge aufweist.

Für die Laufzeit (Anzahl Additionen, die proportional zur Laufzeit ist) ergibt sich dann pro Tabelleneintrag $\Theta(j-i)$, also insgesamt:

$$\sum_{i=1}^{n}\sum_{j=i}^{n}\Theta(j-i) = \Theta\left(\sum_{i=1}^{n}\sum_{j=0}^{n-i} j\right) = \Theta\left(\sum_{i=1}^{n}(n-i)^2\right) = \Theta\left(\sum_{i=0}^{n-1} i^2\right) = \Theta(n^3).$$

Ein alternative Begründung ist die folgende: In der Tabelle mit n^2 Einträgen benötigt jeder Eintrag maximal n Operationen. Hierbei erhalten wir jedoch nur eine obere Schranke und nicht die korrespondierende untere Schranke für die Laufzeit.

1.1.4 Lösen durch dynamische Programmierung

Ein anderer Ansatz ergibt sich aus einer trivialen Rekursionsgleichung für $\sigma(i,j)$. Es gilt folgende Rekursionsgleichung:

$$\sigma(i,j) = \begin{cases} a_i & \text{für } i = j \\ \sigma(i,k) + \sigma(k+1,j) & \text{für ein } k \in [i:j-1] \text{ sofern } i < j \end{cases}$$

In der Regel ist eine direkte Implementierung dieser Rekursionsgleichung zu aufwendig, da meist exponentiell viele Aufrufe erfolgen (siehe zum Beispiel rekursive Berechnung einer Fibonacci-Zahl)! In diesem Fall wären es sogar nur n^3 rekursive Aufraufe, was aber nicht besser als der naive Ansatz ist. Mit Hilfe der *dynamische Programmierung* können wir jedoch effizienter werden, da hier Werte mehrfach berechnet werden.

Die Tabellengröße ist $O(n^2)$. Jeder Eintrag kann mit der Rekursionsgleichung in Zeit $O(1)$ berechnet werden. Dabei wird die Tabelle beginnend von der Mitteldiagonalen aus über alle Nebendiagonalen zur rechten oberen Ecke hin aufgefüllt (siehe auch Abbildung 1.4).

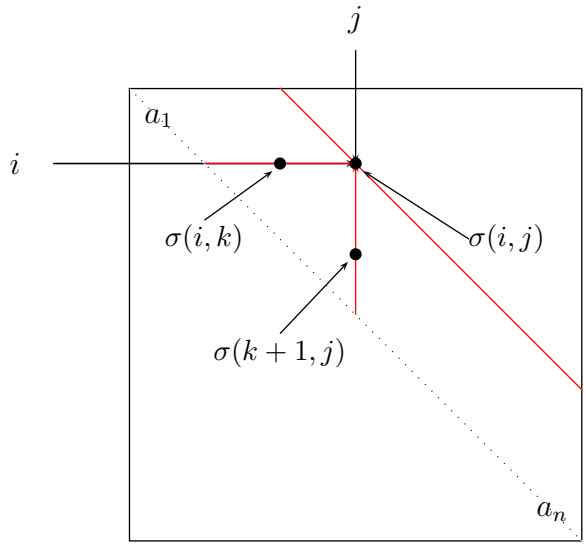


Abbildung 1.4: Skizze: Auffüllen der dynamischen Programmierungstabelle

1.1.5 Divide-and-Conquer-Ansatz

Eine andere Lösungsmöglichkeit erhalten wir einem Divide-and-Conquer-Ansatz, wie in Abbildung 1.5 illustriert. Dabei wird die Folge in zwei etwa gleich lange Folgen aufgeteilt und die Lösung in diesen rekursiv ermittelt.

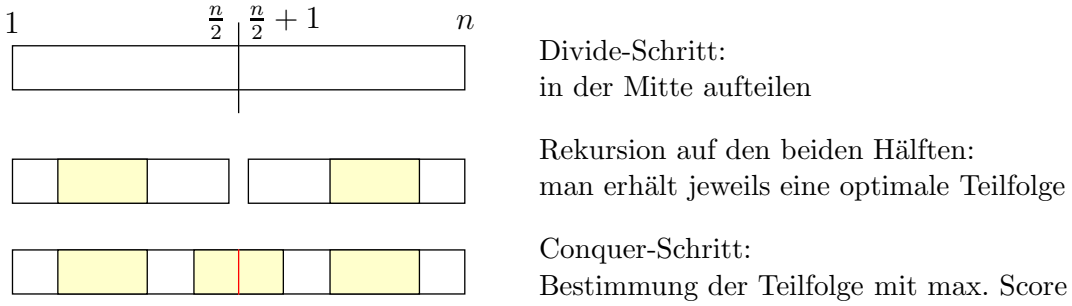
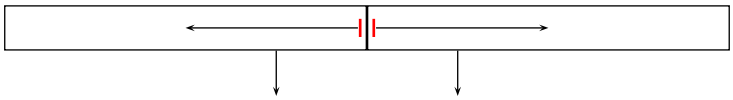


Abbildung 1.5: Skizze: Divide-and-Conquer bei Maximal Scoring Subsequences

Man kann dabei aber auch die optimale Teilfolge in der Mitte zerschneiden, die dann die Elemente $a_{n/2}$ und $a_{n/2+1}$ enthalten muss. Daher muss man zusätzlich von der Mitte aus testen, wie von dort nach rechts bzw. links eine optimale Teilfolge aussieht (siehe auch Abbildung 1.6).

Dazu bestimmen wir jeweils das Optimum der Hälften, d.h

$$\max\{\sigma(i, n/2) \mid i \in [1 : n/2]\} \qquad \text{und} \qquad \max\{\sigma(n/2+1, j) \mid j \in [n/2+1 : n]\}.$$



$\max\{\sigma(i, \frac{n}{2}) \mid i \in [1 : \frac{n}{2}]\}$ $\max\{\sigma(\frac{n}{2}+1, j) \mid j \in [\frac{n}{2}+1 : n]\}$

Abbildung 1.6: Skizze: Conquer-Step

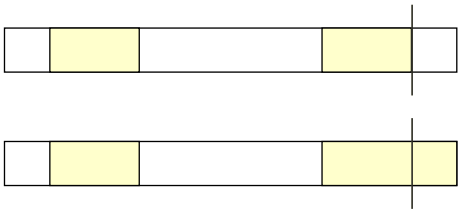
Man überlegt sich leicht, dass die optimale Teilfolge, die die Positionen $n/2$ und $n/2+1$ überdeckt, aus der Konkatenation der beiden berechneten optimalen Teilfolgen in den jeweiligen Hälften bestehen muss.

Für die Laufzeit erhalten wir sofort die folgende Rekursionsgleichung, die identisch zur Laufzeitanalyse von Mergesort ist, da das Bestimmen einer optimalen Teilfolge über die Mitte hinweg in Zeit $O(n)$ (wie oben gesehen) geschehen kann:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n) = O(n \log(n)).$$

1.1.6 Clevere Lösung

Wenn wir wie beim Divide-and-Conquer-Ansatz das Problem nicht in der Mitte aufteilen, sondern am rechten Rand, so können wir (ganz im Gegensatz zum Problem des Sortierens) eine effizientere Lösung finden (siehe auch Abbildung 1.7).



Nur das letzte Feld absplitten.

Am Rand gleich die optimale Teilfolge mitbestimmen, z.B. durch den Versuch die aktuelle Randfolge zu verlängern.

Abbildung 1.7: Skizze: Asymmetrische Divide-and-Conquer

Damit wir nun beim Conquer-Schritt die zusätzliche Zeit einsparen können, bestimmen wir in der linken Rekursion noch die optimale Teilfolge mit, die die letzte Position beinhaltet. Im Conquer-Step wird dann diese Lösung am letzten Rand (die die Position $n-1$ beinhaltet) um die Position n erweitert, wenn der Wert der rekursiven Lösung positiv ist. Andernfalls ist die Folge a_n die gesuchte Lösung, die die Position n beinhaltet.

Man kann diese Idee rekursiv als Divide-and-Conquer implementieren oder iterativ, wie in Abbildung 1.8 angegeben, auflösen. Da im Wesentlichen einmal linear über die Folge gelaufen wird und für jedes Element nur konstante Kosten (Additionen,

```
MSS (real a[], int n)
begin
    int max := 0, ℓ := 1, r := 0 ;                    /* best global solution */
    int rmax := 0, rstart := 1 ;                      /* best solution at right end */
    for (i := 1; i ≤ n; i++) do
        if (rmax > 0) then                            /* equivalent to rmax + a_i > a_i */
            rmax := rmax + a_i;
        else
            rmax := a_i;
            rstart := i;
        if (rmax > max) then
            max := rmax;
            ℓ := rstart;
            r := i;
    return (ℓ, r, max);
end
```

Abbildung 1.8: Algorithmus: Die clevere Lösung

Maximumsbildungen) anfallen, erhalten wir offensichtlich eine Laufzeit von $O(n)$. Da dies eine offensichtlich optimale Lösung ist, halten wir das Ergebnis im folgenden Satz fest.

Theorem 1.1 *Eine Teilfolge mit maximalem Wert einer gegebenen reellen Folge lässt sich in Linearzeit mit konstantem zusätzlichen Platzbedarf bestimmen.*

15.10.24

1.1.7 Zusammenfassung

In der folgenden Tabelle in Abbildung 1.9 sind alle Resultate der vorgestellten Algorithmen noch einmal zusammengefasst.

Algorithmus	Zeit	Platz	Bemerkung
Naiver Algorithmus	$O(n^3)$	$O(n^2)$	Tabelle füllen
Dyn. Programmierung	$O(n^2)$	$O(n^2)$	geschickter füllen
Divide and Conquer	$O(n \log(n))$	$O(n)$	die Eingabelänge ist n
Clevere Lösung	$O(n)$	$n + O(1)$	die Eingabelänge ist n

Abbildung 1.9: Tabelle: Laufzeiten für die MSS Lösungsansätze

In der folgenden Tabelle in Abbildung 1.10 sind explizit die Längen von Folgen angegeben, die in einer Sekunde bzw. einer Minute auf einem gewöhnlichen Rechner mit Stand 2004 (AMD-Athlon, 2GHz, 2GB Hauptspeicher) verarbeitet werden können (mittels eines C-Programms).

Seq-Len	1sec.		1min.
Naive	800	$\xrightarrow{\times 4}$	3.200
Dyn.Prog.	4.200	$\xrightarrow{\times 8}$	32.000
D&C	1.500.000	$\xrightarrow{\times 50}$	75.000.000
Clever	20.000.000	$\xrightarrow{\times 60}$	*1.200.000.000*

Abbildung 1.10: Tabelle: zu verarbeitenden Problemgrößen (Stand 2004)

In der folgenden Tabelle in Abbildung 1.11 ist das gleiche Experiment mit Stand 2008 (Intel Dual-Core, 2.4GHz, 4GB Hauptspeicher) angegeben.

Seq-Len	1sec.		1min.
Naive	1.325	$\xrightarrow{\times 4}$	5.200
Dyn.Prog.	27.500	$\xrightarrow{\times 8}$	220.000
D&C	10.000.000	$\xrightarrow{\times 50}$	500.000.000
Clever	400.000.000	$\xrightarrow{\times 60}$	*24.000.000.000*

Abbildung 1.11: Tabelle: zu verarbeitenden Problemgrößen (Stand(2008)

In der Tabelle in der Abbildung 1.12 ist da gleiche Experiment mit Stand 2014 (Intel Quad Core i7-3770, 3.40 GHz, 32 GB Hauptspeicher) angegeben.

Seq-Len	1sec.		1min.
Naiv	1.500	$\xrightarrow{\times \approx 4}$	5.800
Dyn.Prog.	36.000	$\xrightarrow{\times \approx 8}$	280.000
D&C	18.000.000	$\xrightarrow{\times \approx 50}$	950.000.000
Clever	650.000.000	$\xrightarrow{\times \approx 60}$	*40.000.000.000*

Abbildung 1.12: Tabelle: zu verarbeitenden Problemgrößen (Stand 2014)

In der Tabelle in der Abbildung 1.13 ist da gleiche Experiment mit Stand 2019 (Intel(R) Core(TM) i7-9700,3.00 GHz, 642 GB Hauptspeicher) angegeben.

Seq-Len	1sec.		1min.
Naiv	2.100	$\xrightarrow{\times \approx 4}$	8.300
Dyn.Prog.	65.000	$\xrightarrow{\times \approx 8}$	500.000
D&C	36.000.000	$\xrightarrow{\times \approx 50}$	2.000.000.000
Clever	420.000.000	$\xrightarrow{\times \approx 60}$	*25.000.000.000*

Abbildung 1.13: Tabelle: zu verarbeitenden Problemgrößen (Stand 2019)

Man sieht, dass sich bei den einfachen Algorithmen (Naiv und Dynamische Programmierung) die Größenordnungen kaum ändern, bei den geschickten Varianten hingegen schon. Weiterhin stellt man fest, dass beim linearen Ansatz die Leistung von 2019 gegenüber 2014 zurückgegangen ist. Dies liegt vermutlich in den neuen Architekturen und dem niedrigeren Takt begründet, mit denen einfache Operationen etwas langsamer werden, während komplexe Operationen weiterhin besser vorberechnet werden.

Beachte hierbei, dass die Faktoren bei einer Versechzigfachung der Laufzeit beim naive Algorithmus etwa $\sqrt[3]{60} \approx 4$, bei der dynamischen Programmierung etwa $\sqrt{60} \approx 8$ und beim Cleveren Algorithmus etwa 60 ist. Die Funktion $n \log(n)$ besitzt keine einfache anzugebende Inverse, der Wert 50 entspricht aber in etwa der Inversen für 60 im Bereich der betrachteten Werte von n.

Die kursiven Werte in den Tabellen sind geschätzt, da der benötigte Hauptspeicher nicht zur Verfügung stand.

1.2 All Maximal Scoring Subsequences

Nun wollen wir uns mit der Frage beschäftigen, wenn wir nicht nur eine beste, sondern alle besten bzw. alle möglichen Teilfolgen mit positive Score und zwar nach absteigenden Score erhalten wollen. Zuerst einmal müssen wir uns über die Fragestellung klar werden, d.h. was ist überhaupt die gesuchte Lösung.

1.2.1 Problemstellung

Geben wir zunächst die formale Problemstellung an und diskutieren anschließend die damit verbundenen Probleme.

ALL MAXIMAL SCORING SUBSEQUENCES (AMSS)

Eingabe: Eine Folge $(a_1, \ldots, a_n) \in \mathbb{R}^n$.
Gesucht: Alle disjunkten Teilfolgen von a, die ihren Score maximieren.

Zunächst einmal muss man sich überlegen, was es heißen soll, dass *alle disjunkten Teilfolgen ihren Score maximieren.* Betrachten wir dazu das Beispiel in Abbildung 1.14. Die lange Folge ist keine Lösung, da wir keine überlappenden Folgen haben wollen.

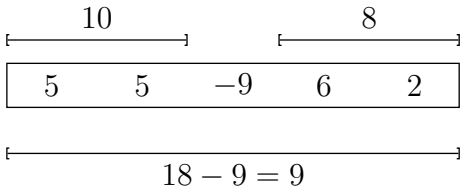


Abbildung 1.14: Beispiel: Maximal bewertete Teilfolgen

Wir geben zwei mögliche Definitionen an, wie man alle maximalen Teilfolgen einer Folge definieren kann. Im Folgenden werden wir zeigen, dass die beiden Definitionen äquivalent sind. A priori ist dies überhaupt nicht klar.

Definition 1.2 (Strukturelle Definition) *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ eine reelle Folge. Eine Teilfolge $a' = (a_i, \ldots, a_j)$ von a heißt* maximal bewertet *(engl. maximal scoring), wenn die beiden folgenden Eigenschaften erfüllt sind:*

(E1) alle echten Teilfolgen von a' haben einen kleineren Score;

(E2) keine echte Oberfolge von a' in a erfüllt E1.

Die Bedingung E1 ist nach unserer vorhergehenden Diskussion klar, da wir keine Teilfolge mit größerem Score, die Teilfolge einer anderen Teilfolge ist, als Lösung verlieren wollen. Nur die Bedingung E1 ist aber zu wenig, da sonst auch alle positiven Elemente im Beispiel in Abbildung 1.14 maximal bewertet wären, was so nicht sinnvoll ist.

Als Bedingung E2 würde man vermutlich zunächst erwarten, dass jede Oberfolge ebenfalls einen kleineren Score als die gegebene Teilfolge besitzen soll. Die ist jedoch zu naiv, wie das vorherige Beispiel mit $a = (5, 5, -9, 6, 2)$ zeigt. Damit wäre die Teilfolge $a' = (6, 2)$ keine maximal bewertete Teilfolge, da die Oberfolge $a = (a_1, \ldots, a_5)$ einen höheren Score besitzt: $9 > 8$. Als Lösungsmenge würde man jedoch sicherlich $\mathrm{MSS}(a) = \{(a_1, a_2), (a_4, a_5)\}$ erwarten. Diese beiden Folgen erfüllen jedoch sowohl E1 als auch E2.

Halten wir jetzt die endgültige Definition noch fest.

ALL MAXIMAL SCORING SUBSEQUENCES (AMSS)

Eingabe: Eine Folge $(a_1, \ldots, a_n) \in \mathbb{R}^n$.
Gesucht: Alle maximal bewerteten Teilfolgen von a.

Neben dieser strukturellen Definition kann man auch noch eine eher algorithmisch angelehnte Definition angeben.

Definition 1.3 (Prozedurale Definition) *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ eine reelle Folge. Eine kürzeste Teilfolge $(a_i, \ldots, a_j)$ von a mit maximalem Score heißt* maximal bewertet *(engl. maximal scoring). Teilfolgen aus $(a_1, \ldots, a_{i-1})$ bzw. $(a_{j+1}, \ldots, a_n)$, die für diese maximal bewertet sind, sind auch für a* maximal bewertet.

Man überlegt sich leicht, dass die Menge aller maximal bewerteten Teilfolgen nach der prozeduralen Definition eindeutig ist. Wir werden später noch sehen, dass dies auch für die strukturelle Definition gilt (das folgt aus der noch zu zeigenden Äquivalenz der beiden Definitionen).

Eigentlich sollte man vorerst beim Begriff *maximal bewertet* immer angeben, ob dies nach der strukturellen oder nach der prozeduralen Definition gemeint ist. Im Folgenden wird dies aber aus dem Kontext klar.

Notation 1.4 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ eine reelle Folge, dann bezeichnet $MSS(a)$ die Menge aller maximal bewerteten Teilfolgen von a.*

Aus der prozeduralen Definition kann sofort ein rekursiver Algorithmus zur Bestimmung aller maximal bewerteter Teilfolgen abgeleitet werden, der in der folgenden Skizze in Abbildung 1.15 veranschaulicht ist. Man bestimmt zunächst eine Maximal Scoring Subsequence und entfernt diese aus der Folge. Für die beiden entstehenden Folgen wird dieser Algorithmus rekursiv aufgerufen.

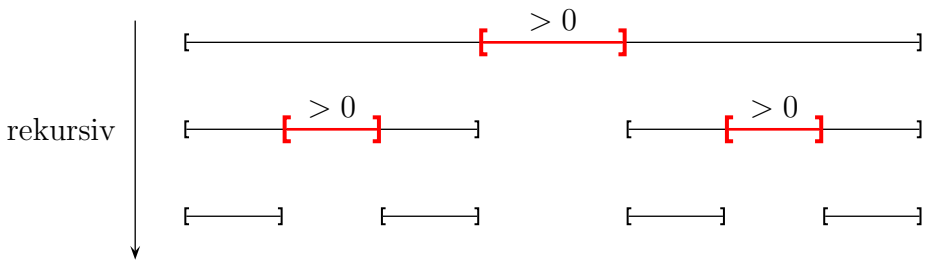


Abbildung 1.15: Skizze: Rekursiver Ansatz für AMSS

Die Laufzeit dieses Algorithmus erfüllt folgende Rekursionsgleichung, da das Auffinden einer Maximal Scoring Subsequence, wie wir im letzten Abschnitt gesehen haben, in Zeit $O(n)$ durchführbar ist:

$$T(n) = O(n) + T(n_1) + T(n_2) \quad \text{mit} \quad n_1 + n_2 < n.$$

Ähnlich zu Quicksort ergibt sich folgende Analyse. Im worst-case benötigt dieser Algorithmus offensichtlich maximal Zeit $O(n^2)$.

Im average-case kann man aus der Analyse von Quicksort herleiten, dass auch dieser Algorithmus einen Zeitbedarf von $O(n \log(n))$ hat. Hierzu muss man jedoch eine geeignete Wahrscheinlichkeitsverteilung annehmen, die in der Rechnung äquivalent zu der von Quicksort ist, die aber nicht unbedingt realistisch sein muss!

1.2.2 Elementare Eigenschaften der strukturellen Definition

In diesem Abschnitt werden wir einige grundlegende Eigenschaften maximal bewerteter Teilfolgen nach der strukturellen Definition herleiten, die es uns erlauben werden, die Äquivalenz der beiden Definitionen maximal bewerteter Teilfolgen zu zeigen. Darauf basierend werden wir im nächsten Abschnitt einen effizienten Algorithmus für die Lösung des All Maximal Scoring Subsequences Problem vorstellen.

Lemma 1.5 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Für jede Teilfolge $a' = (a_i, \ldots, a_j)$ von a sind äquivalent:*

1) a' erfüllt E1.

2) Es gilt für das Minimum aller Präfixe von a in a', dass

$$\sigma(1, i-1) = \min\{\sigma(1, k) \mid k \in [i-1, j]\},$$

und für das Maximum aller Präfixe von a in a', dass

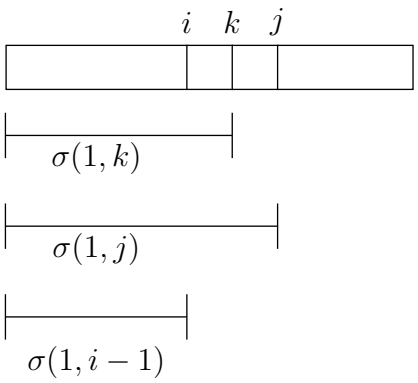
$$\sigma(1, j) = \max\{\sigma(1, k) \mid k \in [i-1, j]\},$$

und dass diese jeweils eindeutig sind!

3) $\forall k \in [i : j] : \sigma(i, k) > 0 \wedge \sigma(k, j) > 0$.

Beweis: $1 \Rightarrow 2$: Wir führen den Beweis durch Widerspruch.

Für das Maximum: Sei $k \in [i-1 : j-1]$ mit $\sigma(1, k) \geq \sigma(1, j)$. Dann ergibt sich die folgende Situation, die in Abbildung 1.16 dargestellt ist.



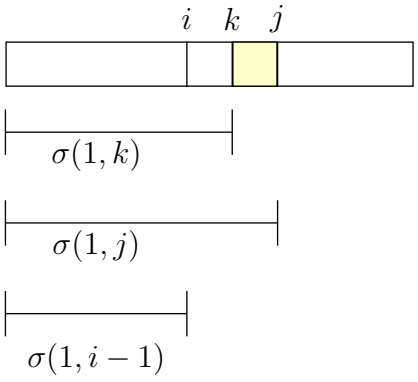
$$\begin{aligned}\sigma(i,k) &= \sigma(1,k) - \sigma(1,i-1)\\ &\geq \sigma(1,j) - \sigma(1,i-1)\\ &= \sigma(i,j)\end{aligned}$$

Abbildung 1.16: Skizze: Fall 1: $\sigma(1,k) \geq \sigma(1,j)$

Daraus ergibt sich ein Widerspruch zur Annahme der Eigenschaft E1, da die echte Teilfolge $(a_i, \ldots, a_k)$ von $(a_i, \ldots, a_j)$ keinen kleineren Score besitzt.

Anmerkung: Der Beweis gilt auch für $k = i-1$. In diesem Fall wäre $\sigma(i,j) \leq 0$, was nicht sein kann.

Für das Minimum: Sei jetzt $k \in [i:j]$ mit $\sigma(1,k) \leq \sigma(1,i-1)$. Dann ergibt sich die folgende Situation, die in Abbildung 1.17 dargestellt ist.



$$\begin{aligned}\sigma(k+1,j) &= \sigma(1,j) - \sigma(1,k)\\ &\geq \sigma(1,j) - \sigma(1,i-1)\\ &= \sigma(i,j)\end{aligned}$$

Abbildung 1.17: Skizze: $\sigma(1,k) \leq \sigma(1,i-1)$

Daraus ergibt sich ein Widerspruch zur Annahme der Eigenschaft E1, da die echte Teilfolge $(a_{k+1}, \ldots, a_j)$ von $(a_i, \ldots, a_j)$ keinen kleineren Score besitzt.

Anmerkung: Der Beweis gilt auch für $k = j$. In diesem Fall wäre $\sigma(i,j) \leq 0$, was nicht sein kann.

2 ⇒ 3 : Da das Minimum eindeutig ist, folgt $\sigma(1, i-1) < \sigma(1, k)$ für alle $k \in [i : j]$ und somit:

$$\sigma(i, k) = \sigma(1, k) - \underbrace{\sigma(1, i-1)}_{<\sigma(1,k)} > 0.$$

Da das Maximum eindeutig ist, folgt $\sigma(1, j) > \sigma(1, k-1)$ für alle $k \in [i : j]$ und somit:

$$\sigma(k, j) = \underbrace{\sigma(1, j)}_{>\sigma(1,k-1)} - \sigma(1, k-1) > 0.$$

3 ⇒ 1 : Sei $a'' = (a_\ell, \ldots, a_m)$ mit $i \leq \ell \leq m \leq j$ sowie $i \neq \ell$ oder $m \neq j$ eine echte Teilfolge von $a' = (a_i, \ldots, a_j)$. Dann gilt wegen der Voraussetzung 2 für a' mit $k = \ell - 1 \in [i : j]$ bzw. $k = m + 1 \in [i : j]$ (und auch $\sigma(i, i-1) = 0 = \sigma(j+1, j)$):

$$\sigma(\ell, m) = \sigma(i, j) - \underbrace{\sigma(i, \ell-1)}_{\geq 0} - \underbrace{\sigma(m+1, j)}_{\geq 0} < \sigma(i, j).$$

Hinweis: $\sigma(i, \ell - 1)$ und $\sigma(m + 1, j)$ sind jeweils größer oder gleich 0, eines davon muss jedoch echt größer als 0 sein, da ansonsten $\ell = i$ und $m = j$ gilt und somit $a' = a''$ gelten würde.

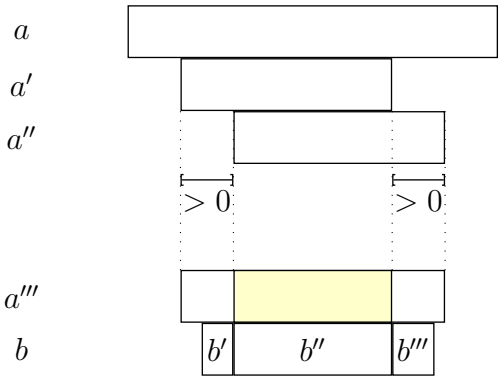
Damit ist gezeigt, dass E1 gilt. ■

Lemma 1.6 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Die maximal bewerteten Teilfolgen von a sind paarweise disjunkt.*

Beweis: Wir führen auch hier den Beweis durch Widerspruch. Seien a' und a'' zwei maximal bewertete Teilfolgen von a. Nehmen wir zunächst an, dass a' eine Teilfolge von a'' ist. Dies kann jedoch nicht sein, da dann a'' eine Oberfolge von a' ist, die E1 erfüllt (da a'' eine maximal bewertete Teilfolge ist). Also erfüllt a' nicht E2 und kann somit keine maximal bewertete Teilfolge sein und wir erhalten den gewünschten Widerspruch. Der Fall, dass a'' eine Teilfolge von a' ist, ist analog.

22.10.24

Seien also a' und a'' jetzt zwei maximal bewertete Teilfolgen von a, die sich überlappen. Dies ist in Abbildung 1.18 illustriert. Wir zeigen jetzt, dass die Folge a''', die als Vereinigung der Folgen a' und a'' (respektive ihrer Indizes) definiert ist, die Eigenschaft E1 erfüllt. Zuerst halten wir noch fest, dass dann $\sigma(a''') > \max\{\sigma('a), \sigma(a'')\}$ gilt. Sei dazu b eine beliebige Teilfolge von a'''. Ist b eine Teilfolge von a' (bzw. a'') dann gilt aufgrund der Eigenschaft E1 von a' (bzw. a'') $\sigma(b) \leq \sigma(a') < \sigma(a''')$ (bzw. $\sigma(b) \leq \sigma(a') < \sigma(a''')$).



wegen E1 & Lemma 1.5 Teil 3).

$\sigma(a''') > \max\{\sigma(a'), \sigma(a'')\}$

Abbildung 1.18: Skizze: Zwei überlappende maximal bewerteten Teilfolgen

Sei also nun b keine Teilfolge von a' oder a''. Sei weiter $b = b' \cdot b'' \cdot b'''$, so dass $b' \cdot b''$ ein Suffix von a' und $b'' \cdot b'''$ ein Präfix von a'' ist. Dann gilt:

$$\begin{aligned}
\sigma(b) &= \sigma(b' \cdot b'') + \sigma(b'' \cdot b''') - \sigma(b'') \\
&\quad \text{da } \sigma(b' \cdot b'') \leq \sigma(a') \text{ wegen E1 von } a' \\
&\quad \text{und } \sigma(b'' \cdot b''') \leq \sigma(a'') \text{ wegen E1 von } a'' \\
&\quad \text{(eine davon ist strikt, da } b'b'' \neq a' \vee b''b''' \neq a'') \\
&< \sigma(a') + \sigma(a'') - \sigma(b'') \\
&= \sigma(a''')
\end{aligned}$$

Dies ergibt den gewünschten Widerspruch zur Eigenschaft E2 von a' und a'', da a''' E1 erfüllt und eine Oberfolge von a' und a'' ist. Beachte, dass eine der Ungleichungen $\sigma(b' \cdot b'') \leq \sigma(a')$ und $\sigma(b'' \cdot b''') \leq \sigma(a'')$ strikt sein muss, da sonst $b = a'''$ wäre. ■

Lemma 1.7 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Jede Teilfolge $a' = (a_k, \ldots, a_\ell)$, die E1 erfüllt, ist Teilfolge einer maximal bewerteten Teilfolge.*

Beweis: Wir führen den Beweis durch Widerspruch. Dazu sei $a' = (a_k, \ldots, a_\ell)$ ein *längstes* Gegenbeispiel. Also a' erfüllt E1 mit maximaler Länge, d.h. $\ell - k$ ist unter allen Gegenbeispielen maximal. Dies ist in Abbildung 1.19 illustriert.

Somit erfüllt a' die Eigenschaft E1, aber **nicht** E2 (sonst wäre a' kein Gegenbeispiel). Somit existiert eine echte Oberfolge $a'' = (a_i, \ldots, a_j)$ von a', die E1 erfüllt. Würde die Folge a'' auch E2 erfüllen, dann wäre a'' eine maximal bewertete Teilfolge, die auch eine Oberfolge von a' ist, d.h. a' wäre kein Gegenbeispiel.

Also erfüllt a'' die Eigenschaft E1, aber nicht E2. Besäße a'' eine maximal bewertete Oberfolge, so wäre diese auch eine maximal bewertete Oberfolge von a' und a' somit kein Gegenbeispiel.

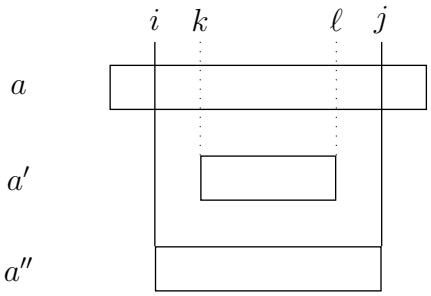


Abbildung 1.19: Skizze: a' ist ein Gegenbeispiel maximaler Länge

Also besitzt a'' keine maximal bewertete Oberfolge und erfüllt E1. Somit ist a'' ein längeres Gegenbeispiel als a', da $j - i > \ell - k$. Dies führt zu einem Widerspruch zur Annahme. ■

Korollar 1.8 *Jedes positive Element ist in einer maximal bewerteten Teilfolge enthalten.*

Beweis: Dies folgt aus dem vorherigen Lemma, da jede einelementige Folge mit einem positiven Wert die Eigenschaft E1 erfüllt. ■

Korollar 1.9 *Innerhalb jeder Teilfolge, die mit keiner maximal bewerteten Teilfolge überlappt, sind die aufaddierten Scores monoton fallend.*

Beweis: Nach dem vorherigen Korollar müssen alle Elemente einer solchen Folge nichtpositiv sein. ■

Lemma 1.10 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Betrachte $\mu = \min\{\sigma(1,k) \mid k \in [0:n]\}$ (bzw. $\mu = \max\{\sigma(1,k) \mid k \in [0:n]\}$). Sei k der größte (bzw. kleinste) Wert mit $\sigma(1,k) = \mu$. Dann beginnt an Position $k+1$ (bzw. endet an Position k) eine maximal bewertete Teilfolge oder es gilt $k = n$ (bzw. $k = 0$).*

Beweis: Betrachte $\sigma(1,i)$ als Funktion von $i \in [0:n]$ wie in Abbildung 1.20 dargestellt.

Allgemein gilt $\sigma(i,j) = \sigma(1,j) - \sigma(1,i-1)$.

Fall 1: Position $k > 0$ befindet sich in keiner maximal bewerteten Teilfolge. Da der folgende Wert a_{k+1} (sofern vorhanden) aufgrund der Definition von k positiv sein muss, ist nach Korollar 1.8 die Position k am Ende einer nicht maximal bewerteten

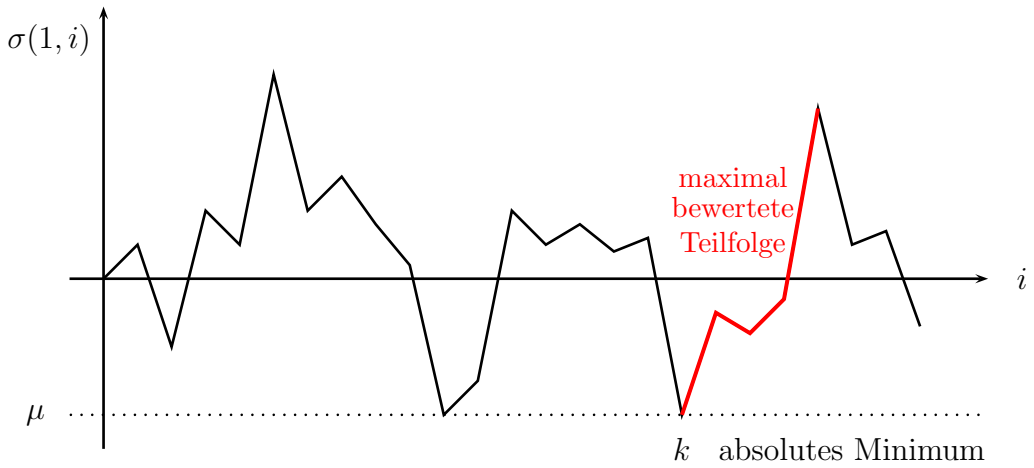


Abbildung 1.20: Skizze: Die Funktion $\sigma(1,i)$

Teilfolge. Also beginnt entweder an Position $k+1$ eine maximal bewertete Teilfolge oder es ist $k = n$.

Fall 2: Position $k > 0$ befindet sich in einer maximal bewerteten Teilfolge a'. Angenommen a' beginnt an Position $i \leq k$. Dann gilt aber nach Definition von k, dass $\sigma(i,k) \leq 0$ ist, was im Widerspruch zu Lemma 1.5 Charakterisierung 3 steht und Fall 2 kann gar nicht eintreten.

Fall 3: Falls die Position $k = 0$ ist, dann ist a_1 positiv und befindet sich nach Korollar 1.8 in einer maximal bewerteten Teilfolge und muss darin an erster Position sein.

Den Beweis für das Maximum von $\sigma(1,i)$ sei dem Leser überlassen. ■

Wir werden von nun an die folgende algorithmische Idee verfolgen: Sei a' eine Teilfolge von a, die die Eigenschaft E1 erfüllt. Wir werden versuchen a' zu einer maximal bewerteten Teilfolge zu verlängern.

Definition 1.11 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ eine Folge reeller Zahlen. Eine Teilfolge a' von a heißt a-MSS, wenn $a' \in \mathrm{MSS}(a)$.*

Lemma 1.12 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Sei a'' eine Teilfolge von a' und a' eine Teilfolge von a. Ist a'' eine a-MSS, dann ist a'' auch a'-MSS.*

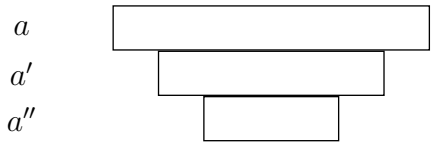


Abbildung 1.21: Skizze: a'' ist a-MSS

Beweis: Sei a'' eine a-MSS und sei sowohl a' eine Teilfolge von a als auch eine Oberfolge von a'', wie in Abbildung 1.21 dargestellt.

Da a'' eine a-MSS ist, erfüllt a'' die Eigenschaften E1 und E2 bezüglich a. Somit erfüllt die Folge a'' diese Eigenschaften auch bezüglich der Oberfolge a'. ■

Lemma 1.13 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Sei $a' = (a_1, \ldots, a_i)$ und $a_k \leq 0$ für alle $k \in [i{+}1 : n]$. Alle maximal bewerteten Teilfolgen von a' sind auch maximal bewertete Teilfolgen von a und umgekehrt.*

Beweis: Wir beweisen beide Implikationen getrennt.

⇐: Dies ist die Aussage von Lemma 1.12, da nach Lemma 1.5 Charakterisierung 3 jede a-MSS eine Teilfolge von a' sein muss.

⇒: Sei a'' eine a'-MSS, wie in Abbildung 1.22 illustriert. Wir führen den Beweis

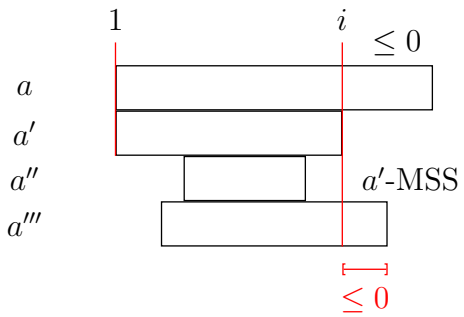


Abbildung 1.22: Skizze: a'' ist eine a'-MSS

durch Widerspruch und nehmen daher an, dass a'' keine a-MSS ist. Dann muss es eine Oberfolge a''' von a'' geben, die E1 erfüllt. Da a'' eine a'-MSS ist, muss diese Oberfolge a''' in den hinteren Teil (ab Position $i+1$) hineinragen. Eine solche Folge a''' kann nach Lemma 1.5 Charakterisierung 3 nicht E1 erfüllen, da der Suffix von a''' ab Position $i+1$ einen nichtpositiven Score besitzt. Dies führt zu dem gewünschten Widerspruch. ■

Lemma 1.14 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Sei $a' = (a_i, \ldots, a_j)$ eine a-MSS und sei $a^L = (a_1, \ldots, a_{i-1})$ und $a^R = (a_{j+1}, \ldots, a_n)$. Dann ist eine Teilfolge a'' von a^L (bzw. a^R) genau dann eine a^L-MSS (bzw. a^R-MSS), wenn $a'' \neq a'$ eine a-MSS ist.*

Beweis: Wir beweisen beide Implikationen getrennt.

$\Leftarrow$: Nach Lemma 1.6 sind maximal bewertete Teilfolgen disjunkt. Somit ist jede andere maximal bewertete Teilfolge eine Teilfolge von a^L oder a^R. Mit Lemma 1.12 ist nun jede maximal bewertete Teilfolge von a ungleich a' entweder eine maximal bewertete Teilfolge von a^L oder a^R.

$\Rightarrow$: Wir führen den Beweis durch Widerspruch. Sei dazu a'' eine a^L-MSS, aber keine a-MSS. Somit erfüllt a'' die Eigenschaft E1. Mit Lemma 1.7 folgt, dass es eine Oberfolge $\overline{a}$ von a'' gibt, die eine a-MSS ist.

Angenommen $\overline{a}$ wäre eine Teilfolge von a^L. Dann wäre nach Lemma 1.12 $\overline{a}$ auch eine a^L-MSS. Somit wären $\overline{a}$ und a'' überlappende a^L-MSS, was Lemma 1.6 widerspricht.

Somit müssen sich $\overline{a}$ und a' überlappen. Da aber beide a-MSS sind, ist dies ebenfalls ein Widerspruch zu Lemma 1.6. ■

Damit können wir nun die Äquivalenz der strukturellen und prozeduralen Definition maximal bewerteter Teilfolgen zeigen.

Theorem 1.15 *Die strukturelle und prozedurale Definition von maximal bewerteten Teilfolgen stimmen überein.*

Beweisidee: Sei $\mathrm{MSS}(a)$ die Menge aller Teilfolgen von a, die maximal bewertete Teilfolgen sind (gemäß der strukturellen Definition, also die die Eigenschaften E1 und E2 erfüllen).

Sei a' eine kürzeste Teilfolge mit maximalem Score. Man überlegt sich leicht, dass diese in a maximal bewertet ist, d.h. dass a' die Eigenschaften E1 und E2 erfüllt sind (insbesondere weil wir eine kürzeste Folge a' mit maximalem Score wählen) und somit $a' = (a_i, \ldots, a_j) \in \mathrm{MSS}(a)$ gilt. Nach Lemma 1.14 gilt dann:

$$\mathrm{MSS}(a) = \{a'\} \cup \mathrm{MSS}(a_1, \ldots, a_{i-1}) \cup \mathrm{MSS}(a_{j+1}, \ldots, a_n).$$

Der vollständige Beweis lässt sich dann formal mittels vollständiger struktureller Induktion führen. ■

1.2.3 Ein Algorithmus zur Lösung

Wir beschreiben jetzt einen Algorithmus zur Ermittlung aller maximal bewerteten Teilfolgen.

- Die Eingabe ist die Folge $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$.
- Die Elemente werden von links nach rechts verarbeitet. Wir betrachten also der Reihe nach jedes Präfixes $a' = (a_1, \ldots, a_m)$ mit $m \in [0 : n]$.
- Dabei merken wir uns disjunkte Teilfolgen $I_1, \ldots, I_k$ eines Präfixes a' von a, die maximal bewertete Teilfolgen des bereits abgearbeiteten Präfixes a' sein werden.
- $I_i = (a_{\ell_i}, \ldots, a_{r_i})$, d.h. $I_i \hat{=} (\ell_i, r_i)$.
- Setze $L_i = \sigma(1, \ell_i - 1)$ und $R_i = \sigma(1, r_i)$.
- Damit gilt $\sigma(I_i) := \sigma(\ell_i, r_i) = R_i - L_i$.

Zu Beginn ist a' die leere Folge und wir setzen $m := 0$ und $k := 0$ und merken uns dazu $S = \sigma(1, m)$ (also zu Beginn $S := 0$) und aktualisieren $S := S + a_m$ nach einer Erhöhung von m. Die Werte von S können vor bzw. nach der Aktualisierung verwendet werden, um ggf. L_i bzw. R_i zu bestimmen.

Das können wir uns für $a' = (a_1, \ldots, a_m) = (+5, -6, +4, -5, +3, -1, +2, -3)$ wie in Abbildung 1.23 veranschaulichen (dabei ist $a_{m+1} = +6$).

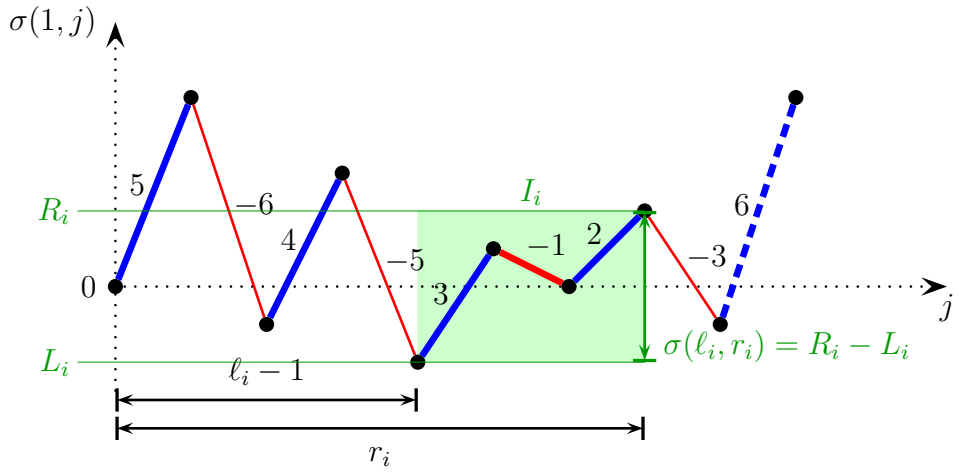


Abbildung 1.23: Skizze: Die Werte ℓ_i, r_i, L_i, R_i und I_i

Wir gehen wie folgt vor: Bearbeite die Folge von links nach rechts und sammle eine Liste von maximal bewerteten Teilfolgen bezüglich des betrachteten Präfixes a'.

24.10.24

Sei a_m das aktuell betrachtete Element der Folge a (nach einer Erhöhung von m):

- Ist $a_m \leq 0$, betrachte die nächste Position $m+1$, da keine maximal bewertete Teilfolge an Position m beginnen oder enden kann (siehe Lemma 1.13).
- Ist $a_m > 0$, inkrementiere k, erzeuge eine neue einelementige Liste $I_k = (m, m)$ und bestimme $L_k := S$ sowie $R_k := S + a_m$. Die Folge (a_m) erfüllt E1, aber nicht notwendigerweise E2 im betrachteten Präfix $a' = (a_1, \ldots, a_m)$ von a.
- Für ein neues I_k wiederhole das Folgende: Durchsuche $I_{k-1}, \ldots, I_1$ (von rechts nach links!) bis ein maximales j gefunden wird, so dass $L_j < L_k$ und betrachte die folgenden drei Fälle genauer. Hierbei gilt immer, dass $I_1, \ldots, I_{k-1}$ maximal bewertete Teilfolgen in $(a_1, \ldots, a_{\ell_k - 1})$ sind (zu Beginn ist jeweils $\ell_{k-1} = m-1$) und I_k zumindest die Eigenschaft E1 erfüllt.

  Fall 1: Es existiert gar kein solches j, siehe Abbildung 1.24.

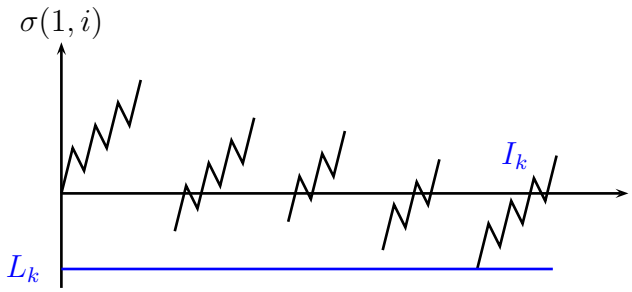


Abbildung 1.24: Skizze: Nichtexistenz von j

  Nach Lemma 1.10 ist I_k der Anfang einer maximal bewerteten Teilfolge in der Teilfolge $a' = (a_1, \ldots, a_m)$. Nach Lemma 1.14 sind $I_1, \ldots, I_{k-1}$ dann auch maximal bewertete Teilfolgen in a'.

  Nach Lemma 1.7 ist I_k auch Teilfolge einer a'-MSS a''. Die Teilfolge I_k ist sogar Anfang von a'', denn sonst hätten einige Präfixe von a'' einen nichtpositiven Score, was ein Widerspruch zu Lemma 1.5 wäre. a' muss sogar der Beginn einer a-MSS sein, da keine Oberfolge von I_k, die vor ℓ_k beginnt, die Eigenschaft E1 erfüllen kann. Somit sind $I_1, \ldots, I_{k-1}$ nach Lemma 1.14 auch je eine a-MSS.

  Im Algorithmus geben wir daher die Teilfolgen $I_1, \ldots, I_{k-1}$ als maximal bewerteten Teilfolgen aus und setzen $I_1 := I_k$, $(\ell_1, r_1) = (\ell_k, r_k)$, $(L_1, R_1) := (L_k, R_k)$ sowie $k := 1$. □

  Fall 2: Sei j maximal mit $L_j < L_k$ und sei $R_j < R_k$, siehe Abbildung 1.25.

  Offensichtlich gilt nach Wahl von j, dass $L_i \geq L_k$ für $i \in [j+1 : k-1]$. Weiterhin gilt $R_i \leq R_j$ für alle $i \in [j+1 : k-1]$. Andernfalls gäbe es ein

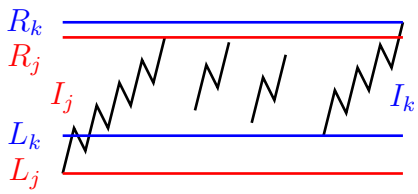


Abbildung 1.25: j maximal mit $L_j < L_k$ und $R_j < R_k$

$i \in [r_j + 1 : \ell_k - 1]$ mit $\sigma(1, i) > R_j$. Sei i ein minimaler solcher Index. Dann besäße I_j die Oberfolge $(a_{\ell_j}, \ldots, a_i)$, die offensichtlich die Eigenschaft E1 erfüllt, was der Eigenschaft E2 von I_j in $(a_1, \ldots, a_{\ell_k - 1})$ widerspricht. Beachte auch, dass wegen Korollar 1.8 alle Elemente außerhalb der Teilfolge, $I_1, \ldots, I_k$ nichtpositiv sein müssen.

Somit erfüllt die Teilfolge $(a_{\ell_j}, \ldots, a_{r_k})$ die Eigenschaft E1, aber nicht notwendigerweise E2. Somit muss ein neue Teilfolge generiert und angefügt werden. Wir bilden aus den Teilfolgen $I_j, \ldots, I_k$ eine neue Teilfolge, d.h. wir setzen $I_j := (\ell_j, r_k)$, $r_j := r_k$, $R_j := R_k$ sowie $k := j$ und wiederholen die Prozedur für $I_k = I_j$ (also mit dem neuen k). Die alten Teilfolgen $I_{j+1}, \ldots, I_k$ müssen dabei natürlich gelöscht werden.

Alle 3 Fälle sind jetzt wieder neu zu betrachten. □

Fall 3: Sei j maximal mit $L_j < L_k$ und $R_j \geq R_k$, siehe Abbildung 1.26.

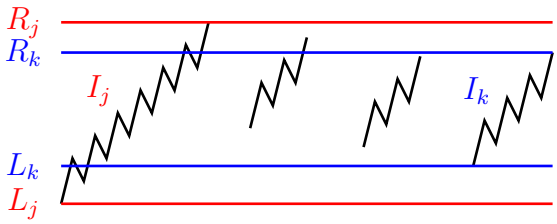


Abbildung 1.26: Skizze: j maximal mit $L_j < L_k$ und $R_j \geq R_k$

Wir zeigen zunächst, dass dann I_j eine a'-MSS mit $a' = (a_1, \ldots, a_m)$ ist.

E1) Dies gilt nach Konstruktion, da I_j $(j < k)$ bereits eine maximal bewertete Teilfolge von $(a_1, \ldots, a_{\ell_k - 1})$ ist und somit die Eigenschaft E1 gilt.

E2) Für einen Widerspruchsbeweis nehmen wir an, es gäbe eine Oberfolge $a'' = (a_i, \ldots, a_s)$ von I_j, die die Eigenschaft E1 erfüllt.

Gilt $r_j \leq s < \ell_k \leq m$, dann wäre a'' auch eine Oberfolge von I_j, die in $(a_1, \ldots, a_{\ell_k - 1})$ liegt, und somit wäre I_j keine maximal bewertete Teilfolge von $(a_1, \ldots, a_{\ell_k - 1})$ gewesen, was aufgrund unserer Konstruktion ein Widerspruch ist.

Also muss $s \in [\ell_k : r_k]$ gelten. Dann gibt es aber offensichtlich Suffixe von a'' mit nichtpositiven Score, was nach Lemma 1.5 Charakterisierung 3) nicht sein kann.

Jetzt zeigen wir noch, dass auch $I_k = (a_{\ell_k}, \ldots, a_{r_k})$ eine maximal bewertete Teilfolge von $a' = (a_1, \ldots, a_m)$ ist. Da I_j eine maximal bewertete Teilfolge von a' ist genügt es nach Lemma 1.14 zu zeigen, dass I_k eine maximal bewertete Teilfolge von $a'' = (a_{r_j+1}, \ldots, a_m)$ ist. Aus Lemma 1.10 folgt aber unmittelbar, dass an Position $\ell_k \leq m$ eine maximal bewertete Teilfolge beginnt. Da jeder echte Präfix von I_k die Folgen I_k als Oberfolge besitzt und diese die Eigenschaft E1 erfüllt, kann kein echter Präfix von I_k maximal bewertet sein. Also muss I_k selbst eine a''-MSS sein und die Behauptung gilt.

In diesem Fall ist algorithmisch also gar nichts zu tun. □

In jedem Fall wird das Durchsuchen der Listen $I_{k-1}, \ldots, I_1$ entweder erneut aufgerufen oder aber alle aufgesammelten Teilfolgen $I_1, \ldots, I_k$ sind maximal bewertete Teilfolgen des Präfixes $a' = (a_1, \ldots, a_m)$ von a (die bereits ausgegebenen maximal bewerteten Folgen von a ignorieren wir einfach).

In Abbildung 1.27 auf Seite 24 ist ein Ablauf des gerade beschriebenen AMSS-Algorithmus für die Sequenz $a = (+3, -2, +3, -3, +2, -4, +3)$ schematisch dargestellt. Hierbei entsprechen die roten Linienzüge den gespeicherten disjunkten Teilfolgen $I_1, \ldots, I_k$, während die grünen Linienzüge die nach Fall 1 ausgegeben maximal bewerteten Teilfolgen darstellen.

1.2.4 Zeitkomplexität

Wir analysieren jetzt die Zeitkomplexität des soeben vorgestellen Algorithmus:

1.) Die Betrachtung von a_m und der eventuell Generierung des neuen I_k kann pro Folgenelement in konstanter Zeit durchgeführt werden. Daraus ergibt sich ein Gesamtaufwand von $O(n)$.

2.) Das Durchsuchen der bislang gesammelten Teilfolgen $I_{k-1}, \ldots, I_1$ wird anschließend analysiert.

3.) Gesamtaufwand von Fall 1: Es existiert kein j, dann geben wir $I_1, \ldots, I_{k-1}$ aus. Dies lässt sich für jedes I_ℓ in konstanter Zeit erledigen, also erhalten wir insgesamt für alle Ausgaben einen Zeitbedarf von $O(n)$. Es kann nur $O(n)$ maximal bewertete Teilfolgen geben, da diese ja nach Lemma 1.6 disjunkt sind.

4.) Gesamtaufwand von Fall 3: Es gilt $L_j < L_k \wedge R_j \geq R_k$. Die Ermittlung des dritten Falles geschieht in konstanter Zeit. Da anschließend das Durchsuchen

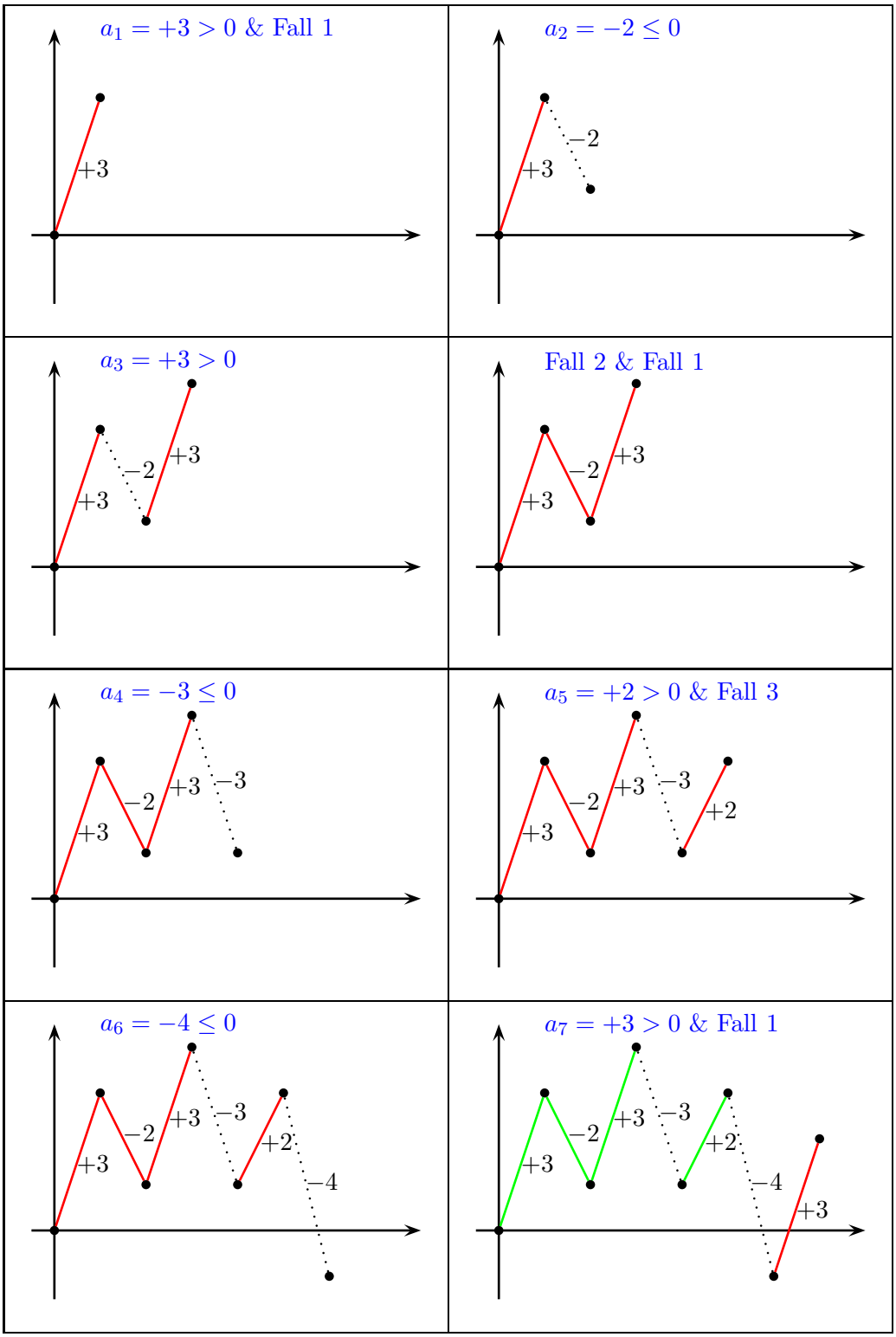


Abbildung 1.27: Beispiel: $a = (+3, -2, +3, -3, +2, -4, +3)$

beendet ist, tritt dieser Fall maximal n-mal auf, also ist der Aufwand für alle diese Fälle höchstens $O(n)$.

5.) Gesamtaufwand von Fall 2: Es gilt $L_j < L_k \wedge R_j < R_k$. Die Verschmelzung von $I_j, \ldots, I_k$ lässt sich in Zeit $O(k-j+1)$ erledigen (explizites Löschen der Teilfolgen). Da insgesamt höchstens $n-1$ Verschmelzungen von einzelnen Teilfolgen möglich sind (spätestens dann müssen sich alle Folgenglieder in einer einzigen Folge befinden), beträgt der Gesamtzeitbedarf $O(n)$.

Dabei stellen wir uns eine Verschmelzung von ℓ Teilfolgen als $\ell-1$ Verschmelzungen von je zwei Teilfolgen vor. Da es insgesamt maximal $n-1$ Verschmelzungen disjunkter Teilfolgen geben kann, folgt obige Behauptung.

Durchsuchen der Listen: Beim Durchsuchen der Listen von Teilfolgen gehen wir etwas geschickter vor. Wir merken uns für jede Teilfolge I_k dabei die Teilfolge I_j, für die $L_j < L_k$ und j maximal ist. Beim nächsten Durchlaufen können wir dann einige Teilfolgen beim Suchen einfach überspringen. Dies ist in der folgenden Abbildung 1.28 schematisch dargestellt. Dabei stellen die grünen Pfeile die gespeicherten Vorgänger bzgl. der L-Werte dar und der blaue der für I_k neu eingeführt wird. Für die Bestimmung von L_k werden die hellen Pfeile abgelaufen.

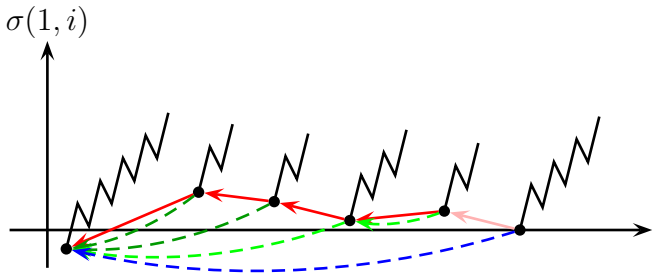


Abbildung 1.28: Skizze: Übersprungene Folge von L_j-Werten beim Durchsuchen

Jetzt müssen wir uns nur noch überlegen, welchen Aufwand alle Durchsuche-Operationen insgesamt haben. Dabei werden die Kosten zum einen auf die Aufrufe (der Durchsucheprozedur) und zum anderen auf die ausgeschlossenen Teilfolgen verteilt.

Wenn bei einem Durchsuchen der Liste ℓ Teilfolgen übersprungen werden, dann werden $\ell+2$ Teilfolgen inspiziert (und ℓ Teilfolgen ausgeschlossen), was Kosten in Höhe von $O(\ell+1)$ verursacht (man beachte, dass $\ell \geq 0$). Die Kosten in Höhe von $O(\ell)$ werden anteilig auf die ausgeschlossenen Teilfolgen umgelegt. Somit erhält jedes ausgeschlossene Teilfolge Kosten von $O(1)$. Die Kosten für den Rest werden auf den Aufruf eines Durchsuch-Durchlaufs umgelegt, die dann ebenfalls konstant sind. Somit fallen jeweils Kosten $O(1)$ für jede ausgeschlossene Teilfolge und jeden Aufruf an.

Anzahl Aufrufe: Es kann maximal so viele Aufrufe geben, wie der Zeitbedarf in der Analyse der Punkte 3, 4 und 5 verbraucht wird, da ja nach jedem Durchsuchen entweder Fall 1, Fall 2 oder Fall 3 ausgeführt wird (die mindestens konstante Kosten verursachen). Also ist die Anzahl Aufrufe $O(n)$.

Anzahl ausgeschlossener Teilfolgen: Zum einen können diese im Fall 1 ausgegeben werden, von diesen kann es maximal $O(n)$ viele geben. Wenn diese Teilfolgen nicht selbst ausgegeben werden, müssen diese irgendwann mit anderen Teilfolgen verschmolzen worden sein (sie können ja nicht einfach verschwinden). Da, wie wir bereits gesehen haben, maximal $O(n)$ Teilfolgen verschmolzen werden, können auch hierdurch höchstens $O(n)$ Teilfolgen ausgeschlossen werden.

Mit unserem kleinen Trick kann also auch das Durchsuchen der Listen von Teilfolgen mit einem Aufwand von insgesamt $O(n)$ bewerkstelligt werden. Halten wir das Resultat noch im folgenden Satz fest.

29.10.24

Theorem 1.16 *Das All Maximal Scoring Subsequences Problem für eine gegebene reelle Folge kann in Linearzeit gelöst werden.*

1.3 Bounded All Maximum Scoring Subsequences (*)

Im vorherigen Abschnitt haben wir quasi über eine lokale Definition die Menge aller maximal bewerteten Teilfolgen bestimmt. Wir können auch einen globalen Ansatz wählen und eine Menge von Teilfolgen einer gegebenen Folge fordern, deren aufaddierte Scores maximal ist. Dies ist jedoch nicht sinnvoll, da dann die Menge aller einelementigen Teilfolgen, die positive Elemente beschreiben, eine Lösung dieses Problems ist. Man überlegt sich leicht, dass dies wirklich ein globales Optimum ist.

Daher wollen wir nun das Problem ein wenig abwandeln, indem wir die Länge der Teilfolgen in der Lösungsmenge sowohl von oben als auch von unten beschränken.

1.3.1 Problemstellung

Es wird zusätzlich eine untere $\underline{\lambda}$ und eine obere Schranke $\overline{\lambda}$ vorgegeben, um die Länge der zu betrachtenden Teilfolgen zu beschränken, wobei natürlich $\underline{\lambda} \leq \overline{\lambda} \in \mathbb{N}$ gilt. Mit $Seq(n,k)$ bezeichnen wir die Menge aller 0-1-Zeichenreihen mit genau k konsekutiven 1-Runs, deren Länge durch $\underline{\lambda}$ nach unten und mit $\overline{\lambda}$ nach oben beschränkt ist.

Notation 1.17 *Sei $k \leq n \in \mathbb{N}$ und $\underline{\lambda} \leq \overline{\lambda} \in \mathbb{N}$, dann bezeichne*

$$Seq(n,k) := \{0^*1^{m_1}0^+1^{m_2}0^+\cdots 0^+1^{m_k}0^* \mid \underline{\lambda} \leq m_i \leq \overline{\lambda}\} \cap \{0,1\}^n \subseteq \{0,1\}^n.$$

Bounded All Maximum Scoring Subsequences (BAMSS)

Eingabe: Eine Folge $(a_1, \ldots, a_n) \in \mathbb{R}^n$, $\underline{\lambda} \leq \overline{\lambda} \in \mathbb{N}$.
Gesucht: Eine Sequenz $s \in \bigcup_{k=0}^{n} Seq(n,k)$, die $\sum_{i=1}^{n} s_i \cdot a_i$ maximiert.

Bemerkung: Durch die Längenbeschränkung können nach unserer alten Definition von allen Maximal Scoring Subsequences die einzelnen Teilfolgen der Lösung überlappen, siehe Abbildung 1.29. Damit ist ein Greedy-Ansatz nicht mehr effizient mög-

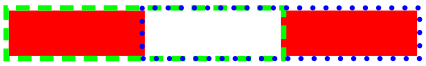


Abbildung 1.29: Skizze: Erlaubte überlappende Sequenzen

lich. Aus diesem Grund wurde das Problem anders gefasst, nämlich wie oben. Mit Hilfe der Menge $Seq(n,k)$ werden aus der Gesamtfolge Teilstücke ausgewählt (nämlich 1-Runs), die wir als Lösungsteilfolgen zulassen wollen.

Weiterhin sollte man beachten, dass für eine gegeben der Folge der Länge n auch für $\underline{\lambda} = 1$ und $\overline{\lambda} = n$ die Lösungen für das BAMSS und AMSS verschieden sein können. Betrachte hierzu die Folge $a = (+3, -2, +3)$. Die eindeutige Lösung für AMSS lautet $\{(a_1, a_2, a_3)\}$ und für BAMSS $\{(a_1), (a_3)\}$.

1.3.2 Lösung mittels Dynamischer Programmierung

Wir definieren wieder eine Tabelle S für $i, k \in [0:n]$ wie folgt:

$$S(i,k) := \max \left\{ \sum_{j=1}^{i} s_j \cdot a_j \ : \ s \in Seq(i,k) \right\}$$

Wir müssen also $S(i,k)$ für alle Werte $i \in [0:n]$, $k \in [0:n]$ berechnen, wobei $\max \emptyset = \max\{\} = -\infty$ (entspricht dem neutralem Element für das Minimum) gilt.

Für diese Tabelle S lässt sich die folgende Rekursionsgleichung aufstellen:

$$\begin{aligned}
S(i,0) &= 0 \quad \text{für } i \in [-1:n], \\
S(i,k) &= -\infty \quad \text{für } i < k \cdot \underline{\lambda} + (k-1), \\
S(i,k) &= \max \left\{ \begin{array}{c} S(i-1,k), \\ S(i-\lambda-1,k-1) + \sum\limits_{j=i-\lambda+1}^{i} a_j \end{array} \ : \ \lambda \in [\underline{\lambda} : \min(\overline{\lambda}, i)] \right\}.
\end{aligned}$$

Die Korrektheit der Rekursionsgleichung ergibt sich aus der Tatsache, dass man sich eine optimale Menge von Teilfolgen anschaut und unterscheidet, ob diese mit einer 0 oder einem 1-Run endet, wie in Abbildung 1.30 illustriert.

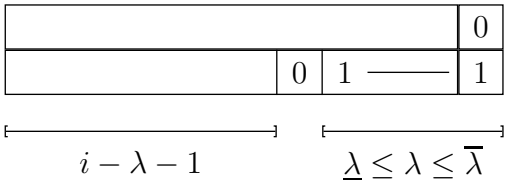


Abbildung 1.30: Skizze: Aufstellen der Rekursionsgleichung

Wie man leicht sieht, gilt:

$$\sum_{j=i-\lambda+1}^{i} a_j = \sum_{j=1}^{i} a_j - \sum_{j=1}^{i-\lambda} a_j$$

Somit kann man die einzelnen Summen effizient berechnen, wenn für alle $i \in [0:n]$ die folgenden Summen bekannt sind: $A(i) := \sum_{j=1}^{i} a_j$. Berechnet man die Werte $A(i)$ für $i \in [0:n]$ vorab, so lässt sich jeder Eintrag der Tabelle S in Zeit $O(\overline{\lambda}-\underline{\lambda}) = O(n)$ berechnen. Falls die Summen zu groß werden sollten, kann man diese auch für einige jeweils kleinere Teilbereiche ermitteln.

Die Gesamtlaufzeit beträgt somit $O(n^2/\underline{\lambda}(\overline{\lambda} - \underline{\lambda})) \leq O(n^2(\overline{\lambda} - \underline{\lambda})) \leq O(n^3)$. In der Praxis sind die Schranken $\underline{\lambda}$ und $\overline{\lambda}$ allerdings so nah beieinander, so dass $\overline{\lambda}-\underline{\lambda} = O(1)$ gilt und somit die Laufzeit bei $O((\overline{\lambda} - \underline{\lambda})n^2) = O(n^2)$ bleibt. Gilt $(\overline{\lambda} - \underline{\lambda}) = O(\underline{\lambda})$, so bleibt es auch bei einer Laufzeit von $O(n^2)$.

Theorem 1.18 *Das BAMSS-Problem lässt sich in Zeit $O(n^2/\underline{\lambda} \cdot (\overline{\lambda} - \underline{\lambda}))$ lösen.*

Wie wir im nächsten Abschnitt sehen werden, können wir das Problem auch effizienter lösen. Die vorhergehende Lösung hat den Vorteil dass man auch die Maximalanzahl der Teilfolgen einer Lösung beschränken kann, was durchaus biologisch sinnvoll sein kann.

1.3.3 Effiziente Lösung mittels Dynamischer Programmierung

Wir definieren jetzt folgende Tabelle S für $i \in [0:n]$ wie folgt:

$$S(i) := \max \left\{ \sum_{j=1}^{i} s_j \cdot a_j \ : \ s \in Seq(i,k) \wedge k \in [0:n] \right\}$$

Wir müssen also in diesem Fall $S(i)$ für alle Werte $i \in [0 : n]$ berechnen, wobei $\max \emptyset = \max\{\} = -\infty$ (entspricht dem neutralem Element) gilt.

Für diese Tabelle S lässt sich die folgende Rekursionsgleichung aufstellen:

$$\begin{aligned} S(i) &= 0 \quad \text{für } i \in [-1 : \underline{\lambda} - 1], \\ S(i) &= \max \left\{ \begin{array}{c} S(i-1), \\ S(i-\lambda-1) + \sum_{j=i-\lambda+1}^{i} a_j \end{array} \; : \; \lambda \in [\underline{\lambda} : \min\{i, \overline{\lambda}\}] \right\}. \end{aligned}$$

Die Korrektheit der Rekursionsgleichung ergibt sich wie vorher. Für eine einfachere Angabe der Rekursionsgleichung benötigen wir auch hier noch die Definition $S(-1) = 0$.

Die Gesamtlaufzeit beträgt somit $O(n(\overline{\lambda} - \underline{\lambda})) \leq O(n^2)$. In der Praxis sind die Schranken $\underline{\lambda}$ und $\overline{\lambda}$ allerdings so nah beieinander, so dass $\overline{\lambda} - \underline{\lambda} = O(1)$ gilt und somit die Laufzeit bei $O((\overline{\lambda} - \underline{\lambda})n) = O(n)$ bleibt.

Theorem 1.19 *Das BAMSS-Problem lässt sich in Zeit $O(n(\overline{\lambda} - \underline{\lambda}))$ lösen.*

1.4 Bounded Maximal Scoring Subsequence (*)

Jetzt wollen wir nur *eine* längenbeschränkte Maximal Scoring Subsequence finden. Man beachte, dass diese kein Teil der Lösung aus dem Problem des vorherigen Abschnittes sein muss! Wir werden zunächst nur eine obere Längenbeschränkung für die gesuchte Folge betrachten. Eine Hinzunahme einer unteren Längenbeschränkung ist nicht weiter schwierig und wird in den Übungen behandelt.

1.4.1 Problemstellung

Wir formalisieren zunächst die Problemstellung.

Bounded Maximal Scoring Subsequence (BMSS)

Eingabe: Eine Folge $(a_1, \ldots, a_n) \in \mathbb{R}^n$ und $\lambda \in \mathbb{N}$.
Gesucht: Eine (zusammenhängende) Teilfolge $(a_i, \ldots, a_j)$ mit $j - i + 1 \leq \lambda$, die $\sigma(i, j)$ maximiert, wobei $\sigma(i, j) = \sum_{\ell=i}^{j} a_\ell$.

1.4.2 Links-Negativität

Um einer effizienten Lösung des Problems näher zu kommen, benötigen wir zuerst den Begriff der Links-Negativität und einer minimalen linksnegativen Partition einer reellen Folge.

Definition 1.20 *Eine Folge $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ reeller Zahlen heißt* linksnegativ, *wenn $\sum_{\ell=1}^{k} a_\ell \leq 0$ für alle $k \in [1 : n-1]$.*

Eine Partition $a = A_1 \cdots A_k$ der Folge a heißt minimal linksnegativ, *wenn für alle $i \in [1 : k]$ A_i linksnegativ ist und $\sigma(A_i) > 0$ für alle $i \in [1 : k-1]$ ist.*

Beispiele:

1.) $(-1, 1, -3, 1, 1, 3)$ ist linksnegativ,

2.) $(2, -3, -4, 5, 3, -3)$ ist **nicht** linksnegativ.

Die Partition $(2)(-3, -4, 5, 3)(-3)$ ist eine minimal linksnegative.

Lemma 1.21 *Jede Folge $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ lässt sich eindeutig in eine minimal linksnegative Partition zerlegen.*

Beweis: Wir führen den Beweis durch Induktion über n.

Induktionsanfang ($n = 1$): Dies folgt unmittelbar aus der Definition.

Induktionsschritt ($n \to n + 1$): Betrachte $a' = (a_0, a_1, \ldots, a_n)$ und sei dann $a = (a_1, \ldots, a_n)$. Nach Induktionsvoraussetzung sei $a = A_1 \cdots A_k$ die(!) minimal linksnegative Partition von a.

Fall 1 ($a_0 > 0$): Indem wir a_0 einfach als Segment an die minimal linksnegative Partition von a voranstellen, erhalten wir eine minimal linksnegative Partition für $a' = (a_0) \cdot A_1 \cdots A_k$.

Fall 2 ($a_0 \leq 0$): Wähle ein minimales i mit

$$a_0 + \sum_{j=1}^{i} \sigma(A_j) > 0.$$

Dann ist $((a_0) \cdot A_1 \cdots A_i) \cdot A_{i+1} \cdots A_k$ eine minimal linksnegative Partition. Hierzu genügt es zu zeigen, dass $(a_0) \cdot A_1 \cdots A_i$ linksnegativ ist. Nach Konstruktion gilt

$\sigma((a_0)\cdot A_1 \cdots A_j) \leq 0$ für $j \in [1 : i-1]$. Auch für jedes echte Präfix $(a_0)\cdot A_1 \cdots A_{j-1} \cdot A'_j$ davon muss $\sigma((a_0) \cdot A_1 \cdots A_{j-1} \cdot A'_j) \leq 0$ sein, da sonst bereits $\sigma(A'_j) > 0$ gewesen sein müsste. Dies kann nicht der Fall sein, da A_j linksnegativ ist, weil $A_1 \cdots A_k$ eine minimale linksnegative Partition von a ist.

Der Beweis der Eindeutigkeit sei dem Leser zur Übung überlassen. ■

Im Folgenden ist die Notation für die minimale linksnegative Partition der Suffixe der untersuchten Folge hilfreich.

Notation 1.22 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Jedes Suffix $a^{(i)} = (a_i, \ldots, a_n)$ von a besitzt dann die eindeutige minimal linksnegative Partition $a^{(i)} = A_1^{(i)} \cdots A_{k_i}^{(i)}$.*

Basierend auf dieser Notation lassen sich linksnegative Zeiger definieren.

Definition 1.23 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ eine reelle Folge und für jedes Suffix ab Position i sei $a^{(i)} = A_1^{(i)} \ldots A_{k_i}^{(i)}$ die zugehörige minimale linksnegative Partition. Sei weiter $A_1^{(i)} = (a_i, \ldots, a_{p(i)})$, dann heißt $p(i)$ der* linksnegative Zeiger *von i.*

Bemerkungen:

- Ist $a_i > 0$, dann ist $p(i) = i$.
- Ist $a_i \leq 0$, dann ist $p(i) > i$ für $i < n$ und $p(n) = n$.

1.4.3 Algorithmus zur Lösung des BMSS-Problems

Wie helfen uns minimale linksnegative Partitionen bei der Lösung des Problems? Wenn wir eine Teilfolge $a' = (a_i, \ldots, a_j)$ als Kandidaten für eine längenbeschränkte MSS gefunden haben, dann erhalten wir den nächstlängeren Kandidaten durch das

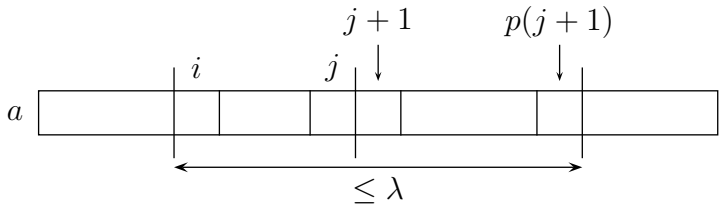


Abbildung 1.31: Skizze: Verlängerung einer maximal Scoring Subsequence

BMSS (real $a[]$, int n, λ)

```
begin
    // score and position of current mss
    int ms := 0,    mi := 1,    mj := 0;
    int i := 1,    j := 0 ;  /* (a_i, ..., a_j) is current candidate subsequence */
    for (i := 1; i ≤ n; i++) do
        while ((a_i ≤ 0) && (i < n)) do
            i++;
        j := max(i, j);
        while ((j < n) && (p[j + 1] < i + λ) && (σ(j + 1, p(j + 1)) > 0)) do
            j := p(j + 1);
            if (σ(i, j) > ms) then
                mi := i;
                mj := j;
                ms := σ(i, j);
end
```

Abbildung 1.32: Algorithmus: Lösung das BMSS Problems

Anhängen der Teilfolge $(a_{j+1}, \ldots, a_{p(j+1)})$. Nach Definition hätte jedes kürzere Stück einen nichtpositiven Score und würde uns nicht weiter helfen. Wir müssen dann nur noch prüfen, ob die obere Längenbeschränkung eingehalten wird. Dies ist in der folgenden Abbildung 1.31 veranschaulicht.

Mit Hilfe der linksnegativen Zeiger können wir das Problem mit dem in Abbildung 1.32 angegebenen Algorithmus leicht lösen. Wir versuchen hier ab jeder Position $i \in [1 : n]$ eine Maximal Scoring Subsequence der Länge höchstens λ zu finden.

Für die Laufzeit halten wir das Folgende fest. Nach maximal einer konstanten Anzahl von Operationen wird entweder i oder j erhöht, somit ist die Laufzeit $O(n)$.

Wir haben jedoch die linksnegativen Zeiger noch nicht berechnet. Wir durchlaufen dazu die Folge vom Ende zum Anfang hin. Treffen wir auf ein Element $a_i > 0$, dann sind wir fertig. Andernfalls verschmelzen wir das Element mit den folgenden Segmenten, bis das neu entstehende Segment einen positiven Score erhält. Dies ist

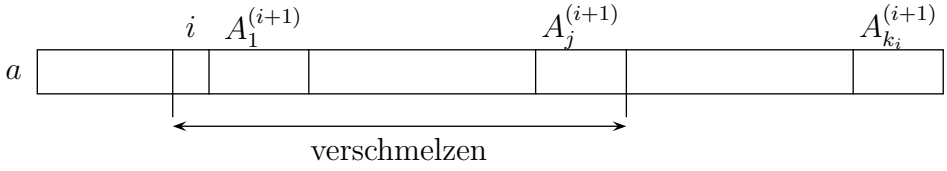


Abbildung 1.33: Skizze: Verschmelzen eines Elements mit Segmenten

in der folgenden Abbildung 1.33 illustriert, wobei nach dem Verschmelzen mit $A_j^{(i)}$ das Segment ab Position i einen positiven Score erhält.

Wir können diese Idee in den in Abbildung 1.34 angegebenen Algorithmus umsetzen, wobei $s(i) := \sigma(i, p(i))$ bezeichnet.

compute_left_negative_pointer (real $a[]$, int n)

```
begin
    int i, p[n], s[n];
    for (i := n; i > 0; i--) do
        p(i) := i;
        s(i) := a_i;
        while ((p(i) < n) && (s(i) ≤ 0)) do
            s(i) := s(i) + s(p(i) + 1);
            p(i) := p(p(i) + 1);
end
```

Abbildung 1.34: Algorithmus: Berechnung linksnegativer Zeiger durch Verschmelzen von Segmenten

Eine simple Laufzeit-Abschätzung ergibt, das die Laufzeit im schlimmsten Fall $O(n^2)$ beträgt. Mit einer geschickteren Analyse können wir jedoch wiederum eine lineare Laufzeit zeigen. Es wird pro Iteration maximal ein neues Segment generiert. In jeder inneren 'while-Schleife' wird ein Segment eliminiert. Es müssen mindestens so viele Segmente generiert wie gelöscht (= verschmolzen) werden. Da nur n Segmente generiert werden, können auch nur n Segmente gelöscht werden und somit ist die Laufzeit $O(n)$. Halten wir das Resultat im folgenden Satz fest.

Theorem 1.24 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ eine Folge reeller Zahlen und $\lambda \in \mathbb{N}$. Dann lässt sich eine Maximal Scoring Subsequence der Länge höchstens λ in Zeit $O(n)$ finden.*

Es gilt sogar der folgende Satz.

Theorem 1.25 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ eine Folge reeller Zahlen und $\underline{\lambda} \leq \overline{\lambda} \in \mathbb{N}$. Dann lässt sich eine Maximal Scoring Subsequence der Länge mindestens $\underline{\lambda}$ und höchstens $\overline{\lambda}$ in Zeit $O(n)$ finden.*

Beweis: Der Beweis sei dem Leser zur Übung überlassen. ■

1.5 Maximal Average Scoring Subsequence (*)

Jetzt wollen wir eine Teilfolge finden, deren Mittelwert (gemittelt über die Länge) maximal ist.

1.5.1 Problemstellung

Bevor wir zur Formalisierung des Problems kommen, führen wir zunächst noch einige abkürzende Notationen ein, die uns bei der Formulierung und dann auch bei der Lösung helfen werden.

Notation 1.26 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$, dann ist*

$$\sigma(i,j) := \sum_{k=i}^{j} a_k, \qquad \ell(i,j) := j - i + 1, \qquad \mu(i,j) := \frac{\sigma(i,j)}{\ell(i,j)}.$$

Damit lässt sich nun das Problem formalisieren.

MAXIMAL AVERAGE SCORING SUBSEQUENCE (MASS)

Eingabe: Eine Folge $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$, $\lambda \in \mathbb{N}$.
Gesucht: Eine (zusammenhängende) Teilfolge $(a_i, \ldots, a_j)$ mit $\ell(i,j) \geq \lambda$, die $\mu(i,j)$ maximiert, d.h. $\mu(i,j) = \max\{\mu(i',j') \mid j' \geq i' + \lambda - 1\}$.

Wozu haben wir hier noch die untere Schranke λ eingeführt? Ansonsten wird das Problem trivial, denn dann ist das maximale Element, interpretiert als eine einelementige Folge, die gesuchte Lösung!

1.5.2 Rechtsschiefe Folgen und fallend rechtsschiefe Partitionen

Zur Lösung des Problems benötigen wir den Begriff einer rechtsschiefen Folge sowie einer fallend rechtsschiefen Partition.

Definition 1.27 *Eine Folge $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ heißt* rechtsschief*, wenn für alle $i \in [1 : n-1]$ gilt: $\mu(1,i) \leq \mu(i+1,n)$.*

Eine Partition $a = A_1 \cdots A_k$ von a heißt fallend rechtsschief*, wenn jedes Segment A_i rechtsschief ist und $\mu(A_i) > \mu(A_j)$ für $i < j$ gilt.*

Anschaulich heißt eine Folge $a = (a_1, \ldots, a_n)$ also rechtsschief, wenn der Durchschnittswert jedes echten Präfix $(a_1, \ldots, a_i)$ kleiner gleich dem Durchschnittswert des korrespondierenden Suffixes $(a_{i+1}, \ldots, a_n)$ ist.

Lemma 1.28 *Seien $a \in \mathbb{R}^n$ und $b \in \mathbb{R}^m$ zwei reelle Folgen mit $\mu(a) < \mu(b)$. Dann gilt $\mu(a) < \mu(ab) < \mu(b)$ (ab sei die konkatenierte Folge).*

Beweis: Es gilt:

$$\begin{aligned}
\mu(ab) &= \frac{\sigma(ab)}{\ell(ab)} \\
&= \frac{\sigma(a) + \sigma(b)}{\ell(ab)} \\
&= \frac{\mu(a) \cdot \ell(a) + \mu(b) \cdot \ell(b)}{\ell(ab)}.
\end{aligned}$$

Somit ist zum einen (da $\mu(b) > \mu(a)$ ist)

$$\begin{aligned}
\mu(ab) &= \frac{\mu(a) \cdot \ell(a) + \mu(b) \cdot \ell(b)}{\ell(ab)} \\
&> \frac{\mu(a) \cdot \ell(a) + \mu(a) \cdot \ell(b)}{\ell(ab)} \\
&= \frac{\mu(a) \cdot (\ell(a) + \ell(b))}{\ell(ab)} \\
&= \frac{\mu(a) \cdot \ell(ab)}{\ell(ab)} \\
&= \mu(a)
\end{aligned}$$

und zum anderen (da $\mu(a) < \mu(b)$ ist)

$$\begin{aligned}
\mu(ab) &= \frac{\mu(a) \cdot \ell(a) + \mu(b) \cdot \ell(b)}{\ell(ab)} \\
&< \frac{\mu(b) \cdot \ell(a) + \mu(b) \cdot \ell(b)}{\ell(ab)} \\
&= \frac{\mu(b) \cdot (\ell(a) + \ell(b))}{\ell(ab)} \\
&= \mu(b).
\end{aligned}$$

Damit ist die Behauptung gezeigt. ■

Korollar 1.29 *Seien $a \in \mathbb{R}^n$ und $b \in \mathbb{R}^m$ zwei reelle Folgen mit $\mu(a) \leq \mu(b)$. Dann gilt $\mu(a) \leq \mu(ab) \leq \mu(b)$.*

Beweis: Gilt $\mu(a) < \mu(b)$, so ist es gerade die stärkere Aussage des vorherigen Lemmas. Gilt $\mu(a) = \mu(b)$, so gilt trivialerweise $\mu(a) = \mu(ab) = \mu(b)$. ■

Aus dem Beweis des vorherigen Lemmas folgt sofort auch das folgende Korollar.

Korollar 1.30 *Seien $a \in \mathbb{R}^n$ und $b \in \mathbb{R}^m$ zwei reelle Folgen mit $\mu(a) > \mu(b)$. Dann gilt $\mu(a) > \mu(ab) > \mu(b)$.*

Lemma 1.31 *Seien $a \in \mathbb{R}^n$ und $b \in \mathbb{R}^m$ zwei rechtsschiefe Folgen mit $\mu(a) \leq \mu(b)$. Dann ist auch die Folge ab rechtsschief.*

Beweis: Sei p ein beliebiges Präfix von ab. Es ist zu zeigen, dass $\mu(p) \leq \mu(q)$ ist, wobei q das zu p korrespondierende Suffix in ab ist, d.h. $ab = pq$.

Fall 1: Sei $p = a$. Dann gilt $\mu(a) \leq \mu(b)$ nach Voraussetzung, und die Behauptung ist erfüllt, da $q = b$ ist.

Fall 2: Sei jetzt p ein echtes Präfix von a, d.h. $a = pa'$. Da a eine rechtsschiefe Folge ist, gilt $\mu(p) \leq \mu(a')$. Mit dem vorherigen Korollar gilt dann:

$$\mu(p) \leq \mu(pa') \leq \mu(a').$$

Somit ist $\mu(p) \leq \mu(a) \leq \mu(b)$.

Mit $\mu(p) \leq \mu(a')$ und $\mu(p) \leq \mu(b)$, folgt

$$\begin{aligned}
\mu(p) &= \frac{\ell(a')\mu(p) + \ell(b)\mu(p)}{\ell(a'b)} \\
&\leq \frac{\ell(a')\mu(a') + \ell(b)\mu(b)}{\ell(a'b)} \\
&= \frac{\sigma(a') + \sigma(b)}{\ell(a'b)} \\
&= \mu(a'b) \\
&= \mu(q).
\end{aligned}$$

Fall 3: Sei $p = ab'$ mit $b = b'b''$. Mit dem vorherigen Korollar folgt, da $b = b'b''$ rechtsschief ist:

$$\mu(b') \leq \mu(\underbrace{b'b''}_{=b}) \leq \mu(b'').$$

Somit gilt $\mu(a) \leq \mu(b) \leq \mu(b'')$. Also gilt mit $\mu(a) \leq \mu(b'')$ und $\mu(b') \leq \mu(b'')$:

$$\begin{aligned}
\mu(p) &= \mu(ab') \\
&= \frac{\sigma(a) + \sigma(b')}{\ell(ab')} \\
&= \frac{\ell(a)\mu(a) + \ell(b')\mu(b')}{\ell(ab')} \\
&\leq \frac{\ell(a)\mu(b'') + \ell(b')\mu(b'')}{\ell(ab')} \\
&= \mu(b'') \\
&= \mu(q).
\end{aligned}$$

Damit ist das Lemma bewiesen. ■

Lemma 1.32 *Jede Folge $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ besitzt eine eindeutige fallend rechtsschiefe Partition.*

Beweis: Wir führen den Beweis durch Induktion über n.

Induktionsanfang ($n = 1$): Klar, nach Definition.

Induktionsschritt ($n \to n + 1$): Wir betrachten eine Folge $(a_1, \ldots, a_n, a_{n+1})$. Sei weiter $A_1 \cdots A_k$ die fallend rechtsschiefe Partition von $(a_1, \ldots, a_n)$.

Gilt $\mu(A_k) > a_{n+1} = \mu(a_{n+1})$, dann ist $A_1 \cdots A_k \cdot (a_{n+1})$ eine neue rechtsschiefe Partition.

Andernfalls bestimmen wir ein maximales i mit $\mu(A_{i-1}) > \mu(A_i \cdots A_k \cdot (a_{n+1}))$. Dann behaupten wir, dass die neue Partition $A_1 \cdots A_{i-1} \cdot (A_i \cdots A_k \cdot (a_{n+1}))$ eine fallend rechtsschiefe ist.

Nach Definition ist (a_{n+1}) rechtsschief. Nach dem vorherigen Lemma 1.31 und der Wahl von i ist dann auch $A_k \cdot (a_{n+1})$ rechtsschief. Weiter gilt allgemein für $j \geq i$, dass $A_j \cdots A_k \cdot (a_{n+1})$ rechtsschief ist. Somit ist auch $A_i \cdots A_k \cdot (a_{n+1})$ rechtsschief.

Nach Konstruktion ist also die jeweils konstruierte Partition eine fallend rechtsschiefe Partition.

Es bleibt noch die Eindeutigkeit zu zeigen. Nehmen wir an, es gäbe zwei verschiedene fallend rechtsschiefe Partitionen. Betrachten wir, wie in der folgenden Abbildung 1.35 skizziert, die jeweils linkesten Teilfolgen in ihren Partition, in denen sich die beiden Partitionen unterscheiden.

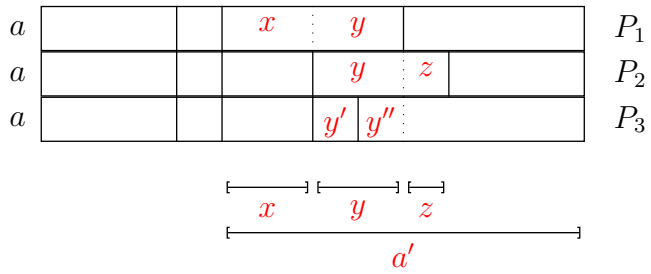


Abbildung 1.35: Skizze: Beweis der Eindeutigkeit der rechtsschiefen Partition

Betrachten wir zuerst den Fall P_1 gegen P_2. Sei zuerst $z = \varepsilon$. Dann gilt $\mu(x) > \mu(y)$, da P_2 eine fallend rechtsschiefe Partition ist. Da xy in P_1 rechtsschief ist, gilt $\mu(x) \leq \mu(y)$ und wir erhalten den gewünschten Widerspruch.

Sei nun $z \neq \varepsilon$. Da yz nach der Partition P_2 rechtsschief ist, gilt $\mu(y) \leq \mu(z)$. Mit Korollar 1.29 folgt, dass $\mu(y) \leq \mu(yz) \leq \mu(z)$.

Nach Wahl der fallend rechtsschiefen Partition P_2 gilt $\mu(x) > \mu(yz)$. Wie wir eben gezeigt haben, gilt auch $\mu(yz) \geq \mu(y)$. Damit ist $\mu(x) > \mu(y)$ und somit ist xy nicht rechtsschief. Dies ist ein Widerspruch zur Annahme, dass P_1 eine fallend rechtsschiefe Partition ist.

Es bleibt noch der Fall P_1 gegen P_3 zu betrachten. Nach P_1 gilt $\mu(x) \leq \mu(y)$ und nach P_3 gilt $\mu(x) > \mu(y')$. Also gilt $\mu(y) > \mu(y')$. Sei y'' so gewählt, dass $y = y'y''$ und sei weiter $y = y' \cdot Y_2 \cdots Y_k$ eine fallende rechtsschiefe Partition von y, die sich aus P_3 bzw. dem ersten Teil des Beweises ergibt. Dann gilt $\mu(y') > \mu(Y_2) > \cdots > \mu(Y_k)$. Dann muss aber $\mu(y') > \mu(y'')$ sein. Somit gilt $\mu(y) > \mu(y') > \mu(y'')$, was ein offensichtlicher Widerspruch ist. ■

1.5.3 Algorithmus zur Konstruktion rechtsschiefer Zeiger

Kommen wir nun zu einem Algorithmus, der die fallend rechtsschiefe Partition zu einer gegebenen Folge konstruiert. Zuerst benötigen wir noch einige Notationen.

Notation 1.33 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Jedes Suffix $a^{(i)} = (a_i, \ldots, a_n)$ von a besitzt eine eindeutige fallend rechtsschiefe Partition: $a^{(i)} = A_1^{(i)} \cdots A_{k_i}^{(i)}$.*

Damit können wir die so genannten rechtsschiefen Zeiger definieren.

Definition 1.34 *Sei* $A_1^{(i)} = (a_i, \ldots, a_{p(i)})$, *dann heißt* $p(i)$ *der* rechtsschiefe Zeiger *von* i.

Für den folgenden Algorithmus benötigen wir noch die folgenden vereinfachenden Notationen.

Notation 1.35 *Im Folgenden verwenden wir der Einfachheit halber die beiden folgenden Abkürzungen:*

$$\begin{aligned} s(i) &:= \sigma(i, p(i)), \\ \ell(i) &:= \ell(i, p(i)) = p(i) - i + 1. \end{aligned}$$

Für die Konstruktion der rechtsschiefen Zeiger arbeiten wir uns wieder vom Ende zum Anfang durch die gegebene Folge durch. Für ein neu betrachtetes Element setzen wir zunächst die rechtsschiefe Folge auf diese einelementige Folge. Ist nun der

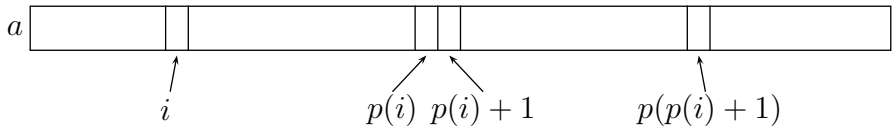


Abbildung 1.36: Skizze: Verschmelzen von Segmenten

Mittelwert des aktuell betrachten Segments kleiner gleich dem Mittelwert des folgenden Segments, so verschmelzen wir diese beiden und betrachten dieses neue Segment als das aktuelle. Andernfalls haben wir die Eigenschaft einer fallend rechtsschiefen Partition sichergestellt. Dies ist in der Abbildung 1.36 schematisch dargestellt.

Da diese eindeutig ist, erhalten wir zum Schluss die gewünschte fallend rechtsschiefe Partition samt aller rechtsschiefer Zeiger. Somit können wir den in Abbildung 1.37 angegebenen Algorithmus zur Konstruktion rechtsschiefer Zeiger formalisieren.

Lemma 1.36 *Die rechtsschiefen Zeiger lassen sich in Zeit* $O(n)$ *berechnen.*

Beweis: Analog zum Beweis der Laufzeit des Algorithmus zur Konstruktion linksnegativer Zeiger. ■

```
compute_rightskew_pointer (real a[], int n)
begin
    for (i := n; i > 0; i--) do
        p(i) := i;
        s(i) := a_i;
        ℓ(i) := 1;
        while ((p(i) < n) && (s(i)/ℓ(i) ≤ s(p(i)+1)/ℓ(p(i)+1))) do
            s(i) := s(i) + s(p(i)+1);
            ℓ(i) := ℓ(i) + ℓ(p(i)+1);
            p(i) := p(p(i)+1);
end
```

Abbildung 1.37: Algorithmus: Rechtsschiefe Zeiger

1.5.4 Elementare Eigenschaften von MASS

Bevor wir zur Bestimmung von Teilfolgen vorgegebener Mindestlänge mit einem maximalem Mittelwert kommen, werden wir erst noch ein paar fundamentale Eigenschaften festhalten.

Lemma 1.37 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ und $(a_i, \ldots, a_j)$ eine kürzeste Teilfolge von a der Länge mindestens λ, die deren Average Score $\mu(i,j)$ maximiert. Dann gilt $\ell(i,j) = j - i + 1 \leq 2\lambda - 1$.*

Beweis: Der Beweis sei dem Leser zur Übung überlassen. ■

Lemma 1.38 *Seien $a \in \mathbb{R}^n$, $b \in \mathbb{R}^m$ und $c \in \mathbb{R}^k$ drei reelle Folgen, die die Beziehung $\mu(a) < \mu(b) < \mu(c)$ erfüllen. Dann gilt $\mu(ab) < \mu(abc)$.*

Zunächst geben wir ein Gegenbeispiel an, um zu zeigen, dass im vorherigen Lemma die Bedingung $\mu(a) < \mu(b)$ notwendig ist. Sei $a = 11$, $b = 1$ und $c = 3$, dann ist $\mu(ab) = \frac{12}{2} = 6$ sowie $\mu(abc) = \frac{15}{3} = 5$.

Beweis: Nach Korollar 1.30 gilt $\mu(a) < \mu(ab) < \mu(b) < \mu(c)$. Somit gilt insbesondere $\mu(ab) < \mu(c)$. Nach Korollar 1.30 gilt dann $\mu(ab) < \mu(abc) < \mu(c)$ und das Lemma ist bewiesen. ■

Lemma 1.39 *Seien $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ und $p \in \mathbb{R}^\ell$ und sei $A_1 \cdots A_k$ die fallend rechtsschiefe Partition von a. Sei weiter m maximal gewählt, so dass*

$$\mu(p \cdot A_1 \cdots A_m) = \max\{\mu(p \cdot A_1 \cdots A_i) \mid i \in [0:k]\}.$$

Es gilt genau dann $\mu(p \cdot A_1 \cdots A_i) > \mu(A_{i+1})$, wenn $i \geq m$ gilt.

Beweis: $\Rightarrow$: Sei i so gewählt, dass $\mu(p \cdot A_1 \cdots A_i) > \mu(A_{i+1})$ gilt. Da $A_1 \cdots A_k$ die fallend rechtsschiefe Partition von a ist, gilt $\mu(A_1) > \mu(A_2) > \cdots > \mu(A_k)$. Dann gilt auch $\mu(p \cdot A_1 \cdots A_i) > \mu(A_{i+1}) > \mu(A_{i+2}) > \cdots > \mu(A_k)$. Nach Korollar 1.30 gilt dann $\mu(p \cdot A_1 \cdots A_i) > \mu(p \cdot A_1 \cdots A_{i+1}) > \mu(A_{i+1}) > \cdots > \mu(A_k)$. Mit Korollar 1.30 gilt also auch $\mu(p \cdot A_1 \cdots A_i) > \mu(p \cdot A_1 \cdots A_j) > \mu(A_j) > \cdots > \mu(A_k)$ für alle $j \in [i:k]$. Aufgrund der Definition von m muss also $i \geq m$ gelten.

$\Leftarrow$: Zum Beweis der Behauptung unterscheiden wir die folgenden beiden Fälle, je nachdem, ob $\mu(p \cdot A_1 \cdots A_m) > \mu(A_{m+1})$ oder ob $\mu(p \cdot A_1 \cdots A_m) \leq \mu(A_{m+1})$ gilt.

Fall 1: Es gilt $\mu(p \cdot A_1 \cdots A_m) > \mu(A_{m+1}) > \mu(A_{m+2}) > \cdots > \mu(A_k)$. Korollar 1.30 liefert $\mu(p \cdot A_1 \cdots A_{m+1}) > \mu(A_{m+1}) > \mu(A_{m+2}) > \cdots > \mu(A_k)$. Nach wiederholter Anwendung von Korollar 1.30 gilt dann für $j \in [1 : k-m]$ natürlich auch $\mu(p \cdot A_1 \cdots A_{m+j}) > \mu(A_{m+j}) > \mu(A_{m+j+1}) > \cdots > \mu(A_k)$. Also gilt für $i \geq m$ $\mu(p \cdot A_1 \cdots A_i) > \mu(A_i) > \mu(A_{i+1}) > \cdots > \mu(A_k)$ und somit die Behauptung.

Fall 2: Es gilt $\mu(p \cdot A_1 \cdots A_m) \leq \mu(A_{m+1})$. Nach Korollar 1.29 gilt nun allerdings $\mu(p \cdot A_1 \cdots A_m) \leq \mu(p \cdot A_1 \cdots A_{m+1}) \leq \mu(A_{m+1})$. Aufgrund der Wahl von m (das Maximum von $\mu(p \cdot A_1 \cdots A_i)$ wird für $i = m$ angenommen) gilt dann $\mu(p \cdot A_1 \cdots A_m) = \mu(p \cdot A_1 \cdots A_{m+1})$. Daraus ergibt sich ein Widerspruch zur Maximalität von m. Damit ist die Behauptung des Lemmas gezeigt. ■

1.5.5 Ein Algorithmus für MASS

Somit können wir einen Algorithmus zur Lösung unseres Problems angeben, der im Wesentlichen auf Lemma 1.39 basiert. Wir laufen von links nach rechts durch die Folge, und versuchen, ab Position i die optimale Sequenz mit maximalem Mittelwert zu finden. Nach Voraussetzung hat diese mindestens die Länge λ und nach Lemma 1.37 auch höchstens die Länge $2\lambda - 1$. Dies ist in Abbildung 1.38 illustriert.

Ein einfacher Algorithmus würde jetzt alle möglichen Längen aus dem Intervall $[\lambda : 2\lambda - 1]$ ausprobieren. Der folgende, in Abbildung 1.39 angegebene Algorithmus würde dies tun, wenn die Prozedur `locate` geeignet implementiert wäre, wobei `locate` einfach die Länge einer optimalen Teilfolge zurückgibt. Würden wir `locate`

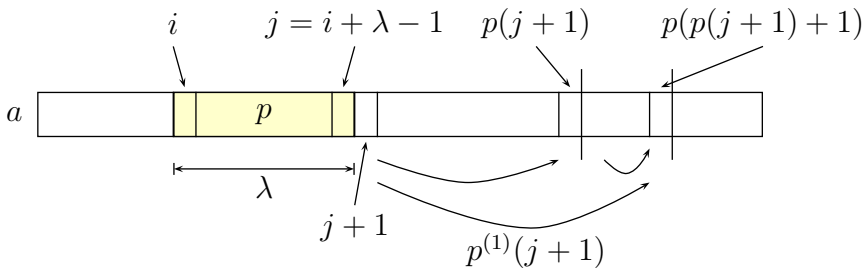


Abbildung 1.38: Skizze: Auffinden der optimalen Teilfolge

mittels einer linearen Suche implementieren, so würden wir einen Laufzeit von $O(n\lambda) = O(n^2)$ erhalten.

Mit Hilfe von Lemma 1.39 können wir aber auch geschickter vorgehen. Das Lemma besagt nämlich, dass wenn wir rechts vom optimalen Schnitt sind, der Mittelwert größer als der des folgenden Segments ist. Links vom Optimum gilt, dass der Wert kleiner als der Mittelwert vom folgenden Segment ist. Somit könnten wir hier eine binäre Suche ausnutzen.

Allerdings kennen wir die Grenzen der Segmente nicht explizit, sondern nur als lineare Liste über die rechtsschiefen Zeiger. Daher konstruieren wir uns zu den rechtsschiefen Zeiger noch solche, die nicht nur auf das nächste Segment, sondern auch auf das 2^k-te folgende Segment angibt. Dafür definieren wir erst einmal formal die Anfänge der nächsten 2^k-ten Segment sowie die darauf basierenden iterierten

MASS (real $a[]$, int n, λ)

```
begin
    int mi := 1;
    int mj := λ;
    int mm =: μ(mi, mj);
    for (i := 1; i ≤ n; i++) do
        j := i + λ − 1;
        if (μ(i, j) ≤ μ(j + 1, p(j + 1))) then
            j := locate(i, j);
        if (μ(i, j) > mm) then
            mi := i;
            mj := j;
            mm := μ(mi, mj);
end
```

Abbildung 1.39: Algorithmus: MASS

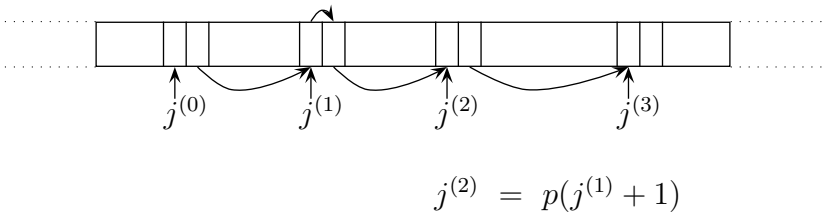


Abbildung 1.40: Skizze: Definition der Werte $j^{(\cdot)}$

rechtsschiefen Zeiger wie folgt:

$$\begin{aligned} j^{(0)} &:= j, \\ j^{(k)} &= \min\{p(j^{(k-1)} + 1), n\}, \\ p^{(0)}(i) &= p(i), \\ p^{(k)}(i) &= \min\{p^{(k-1)}(p^{(k-1)}(i) + 1), n\}. \end{aligned}$$

Hierbei gibt $p^{(k)}(i)$ das Ende nach 2^k Segmenten an. Dies ist in folgenden Abbildungen illustriert, in Abbildung 1.40 die Definition von $j^{(\cdot)}$ und in Abbildung 1.41 die Definition von $p^{(\cdot)}$.

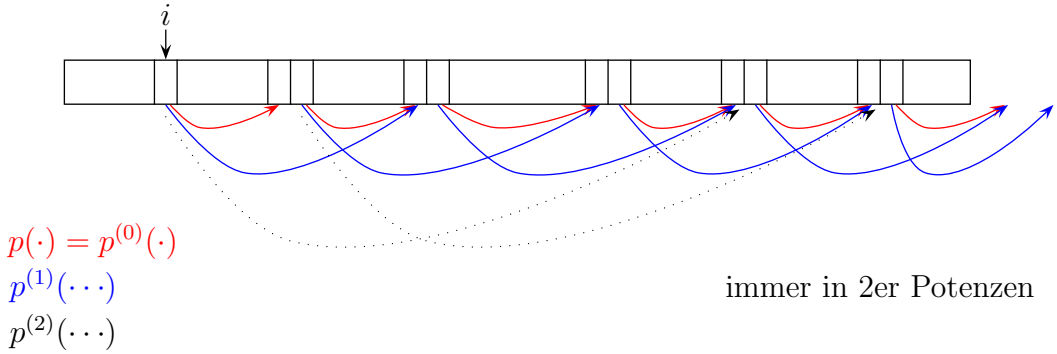


Abbildung 1.41: Skizze: Definition der Werte $p^{(\cdot)}$

Definition 1.40 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ eine reelle Folge und $p(\cdot)$ die zugehörigen rechtsschiefen Zeiger. Der* k-te iterierte rechtsschiefe Zeiger *$p^{(k)}(i)$ ist rekursiv definiert durch $p^{(k)}(i) := \min\{p^{(k-1)}(p^{(k-1)}(i) + 1), n\}$ und $p^{(0)}(i) := p(i)$.*

Diese iterierten rechtsschiefen Zeiger lassen sich aus den rechtsschiefen Zeigern in Zeit $O(\log(\lambda))$ berechnen, da wir ja maximal die 2λ-iterierten rechtsschiefen Zeiger aufgrund der oberen Längenbeschränkung von $2\lambda - 1$ benötigen.

Mit Hilfe dieser iterierten rechtsschiefen Zeiger können wir jetzt die binäre Suche in der Prozedur `locate` wie im folgenden Algorithmus implementieren, der in Abbildung 1.42 angegeben ist. Hierbei ist zu beachten, dass die erste zu testende Position bei der binären Suche bei maximal $\log(2\lambda - 1) - 1 \leq \log(\lambda)$ liegt.

locate (int i, j)

```
begin
    for (k := log(λ); k ≥ 0; k--) do
        if ((j ≥ n) || (μ(i, j) > μ(j + 1, p(j + 1)))) then
            return j;
        if ((p^(k)(j + 1) < n) &&
        (μ(i, p^(k)(j + 1)) ≤ μ(p^(k)(j + 1) + 1, p(p^(k)(j + 1) + 1)))) then
            j := p^(k)(j + 1);
    if ((j < n) && (μ(i, j) ≤ μ(j + 1, p(j + 1)))) then
        j := p(j + 1);
    return j;
end
```

Abbildung 1.42: Algorithmus: locate(i, j)

Die Strategie der Suche ist noch einmal in der folgenden Abbildung 1.43 illustriert.

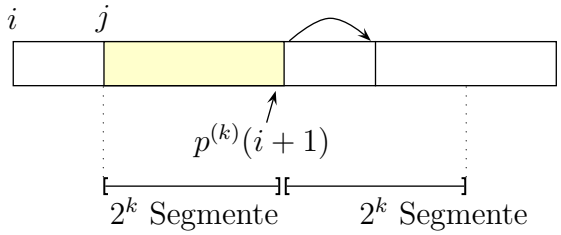


Abbildung 1.43: Skizze: Strategie von `locate`

Zusammenfassend erhalten wir das folgende Theorem.

Theorem 1.41 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$. Eine kürzeste Teilfolge der Länge mindestens λ mit maximalem durchschnittlichem Score kann in Zeit $O(n \log(\lambda))$ gefunden werden.*

Wir wollen hier noch erwähnen, dass mittlerweile für das MASS-Problem Algorithmen mit linearer Laufzeit bekannt sind.

1.6 Weighted Maximal Average Scoring Subsequence (*)

In diesem Abschnitt skizzieren wir einen linearen Algorithmus für die Bestimmung einer optimalen Teilfolge bezüglich des Mittelwerts. Darüber hinaus ist dies sogar auch für einen obere Längenbeschränkung möglich.

1.6.1 Problemstellung

Das Problem lässt sich nun wie folgt in etwas allgemeinerer Fassung formalisieren.

WEIGHTED MAXIMAL AVERAGE SCORING SUBSEQUENCE (WMASS)

Eingabe: Eine Folge $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ mit Gewichten $w = (w_1, \ldots, w_n) \in \mathbb{N}^n$ und $\lambda \in \mathbb{N}$.

Gesucht: Eine (zusammenhängende) Teilfolge $(a_i, \ldots, a_j)$ mit $\ell(i,j) \geq \lambda$, die den Wert $\mu(i,j) = \sigma(i,j)/\ell(i,j)$ maximiert, wobei $\sigma(i,j) = \sum_{k=1}^{j} a_k$ und $\ell(i,j) = \sum_{k=i}^{j} w_k$.

Setzt man $w_k = 1$ für alle $k \in [1:n]$ so erhält man die Definition aus dem vorherigen Abschnitt bzw. das entsprechende Analogon der ersten Abschnitte. Die Gewichte müssen keine natürlichen Zahlen sein, man kann auch positive reelle Zahlen verwenden. Auch die Einführung einer oberen Schranke ist möglich, aber der Kürze wegen wollen hier nicht näher darauf eingehen, sondern verweisen auf die Literatur von Chung und Lu.

Wozu haben wir hier noch die untere Schranke λ eingeführt? Ansonsten wird das Problem trivial, denn für $w_i = 1$ ist dann das maximale Element, interpretiert als eine einelementige Folge, die gesuchte Lösung! Für beliebige Gewichte ist dann das Element mit dem maximalen relativen Verhältnis a_i/w_i die Lösung!

Bevor wir zur Bestimmung von Teilfolgen vorgegebener Mindestlänge mit einem maximalem Mittelwert kommen, werden wir erst noch eine fundamentale Eigenschaft festhalten.

Lemma 1.42 *Sei $a = (a_1, \ldots, a_n) \in \mathbb{R}^n$ mit $w = (1, \ldots, 1)$ und $(a_i, \ldots, a_j)$ eine kürzeste Teilfolge von a der Länge mindestens λ, die deren Average Score $\mu(i,j)$ maximiert. Dann gilt $\ell(i,j) = j - i + 1 \leq 2\lambda - 1$.*

Beweis: Der Beweis sei dem Leser zur Übung überlassen. ■

1.6.2 Elementare Eigenschaften

Zuerst halten wir einige fundamentale Eigenschaften über den gewichteten Durchschnitt von Teilfolgen fest.

Lemma 1.43 *Sei $(a_1, \ldots, a_n) \in \mathbb{R}^n$ eine Folge mit Gewichten $(w_1, \ldots, w_n) \in \mathbb{N}^n$. Dann gilt für alle $x \leq y < z \in [1:n]$:*

i) $\mu(x,y) < \mu(y+1,z) \quad \Leftrightarrow \quad \mu(x,z) < \mu(y+1,z) \quad \Leftrightarrow \quad \mu(x,y) < \mu(x,z),$

ii) $\mu(x,y) > \mu(y+1,z) \quad \Leftrightarrow \quad \mu(x,z) > \mu(y+1,z) \quad \Leftrightarrow \quad \mu(x,y) > \mu(x,z).$

Beweis: Es gelte $\mu(x,y) < \mu(y+1,z)$, d.h. $\sigma(x,y) < \frac{\ell(x,y)}{\ell(y+1,z)}\sigma(y+1,z)$, dann gilt:

$$\begin{aligned}
\mu(x,z) &= \frac{\sigma(x,y)+\sigma(y+1,z)}{\ell(x,z)} \\
&< \frac{\ell(x,y)\sigma(y+1,z)}{\ell(x,z)\ell(y+1,z)} + \frac{\sigma(y+1,z)}{\ell(x,z)} \\
&= \frac{\sigma(y+1,z)}{\ell(x,z)}\left(\frac{\ell(x,y)}{\ell(y+1,z)}+1\right) \\
&= \frac{\sigma(y+1,z)}{\ell(x,z)}\left(\frac{\ell(x,y)+\ell(y+1,z)}{\ell(y+1,z)}\right) \\
&= \frac{\sigma(y+1,z)}{\ell(y+1,z)} \\
&= \mu(y+1,z).
\end{aligned}$$

Es gelte $\mu(x,z) < \mu(y+1,z)$, d.h. $\sigma(y+1,z) > \frac{\ell(y+1,z)}{\ell(x,z)}\sigma(x,z)$, dann gilt:

$$\begin{aligned}
\mu(x,y) &= \frac{\sigma(x,z)-\sigma(y+1,z)}{\ell(x,y)} \\
&< \frac{1}{\ell(x,y)}\left(\sigma(x,z) - \frac{\ell(y+1,z)\sigma(x,z)}{\ell(x,z)}\right) \\
&= \frac{\sigma(x,z)}{\ell(x,z)}\left(\frac{\ell(x,z)}{\ell(x,y)} - \frac{\ell(y+1,z)}{\ell(x,y)}\right) \\
&= \frac{\sigma(x,z)}{\ell(x,z)} \\
&= \mu(x,z).
\end{aligned}$$

Es gelte $\mu(x,y) < \mu(x,z)$, d.h. $\sigma(x,z) > \ell(x,z)\frac{\sigma(x,y)}{\ell(x,y)}$, dann gilt:

$$\begin{aligned}
\mu(x,y) &= \frac{\sigma(x,y)}{\ell(x,y)}\left(\frac{\ell(x,z)-\ell(x,y)}{\ell(y+1,z)}\right)\\
&= \frac{1}{\ell(y+1,x)}\left(\frac{\ell(x,z)\sigma(x,y)}{\ell(x,y)} - \frac{\ell(x,y)\sigma(x,y)}{\ell(x,y)}\right)\\
&< \frac{1}{\ell(y+1,x)}\left(\sigma(x,z)-\sigma(x,y)\right)\\
&= \frac{\sigma(y+1,z)}{\ell(y+1,x)}\\
&= \mu(y+1,z).
\end{aligned}$$

Teil ii) wird völlig analog bewiesen. ■

Das folgende Korollar ergibt sich unmittelbar aus dem vorhergenden Lemma.

Korollar 1.44 *Sei $(a_1,\ldots,a_n) \in \mathbb{R}^n$ eine Folge mit Gewichten $(w_1,\ldots,w_n) \in \mathbb{N}^n$. Dann gilt für alle $x < y < z \in [1:n]$:*

i) $\mu(x,y) \le \mu(y+1,z) \quad\Leftrightarrow\quad \mu(x,z) \le \mu(y+1,z) \quad\Leftrightarrow\quad \mu(x,y) \le \mu(x,z),$

ii) $\mu(x,y) \ge \mu(y+1,z) \quad\Leftrightarrow\quad \mu(x,z) \ge \mu(y+1,z) \quad\Leftrightarrow\quad \mu(x,y) \ge \mu(x,z).$

1.6.3 Generischer Algorithmus und seine Korrektheit

In diesem Abschnitt werden wir einen generischen Algorithmus für das Problem vorstellen und seine Korrektheit beweisen. Für die Beschreibung des Algorithmus benötigen wir noch die folgenden Notationen.

Notation 1.45 *Sei $(a_1,\ldots,a_n) \in \mathbb{R}^n$ eine Folge mit Gewichten $(w_1,\ldots,w_n) \in \mathbb{N}^n$ und $\lambda \in \mathbb{N}$, dann ist:*

- $\varphi(x,y) = \max\{z \in [x:y] \,:\, \mu(x,z) = \min\{\mu(x,z') \,:\, z' \in [x:y]\}\}$;
- $j_0 = \min\{j \in \mathbb{N} \,:\, \ell(1,j) \ge \lambda\}$;
- $r_j = \max\{i \in \mathbb{N} \,:\, \ell(i,j) \ge \lambda\}$;
- $i_j^* = \max\{k \in [1:r_j] \,:\, \mu(k,j) = \max\{\mu(i,j) \,:\, i \in [1:r_j]\}\}$;
- $j^* = \min\big\{j \in [1:n] \,:\, \mu(i_j^*,j) = \max\{\mu(i,j) \,:\, j \in [j_0:n] \wedge i \in [1:r_j]\}\big\}$.

Die Funktion $\varphi(x, y)$ liefert die Endposition eines längsten Präfixes von $(a_x, \ldots, a_y)$, die einen minimalen Average Score unter allen Präfixen besitzt. j_0 beschreibt die erste Endposition, an der eine Sequenz mit maximalen Average Score enden kann. Weiter beschreibt r_j die größte Indexposition i, die für eine Betrachtung der Teilfolge (i, j) möglich ist, andernfalls wird die untere Schranke λ verletzt. Der Wert i_j^* beschreibt die letztmögliche Startposition einer Teilfolge, die an Position j endet und den Average Score maximiert. Letztendlich ist j^* die Endposition der ersten Teilfolge, die den Average Score maximiert.

WMASS (real $a[]$, real $w[]$, int n, int λ)

```
begin
    int opt := 0;
    int L := 0;
    int R := 0;
    int i_{j_0-1} := 1;
    for (j := j_0; j ≤ n; j++) do
        i_j := Best(i_{j-1}, r_j, j);
        if (μ(i_j, j) > opt) then
            opt =: μ(i_j, j);
            L := i_j;
            R := j;
end

Best(ℓ, r, j)
begin
    int i := ℓ;
    while ((i < r) && (μ(i, φ(i, r − 1)) ≤ μ(i, j))) do
        i := φ(i, r − 1) + 1;
    return i;
end
```

Abbildung 1.44: Algorithmus: Generischer Algorithmus für WMASS

In dem in Abbildung 1.44 angegebenen Algorithmus bestimmen wir für jede mögliche Endposition j den zugehörigen Startwert i_j, der den Average Score maximiert. Dies wird mithilfe der Prozedur Best erreicht. Im Folgenden versuchen wir zuerst die Korrektheit zu beweisen. Im nächsten Abschnitt werden wir dann eine effiziente Implementierung vorstellen (hier ist ja die Realisierung der Funktion φ noch völlig offen). Wir beweisen zunächst eine wichtige Eigenschaft der Prozedur Best.

Lemma 1.46 *Für $\ell \leq r \leq j$ liefert* Best(ℓ, r, j) *den größten Index $i \in [\ell : r]$, der $\mu(i, j)$ maximiert.*

Beweis: Sei $i^* = \operatorname{argmax}\{\mu(i,j) : i \in [\ell : r]\}$. Ist das Maximum nicht eindeutig, so wählen wir den größten Index. Weiter sei $i_j := \textsc{Best}(\ell, r, j)$.

Wir führen den Beweis durch Widerspruch, also gilt $i_j \neq i^*$. O.B.d.A nehmen wir an, dass j der kleinste Index ist, für den Best nicht korrekt ist.

Fall 1 ($i_j < i^*$): Somit gilt auch $i_j < r$, da $i^* \leq r$ gelten muss. Aufgrund der while-Schleife im Algorithmus 1.44 gilt, dass $\mu(i_j, \varphi(i_j, r-1)) > \mu(i_j, j)$.

Da nach Definition von i^* gilt, dass $\mu(i_j, j) \leq \mu(i^*, j)$, folgt mit Korollar 1.44: $\mu(i_j, i^* - 1) \leq \mu(i_j, j)$. Somit gilt insgesamt:

$$\mu(i_j, i^* - 1) \leq \mu(i_j, j) < \mu(i_j, \varphi(i_j, r-1)).$$

Dies ist offensichtlich ein Widerspruch zur Definition von φ.

Fall 2 ($i_j > i^*$): Somit gilt $\ell < r$, da sonst $\ell = r$ und damit $i^* = r = i_j$.

Da $\ell < r$ und $i_j > i^* \geq \ell$ gilt, wird die while-Schleife im Algorithmus 1.44 mindestens einmal durchlaufen (da er ja $i_j > i^* \geq \ell$ zurückliefert). Daher gilt:

$$\exists i \in [\ell : r] : i < r \wedge \mu(i, \varphi(i, r-1)) \leq \mu(i, j) \wedge i \leq i^* < \varphi(i, r-1) + 1.$$

Wenn $i = i^*$, folgt mit Korollar 1.44:

$$\mu(i^*, \varphi(i^*, r-1)) \leq \mu(i^*, j) \leq \mu(\varphi(i^*, r-1) + 1, j).$$

Dies ist aber ein Widerspruch zur Definition von i^*, da $\varphi(i^*, r-1) + 1 > i^*$.

Also ist im Folgenden $i < i^*$ und es gilt (aufgrund der Definition von i^*):

$$\begin{aligned}
\mu(i^*, j) &\geq \mu(i, j) \\
&\quad \text{Korollar 1.44 auf letzte Ungleichung} \\
&\geq \mu(i, i^* - 1) \\
&\quad \text{da } i^* - 1 \in [i : r-1] \text{ und Defintion von } \varphi \\
&\geq \mu(i, \varphi(i, r-1)) \\
&\quad \text{da } i^* - 1 < \varphi(i, r-1) \\
&\quad \text{liefert das Korollar 1.44 auf die letzte Ungleichung} \\
&\geq \mu(i^*, \varphi(i, r-1))
\end{aligned}$$

Somit gilt also $\mu(i^*, j) \geq \mu(i^*, \varphi(i, r-1))$. Korollar 1.44 angewendet auf diese Ungleichung liefert

$$\mu(\varphi(i, r-1) + 1, j) \geq \mu(i^*, j).$$

Da $i^* < \varphi(i, r-1) + 1$ gilt, ist das ein Widerspruch zur Definition von i^*. ∎

Basierend auf dem letzten Lemma zeigen wir jetzt die Korrektheit des generischen Algorithmus.

Theorem 1.47 *Algorithmus* WMASS *ist korrekt.*

Beweis: Da der Algorithmus WMASS alle möglichen $j \in [j_0 : n]$ durchprobiert und für $j = j^*$ als Ergebnis i_{j^*} liefert, genügt es zu zeigen, dass $i_{j^*} = i^*_{j^*}$.

Fall 1 ($j^* = j_0$): Da $i_{j_0-1} = 1$, gilt aufgrund des Algorithmus $i_{j_0} = \textsc{Best}(1, r_j, j_0)$. Da $j_0 = j^* \in [1 : r_j]$ sein muss, folgt aus Lemma 1.46 die Korrektheit.

Fall 2 ($j^* > j_0$): Irgendwann erfolgt im Laufe des Algorithmus WMASS der Aufruf $\textsc{Best}(i_{j*-1}, r_{j^*}, j^*)$. Wegen Lemma 1.46 genügt es also zu zeigen, dass $i^*_{j^*-1} \leq i^*_{j^*}$.

Dies zeigen wir durch einen Widerspruchsbeweis. Sei im Folgenden für $j \in [j_0 : n]$ wieder $i_j := \textsc{Best}(i_{j-1}, r_j, j)$. Wir nehmen also an, dass ein $j \in [j_0 : j^* - 1]$ existiert mit $i_{j-1} \leq i^*_{j^*} < i_j$.

Zunächst gilt aufgrund der Definition von j^* und $i^*_{j^*} < i_j$, dass $\mu(i^*_{j^*}, j^*) > \mu(i_j, j^*)$. Mit dem Lemma 1.43 folgt dann

$$\mu(i^*_{j^*}, i_j - 1) > \mu(i^*_{j^*}, j^*). \tag{1.1}$$

Mit $i_{j-1} \leq i^*_{j^*} < i_j \leq r_j$ und dem Lemma 1.46 folgt:

$$\begin{aligned} \mu(i_j, j) &\geq \mu(i^*_{j^*}, j) \\ &\quad \text{mit Korollar 1.44 auf die letzte Ungleichung} \\ &\geq \mu(i^*_{j^*}, i_j - 1) \\ &\quad \text{wegen der Ungleichung (1.1)} \\ &> \mu(i^*_{j^*}, j^*). \end{aligned}$$

Das ist aber ein Widerspruch zur Definition von j^*. ∎

Wir wollen noch kurz die Laufzeit des generischen Algorithmus analysieren, vorausgesetzt, dass die Funktion φ in konstanter Zeit bestimmt werden kann. In der Schleife wird für j das Intervall $[i_{j-1} : i_j]$ durchlaufen. Für $j + 1$ anschließend das Intervall $[i_j : i_{j+1}]$. Somit ist die Laufzeit aller Best-Aufrufe im worst-case proportional zur Anzahl der Elemente in $\bigcup_{j=j_0}^{n} [i_{j-1} : i_j] = [1 : n]$. Da aber nur die Elemente i_j dabei mehrfach vorkommen können und beim mehrmaligen Vorkommen jeweils j inkrementiert wird, ist die Laufzeit im worst-case linear und wir haben das folgende Lemma bewiesen.

Lemma 1.48 WMASS *kann in linearer Zeit gelöst werden, vorausgesetzt, die Funktion φ kann in konstanter Zeit berechnet werden.*

1.6.4 Linksschiefe Folgen und steigend linksschiefe Partitionen

Zur Lösung des Problems müssen wir nur noch zeigen, wie sich die Funktion φ effizient berechnen lässt. Hierzu benötigen wir den Begriff einer linksschiefen Folge sowie einer steigend linksschiefen Partition.

Definition 1.49 *Eine Folge $(a_1, \ldots, a_n) \in \mathbb{R}^n$ mit Gewichten $(w_1, \ldots, w_n) \in \mathbb{N}^n$ heißt* linksschief*, wenn für alle $i \in [1 : n-1]$ gilt: $\mu(1, i) \geq \mu(i+1, n)$.*

Eine Partition $a = A_1 \cdots A_k$ von $a = (a_1, \ldots, a_n)$ heißt steigend linksschief*, wenn jedes Segment A_i linksschief ist und $\mu(A_i) < \mu(A_j)$ für alle $i < j$ gilt.*

Anschaulich heißt eine Folge $a = (a_1, \ldots, a_n)$ also linksschief, wenn der Durchschnittswert jedes echten Präfix $(a_1, \ldots, a_i)$ größer gleich dem Durchschnittswert des korrespondierenden Suffixes $(a_{i+1}, \ldots, a_n)$ ist.

Lemma 1.50 *Sei $a \in \mathbb{R}^n$ bzw. $b \in \mathbb{R}^m$ mit Gewicht $w \in \mathbb{N}^n$ bzw. $w' \in \mathbb{N}^m$ jeweils eine linksschiefe Folge mit $\mu(a) \geq \mu(b)$. Dann ist auch die Folge ab mit Gewicht ww' linksschief.*

Beweis: Sei p ein beliebiges Präfix von ab. Es ist zu zeigen, dass $\mu(p) \geq \mu(q)$ ist, wobei q das zu p korrespondierende Suffix in ab ist, d.h. $ab = pq$.

Fall 1: Sei $p = a$. Dann gilt $\mu(a) \geq \mu(b)$ nach Voraussetzung, und die Behauptung ist erfüllt, da $q = b$ ist.

Fall 2: Sei jetzt p ein echtes Präfix von a, d.h. $a = pa'$. Da a eine linksschiefe Folge ist, gilt $\mu(p) \geq \mu(a')$. Mit dem Korollar 1.44 gilt dann:

$$\mu(p) \geq \mu(pa') \geq \mu(a').$$

Somit ist $\mu(p) \geq \mu(a) \geq \mu(b)$.

Mit $\mu(p) \geq \mu(a')$ und $\mu(p) \geq \mu(b)$, folgt

$$\begin{aligned} \mu(p) &= \frac{\ell(a')\mu(p) + \ell(b)\mu(p)}{\ell(a'b)} \\ &\geq \frac{\ell(a')\mu(a') + \ell(b)\mu(b)}{\ell(a'b)} \\ &= \frac{\sigma(a') + \sigma(b)}{\ell(a'b)} \end{aligned}$$

$$\begin{aligned} &= \frac{\sigma(a'b)}{\ell(a'b)} \\ &= \mu(a'b) \\ &= \mu(q). \end{aligned}$$

Fall 3: Sei $p = ab'$ mit $b = b'q$. Mit dem Korollar 1.44 folgt, da $b = b'q$ linksschief ist:

$$\mu(b') \geq \mu(\underbrace{b'q}_{=b}) \geq \mu(q).$$

Somit gilt $\mu(a) \geq \mu(b) \geq \mu(q)$. Also gilt mit $\mu(a) \geq \mu(q)$ und $\mu(b') \geq \mu(q)$:

$$\begin{aligned} \mu(p) &= \mu(ab') \\ &= \frac{\sigma(ab')}{\ell(ab')} \\ &= \frac{\sigma(a) + \sigma(b')}{\ell(ab')} \\ &= \frac{\ell(a)\mu(a) + \ell(b')\mu(b')}{\ell(ab')} \\ &\geq \frac{\ell(a)\mu(q) + \ell(b')\mu(q)}{\ell(ab')} \\ &= \mu(q). \end{aligned}$$

Damit ist das Lemma bewiesen. ■

Lemma 1.51 *Jede Folge $a \in \mathbb{R}^n$ mit Gewichten $w \in \mathbb{N}^n$ besitzt eine eindeutig steigend linksschiefe Partition.*

Beweis: Wir führen den Beweis durch Induktion über n.

Induktionsanfang ($n = 1$): Klar, nach Definition.

Induktionsschritt ($n \to n + 1$): Wir betrachten eine Folge $(a_1, \ldots, a_n, a_{n+1})$. Sei weiter $A_1 \cdots A_k$ die steigend linksschiefe Partition von $(a_1, \ldots, a_n)$.

Gilt $\mu(A_k) < a_{n+1}/w_{n+1} = \mu(a_{n+1})$, dann ist $A_1 \cdots A_k \cdot (a_{n+1})$ eine neue steigend linksschiefe Partition.

Andernfalls bestimmen wir ein maximales i mit $\mu(A_{i-1}) < \mu(A_i \cdots A_k \cdot (a_{n+1}))$. Dann behaupten wir, dass die neue Partition $A_1 \cdots A_{i-1} \cdot (A_i \cdots A_k \cdot (a_{n+1}))$ eine steigend linksschiefe ist.

Nach Definition ist (a_{n+1}) linksschief. Nach dem vorherigen Lemma 1.50 und der Wahl von i ist dann auch $A_k \cdot (a_{n+1})$ linksschief. Weiter gilt allgemein für $j \geq i$, dass $A_j \cdots A_k \cdot (a_{n+1})$ linksschief ist. Somit ist auch $A_i \cdots A_k \cdot (a_{n+1})$ linksschief.

Nach Konstruktion ist also die jeweils konstruierte Partition eine steigend linksschiefe Partition.

Es bleibt noch die Eindeutigkeit zu zeigen. Nehmen wir an, es gäbe zwei verschiedene steigend linksschiefe Partitionen. Betrachten wir, wie in der folgenden Abbildung 1.45 skizziert, die jeweils linkesten Teilfolgen in ihren Partition, in denen sich die beiden Partitionen unterscheiden.

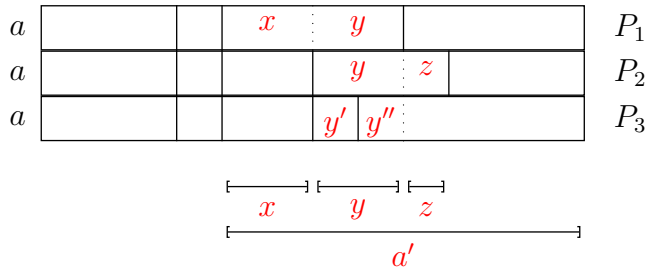


Abbildung 1.45: Skizze: Beweis der Eindeutigkeit der linksschiefen Partition

Betrachten wir zuerst den Fall P_1 gegen P_2 Sei zuerst $z = \varepsilon$. Dann gilt $\mu(x) < \mu(y)$, da P_2 eine steigend linksschiefe Partition ist. Da xy in P_1 linksschief ist, gilt weiterhin $\mu(x) \geq \mu(y)$ und wir erhalten den gewünschten Widerspruch.

Sei nun $z \neq \varepsilon$. Da yz nach der Partition P_2 linksschief ist, gilt $\mu(y) \geq \mu(z)$. Mit Korollar 1.44 folgt, dass $\mu(y) \geq \mu(yz) \geq \mu(z)$.

Nach Wahl der steigend linksschiefen Partition P_2 gilt $\mu(x) < \mu(yz)$. Wie wir eben gezeigt haben, gilt auch $\mu(yz) \leq \mu(y)$. Damit ist $\mu(x) < \mu(y)$ und somit ist xy nicht linksschief. Dies ist ein Widerspruch zur Annahme, dass P_1 eine steigend linksschiefe Partition ist.

Es bleibt noch der Fall P_1 gegen P_3 zu betrachten. Nach P_1 gilt $\mu(x) \geq \mu(y)$ und nach P_3 gilt $\mu(x) < \mu(y')$. Also gilt $\mu(y) < \mu(y')$. Sei y'' so gewählt, dass $y = y'y''$ und sei weiter $y = y' \cdot Y_2 \cdots Y_k$ eine steigend linksschiefe Partition von y, die sich aus P_3 ergibt. Dann gilt $\mu(y') < \mu(Y_2) < \cdots < \mu(Y_k)$. Dann muss aber $\mu(y') < \mu(y'')$ sein. Somit gilt $\mu(y) < \mu(y') < \mu(y'')$, was ein offensichtlicher Widerspruch ist. ■

Wir merken noch an, dass sich aus diesem Beweis unmittelbar ein Konstruktionsalgorithmus für die steigende linksschiefe Partition einer gegebenen gewichteten Folge ergibt. Die Laufzeit ist auch hier wieder linear in der Länge der Folge.

Korollar 1.52 *Für eine Folge $a \in \mathbb{R}^n$ mit Gewichten $w \in \mathbb{N}^n$ kann ihre steigend linksschiefe Partition in Zeit $O(n)$ berechnet werden.*

Beweis: Wir müssen nur die Laufzeit beweisen. Für jedes neues Folgenglied a_i wird ein neues Segment generiert. Die Anzahl der benötigten Verschmelzungen kann insgesamt nicht größer als die Anzahl der insgesamt generierten Segmente sein. Da diese $O(n)$ ist, kann es auch nur $O(n)$ Verschmelzungen geben und somit ist die Gesamtlaufzeit linear. ■

Das folgende Lemma zeigt, wie uns die steigende linksschiefe Partition für die Berechnung der Funktion φ in den benötigten Fällen hilft.

Lemma 1.53 *Sei $i \leq r \leq j \in \mathbb{N}$ und sei $(a_i, \ldots, a_j) \in \mathbb{R}^{j-i+1}$ eine reelle Folge mit Gewichten $(w_i, \ldots, w_j) \in \mathbb{N}^{j-i+1}$. Weiter bezeichne $p(m)$ die Endposition des Segments einer steigend linksschiefen Partition von $(a_i, \ldots, a_{r-1})$, die an Position m beginnt. Dann gilt $\varphi(i, r-1) = p(i)$.*

Beweis: Sei $A_1 \cdots A_k$ die steigend linksschiefe Partition von $(a_i, \ldots, a_{r-1})$. Für einen Widerspruchsbeweis nehmen wir an, dass $\varphi(i, r-1) \neq p(i)$.

Fall 1 ($\boldsymbol{\varphi(i, r-1) > p(i)}$): Da für jedes $m \in [1:k]$ das Segment A_m linksschief ist, gilt für jedes Präfix A'_m von A_m, dass $\mu(A'_m) \geq \mu(A_m)$. Da $\mu(A_1) < \mu(A_m)$ für alle $m \in [2:k]$ (steigend linksschiefe Partition), gilt mit Lemma 1.43 für ein $m \in [2:k]$ also

$$\mu(i, \varphi(i, r-1) = \mu(A_1 \cdot A_2 \cdots A_{m-1} \cdot A'_m) > \mu(A_1) = \mu(i, p(i))$$

Dies ist ein Widerspruch zur Definition von φ.

Fall 2 ($\boldsymbol{\varphi(i, r-1) < p(i)}$): Dann ist $(a_i, \ldots, a_{\varphi(i,r-1)})$ ein Präfix von A_1. Da A_1 linksschief ist, gilt $\mu(i, \varphi(i, r-1) \geq \mu(\varphi(i, r-1) + 1, p(i))$. Aufgrund von Korollar 1.44 gilt dann allerdings auch $\mu(i, \varphi(i, r-1) \geq \mu(i, p(i))$, was ein Widerspruch zur Definition von φ ist. ■

In Abbildung 1.46 ist der so modifizierte Algorithmus zur Lösung von WMASS angegeben. Beachte hierbei, dass p nur partiell definiert ist. Für $i \in [i_{j-1} : r_j - 1]$ gibt $p(i)$ die Endposition des Segments, das an Position i beginnt, einer steigend

```
WMASS (real a[], real w[], int n, int λ)
begin
    int opt := 0;
    int L := 0;
    int R := 0;
    int p[1 : n] ;                                   /* not initialized */
    int i_{j_0-1} := 1;
    for (j := j_0; j ≤ n; j++) do
        update increasing leftskew partition p of (a_{i_{j-1}}, ..., a_{r_j-1});
        i_j := Best(i_{j-1}, r_j, j);
        if (μ(i_j, j) > opt) then
            opt := μ(i_j, j);
            L := i_j;
            R := j;
end

Best(ℓ, r, j)
begin
    int i := ℓ;
    while ((i < r) && (μ(i, p(i)) ≤ μ(i, j))) do
        i := p(i) + 1;
    return i;
end
```

Abbildung 1.46: Algorithmus: WMASS

linksschiefen Partition von $(a_{i_{j-1}}, \ldots, a_{r_j-1})$. Für alle anderen Werte ist $p(i)$ entweder undefiniert oder enthält einen Wert, der ohne Bedeutung ist.

Die Prozedur UPDATE wird konsekutiv für die Teilfolgen $(a_{i_{j-1}}, \ldots, a_{r_j-1})$ aufgerufen, wobei wir immer schon für einen Präfix der Folge (eventuelle auch einem leeren, wie zu Beginn) die eindeutige steigende linksschiefe Partition bestimmt haben. Wir müssen diese also nur noch ergänzen, wie im Lemma 1.51 beschrieben. Somit bestimmen wir mittels UPDATE Stück für Stück die eindeutige steigende linksschiefe Partion von a, die sich in linearer Zeit berechnen lässt. Somit sind die zusätzlichen Berechnungskosten zum generischen Algorithmus $O(n)$ und wir erhalten folgenden Satz.

Theorem 1.54 *Für eine Folge $a \in \mathbb{R}^n$ mit Gewichten $w \in \mathbb{N}^n$ kann* WMASS *in Zeit $O(n)$ gelöst werden.*

Suffix Trees Revisited 2

2.1 Definition von Suffix Tries und Suffix Trees

In diesem Kapitel betrachten wir einige Algorithmen zur Konstruktion von Suffix Tries bzw. Suffix Trees. Letztere werden zur effizienten Suche von Wörtern und Teilstrings (z.B. Tandem-Repeats) im nächsten Kapitel benötigt.

2.1.1 Σ-Bäume und (kompakte) Σ^+-Bäume

Zunächst definieren wir die so genannten Σ-Bäume.

Definition 2.1 *Sei Σ ein Alphabet. Ein* Σ-Baum *ist ein gewurzelter Baum mit Kantenmarkierungen aus Σ, so dass kein Knoten zwei ausgehende Kanten mit derselben Markierung besitzt. Ein Σ-Baum wird oft auch als* Trie *bezeichnet.*

In Abbildung 2.1 ist ein Beispiel eines Σ-Baumes mit $\Sigma = \{a, b\}$ angegeben.

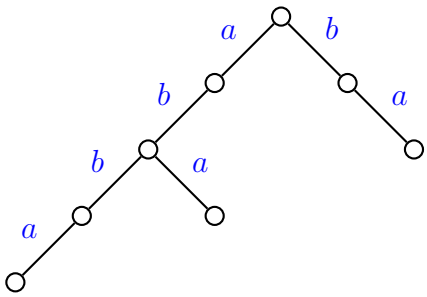


Abbildung 2.1: Beispiel: Ein Σ-Baum T_1 mit $\Sigma = \{a, b\}$

Erlaubt man als Kantenmarkierung Zeichenreihen über Σ, so erhält man die so genannten Σ^+-Bäume.

Definition 2.2 *Sei Σ ein Alphabet. Ein* Σ^+-Baum *ist ein gewurzelter Baum mit Kantenmarkierungen aus Σ^+, so dass kein Knoten zwei ausgehende Kanten besitzt, deren Markierungen mit demselben Zeichen aus Σ beginnen.*

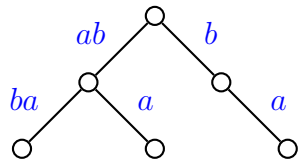


Abbildung 2.2: Beispiel: Ein Σ^+-Baum T_2 mit $\Sigma = \{a, b\}$

In Abbildung 2.2 ist ein Beispiel eines Σ^+-Baumes mit $\Sigma = \{a, b\}$ angegeben.

Werden in Σ^+-Bäumen Knoten mit nur einem Kind verboten, so erhalten wir kompakte Σ^+-Bäume.

Definition 2.3 *Ein Σ^+-Baum heißt* kompakt*, wenn es keinen Knoten außer der Wurzel mit nur einem Kind gibt. Ein solcher kompakter Σ^+-Baum wird auch als* kompaktifizierter Trie *bezeichnet.*

In Abbildung 2.3 ist ein Beispiel eines kompakten Σ^+-Baumes mit $\Sigma = \{a, b\}$ angegeben.

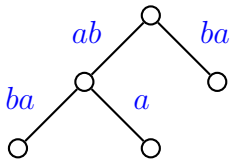


Abbildung 2.3: Beispiel: Ein kompakter Σ^+-Baum T_3 mit $\Sigma = \{a, b\}$

2.1.2 Grundlegende Notationen und elementare Eigenschaften

In diesem Abschnitt wollen wir noch einige grundlegende Notationen und Eigenschaften von Σ^+-Bäumen einführen.

Sei im Folgenden T ein Σ^+-Baum.

- Für einen Knoten $v \in V(T)$ bezeichnen wir mit $\text{path}(v)$ die Konkatenation der Kantenmarkierungen auf dem Pfad von der Wurzel zu v, siehe auch Abbildung 2.4, und $|\text{path}(v)|$ wird als *Worttiefe* von v bezeichnet.
- Wir sagen, dass eine Zeichenreihe $x \in \Sigma^*$ von T *dargestellt* wird, wenn es einen Knoten $v \in V(T)$ und ein $y \in \Sigma^*$ gibt, so dass $\text{path}(v) = xy$.

  In Abbildung 2.4 stellt der Σ^+-Baum unter anderen das Wort *abb* dar.

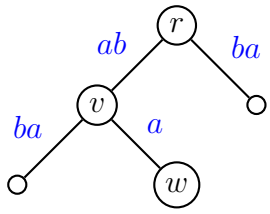


Abbildung 2.4: Beispiel: $\text{path}(v) = ab$ und $\text{path}(w) = aba$ in T_3

- Gilt $\text{path}(v) = x$ für einen Knoten $v \in V(T)$ und eine Zeichenreihe $x \in \Sigma^*$, dann schreiben wir $\overline{x}$ für v.

  In Abbildung 2.4 gilt im angegebenen Σ^+-Baum beispielsweise $\overline{ab} = v$ und $\overline{\varepsilon} = r$ bezeichnet die Wurzel dieses Baumes.

- Mit $\text{words}(T)$ bezeichnen wir die Menge der Wörter, die im Σ^+-Baum T dargestellt werden. Formal lässt sich das wie folgt definieren:

  $$\text{words}(T) = \{x \ : \ \exists v \in V(T), y \in \Sigma^* : \text{path}(v) = xy\}\,.$$

  In Abbildung 2.4 gilt für den Σ^+-Baum T beispielsweise

  $$\text{words}(T) = \{\varepsilon, a, ab, aba, abb, abba, b, ba\},$$

  wobei ε wie üblich das leere Wort bezeichnet.

  Für die Bäume aus den Abbildungen 2.1, 2.2 und 2.3 gilt

  $$\text{words}(T_1) = \text{words}(T_2) = \text{words}(T_3).$$

2.1.3 Suffix Tries und Suffix Trees

Bevor wir zu Suffix Tries und Suffix Trees kommen, wiederholen wir erst noch die Definitionen und zugehörigen Notationen zu Präfixen, Suffixen und Teilwörtern.

Notation 2.4 *Sei Σ ein Alphabet und $w \in \Sigma^*$. Sei weiter v ein* Teilwort *von w (d.h. es gibt $x, y \in \Sigma^*$ mit $w = xvy$), dann schreiben wir auch $v \sqsubseteq w$.*

Beobachtung 2.5 *Seien $v, w \in \Sigma^*$ und $¢, \$ \notin \Sigma$, dann ist v genau dann ein Präfix (bzw. Suffix) von w, wenn $¢v \sqsubseteq ¢w$ (bzw. $v\$ \sqsubseteq w\$$) gilt.*

Damit kommen wir zur Definition eines Suffix Tries.

Definition 2.6 *Sei Σ ein Alphabet. Ein* Suffix Trie *für ein Wort $t \in \Sigma^*$ ist ein Σ-Baum T mit der Eigenschaft $words(T) = \{w \in \Sigma^* \ : \ w \sqsubseteq t\}$.*

In Abbildung 2.5 ist der Suffix Trie für das Wort *abbab* angegeben.

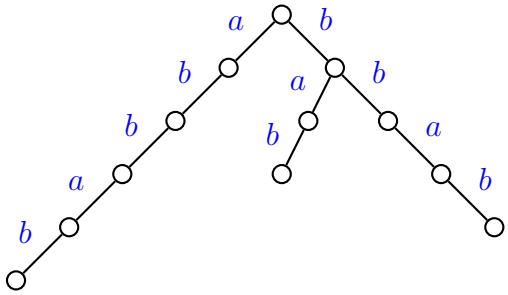


Abbildung 2.5: Beispiel für einen Suffix Trie mit $t = abbab$

Kommen wir nun zur Definition eines Suffix-Baumes.

Definition 2.7 *Sei Σ ein Alphabet und $t \in \Sigma^*$. Ein* Suffix-Baum *(engl.* Suffix Tree*) für t ist ein kompakter Σ^+-Baum $T = T(t)$ mit $words(T) = \{w \in \Sigma^* : w \sqsubseteq t\}$.*

In Abbildung 2.6 ist links ein Suffix-Baum für das Wort *abbab* angegeben.

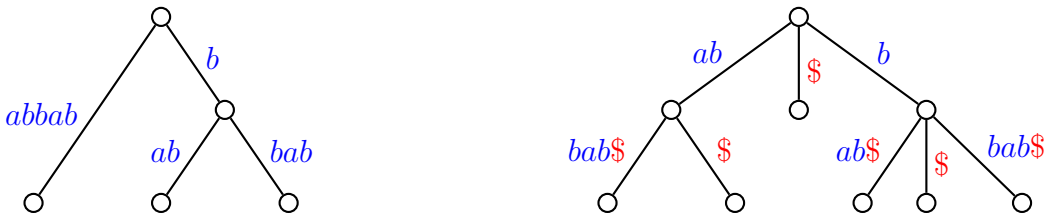


Abbildung 2.6: Beispiel: Suffix-Bäume für $t = abbab$ und $t = abbab\$$

Warum diese Bäume Suffix Tries bzw. Suffix Trees statt Infix Tries bzw. Infix Trees heißen, wird gleich in den folgenden Bemerkungen klar.

Bemerkungen: Halten wir zunächst ein paar nützliche Eigenschaften von Suffix Tries bzw. -Trees fest.

- Oft betrachtet man statt des Suffix Tries bzw. Suffix Trees für $t \in \Sigma^*$ den Suffix Trie bzw. Suffix Tree für $t\$ \in (\Sigma \cup \{\$\})^*$, wobei $\$ \notin \Sigma$ gilt. Der Vorteil dieser Betrachtungsweise ist, dass dann jedes Suffix von t zu einem Blatt des Suffix Tries bzw. Suffix Trees für t korrespondiert. Damit wird dann auch die Bezeichnung Suffix Trie bzw. Suffix Tree klar.

- Als Kantenmarkierungen werden keine Zeichenreihen verwendet, sondern so genannte *Referenzen* (Start- und Endposition) auf das Originalwort. Für ein Beispiel siehe auch Abbildung 2.7. Statt ab wird dort die Referenz $(1, 2)$ für t_1t_2 im Wort $t = abbab\$$ angegeben. Man beachte, dass ab mehrfach in t vorkommt, nämlich ab Position 1 und ab Position 4. Die Wahl der Referenz wird dabei nicht willkürlich sein. Wir betrachten dazu die Kante mit Kantenmarkierung w, deren Referenz bestimmt werden soll. Im Unterbaum des Suffix-Baumes, der am unteren Ende der betrachteten Kante gewurzelt ist, wird das längste im Unterbaum dargestelle Suffix s betrachtet. Dann muss auch $w \cdot s$ ein Suffix von t sein und als Referenz für die Kantenmarkierung w wird $(|t| - |w \cdot s| + 1, |t| - |s|)$ gewählt.

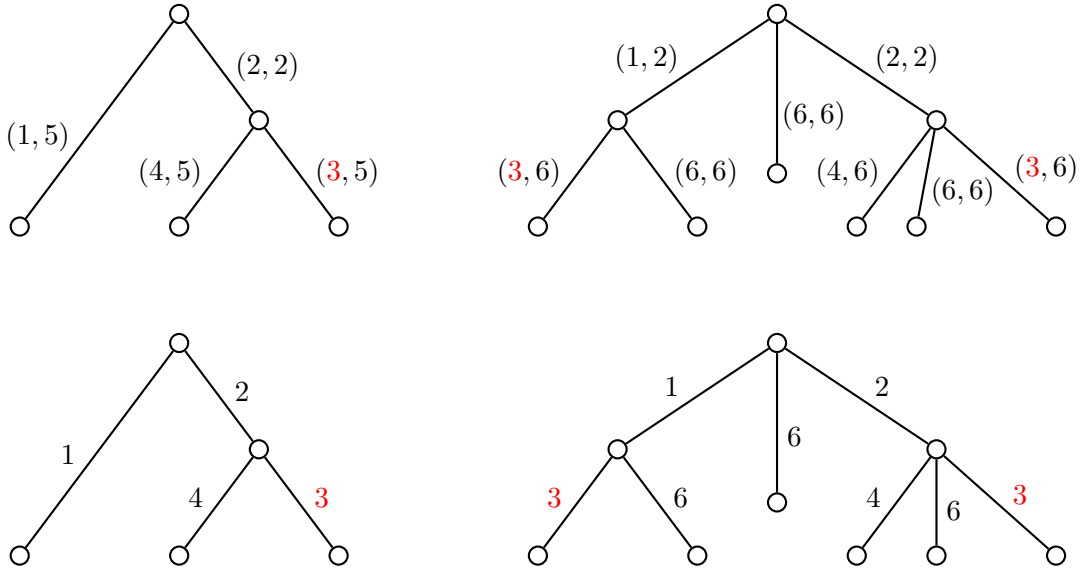


Abbildung 2.7: Beispiel: Suffix-Bäume für $t = abbab$ bzw. $t = abbab\$$ (links bzw. rechts) mit normalen (oben) bzw. reduzierten (unten) Referenzen als Kantenmarkierungen

- Der Platzbedarf für einen Suffix-Baum beträgt mit dieser Referenzdarstellung $O(|t|)$. Würde man hingegen die Zeichenreihen an die Kanten schreiben, so könnte der Platzbedarf auch auf $\Theta(|t|^2)$ ansteigen. Der Beweis ist dem Leser als Übungsaufgabe überlassen.

- In der Literatur wird der Suffix Trie manchmal auch als *atomic suffix tree* und der Suffix Tree selbst als *compact suffix tree* bezeichnet. Dies führt oft zu Verwirrung, insbesondere, da heutzutage oft unter einem kompakten Suffix-Baum (compact suffix tree) etwas ganz anderes verstanden wird.

2.2 Repräsentationen von Bäumen

Zuerst überlegen wir uns einige Möglichkeiten, wie man Bäume im Allgemeinen und Suffix-Bäume im Speziellen überhaupt speichern kann. In einem Baum muss man im Wesentlichen die Kinder eines Knoten speichern. Dafür gibt es verschiedene Möglichkeiten, die jeweils ihre Vor- und Nachteile haben.

Allgemein erinnern wir uns zuerst daran, dass ein Baum mit m Blättern, der keinen Knoten mit nur einem Kind besitzt, maximal $m-1$ innere Knoten haben kann. Somit besitzt ein Suffix-Baum für t weniger als $|t|$ innere Knoten und somit insgesamt weniger als $2|t|$ Knoten. Der Leser möge sich überlegen, was passiert, wenn in einem Suffix-Baum die Wurzel nur ein Kind besitzt.

Für eine detailliertere Analyse in der Bioinformatik nehmen wir an, dass sowohl Integers als auch Verweise für Ihre Speicherung 4 Bytes benötigen (32-Bit-Architektur). Bei Annahme einer 64-Bit-Architektur wären es hingegen 8 Bytes und in den folgenden detaillierte Analysen verdoppelt sich der Speicherbedarf in Bytes in etwa. Für die Kantenmarkierungen werden wir in der Regel nicht die volle Referenz (i, j) speichern, sondern oft nur die Startposition i. Da in einem Suffixbaum alle Suffixe gespeichert sind, muss von dem Knoten, den wir über die Kante (i, j) erreichen, eine Kante mit der Startposition $j + 1$ beginnen. Wenn wir uns diese merken (meist als erste ausgehende Kante in unseren Darstellungen), können wir daraus die Endposition j rekonstruieren. Hierfür ist es wichtig, dass wir für ein Teilwort w von t die Referenz geeignet wählen (d.h. wie im vorherigen Abschnitt beschrieben).

Im Folgenden ist bei der Zeitanalyse der verschiedenen Darstellungen von Bäumen mit Zeit die Zugriffszeit auf ein Kind gemeint, zu dem die Kantenmarkierung mit einem bestimmten Zeichen beginnt.

2.2.1 Darstellung der Kinder mit Feldern

Die Kinder eines Knotens lassen sich sehr einfach mit Hilfe eines Feldes der Größe $|\Sigma|$ darstellen.

- Platz: $O(|t| \cdot |\Sigma|)$.

  Dies folgt daraus, dass für jeden Knoten ein Feld mit Platzbedarf $O(|\Sigma|)$ benötigt wird.

- Zeit: $O(1)$.

  Die üblichen Realisierungen von Feldern erlauben einen Zugriff in konstanter Zeit auf die einzelnen Feldelemente.

Der Zugriff ist also sehr schnell, wo hingegen der Platzbedarf, insbesondere bei großen Alphabeten, doch sehr groß werden kann.

Für eine DNA-Sequenz sind also pro Knoten maximal 20 Bytes nötig (16 Bytes für die Verweise und 4 Bytes für die Referenz auf die Kantenmarkierung). Da es etwa doppelt so viele Knoten wie Blätter (also Nukleotide in der gegebenen DNA-Sequenz) gibt, sind als pro Nukleotid maximal 40 Bytes nötig.

Für ein Protein sind also pro Knoten maximal 84 Bytes nötig (80 Bytes für die Verweise und 4 Bytes für die Referenz auf die Kantenmarkierung). Da es etwa doppelt so viele Knoten wie Blätter (also Aminosäuren im gegebenen Protein) gibt, sind als pro Aminosäure maximal 168 Bytes nötig.

Für die zweite Referenz müssen wir uns noch explizit das Kind merken, deren Referenz im Wesentlichen das Ende der gesuchten Referenz angibt. Dazu sind im Falle von DNA bzw. Proteinen noch jeweils 2 Bits bzw. 5 Bits je inneren Knoten nötig (wenn man die Kinder in konstanter Zeit finden will).

Insgesamt sind daher pro Nukleotid bzw. pro Aminosäure etwa 42 Bytes bzw. etwa 173 Bytes nötig.

2.2.2 Darstellung der Kinder mit Listen

Eine andere Möglichkeit ist, die Kinder eines Knotens in einer linearen Liste zu verwalten, wobei das erste Kind einer Liste immer dasjenige ist, das für den zweiten Referenzwert der inzidenten Kante zu seinem Elter nötig ist.

- Platz: $O(|t|)$.

  Für jeden Knoten ist der Platzbedarf proportional zur Anzahl seiner Kinder. Damit ist Platzbedarf insgesamt proportional zur Anzahl der Knoten des Suffix-Baumes, da jeder Knoten (mit Ausnahme der Wurzel) das Kind eines Knotens ist. Im Suffix-Baum gilt, dass jeder Knoten entweder kein oder mindestens zwei Kinder hat. Für solche Bäume ist bekannt, dass die Anzahl der inneren Knoten kleiner ist als die Anzahl der Blätter. Da ein Suffix-Baum für einen Text der Länge m maximal m Blätter besitzt, folgt daraus die Behauptung für den Platzbedarf.

- Zeit: $O(|\Sigma|)$.

  Leider ist hier die Zugriffszeit auf ein Kind sehr groß, da im schlimmsten Fall (aber größenordnungsmäßig auch im Mittel) die gesamte Kinderliste eines Knotens durchlaufen werden muss und diese bis zu $|\Sigma|$ Elemente umfassen kann.

In vielen Anwendungen wird jedoch kein direkter Zugriff auf ein bestimmtes Kind benötigt, sondern eine effiziente Traversierung aller Kinder. Dies ist mit der Geschwisterliste aber auch wieder in konstanter Zeit pro Kind möglich, wenn keine besondere Ordnung auf den Kindern berücksichtigt werden soll.

Pro Knoten des Baumes sind also maximal 12 Bytes nötig (8 Bytes für die Verweise und 4 Bytes für die Kantenmarkierung). Für die gespeicherte Sequenz sind also 24 Bytes pro Element nötig. Bei geschickterer Darstellung der Blätter (hier ist ja kein Verweis auf sein ältestes Kind nötig) kommen wir mit 20 Bytes aus.

Wenn man eine Ordnung in den Listen aufrecht erhalten will, so muss man auf die reduzierten Referenzen verzichten und pro Nukleotid nochmals 4–8 Bytes opfern (je nachdem, wie geschickt man die Referenzen an den zu Blättern inzidenten Kanten implementiert).

2.2.3 Darstellung der Kinder mit balancierten Bäumen

Die Kinder lassen sich auch mit Hilfe von balancierten Suchbäumen (AVL-, Rot-Schwarz-, B-Bäume, etc.) verwalten:

- Platz: $O(|t|)$

  Da der Platzbedarf für einen Knoten ebenso wie bei linearen Listen proportional zur Anzahl der Kinder ist, folgt die Behauptung für den Platzbedarf unmittelbar.

- Zeit: $O(\log(|\Sigma|))$.

  Da die Tiefe von balancierten Suchbäumen logarithmisch in der Anzahl der abzuspeichernden Schlüssel ist, folgt die Behauptung unmittelbar.

  Auch hier ist eine Traversierung der Kinder eines Knotens in konstanter Zeit pro Kind möglich, indem man einfach den zugehörigen balancierten Baum traversiert.

Auch hier sind für eine Sequenz also pro Element maximal etwa 24-26 Bytes nötig. Dies folgt daraus, das balancierte Bäume etwa genauso viele Links besitzen wie Listen (wenn man annimmt, dass die Daten auch in den inneren Knoten gespeichert werden und man auf die unnötigen Referenzen in den Blättern verzichtet, sonst erhöht sich der Platzbedarf jeweils um etwa einen Faktor 2). Es werden jedoch noch zusätzlich 1–2 Bytes für die Verwaltung der Balancierung benötigt.

31.10.24

2.2.4 Darstellung des Baumes mit einer Hash-Tabelle

Eine weitere Möglichkeit ist die Verwaltung der Kinder aller Knoten in einem einzigen großen Feld der Größe $O(|t|)$. Um nun für ein Knoten auf ein spezielles Kind zuzugreifen wird dann eine *Hashfunktion* verwendet:

$$h : V \times \Sigma \to \mathbb{N} : (v, a) \mapsto h(v, a).$$

Hierbei interpretieren wir die Knotenmenge V in geeigneter Weise als natürliche Zahlen, d.h. $V \subset \mathbb{N}$. Zu jedem Knoten und dem Symbol, die das Kind identifizieren, wird ein Index des globalen Feldes bestimmt, an der die gewünschte Information enthalten ist (also hier die entsprechende Kantenmarkierung).

Leider bilden Hashfunktionen ein relativ großes Universum von potentiellen Referenzen (hier Paare von Knoten und Symbolen aus Σ, also mit einer Mächtigkeit von $\Theta(|t| \cdot |\Sigma|)$) auf ein kleines Intervall ab (hier Indizes aus $[1 : \ell]$ mit $\ell = \Theta(|t|)$). Daher sind so genannte *Kollisionen* prinzipiell nicht auszuschließen. Ein Beispiel ist das so genannte *Geburtstagsparadoxon*. Ordnet man jeder Person in einem Raum eine Zahl aus dem Intervall $[1 : 366]$ zu (nämlich ihren Geburtstag), dann ist bereits ab 23 Personen die Wahrscheinlichkeit größer als 50%, dass zwei Personen denselben Wert erhalten. Also muss man beim Hashing mit diesen Kollisionen leben und diese geeignet auflösen. Für die Details der Kollisionsauflösung verweisen wir auf andere Vorlesungen.

Um solche Kollisionen überhaupt festzustellen, enthält jeder Feldeintrag i neben den normalen Informationen noch die Informationen, wessen Kind er ist und über welches Symbol er von seinem Elter erreichbar ist. Somit lassen sich Kollisionen leicht feststellen und die üblichen Operationen zur Kollisionsauflösung anwenden.

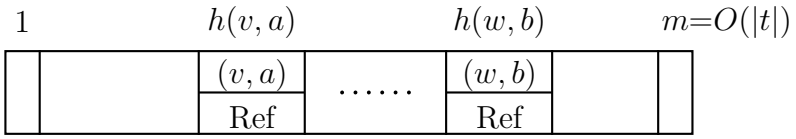


Abbildung 2.8: Skizze: Realisierung mittels eines Feldes und Hashing

- Platz: $O(|t|)$

  Das folgt unmittelbar aus der obigen Diskussion.

- Zeit: $O(1)$ (bei einer perfekten Hashfunktion)

  Im Wesentlichen erfolgt der Zugriff in konstanter Zeit, wenn man voraussetzt, dass sich die Hashfunktion einfach (d.h. in konstanter Zeit) berechnen lässt und dass sich Kollisionen effizient auflösen lassen.

Hier sind für jeden Eintrag 4 Bytes für die Kantenmarkierung nötig. Für die Verwaltung (ein Verweis auf den Elter und das Zeichen, das das Kind identifiziert) sind 5 Bytes nötig. Um die Endposition zu merken, muss man sich noch das entsprechende Kind merken, was ein weiteres Byte kostet (für das Zeichen, über das das Kind erreicht wird). Da es maximal $2|t|$ Kanten gibt, sind somit maximal 20 Bytes pro Element nötig. Da beim Hashing für eine effiziente Kollisionsauflösung das Feld nur bis zu etwa 80% gefüllt sein darf, muss das Feld entsprechend größer gewählt werden, so dass der Speicherbedarf insgesamt maximal 25 Bytes pro Element der Sequenz beträgt.

2.2.5 Speicherplatzeffiziente Feld-Darstellung

Man kann die Methode der linearen Listen auch selbst mit Hilfe eines Feldes implementieren. Dabei werden die Geschwisterlisten als konsekutive Feldelemente abgespeichert. Begonnen wird dabei mit der Geschwisterliste des ältesten Kindes der Wurzel, also mit den Kindern der Wurzel, die dann ab Position 1 des Feldes stehen. Die Wurzel selbst wird dabei nicht explizit dargestellt.

Auch hier unterscheidet man wieder zwischen internen Knoten und Blättern. Ein interner Knoten belegt zwei Feldelemente, ein Blatt hingegen nur ein Feldelement. Ein innerer Knoten hat einen Verweis auf das Feldelement, in dem die Geschwisterliste des ältesten Kindes konsekutiv abgespeichert wird. Im zweiten Feldelement steht die Startposition der Kantenmarkierung innerhalb von t für die Kante, die in den Knoten hineinreicht. Für ein Blatt steht nur die Startposition der Kantenmarkierung innerhalb von t für die Kante, die in das Blatt hineinreicht.

Ein Beispiel ist in Abbildung 2.9 für $t = abbab\$$ angegeben. Zur Unterscheidung der beiden Knotentypen wird ein Bit spendiert. Dies ist in der Abbildung 2.9 als hochgestelltes B für die Blätter zu erkennen (die anderen sind innere Knoten). Des Weiteren müssen mit Hilfe eines weiteren Bits auch noch die Enden der Geschwisterlisten markiert werden. Dies ist in Abbildung 2.9 mit einem tiefergestellten $*$ markiert. Verweise sind dort als rote Zahlen geschrieben.

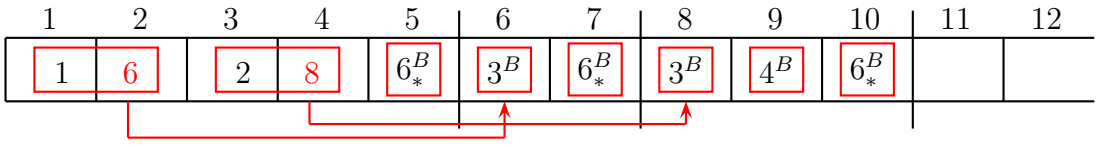


Abbildung 2.9: Beispiel: Feld-Darstellung des Suffix-Baumes aus Abbildung 2.6 für $t = abbab\$$

Es bleibt nur die Frage, wie man das Ende einer Kantenmarkierung findet. Das ist jedoch sehr einfach: Hat man den Beginn der Kantenmarkierung, so folgt man dem

Verweis dieses Knotens. Im folgenden Knoten steht dann der Beginn der nächsten Kantenmarkierung; dieser muss nur noch um es erniedrigt werden. Handelt es sich um ein Blatt, so gibt es keinen Folgeverweis, aber die Endposition ist dann das Ende, also $|t|$. Hierfür ist es aber wichtig, dass die Knoten der Geschwisterliste in der richtigen Reihenfolge abgespeichert werden, nämlich das Kind, dessen zugehörige Kantenmarkierung am frühestens in t auftritt, als erstes (die Reihenfolge der anderen Kinder ist jedoch egal).

- Platz: $O(|t|)$.

  Wie bei Listen

- Zeit: $O(|\Sigma|)$.

  Leider ist hier die Zugriffszeit auf ein Kind sehr groß, da im schlimmsten Fall (aber größenordnungsmäßig auch im Mittel) die gesamte Kinderliste eines Knotens durchlaufen werden muss und diese bis zu $|\Sigma|$ Elemente umfassen kann.

  Auch hier ist eine Traversierung der Kinder eines Knotens in konstanter Zeit pro Kind möglich, indem man einfach im Feld weiterläuft bis man auf eine Markierung stößt, die das Ende der Geschwisterliste markiert.

Pro inneren Knoten wird nun eine Zahl, ein Verweis und zwei Bits gespeichert, pro Blatt eine Zahl und zwei Bits. Spendiert man von den 4 Bytes jeweils zwei Bits um die Bits dort zu Markierung von Blättern und dem Ende von Geschwisterlisten zu speichern, kommt man mit maximal 12 Bytes pro Sequenzelement aus.

2.3 WOTD-Algorithmus

Nun wollen wir einen einfachen Algorithmus zur Konstruktion von Suffix-Bäumen angeben. Der so genannte *WOTD-Algorithmus* (für write-only-top-down) fügt die Kanten in den Suffix-Baum von der Wurzel beginnend (daher top-down) ein.

2.3.1 Die Konstruktion

Zuerst werden alle möglichen Suffixe des Wortes $t\$$ (mit $t \in \Sigma^*$ und $\$ \notin \Sigma$) bestimmt und hierfür der rekursive Algorithmus mit der Wurzel und dieser Menge von Suffixen aufgerufen. Die Menge von Suffixen wird dabei natürlich durch deren Startpositionen repräsentiert. Der rekursive Algorithmus konstruiert nun einen Baum mit der angegeben Wurzel, der alle Wörter in der übergebenen Menge darstellen soll.

WOTD (char t[])

```
begin
    set of words S(t) := {y : ∃x ∈ Σ* : xy = t$};      /* all suffixes of t$ */
    node r;                                            /* root of the suffix-tree */
    WOTD_REC(S(t), r);
end

WOTD_REC(set of words S, node v)
begin
    sort S according to the first character using bucket-sort;
    let S_c := {x ∈ S : x = c = $ ∨ ∃z ∈ Σ*$ : x = cz} for all c ∈ Σ ∪ {$};
    for (c ∈ Σ ∪ {$}) do
        if (|S_c| = 1) then
            append a new leaf to v with edge label w ∈ S_c;
        else if (|S_c| > 1) then
            // determine a longest common prefix in S_c
            let p be a longest word in {p' ∈ Σ* : ∀x ∈ S_c: ∃z ∈ Σ*$): x = p'z};
            append a new node w to v with edge label p;
            S'_c := p^-1 · S_c := {z ∈ Σ*$ : ∃x ∈ S_c: x = pz};
            WOTD_REC(S'_c,w);
end
```

Abbildung 2.10: Algorithmus: WOTD (Write-Only-Top-Down)

Im rekursiven Algorithmus wird die übergebenen Menge von Wörtern nach dem ersten Buchstaben sortiert. Hierzu wird vorzugsweise ein Bucket-Sort verwendet. Rekursiv wird dann daraus der Suffix-Baum aufgebaut. Für den eigentlichen Aufbau traversiert man den zu konstruierenden Suffix-Baum top-down. Für jeden Knoten gibt es eine Menge S von Zeichenreihen (Suffixe von t\$), die noch zu verarbeiten sind. Diese wird dann nach dem ersten Zeichen (mittels eines Bucket-Sorts) sortiert und liefert eine Partition der Zeichenreihe in Mengen S_c mit demselben ersten Zeichen für $c \in \Sigma \cup \{\$\}$. Für jede nichtleere Menge S_c wird ein neues Kind generiert, wobei die inzidente Kante als Kantenmarkierung das längste gemeinsame Präfix p aus S_c erhält. Dann werden von jeder Zeichenreihe aus S_c dieses Präfix p entfernt und in einer neuen Menge S'_c gesammelt (beachte hierbei, dass Suffixe von Suffixen von t\$ wiederum Suffixe von t\$ sind). An dem neu konstruierten Kind wird nun die Prozedur rekursiv mit der Menge S'_c aufgerufen. Siehe hierzu den Algorithmus in Abbildung 2.10.

Da Werte immer nur geschrieben und nie modifiziert werden, erklärt sich nun der Teil write-only aus WOTD.

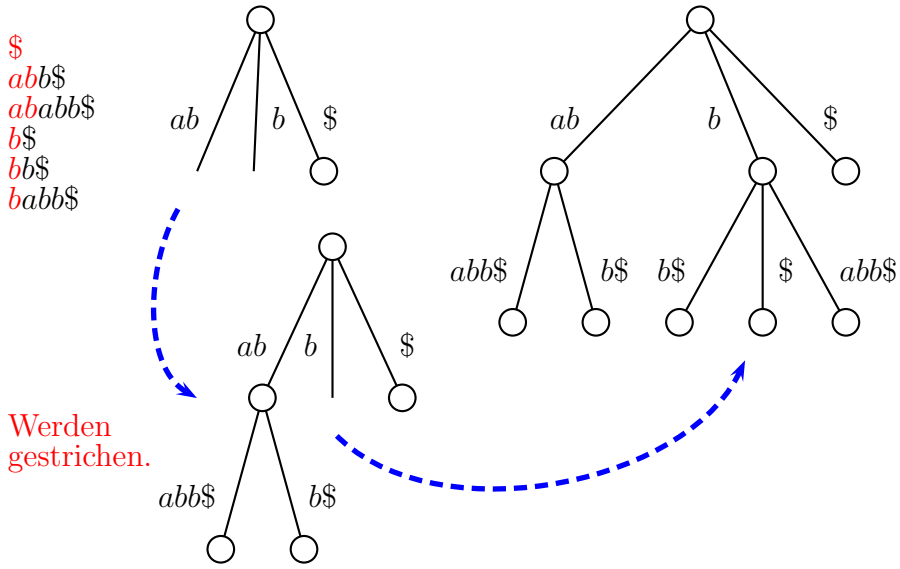


Abbildung 2.11: Beispiel: Konstruktion eines Suffix-Baumes für $t = ababb\$$ mittels WOTD

In der Implementierung wird man die Menge S (bzw. S_c) nicht wirklich als Menge von Wörtern implementieren, da sonst die Menge S bereits quadratischen Platzbedarf hätte. Da S jeweils eine Menge von Suffixen beschreibt, wird man diese Menge als Menge der Anfangspositionen der Suffixe in S realisieren.

Ein Beispiel für den Ablauf dieses Algorithmus mit dem Wort $t = ababb\$$ ist in Abbildung 2.11 angegeben. Wenn man den WOTD-Algorithmus für Worte ohne das Endsymbol \$, verwendet, entstehen nicht notwendigerweise Suffix-Bäume, es kann dann interne Konten mit nur einem Kind geben (betrachte beispielweise das Wort $t = abab$).

2.3.2 Zeitbedarf

Wir wollen nun den Zeitbedarf des Algorithmus analysieren. Für jeden Knoten wird ein Bucket-Sort nach dem ersten Zeichen ausgeführt. Dies geht in Zeit $O(|S|)$. Wir verteilen diese Kosten auf das jeweils erste Zeichen in den Mengen S_c. Somit erhält jedes erste Zeichen in S_c konstante Kosten.

Zur Ermittelung des längsten gemeinsamen Präfixes p in S_c werden in jeder Zeichenreihe in S_c maximal $|p|$ Vergleiche ausgeführt. Das erste Zeichen ist in allen Wörtern in S_c sind nach dem Bucket-Sort gleich (und muss nicht verglichen werden), nach Definition von p muss es an Position $p + 1$ in S_c zwei verschiedene Zeichen geben. Diese Kosten verteilen wir nun auf die Zeichen des längsten gemeinsamen Präfi-

xes. Da nach Vorsortierung $|p| \geq 1$ gilt, erhält auch hier wieder jedes Zeichens des längsten gemeinsamen Präfixes konstante Kosten.

Da anschließend das längste gemeinsame Präfix entfernt wird, enthält jedes Zeichen aus $S(t)$ maximal konstant viele Einheiten. Da für ein Wort t der Länge n gilt, dass die Anzahl der Zeichen aller Suffixe von t gleich $\binom{n+1}{2}$ ist, beträgt die Laufzeit im worst-case $O(n^2)$.

Theorem 2.8 *Für ein Wort $t \in \Sigma^n$ mit $n \in \mathbb{N}$ kann der zugehörige Suffix-Baum mit dem WOTD-Algorithmus im worst-case in Zeit $O(n^2)$ konstruiert werden.*

In der Praxis bricht der Algorithmus ja ab, wenn $|S_c| = 1$ ist. Für jeden internen Knoten v eines Suffix-Baumes gilt im average-case $|\text{path}(v)| = O(\log_{|\Sigma|}(n))$. Somit werden im average-case maximal die ersten $O(\log_{|\Sigma|}(n))$ Zeichen eines Wortes aus $S(t)$ betrachtet. Daraus ergibt sich dann im average-case eine Laufzeit von $O(n \log_{|\Sigma|}(n))$.

Theorem 2.9 *Für ein Wort $t \in \Sigma^n$ mit $n \in \mathbb{N}$ kann der zugehörige Suffix-Baum mit dem WOTD-Algorithmus im average-case in Zeit $O(n \log_{|\Sigma|}(n))$ konstruiert werden.*

2.4 Der Algorithmus von Ukkonen

In diesem Abschnitt wollen wir einen Algorithmus vorstellen, der im worst-case eine lineare Laufzeit zur Konstruktion eines Suffix-Baumes besitzt.

2.4.1 Suffix-Links

Zunächst einmal benötigen wir für die Darstellung des Algorithmus den Begriff eines Suffix-Links.

Definition 2.10 *Sei $t \in \Sigma^*$ und sei T der Suffix-Baum zu t. Zu jedem inneren Knoten $\overline{aw}$ von T mit $a \in \Sigma$, $w \in \Sigma^*$ (also mit Ausnahme der Wurzel) ist der* Suffix-Link *des Knotens $\overline{aw}$ als $\overline{w}$ definiert (in Zeichen $slink(\overline{aw}) = \overline{w}$).*

In Abbildung 2.12 sind die Suffix-Links für den Suffix-Baum für $t = ababb$ als gestrichelte Linien dargestellt.

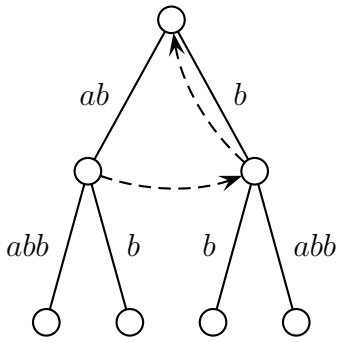


Abbildung 2.12: Beispiel: Die Suffix-Links im Suffix-Baum für $t = ababb$

Als nächstes sollten wir uns noch überlegen, dass Suffix-Links überhaupt wohldefiniert sind. Es könnte ja durchaus sein, dass es einen Knoten $\overline{aw}$ für $a \in \Sigma$ und $w \in \Sigma^*$ gibt, aber es keinen Knoten v mit $\text{path}(v) = w$ gibt.

Wir müssen also Folgendes zeigen: Wenn $\overline{aw}$ ein innerer Knoten des Suffix-Baums ist, dann ist auch $\overline{w}$ ein innerer Knoten. Beachte dabei, dass nach Definition $\overline{aw}$ nicht die Wurzel sein kann.

Ist $\overline{aw}$ ein innerer Knoten des Suffix-Baumes (ungleich der Wurzel), dann muss es Zeichen $b \neq c \in \Sigma$ geben, so dass sowohl $awb \sqsubseteq t$ als auch $awc \sqsubseteq t$ gilt. Dabei sind bx und cx' mit $x, x' \in \Sigma^*$ die Kantenmarkierungen von (mindestens) zwei ausgehenden Kanten aus dem Knoten $\overline{aw}$. Dies folgt aus der Definition des Suffix-Baumes als kompakter Σ^+-Baum und ist in Abbildung 2.13 schematisch dargestellt.

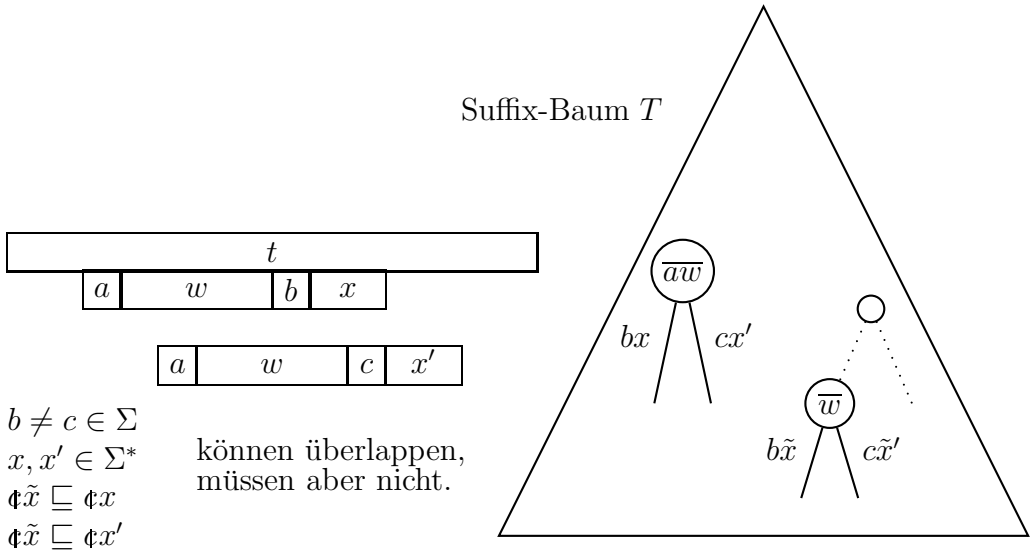


Abbildung 2.13: Schema: Suffix-Links sind wohldefiniert

Gilt jedoch sowohl $awb \sqsubseteq t$ als auch $awc \sqsubseteq t$, dann gilt auch $wb \sqsubseteq t$ und $wc \sqsubseteq t$. Somit muss $\overline{w}$ ein Knoten im Suffix-Baum für t sein.

Man beachte, dass man für Blätter eines Suffix-Baumes auf diese Weise keine Suffix-Links definieren kann. Hier kann es passieren, dass für ein Blatt $\overline{aw}$ kein Knoten $\overline{w}$ im Suffix-Baum existiert. Der Leser möge sich solche Beispiele selbst überlegen.

2.4.2 Verschachtelte Suffixe und verzweigende Teilwörter

Für die folgende Beschreibung des Algorithmus von Ukkonen zur Konstruktion von Suffix-Bäumen benötigen wir noch einige Definitionen und Notationen, die in diesem Abschnitt zur Verfügung gestellt werden.

Definition 2.11 *Sei $t \in \Sigma^*$. Ein Suffix $s \sqsubseteq t$ heißt* verschachtelt *(engl.* nested*), wenn sowohl $s\$ \sqsubseteq t\$$ als auch $sa \sqsubseteq t$ für ein $a \in \Sigma$ gilt.*

In Abbildung 2.14 ist ein verschachteltes Suffix s von t schematisch dargestellt. Man beachte hierbei, dass anders als in der Skizze das verschachtelte Vorkommen mit dem Suffix auch überlappen kann.

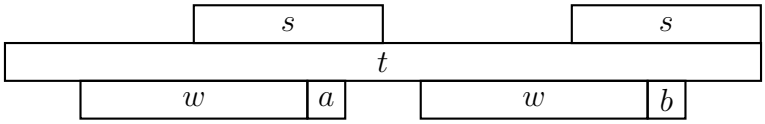


Abbildung 2.14: Skizze: verschachteltes Suffix s und rechtsverzweigendes Teilwort w

Definition 2.12 *Sei $t \in \Sigma^*$. Ein Teilwort $w \sqsubseteq t$ heißt* rechtsverzweigend *(engl.* rightbranching*), wenn es $a \neq b \in \Sigma$ mit $wa \sqsubseteq t$ und $wb \sqsubseteq t$ gibt.*

Ein rechtsverzweigendes Teilwort w von t ist in Abbildung 2.14 schematisch dargestellt Man beachte, dass anders als in der Skizze das Vorkommen des rechtsverzweigenden Teilwortes auch überlappen kann.

Für das Wort $t = abbabbba$ ist beispielsweise ba oder bba ein verschachteltes Suffix und bb ein rechtsverzweigendes Teilwort.

2.4.3 Idee von Ukkonens Algorithmus

Sei $t = t_1 \cdots t_n \in \Sigma^*$. Der Algorithmus konstruiert dann sukzessive Suffix-Bäume $T^1, \ldots, T^n$ mit $T^i = T(t_1 \cdots t_i)$. Nach Definition stellt T^i alle Teilwörter von $t_1 \cdots t_i$ und T^{i+1} alle Teilwörter von $t_1 \cdots t_i \cdot t_{i+1}$ dar, d.h words$(T^i) = \{w : w \sqsubseteq t_1 \cdots t_i\}$ und words$(T^{i+1}) = \{w : w \sqsubseteq t_1 \cdots t_{i+1}\}$.

T^1 ist offensichtlich leicht zu erstellen. Da jedes Teilwort von $t_1 \cdots t_i$ auch ein Teilwort von $t_1 \cdots t_i \cdot t_{i+1}$ ist, ist T^{i+1} quasi eine Erweiterung von T^i. Wir müssen uns nur noch überlegen, wie wir T^{i+1} aus T^i konstruieren können.

Im Weiteren verwenden wir die folgenden Abkürzungen: $x := t_1 \cdots t_i$ und $a := t_{i+1}$. Also müssen wir in T^i alle Teilwörter aus xa einfügen, die keine Teilwörter von x sind, um T^{i+1} zu erhalten. Dazu halten wir erst noch einige Notationen fest.

Notation 2.13 $I := \{w \in \Sigma^* : w \sqsubseteq xa \wedge w \not\sqsubseteq x\}$.

Halten wir zuerst die folgenden beiden offensichtlichen Lemmata fest.

Lemma 2.14 *Alle Wörter aus I sind Suffixe von xa.*

Beweis: Dies folgt unmittelbar aus der Definition von I. ■

Lemma 2.15 *Sei $sa \in I$, dann ist $\overline{sa}$ in T^{i+1} ein Blatt.*

Beweis: Da sa nach dem vorhergehenden Lemma ein Suffix von xa ist und sa kein Teilwort von x ist, folgt die Behauptung. ■

Damit können wir noch folgende wichtige Notation einführen.

Notation 2.16 $I^* := \{sa \in I : \overline{s}$ *ist kein Blatt in* $T^i\} \subseteq I$.

Für alle Wörter in $sa \in I \setminus I^*$ ist nicht sonderlich viel zu tun, da es sich bei $\overline{s}$ um ein Blatt in T^i handelt. Für die Konstruktion von T^{i+1} wird an die Kantenmarkierung des zu $\overline{s}$ inzidenten Blattes nur ein a angehängt und die Bezeichnung dieses Blattes wird zu $\overline{sa}$. Somit sind alle Wörter aus $I \setminus I^*$ nun ebenfalls dargestellt. Die eigentliche Arbeit besteht also nur bei der Darstellung der Wörter aus I^*.

05.11.24

Lemma 2.17 *Für ein Suffix sa von xa gilt genau dann $sa \in I^*$, wenn s ein verschachteltes Suffix von x ist und $sa \not\sqsubseteq x$.*

Beweis: $\Rightarrow$: Da $sa \in I^* \subseteq I$ gilt, gilt nach Definition von I, dass $sa \not\sqsubseteq x$.

Da $sa \in I^*$ ist, ist $\overline{s}$ kein Blatt in $T^i = T(x)$. Somit muss s sowohl ein Suffix von x als auch ein echtes Teilwort von x sein, d.h. s ist ein verschachteltes Suffix von x.

$\Leftarrow$: Da s ein verschachteltes Suffix von x ist, ist sa insbesondere ein Suffix von xa. Da außerdem $sa \not\sqsubseteq x$, folgt $sa \in I$.

Da s ein verschachteltes Suffix von x ist, gibt es ein $b \in \Sigma$, so dass auch $sb \sqsubseteq x$ gilt (mit $b \neq a$). Somit kann $\overline{s}$ kein Blatt in $T^i = T(x)$ sein und es folgt $sa \in I^*$. ■

Definition 2.18 *Das längste verschachtelte Suffix von x heißt* aktives Suffix *und wird mit $\alpha(x)$ bezeichnet.*

Als Beispiel hierfür ist das aktive Suffix von $x = ababb$ gleich $\alpha(x) = b$ und von $x = abbab$ gleich $\alpha(x) = ab$. Beachte, dass für das leere Wort kein aktives Suffix definiert ist.

Lemma 2.19 *Sei $x \in \Sigma^+$ und $a \in \Sigma$. Es gelten die folgenden Aussagen:*

1. *Für alle Suffixe s von x gilt: s ist genau dann verschachtelt, wenn $|s| \leq |\alpha(x)|$.*
2. *Für alle Suffixe s von x gilt: $sa \in I^* \Leftrightarrow |\alpha(x)a| \geq |sa| > |\alpha(xa)|$*
3. *$\alpha(xa)$ ist ein Suffix von $\alpha(x) \cdot a$.*
4. *Ist $sa = \alpha(xa)$ und $s \neq \alpha(x)$, dann ist s ein verschachteltes Suffix, sogar ein rechtsverzweigendes Suffix von x.*

Beweis: Zu 1.: $\Rightarrow$: Da $\alpha(x)$ das längste verschachtelte Suffix ist, folgt die Behauptung unmittelbar.

$\Leftarrow$: Ist s ein Suffix von x und gilt $|s| \leq |\alpha(x)|$, so ist s ein Suffix von $\alpha(x)$. Da $\alpha(x)$ ein verschachteltes Suffix von x ist, kommt es in x ein weiteres Mal als Teilwort vor und somit auch s. Also ist s ein verschachteltes Suffix. Dies ist in Abbildung 2.15 illustriert.

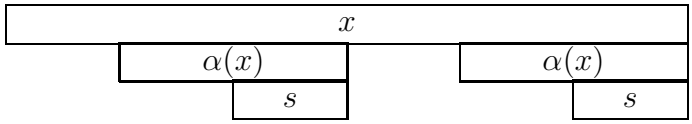


Abbildung 2.15: Skizze: Das Suffix s von x mit $|s| \leq |\alpha(x)|$ ist verschachtelt

Zu 2.: Es gilt offensichtlich mit Lemma 2.17 und mit Teil 1. für jeden Suffix s von x (und somit jeden Suffix sa von xa):

$$\begin{aligned} sa \in I^* &\Leftrightarrow s \text{ ist ein verschachteltes Suffix von } x \text{ und } sa \not\sqsubseteq x \\ &\Leftrightarrow |s| \leq |\alpha(x)| \text{ und } sa \text{ ist kein verschachteltes Suffix von } xa \\ &\Leftrightarrow |s| \leq |\alpha(x)| \wedge |sa| > |\alpha(xa)| \\ &\Leftrightarrow |\alpha(x)a| \geq |sa| > |\alpha(xa)| \end{aligned}$$

Zu 3.: Offensichtlich sind $\alpha(xa)$ und $\alpha(x)a$ Suffixe von xa. Nehmen wir an, dass $\alpha(xa)$ kein Suffix von $\alpha(x) \cdot a$ wäre. Da $\alpha(xa)$ das längste verschachtelte Suffix von xa ist, muss es nochmals als Teilwort in x auftreten. Dann kann aber nicht $\alpha(x)$ das längste verschachtelte Suffix von x sein. Diese Situation ist in Abbildung 2.16 illustriert.

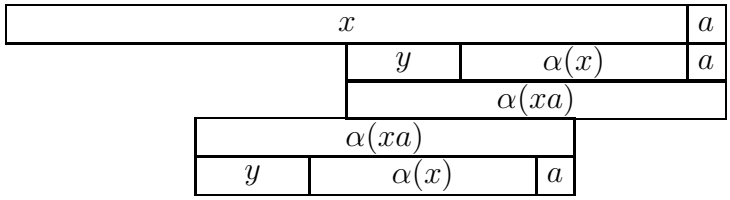


Abbildung 2.16: Skizze: Die Suffixe $\alpha(x) \cdot a$ und $\alpha(xa)$ von xa und ihre Wiederholungen in xa

Zu 4.: Sei also $\alpha(xa) = sa$ und $\alpha(x) \neq s$. Da $\alpha(xa) = sa$ ein verschachteltes Suffix von xa ist, muss auch s ein verschachteltes Suffix von x sein. Somit ist $|\alpha(x)| > |s|$ (da ja nach Voraussetzung $s \neq \alpha(x)$) und s ist ein echtes Suffix von $\alpha(x)$. Nach Definition taucht also $\alpha(x)$ (und damit auch s) ein weiteres Mal in x auf. Da sa ein längstes verschachteltes Suffix von xa ist, muss bei dem weitere Vorkommen von $\alpha(x)$ direkt dahinter ein Zeichen $b \in \Sigma$ mit $b \neq a$ auftauchen (siehe auch die

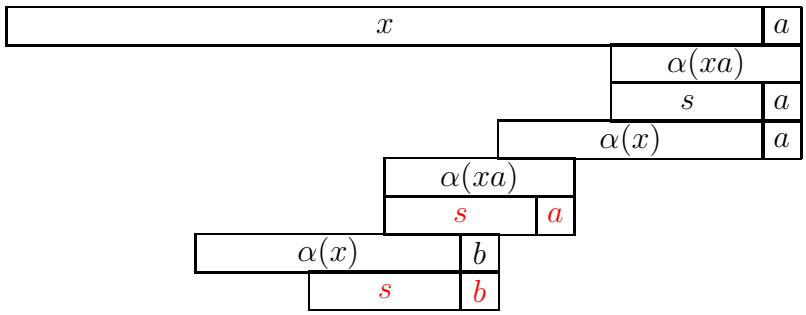


Abbildung 2.17: Skizze: Die Suffixe $\alpha(x) \cdot a$ und $\alpha(xa)$ von xa

Illustration in Abbildung 2.17). Also ist s ein rechtsverzweigendes Suffix von x, da in x sowohl sa als auch sb auftritt.

Somit ist der gesamte Beweis abgeschlossen. ■

2.4.4 Ukkonens Online Algorithmus

Die Aussage 2 des Lemmas 2.19 gibt uns eine Charakterisierung derjenigen Suffixe, die in T^i noch dargestellt werden müssen, um T^{i+1} zu erhalten. Mit Hilfe von Aussage 3 des Lemmas 2.19 wissen wir weiter, dass die noch darzustellenden Suffixe Suffixe des alten aktiven Suffixes verlängert um a sind.

Wie können wir nun alle neu darzustellenden Suffixe von xa finden? Wir beginnen mit dem Suffix $\alpha(x)$ in T^i und versuchen dort das Zeichen a anzuhängen. Anschließend durchlaufen wir alle Suffixe von $\alpha(x)$ in abnehmender Länge (was sich elegant durch die Suffix-Links erledigen lässt) und hängen jeweils ein a an. Wir enden mit dieser Prozedur, wenn für ein Suffix s von $\alpha(x)$ das Wort sa bereits im Suffix-Baum dargestellt ist, d.h. die einzufügende Kante mit Kantenmarkierung a bereits existiert (bzw. eine Kanten, deren Kantenmarkierung nach (einem Suffix von) s bereits ein a enthält). Nach dem vorhergehenden Lemma erhalten wir durch Ablaufen der Kantenmarkierung a den aktiven Suffix $\alpha(xa)$ von T^{i+1}. Diese Position wird dann als Startposition für die Erweiterung auf den Suffix-Baum T^{i+1} verwenden.

Wir müssen jedoch noch berücksichtigen, dass wir auf einen Suffix s von $\alpha(x)$ treffen, wobei $\overline{s}$ kein Knoten in T^i ist. In diesem Fall müssen wir in die Kante, an der die

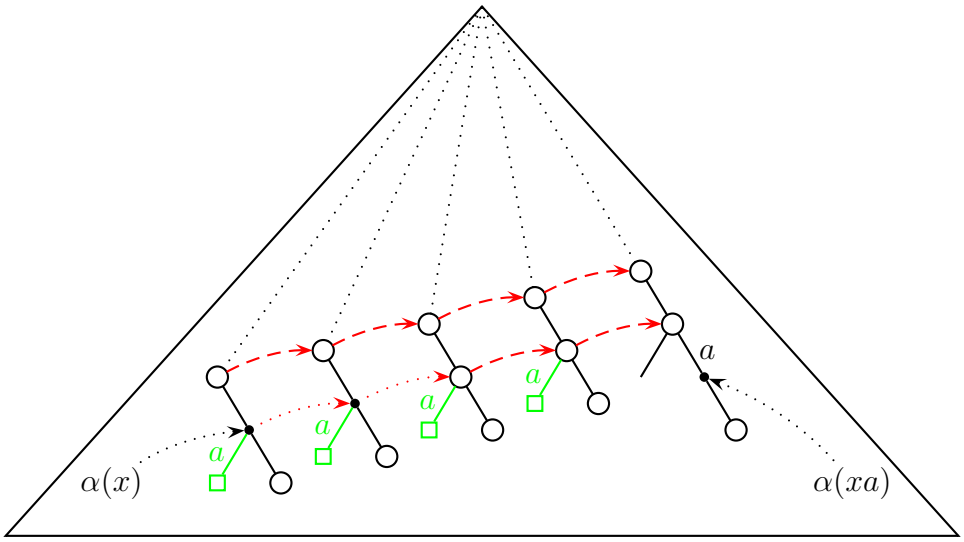


Abbildung 2.18: Skizze: Übergang von T^i zu T^{i+1} in Ukkonens Algorithmus

Darstellung von s endet, aufbrechen und an der geeigneten Stelle einen neuen Knoten einfügen.

Die Erweiterung eines Suffix-Baumes für x zu einem Suffix-Baum für xa ist in Abbildung 2.18 schematisch dargestellt. Dort sind die rot gepunkteten Linien die noch nicht vorhanden Suffix-Links, denen wir gerne folgen möchten, die aber gar nicht existieren, da deren Anfangspunkt in T^i gar nicht zu einem Knoten im Baum T^i gehören (diese werden aber im Laufe der Erweiterung hinzugefügt). Die rot gestrichelten Linien sind wirklich vorhandene Suffix-Links in T^i .

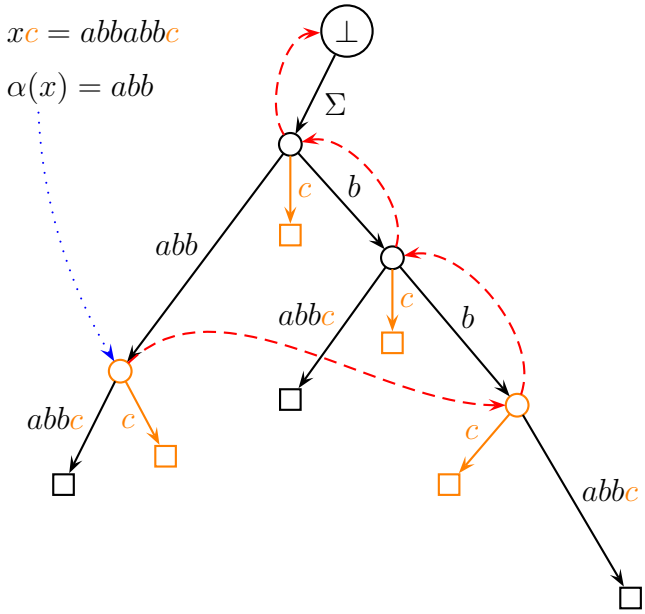


Abbildung 2.19: Beispiel: Konstruktion von $T(abbabbc)$ aus $T(abbabb)$

In Abbildung 2.19 ist ein Beispiel für eine Erweiterung eines Suffix-Baumes angegeben. Dort wird auch das mehrfache Aufbrechen von Kanten illustriert. Die Erweiterung ist orangefarben dargestellt.

In diesem Beispiel haben wir noch eine virtuelle Superwurzel $\perp$ eingeführt. Mit $\text{slink}(\overline{\varepsilon}) = \perp$ können wir sicherstellen, dass jeder innere Knoten einen Suffix-Link besitzt. Dazu nehmen wir noch eine Baum-Kante von $\perp$ zu $\overline{\varepsilon}$ an, die mit allen Zeichen aus Σ markiert ist. Damit stellen wir sicher, dass wir wieder zur Wurzel zurückkehren können, wenn wir eigentlich fälschlicherweise dem Suffix-Link von $\overline{\varepsilon}$ zu $\perp$ gefolgt sind.

Wie wir noch sehen werden, ist der einzige Grund für die Einführung von $\perp$, dass die Beschreibung von Ukkonens Algorithmus einheitlicher wird, da wir beim Folgen der Suffix-Links dann immer irgendwann auf einen Knoten stoßen werden, von dem eine Kante mit dem Zeichen $a \in \Sigma$ ausgeht.

Ukkonen (string $t = t_1 \cdots t_n$)

```
begin
    tree T := T(t_1);
    string α(t_1) := ε;
    for (i := 1; i < n; i++) do
        string x := t_1 ··· t_i;
        char a := t_{i+1};
        string s := α(x);
        while (sa is not represented in T) do
            insert sa in T;
            s := s_2 ··· s_{|s|};                          /* via suffix-links */
                                  /* Note that s_2 ··· s_{|s|} := ⊥ iff s = ε */
        α(xa) := sa ;                  /* Note that ⊥ · a := ε for any a ∈ Σ! */
end
```

Abbildung 2.20: Algorithmus: Abstrakte Fassung von Ukkonens Algorithmus

Wie finden wir in diesem Beispiel den Suffix-Link von $\alpha(x) = abb$? Im Suffix-Baum für *abbabb* gibt es ja keinen Knoten $\overline{abb}$. Wir folgen stattdessen dem Suffix-Link des Knotens, der als erstes oberhalb liegt: $\overline{\varepsilon}$. Somit landen wir mit dem Suffix-Link im Knoten $\bot$. Vom Knoten $\overline{\varepsilon}$ mussten wir ja noch die Zeichenreihe *abb* ablesen um bei der Position von $\alpha(x)$ zu landen. Dies müssen wir jetzt auch vom Knoten $\bot$ aus tun. Damit landen wir dann eigentlich beim Knoten $\overline{bb}$, der aber dummerweise im Suffix-Baum für *abbabb* auch nicht existiert (der aber jetzt eingefügt wird).

Für das weitere Folgen des Suffix-Links von $\overline{bb}$ (wobei es den Suffix-Link ja noch nicht gibt) starten wir wieder von $\overline{b}$ aus. Nach Folgen des Suffix-Links landen wir in $\overline{\varepsilon}$. Nun laufen wir noch das Wort b ab, um die eigentlich Position $\overline{b}$ im Suffix-Baum zu finden, an der wir weitermachen. Die folgenden Schritte sind nun einfach und bleiben dem Leser zur Übung überlassen.

07.11.24

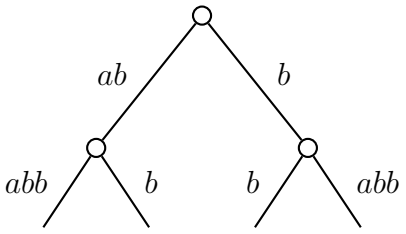
Die sich aus dieser Diskussion ergebende, erste abstrakte Fassung von Ukkonens Algorithmus ist in Abbildung 2.20 angegeben.

Um dies etwas algorithmischer beschreiben zu können, benötigen wir erst noch eine angemessene Darstellung der vom Suffix-Baum T dargestellten Wörter, die wir als Lokation bezeichnen wollen (in Zeichen $\mathrm{loc}(v)$).

Definition 2.20 *Sei $t \in \Sigma^*$ und sei $T = T(t)$ der zu t zugehörige Suffix-Baum. Für $s \in words(T)$ ist $\mathrm{loc}(s) = (\overline{u}, v) \equiv (\overline{u}, j + |u|, j + |s| - 1)$ eine* Lokation *von s in T, wenn $\overline{u} \in V(T)$, $s = uv$ für ein $v \in \Sigma^*$ und $t_j \cdots t_{j+|s|-1} = s$ gilt, wobei j minimal gewählt wird. Ist $\overline{s}$ ein Blatt in T, dann muss darüber hinaus $u \neq s$ gelten.*

Für eine Lokation $(\overline{v}, i, j)$ eines Blattes schreiben wir darüber hinaus im Folgenden $(\overline{v}, i, \infty)$, da wir für ein Blatt immer $j = |t|$ wählen können. Diese Schreibweise bezeichnet man als *offene Lokation* bzw. auch als *offene Referenz*, da der Endpunkt keine Rolle spielt. Das ist auch ein Grund, warum in Lokationen keine Blätter auftreten dürfen.

Die Definition wollen wir uns in Abbildung 2.21 noch einmal an einem Beispiel genauer anschauen.



$$
\begin{array}{lcll}
\mathrm{loc}(\varepsilon) & = & (\overline{\varepsilon}, \varepsilon) & = & (\overline{\varepsilon}, 1, 0) \\
\mathrm{loc}(ab) & = & (\overline{ab}, \varepsilon) & = & (\overline{ab}, 3, 2) \\
\mathrm{loc}(abb) & = & (\overline{ab}, b) & = & (\overline{ab}, 5, \infty) \\
\mathrm{loc}(bab) & = & (\overline{b}, ab) & = & (\overline{b}, 3, 4) \\
\mathrm{loc}(abab) & = & (\overline{ab}, ab) & = & (\overline{ab}, 3, 4) \\
\mathrm{loc}(abab) & = & (\overline{\varepsilon}, abab) & = & (\overline{\varepsilon}, 1, 4)
\end{array}
$$

Abbildung 2.21: Beispiel: Einige Lokationen im Suffix-Baum für *ababb*

Für unseren Algorithmus benötigen wir insbesondere so genannte kanonische Lokationen.

Definition 2.21 *Eine Lokation* $\mathrm{loc}(s) = (\overline{v}, w) \equiv (\overline{v}, i, j)$ *heißt* kanonisch*, wenn für jede andere Lokation* $\mathrm{loc}(s) = (\overline{v'}, w') \equiv (\overline{v'}, i', j')$ *gilt, dass* $|v| > |v'|$.

Für die in Abbildung 2.21 angegebenen Lokationen sind die ersten fünf kanonische, die sechste jedoch nicht.

Für eine kanonische Lokation gilt also das Folgende:

- Wenn $\overline{s}$ ein innerer Knoten (Verzweigungsknoten) in T ist, dann gilt
$$\mathrm{loc}(s) := (\overline{s}, \varepsilon) \equiv (\overline{s}, j + |s|, j + |s| - 1),$$
wobei $t_j \cdots t_{j+|s|-1} = s$ und j minimal gewählt ist.
- Wenn $\overline{s}$ ein Blatt ist, d.h. $\overline{u} \xrightarrow{v} \overline{s}$ und $s = uv$, dann gilt
$$\mathrm{loc}(s) := (\overline{u}, v) \equiv (\overline{u}, j + |u|, \infty),$$
wobei $t_j \cdots t_{j+|uv|-1} = uv = s$ und j minimal gewählt ist.
- Wenn kein Knoten $\overline{s}$ in T existiert, dann existiert eine Kante $\overline{u} \xrightarrow{vw} \overline{uvw}$ mit $s = uv$, $v \neq \varepsilon$, $w \neq \varepsilon$ und es gilt
$$\mathrm{loc}(s) := (\overline{u}, v) \equiv (\overline{u}, j + |u|, j + |s| - 1),$$
wobei $t_j \cdots t_{j+|uv|-1} = uv = s$ und j minimal gewählt ist.

Ukkonen (string $t = t_1 \cdots t_n$)

```
begin
    tree T := T(t_1);
    ref (v̄, w) := (r(T), ε);                          /* r(T) is the root of T */
    for (i := 1; i < n; i++) do
        node x := y := NIL;
        while (not T.lookup((v̄, w), t_{i+1})) do   /* i.e., while((v̄, w·t_{i+1}) ∉ T) */
            y := T.insert((v̄, w), t_{i+1}) ;        /* returns parent of new leaf */
            if (x ≠ NIL) then
                slink(x) := y;
            x := y;
            (v̄, w) := canonize((slink(v̄), w));
        if (x ≠ NIL) then
            slink(x) := v̄;                  /* Note that always w = ε if x ≠ NIL */
        (v̄, w) := canonize((v̄, w · t_{i+1}));
end
```

Abbildung 2.22: Algorithmus: Ukkonens Algorithmus

Aus unserer Diskussion ergibt sich nun Ukkonens Algorithmus, der im Pseudo-Code in Abbildung 2.22 angegeben ist. In der Regel wird dabei in einer realen Implementierung allerdings w dabei als Referenz (j, k) angegeben, d.h. $w = t_j \cdots t_k$. Wir gehen darauf im Folgenden allerdings nicht immer im Detail ein, siehe aber auch Abbildung 2.23. Bei dieser Realisierung wird die Referenz (j, k) allerdings immer auf einen Suffix von $t_1 \cdots t_i$ verweisen, d.h. es gilt $k = i$.

Hierbei schaut die Prozedur lookup$((\overline{v}, w), a)$ nach, ob im Baum T das Wort vwa dargestellt wird, d.h. ob man ab der kanonischen Lokation $(\overline{v}, w)$ in T den Buchstaben a weiterverfolgen kann. Ist $w = \varepsilon$, dann wird nur überprüft, ob $\overline{v}$ eine ausgehende Kante hat deren Kantenmarkierung mit a beginnt. Ist $w \neq \varepsilon$, dann wird die ausgehende Kante von $\overline{v}$ betrachtet, deren Kantenmarkierung mit dem ersten Buchstaben von w beginnt. In dieser Kantenmarkierung wird jetzt nur der Buchstabe an Position $|w| + 1$ mit a verglichen. Dabei liefert lookup `false`, wenn $(\overline{v}, w \cdot a)$ keine Lokation in T ist, andernfalls wird `true` zurückgeliefert. Der Leser möge sich selbst davon überzeugen, dass in dem im Algorithmus benötigten Fall (nach dem Ende der while-Schleife) der Knoten $\overline{vw}$ in T wirklich existieren muss.

Man beachte, dass die Buchstaben an den Positionen 2 mit $|w|$ nicht mit der entsprechenden Kantenmarkierung verglichen werden müssen, da diese nach Konstruktion übereinstimmen müssen. Dies folgt daraus, dass das zur Lokation $(\overline{v}, w)$ gehörige Wort vw nach Konstruktion von Ukkonens Algorithmus in T zwingend dargestellt sein muss.

Für jeden Aufruf lookup$((\overline{v}, w), t_{i+1})$ (bzw. lookup$((\overline{v}, j, k), t_{i+1})$) bedeutet dies, dass man für $w = \varepsilon$ (also $j > k$) eine ausgehende Kanten von $\overline{v}$ finden muss, deren Kantenmarkierung mit t_{i+1} beginnt. Andernfalls muss man zuerst eine ausgehende Kanten von $\overline{v}$ finden, deren Kantenmarkierung (p, q) mit t_j beginnt, d.h. ob $t_j = t_p$; falls ja, muss noch die Bedingung $t_{i+1} = t_{p+k-j+1}$ geprüft werden. Falls einer der Fälle eintritt, liefert lookup den Wert `true`, sonst `false`.

Die Prozedur canonize macht aus einer gegebenen Lokation $(\overline{v}, w)$ eine kanonische, indem sie vom Knoten $\overline{v}$ soweit im Suffix-Baum die Zeichenreihe w verfolgt, bis die kanonische Lokation gefunden wird. Man beachte da, dass der Aufwand proportional zur Anzahl überlaufener Kanten plus 1 ist, da jeweils nur das erste Zeichen einer ausgehenden Kante betrachtet werden muss (die folgenden Zeichen dieser Kantenmarkierung müssen dann, wie schon diskutiert, auch wieder alle übereinstimmen).

Wie man in Abbildung 2.19 sehen kann, kann es auch wirklich passieren, dass man bei canonize mehrere Kanten überlaufen muss. Dort ist die kanonische Lokation des Suffixes $\alpha(x) = abb$ gerade $(\overline{\varepsilon}, abb)$. Folgt man dem zugehörigen Suffix-Link, so gelangt man zur Lokation $(\bot, abb)$. Um die zugehörige kanonische Lokation $(\overline{b}, b)$ zu erhalten, muss man zwei Knoten für das Wort abb im Baum hinabsteigen.

Innerhalb von canonize muss für eine Lokation $(\overline{v}, w)$ (also $(\overline{v}, j, k)$) zuerst die von $\overline{v}$ ausgehende Kante mit Kantenmarkierung (p, q) zu $\overline{vw'}$ bestimmt werden, die mit t_j beginnt, d.h. $t_j = t_p$. Ist $|w'| > |w|$ (also $q - p > k - j$) oder ist $|w'| = |w|$ (also $q - p = k - j$) und $\overline{vw}$ ein Blatt, dann ist die Lokation kanonisch. Ansonsten wird in Canonize versucht die Lokation $(\overline{vw'}, w'')$ mit $w = w'w''$ (also $(\overline{vw'}, j+(q-p+1), k)$) zu kanonisieren.

Die Prozedur insert$((\overline{v}, w), a)$ fügt an der Lokation $(\overline{v}, w)$ eine Kante mit Kantenmarkierung a zu einem neuen Blatt ein. War $(\overline{v}, w)$ bereits ein Knoten im Suffix-Baum, so wird die neue Kante dort nur angehängt. Beschreibt die Lokation $(\overline{v}, w)$ eine Position innerhalb einer Kante, so wird in diese Kante an der zur Lokation $(\overline{v}, w)$ entsprechenden Stelle ein neuer Knoten eingefügt, an den die Kante zu dem neuen Blatt angehängt wird. Man überlegt sich leicht, dass dies in konstanter Zeit erledigt werden kann. Der zu dem neuen Blatt adjazente innere Knoten kann dann leicht wie gefordert zurückgeliefert werden.

Dabei muss auch noch ein Suffix-Link vom vorhergehenden Knoten auf diesen neu eingefügten Knoten gesetzt werden. Mit einer geeigneten Buchhaltung kann auch dies in konstanter Zeit erledigt werden, da der Knoten, der den neuen Suffix-Link auf den neu eingefügten bzw. aktuell betrachteten Knoten erhält, gerade eben vorher betrachtet wurde.

Für jeden Aufruf insert$((\overline{v}, w), t_i)$ (also insert$((\overline{v}, j, k), t_i)$) wird im Falle $w = \varepsilon$ (also $j > k$) nur ein Blatt mit Kantenmarkierung $(i + 1, \infty)$ an den Knoten $\overline{v}$ gehängt.

```
Ukkonen (string t = t_1 ··· t_n)
begin
    tree T := T(t_1);
    ref (v̄, j, k) := (r(T), 2, 1);                    /* Note that always k = i */
    for (i := 1; i < n; i++) do
        node x := y := NIL;
        while (not T.lookup((v̄, j, k), t_{i+1})) do      /* while((v̄, j, k+1) ∉ T) */
            y := T.insert((v̄, j, k), t_{i+1}) ;      /* returns parent of new leaf */
            if (x ≠ NIL) then
                slink(x) := y;
            x := y;
            (v̄, j, k) := canonize((slink(v̄), j, k));
        if (x ≠ NIL) then
            slink(x) := v̄;                /* Note that always j > k if x ≠ NIL */
        (v̄, j, k) := canonize(v̄, j, k + 1);
end
```

Abbildung 2.23: Algorithmus: Ukkonens Algorithmus mit Referenzen

Andernfalls muss die Kante mit Kantenmarkierung (p, q) vom Knoten $\overline{v}$ zum Knoten z mit $t_p = t_j$ aufgebrochen werden. Es wird ein neuer Knoten y eingefügt, wobei y den Knoten z als das Kind von $\overline{v}$ ersetzt und die Kantemarkierung $(p, p + k - j)$ erhält. z wird dann zu einem Kind von y mit Kantenmarkierung $(p + k - j + 1, q)$. Weiter bekommt y ein neues Blatt als Kind mit Kantenmarkierung $(i + 1, \infty)$.

Abschließend geben wir noch die Fassung von Ukkonens Algorithmus mit Referenzen in Abbildung 2.23 an.

2.4.5 Zeitanalyse

Es bleibt noch die Analyse der Zeitkomplexität von Ukkonens Algorithmus. Das Einfügen eines neuen Knotens kann, wie bereits diskutiert, in konstanter Zeit geschehen. Da ein Suffix-Baum für t maximal $O(|t|)$ Knoten besitzt, ist der Gesamtzeitbedarf hierfür $O(|t|)$.

Der Hauptaufwand liegt in der Kanonisierung der Lokationen. Für einen Knoten kann eine Kanonisierung nämlich mehr als konstante Zeit kosten. Bei der Kanonisierung wird jedoch die Lokation $(\overline{v}, w)$ zu $(\overline{vv'}, w')$ für $v' \in \Sigma^+$ und $w' \in \Sigma^*$ mit $v'w' = w$. Damit wandert das Ende der Zeichenreihe v des in der Lokation verwendeten internen Knotens bei einer Kanonisierung immer weiter zum Wortende von t hin. Zwar wird $\overline{v}$ beim Ablaufen der Suffix-Links verkürzt, aber dies geschieht nur

am vorderen Ende. Somit kann eine Verlängerung am Wortende von v bei der verwendeten Lokation $(\overline{v}, w)$ maximal $|t|$ Mal auftreten. Also haben alle Aufrufe zur Kanonisierung eine lineare Laufzeit.

Theorem 2.22 *Ein Suffix-Baum für ein Wort $t = t_1 \cdots t_n \in \Sigma^n$ kann mit Ukkonens Algorithmus in Zeit $O(n)$ mit Platzbedarf $O(n)$ konstruiert werden.*

Wir erwähnen hier noch, dass der Algorithmus folgende Eigenschaft einer Darstellung eines Suffix-Baumes benötigt: Man muss von einem Knoten das Kind finden können, das sich über diejenige Kanten erreichen lässt, dessen erstes Zeichen der Kantenmarkierung vorgegeben ist. Verwendet man also für die Repräsentation aus Platzgründen keine Felder, so erhöht sich der Aufwand in der Regel um den Faktor $|\Sigma|$, was bei kleinen Alphabeten noch tolerierbar ist.

Des Weiteren wollen wir noch anmerken, dass in manchen Lehrbüchern eine Laufzeit für Ukkonens Algorithmus von $O(n \log(n))$ angegeben wird. Dabei wird jedoch statt des uniformen Kostenmaßes das logarithmische Kostenmaß verwendet (also die Bit-Komplexität). In diesem Fall hat jedoch der Suffix-Baum selbst schon die Größe $O(n \log(n))$, da dort ja Zahlen (Referenzen und Zeiger) aus dem Intervall $[0 : n]$ vorkommen, deren Darstellung ja $\Theta(\log(n))$ Bits benötigt. Daher handelt es sich auch in diesem Modell um einen asymptotisch optimalen Algorithmus zur Erstellung von Suffix-Bäumen, da die Ausgabegröße bereits $O(n \log(n))$ betragen kann. Man beachte jedoch, dass die Eingabegröße nur $\Theta(n \log(|\Sigma|))$ beträgt. Im logarithmischen Kostenmaß ist also der Suffix-Baum im Allgemeinen größer als die Eingabe!

12.11.24

Repeats 3

3.1 Exakte und maximale Repeats

In diesem Kapitel wollen wir uns mit Wiederholungen (so genannten) Repeats in Zeichenreihen (insbesondere in Genomen) beschäftigen. Dazu müssen wir uns vor allem überlegen, wie man interessante Wiederholungen charakterisiert.

3.1.1 Erkennung exakter Repeats

Zunächst einmal müssen wir formal definieren, was wir unter einer Wiederholung verstehen wollen. Dazu benötigen wir erst noch die Begriffe einer Hamming- und Alignment-Distanz.

Definition 3.1 *Seien $s, t \in \Sigma^n$, dann ist die* Hamming-Distanz *von s und t definiert als $\delta_H(s,t) := |\{i \in [1:n] : s_i \neq t_i\}|$.*

Definition 3.2 *Seien $s, t \in \Sigma^*$, dann ist die* Alignment-Distanz *von s und t definiert als $\delta_A(s,t) := \min\left\{d(\hat{s},\hat{t}) : (\hat{s},\hat{t}) \in \mathcal{A}(s,t)\right\}$ wobei $\mathcal{A}(s,t)$ die Menge der Alignments von s und t und $d(\hat{s},\hat{t})$ ein Distanzmaß für ein Alignment ist.*

Als Alignment-Distanz kann beispielsweise die Anzahl von Substitutionen, Insertionen und Deletionen im Alignment $(\hat{s},\hat{t})$ (also z.B. die EDIT-Distanz) verwendet werden. Nun können wir definieren, was eine Wiederholung sein soll.

Definition 3.3 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ eine Zeichenreihe. Ein Paar $((i_1,j_1),(i_2,j_2))$ mit $(i_1,j_1) \neq (i_2,j_2)$ heißt*

- exaktes Paar*, wenn $t_{i_1} \cdots t_{j_1} = t_{i_2} \cdots t_{j_2}$. Das Wort $t_{i_1} \cdots t_{j_1}$ wird dann auch als* exaktes Repeat *bezeichnet. Mit $\mathcal{R}(t)$ bezeichnen wir die Menge aller exakten Paare und mit $\overline{\mathcal{R}}(t)$ die Menge aller exakten Repeats.*
- k-mismatch Repeat*, wenn $\delta_H(t_{i_1} \cdots t_{j_1}, t_{i_2} \cdots t_{j_2}) \leq k$. Mit $\mathcal{R}_k^H(t)$ bezeichnen wir die Menge aller k-mismatch Repeats.*
- k-difference Repeat*, wenn $\delta_A(t_{i_1} \cdots t_{j_1}, t_{i_2} \cdots t_{j_2}) \leq k$. Mit $\mathcal{R}_k^A(t)$ bezeichnen wir die Menge aller k-difference Repeats.*

In Abbildung 3.1 sind schematisch zwei solcher Repeats dargestellt. Man beachte dabei, dass nach Definition ein Repeat sich mit sich selbst überlappen kann (wie bei den roten Teilwörtern) oder es auch mehr als zwei Vorkommen eines Repeats geben kann.

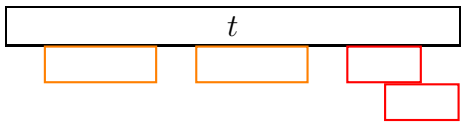


Abbildung 3.1: Skizze: Schematische Darstellung von Repeats

Wir behaupten nun, dass wir alle solchen exakten Repeats in „linearer Zeit“ finden können. Dies folgt aus der Tatsache, dass alle exakten Repeats von t zu Lokationen eines Suffix-Baums $T(t\$)$ korrespondieren, die sich nicht auf einer zu einem Blatt inzidenten Kante befinden. Sobald man sich im Suffix-Baum nicht auf einer zu einem Blatt inzidenten Kanten befindet, gibt es Verlängerungsmöglichkeiten, so dass man in mindestens zwei *verschiedenen* Blättern landen kann. Also ist das betrachtete Wort jeweils ein Präfix von mindestens zwei verschiedenen Suffixen, d.h. das betrachtete Wort beginnt an mindestens zwei verschiedenen Positionen in t.

In Abbildung 3.2 ist für das Wort *abbbab* der Suffix-Baum für *abbbab*\$ angegeben und einige darin enthaltene exakte Repeats durch Angabe von exakten Paaren.

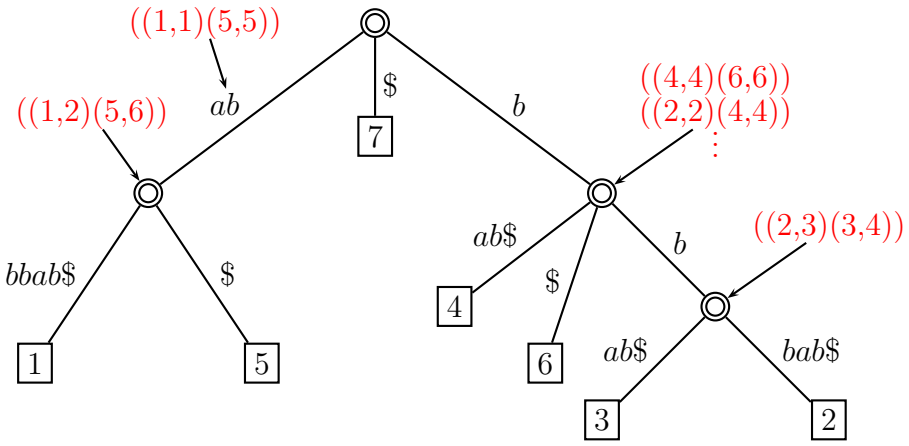


Abbildung 3.2: Beispiel: Suffix-Baum für $t\$ = abbbab\$$ und die exakten Repeats

Durchlaufen wir nun den Suffix-Baum $T(t\$)$ gekürzt um alle Blätter und die dazu inzidenten Kanten mit einer Tiefensuche, so können wir alle dort zu den noch enthaltenen Lokationen korrespondierenden Zeichenreihen ausgeben, und geben damit alle exakten Repeats aus.

Theorem 3.4 *Sei $t \in \Sigma^n$, dann lassen sich alle exakten Repeats als Wörter bzw. Referenzen in Zeit $O(n+k)$ ermitteln, wobei k die Anzahl aller Zeichen in den exakten Repeats bzw. die Anzahl der exakten Repeats ist.*

Somit ist hier mit linearer Zeit gemeint, dass der Algorithmus linear in der Eingabe- und Ausgabegröße läuft. Man beachte, dass es durchaus Wörter über Σ der Länge n geben kann, die $\Theta(n^2)$ exakte Repeats besitzen. Wir merken noch an, dass dieser Algorithmus optimal ist.

Somit haben wir aber nur die exakten Repeats ausgegeben, wir wissen jedoch nicht wo diese auftreten. Daher ist in der Regel ein Algorithmus zum Auffinden aller exakten Paare interessanter. Auch diesen können wir analog zu denen der exakten Repeats definieren. Der Algorithmus zum Auffinden aller exakten Paare geht dabei wie folgt vor:

1. Konstruktion von $T(t\$)$ in Zeit $O(n)$.
2. Tiefensuche durch $T(t\$)$ in Zeit $O(n)$. Während der Tiefensuche führe folgende Schritte aus:
   (a) Jedes Blatt liefert die Indexposition zurück, an der das aufgefundene Suffix beginnt. Der Zeitbedarf hierfür ist insgesamt $O(n)$.
   (b) Jeder innere Knoten liefert eine Liste der Blätter, die von diesem Knoten erreichbar sind, an seinen Elter zurück. Der Zeitbedarf hierfür ist insgesamt $O(n)$, da zum einen immer nur ein Zeiger auf den Beginn und das Ende der Liste übergeben wird und zum anderen, da das Zusammenhängen der Listen der Kinder insgesamt in Zeit $O(n)$ zu realisieren ist (wenn man sich auch jeweils das Ende der Liste merkt).
   (c) Für einen inneren Knoten $\overline{v}$ mit der Blattliste L und mit Elter $\overline{w}$ generiere die exakten Paare $((i, i+\ell-1), (j, j+\ell-1))$ mit $i < j \in L$ und $\ell \in [|w|+1 : |v|]$. Dies lässt sich insgesamt in Zeit $O(n+|\text{output}|)$ erledigen. Nach der vorherigen Diskussion beschreibt jedes ausgegebene exakte Paar einen exakten Repeat. Man überlegt sich leicht, dass alle ausgegeben exakten Paare paarweise verschieden sind.

Theorem 3.5 *Sei $t \in \Sigma^n$, dann lassen sich alle exakten Paare in Zeit $O(n+k)$ ermitteln, wobei k die Anzahl der exakten Paare ist.*

Wir merken noch an, dass dieser Algorithmus optimal ist.

Betrachten wir das Wort $a^n \in \Sigma^*$, dann enthält es nach unserer Definition $\Theta(n^3)$ exakte Paare. Es gilt nämlich, dass für alle $\ell \in [1 : n-1]$ und $i_1 < i_2 \in [1 : n-\ell+1]$

das Paar $((i_1, i_1+\ell-1),(i_2, i_2+\ell-1))$ ein exaktes Paar ist. Andererseits existieren nur $n-1$ exakte Repeats a^i in a^n mit $i \in [1:n-1]$.

Somit ist auch hier mit linearer Zeit gemeint, dass der Algorithmus linear in der Eingabe- und Ausgabegröße läuft.

Man kann diesen Algorithmus auch leicht so modifizieren, dass er nur exakte Paare ausgibt, die einen Repeat mit einer Mindestlänge ausgibt. Man muss dann nur innere Knoten berücksichtigen, deren Worttiefe hinreichend groß ist.

3.1.2 Charakterisierung maximaler Repeats

Wie wir gesehen haben, gibt es für die Bestimmung exakter Repeats bzw. Paare einen optimalen Algorithmus. Dennoch muss dieser nicht sehr effizient sein, insbesondere dann nicht, wenn viele exakte Repeats bzw. Paare vorkommen. Daher werden wir die Problemstellung überarbeiten und versuchen uns auf die interessanten Repeats zu beschränken.

Definition 3.6 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei $t' = t'_0 \cdots t'_{n+1} = ¢t_1 \cdots t_n\$$ mit $¢ \neq \$ \notin \Sigma$. Ein Tripel (i,j,ℓ) mit $\ell \in [1:n]$ sowie $i < j \in [1 : n-\ell+1]$ heißt* maximales Paar*, wenn $t_i \cdots t_{i+\ell-1} = t_j \cdots t_{j+\ell-1}$ und $t'_{i-1} \neq t'_{j-1}$ sowie $t'_{i+\ell} \neq t'_{j+\ell}$ gilt. Ist (i,j,ℓ) ein maximales Paar, dann ist $t_i \cdots t_{i+\ell-1}$ ein* maximaler Repeat*.*

Mit $\mathcal{R}_{\max}(t)$ bezeichnet man die Menge aller maximalen Paare *von t und mit $\overline{\mathcal{R}}_{\max}(t) = \{t_i \cdots t_{i+\ell-1} : (i,j,\ell) \in \mathcal{R}_{\max}(t)\}$ die* Menge aller maximaler Repeats *von t.*

Anschaulich bedeutet dies, dass ein Repeat maximal ist, wenn jede Verlängerung dieses Repeats um ein Zeichen kein Repeat mehr ist. Es kann jedoch Verlängerungen geben, die wiederum ein maximales Repeat sind. Für $t = a\boxed{aa}bb\boxed{aa}ab$ ist aa mit dem maximalen Paar $(2,6,2)$ ein maximales Repeat, aber auch $aaab$ ist ein maximales Repeat. Des Weiteren ist für aa das Paar $(2,7,2)$ nicht maximal, aber das zugehörige Repeat ist maximal (s.o., wegen $(2,6,2)$).

Lemma 3.7 *Sei $t \in \Sigma^*$ und sei $T = T(¢t\$)$ der Suffix-Baum für $¢t\$$ mit $¢, \$ \notin \Sigma$. Wenn $\alpha \in \overline{\mathcal{R}}_{\max}(t)$, dann existiert ein innerer Knoten v in T mit $path(v) = \alpha$.*

Beweis: Betrachten wir das zu α gehörige maximale Paar (i,j,ℓ). Nach Definition des maximalen Paares gilt insbesondere $t'_{i+\ell} \neq t'_{j+\ell}$. Somit muss also α ein rechtsverzweigendes Teilwort von t' und daher muss $\overline{\alpha}$ ein innerer Knoten in T sein. ■

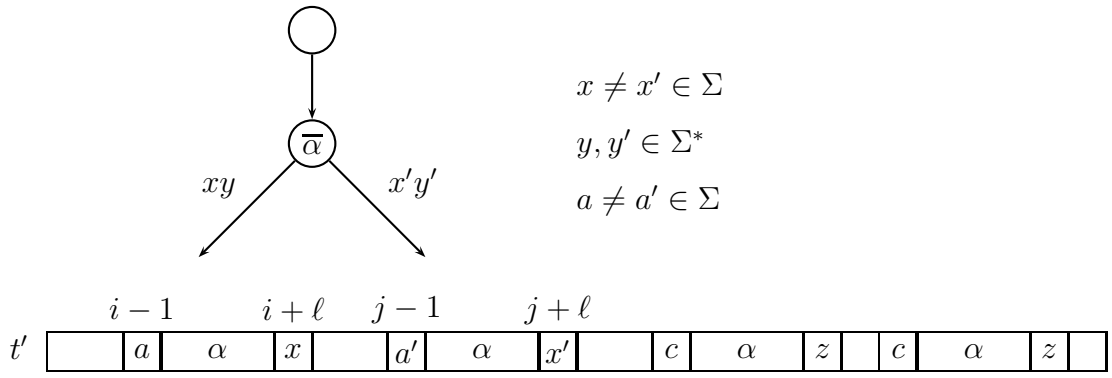


Abbildung 3.3: Skizze: Maximale Repeats enden an inneren Knoten

Die Beweisidee des Lemmas ist noch einmal in Abbildung 3.3 illustriert. Man sollte sich auch klar machen, dass es für ein maximales Repeat α zwei Vorkommen in t existieren müssen, an denen sich die angrenzenden Zeichen unterscheiden (d.h. $t'_{i-1} \neq t'_{j-1}$ und $t'_{i+\ell} \neq t'_{j+\ell}$, wenn das maximale Paar (i, j, ℓ) das maximale Repeat α beschreibt). Es kann durchaus zwei Vorkommen von α in t geben, so dass sich die Wörter nach vorne oder hinten zu längeren Repeats verlängern lassen.

Wir erhalten damit unmittelbar noch das folgende Korollar.

Korollar 3.8 *Ein Wort $t \in \Sigma^*$ besitzt höchstens $|t|$ viele maximale Repeats.*

Beweis: Jedem maximalen Repeat entspricht nach dem vorhergehenden Lemma ein innerer Knoten und in einem Suffix-Baum für ¢t\$ mit $|t| + 2$ Blättern kann es maximal $|t| + 1$ viele innere Knoten geben. In der obigen Formel kann nur dann das Maximum angenommen werden, wenn jeder innere Knoten genau zwei Kinder hat. Da die Wurzel jedoch mindestens drei Kinder hat (über die Kanten, die mit ¢, \$ und einem $a \in \Sigma$ beginnen), kann es maximal $|t|$ innere Knoten geben. ■

Wir merken noch an, dass die Anzahl der maximalen Paare keineswegs linear in der Länge des Textes t begrenzt sein muss. Dies sei dem Leser zur Übung überlassen.

Wie man im Beweis von Lemma 3.7 bemerkt, nutzen wir hier nur aus, dass nach einem maximalen Repeat eines maximalen Paares (i, j, ℓ) die Zeichen unterschiedlich sind (d.h. $t'_{i+\ell} \neq t'_{j+\ell}$), aber nicht die Zeichen unmittelbar davor (d.h. $t'_{i-1} \neq t'_{j-1}$). Um diese bei der Bestimmung maximaler Repeats auch noch berücksichtigen zu können, benötigen wir noch die folgende Definition.

Definition 3.9 *Sei $t \in \Sigma^n$ und sei $t' = t'_0 \cdots t'_{n+1} = ¢t\$$.*

- *Das* Linkszeichen *von Position $i \in [1:n]$ ist definiert als t'_{i-1}.*
- *Das* Linkszeichen *eines Blattes $\overline{s} \neq \overline{¢t\$}$ von $T(¢t\$)$ ist das Zeichen $t'_{n-|s|+1}$.*
- *Ein innerer Knoten von $T(¢t\$)$ heißt* linksdivers, *wenn in seinem Teilbaum zwei Blätter mit einem verschiedenen Linkszeichen existieren.*

In Abbildung 3.4 ist noch einmal die Definition der Linkszeichen von Blättern illustriert.

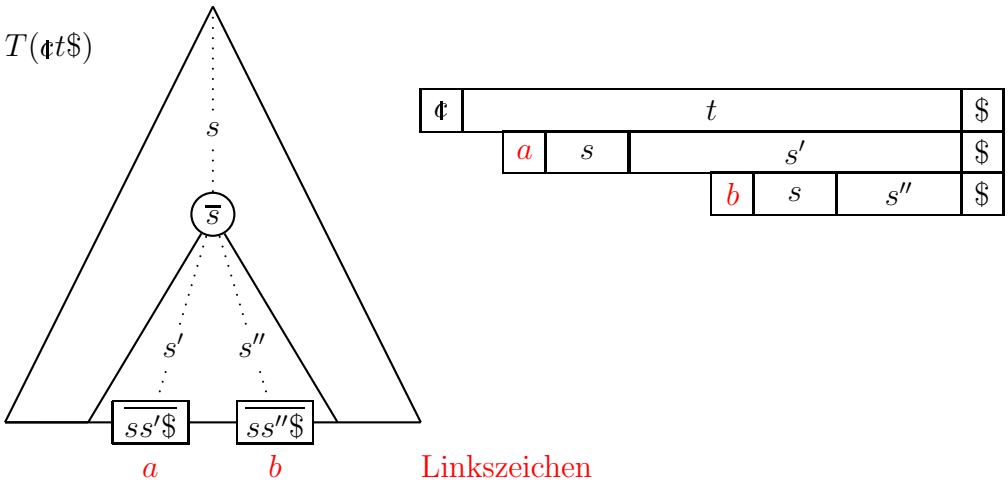


Abbildung 3.4: Skizze: Definition von Linkszeichen

In der Definition ist für einen linksdiversen Knoten v nur gefordert, dass im Teilbaum von v zwei verschiedenen Linkszeichen auftreten müssen. Man kann sich aber leicht überlegen, dass es dann sogar zwei Blätter in den Teilbäumen von zwei verschiedenen Kindern von v mit unterschiedlichem Linkszeichen geben muss.

Theorem 3.10 *Sei $t \in \Sigma^*$, $t' = ¢t\$$ und $T = T(t')$ der Suffix-Baum für t'. Die Zeichenreihe $s \in \Sigma^*$ ist genau dann ein maximaler Repeat von t, wenn der Knoten $\overline{s}$ in T existiert und linksdivers ist.*

Beweis: $\Leftarrow$: Da $\overline{s}$ linksdivers ist, muss es zwei Blätter $\overline{v}$ und $\overline{w}$ im Teilbaum von $\overline{s}$ geben, die unterschiedliche Linkszeichen besitzen.

Wir halten zunächst fest, dass dann $\overline{s}$ zwei verschiedene Kinder besitzen muss, in deren Teilbäumen die Blätter mit den unterschiedlichen Linkszeichen auftreten müssen. Dieser Fall ist in Abbildung 3.5 illustriert.

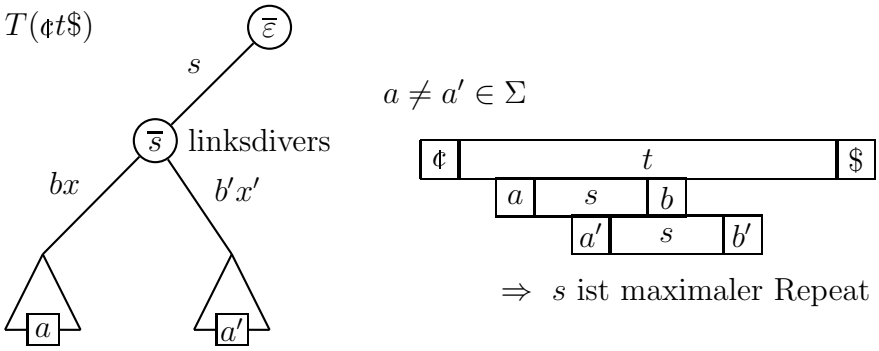


Abbildung 3.5: Skizze: $\overline{s}$ ist linksdivers

Seien bx und $b'x'$ mit $b \neq b' \in \Sigma$ und $x, x' \in \Sigma^*$ die beiden Kantemarkierungen zu den zwei Kindern, die die verschiedenen Linkszeichen im Unterbaum besitzen. Weiter seien $a \neq a' \in \Sigma$ die beiden verschiedenen Linkszeichen, wobei a das Linkszeichen des Blattes ist, der im über die Kante bx erreichbaren Teilbaum liegt. Dann muss sowohl asb als auch $a'sb'$ ein Teilwort von $t' = ¢t\$$ sein. Somit gibt es ein maximales Paar, das genau diese Vorkommen in t' beschreibt und somit ist s ein maximaler Repeat.

$\Rightarrow$: Sei nun s ein maximaler Repeat in t, dann muss es nach Definition ein maximales Paar (i, j, ℓ) geben mit $t_i \cdots t_{i+\ell-1} = t_j \cdots t_{j+\ell-1} = s$, $t'_{i-1} \neq t'_{j-1}$ und $t'_{i+\ell} \neq t'_{j+\ell}$. Somit ist s ein rechtsverzweigendes Teilwort von $¢t\$$ und $\overline{s}$ muss dann ein interner Knoten sein. Da ja $t'_{i-1} \neq t'_{j-1}$ ist, ist das Linkszeichen vom Blatt $\overline{t'_i \cdots t'_{n+1}}$ ungleich dem Linkszeichen von Blatt $\overline{t'_j \cdots t'_{n+1}}$. Da weiter $t'_{i+\ell} \neq t'_{j+\ell}$ ist, tauchen die beiden unterschiedlichen Linkszeichen in zwei verschiedenen Teilbäumen der Kinder von $\overline{s}$ auf und $\overline{s}$ ist somit linksdivers. Dies ist auch in Abbildung 3.6 illustriert. ■

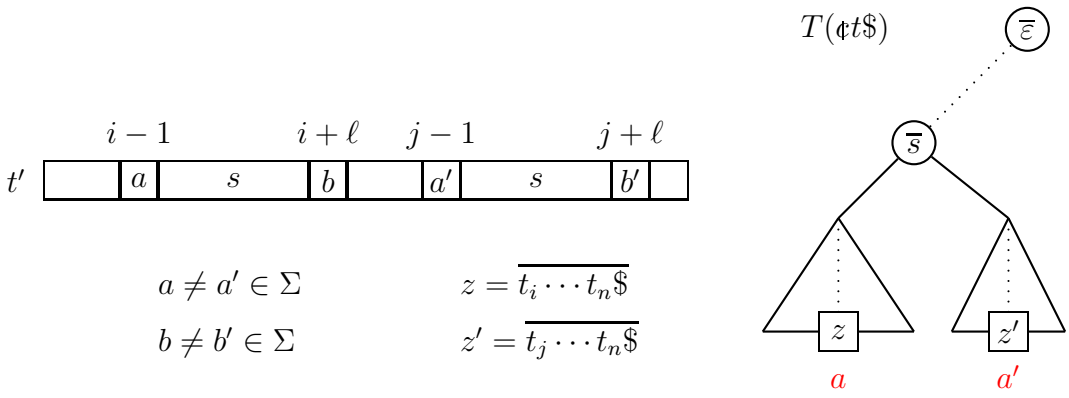


Abbildung 3.6: Skizze: Ein maximales Repeat, das durch sein zugehöriges maximales Paar einen internen Knoten im zugehörigen Suffix-Baum induziert

Damit erhalten auch eine kompakte Darstellung von maximalen Repeats in t. Wir müssen nur den Suffix-Baum für ¢t\$ konstruieren und dort alle nicht linksdiversen Knoten (inklusive der Blätter) und die dazu inzidenten Kanten entfernen. Man sieht leicht, dass der restlichen Baum zusammenhängend ist und nach dem vorherigen Satz alle maximalen Repeats charakterisiert.

3.1.3 Erkennung maximaler Repeats

Wir müssen nun nur noch einen Algorithmus zur Erkennung linksdiverser Knoten entwickeln.

Wir werden das Problem auch hier wieder mit einer Tiefensuche durch den Suffix-Baum $T = T(¢t\$)$ erledigen. Dazu bestimmen wir an den Blättern das jeweilige Linkszeichen. Hierfür müssen wir für ein Blatt nur wissen, ab welcher Position i das zugehörige Suffix beginnt, dann ist t'_{i-1} das zugehörige Linkszeichen.

An den inneren Knoten bekommen wir von den Kindern entweder die Information zurück, ob das Kind linksdivers ist, oder das Zeichen das an allen Blättern des Teilbaum des Kindes als Linkszeichen notiert ist. Liefern alle Kinder dasselbe Zeichen zurück, so ist der Knoten nicht linksdivers und wir geben dieses Zeichen an seinen Elter zurück. Sind die von den Kindern zurückgelieferten Zeichen unterschiedlich oder ist eines der Kinder linksdivers, so geben wir an den Elter die Information linksdivers zurück.

Die Laufzeit an jedem Blatt ist konstant und an den inneren Knoten proportional zur Anzahl der Kinder des Knotens. Da die Summe der Anzahl der Kinder aller Knoten eines Baumes mit n Knoten gerade $n-1$ ist, ist die Gesamtlaufzeit zur Bestimmung linksdiverser Knoten linear in der Größe des Suffix-Baumes T und somit $O(|t|)$. Wie im vorherigen Abschnitt lassen sich somit die maximalen Repeats sehr leicht ausgeben.

Theorem 3.11 *Sei $t \in \Sigma^n$, dann lassen sich alle maximalen Repeats als Wörter bzw. Referenzen in Zeit $O(n + k)$ ermitteln, wobei k die Anzahl aller Zeichen in den maximalen Repeats bzw. die Anzahl der maximalen Repeats ist.*

Jetzt müssen wir nur alle maximalen Paare (Position und Länge des Repeats in der Zeichenreihe) generieren. Hierfür konstruieren wir für jeden Knoten $\overline{v}$ ein Feld $L_{\overline{v}}[\cdot]$ von Listen, wobei das Feld über die Zeichen des Alphabets Σ indiziert ist und die Listenelemente Positionen innerhalb von t (also Werte aus $[1 : |t|]$) sind. Intuitiv bedeutet dies für einen Knoten $\overline{v}$, dass in seiner Liste für das Zeichen $a \in \Sigma$ die Positionen gespeichert sind, für die a ein Linkszeichen im zugehörigen Teilbaum ist.

An den Blättern erzeugen wir das Feld von Listen wie folgt. Sei b das Linkszeichen des Blattes, dessen zugehöriges Suffix an Position i beginnt. Dann enthalten alle Feldelemente die leere Liste mit Ausnahme für das Zeichen $b \in \Sigma$, die eine einelementige Liste mit Listenelement i enthält. Dies ist in Abbildung 3.7 illustriert.

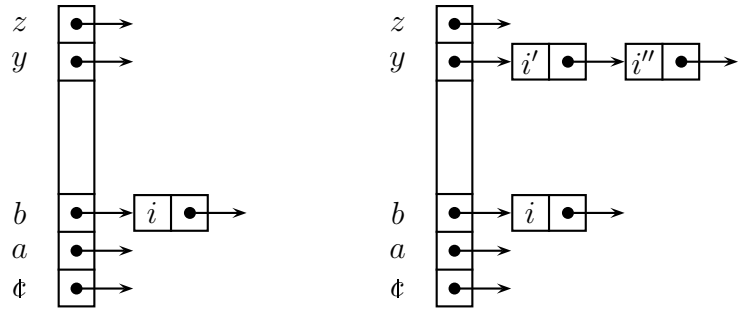


Abbildung 3.7: Skizze: Das Feld der Listen $L_{\overline{v}}$ für ein Blatt $\overline{v}$ mit Linkszeichen $b \in \Sigma$ (rechts) und für einen inneren Knoten (links) mit $\Sigma = \{a, \ldots, z\}$

An den inneren Knoten erzeugen wir dieses Feld, indem wir für jedes Feldelement $a \in \Sigma$ die entsprechenden Listen der Kinder aneinanderhängen. Bevor wir für ein Kind die Listen an die Listen des Knotens $\overline{v}$ anhängen, geben wir erst noch die maximalen Paare wie folgt aus: für jedes Paar von Kindern $\overline{v'} \neq \overline{v''}$ von $\overline{v}$ und für alle $a \neq b \in \Sigma \cup \{¢\}$ erzeuge für alle $i \in L_{\overline{v'}}[a]$ und $j \in L_{\overline{v''}}[b]$ die maximalen Paare $(i, j, |v|)$. Wenn wir das Kind $\overline{v''}$ neu aufnehmen, dann können wir statt der Liste vom Kind $\overline{v'}$ auch die bislang bereits konstruierte Listen von $\overline{v}$ hernehmen.

14.11.24

Für die Laufzeit ist wichtig, dass die Listen konkateniert und nicht kopiert werden, da die Listen ja sehr lang werden können. Prinzipiell muss dazu im Feld jeweils auch noch ein Zeiger auf das letzte Feldelement gespeichert werden. Die Idee der Konkatenation der Felderlisten der Kinder ist (ohne die Zeiger auf das das letzte Listenelement) in Abbildung 3.8 illustriert.

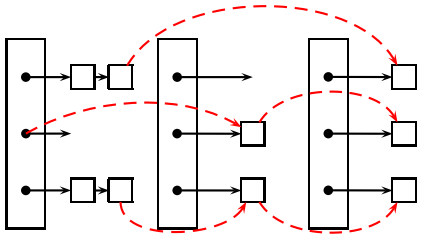


Abbildung 3.8: Skizze: Konkatenation der Felder von Listen zu einem neuen Feld von Listen

Der Aufwand der Konkatenation der Felder von Listen ist wiederum proportional zur Summe der Grade über alle Baumknoten und ist damit wieder linear (allerdings

mit Aufwand $O(|\Sigma|)$ pro Knoten), d.h. insgesamt $O(|\Sigma| \cdot |t|)$. Allerdings kann die Ausgabe der maximalen Paare wiederum mehr Zeit benötigen, jedoch nur konstante Zeit pro maximales Paar. Somit ist die Laufzeit wiederum linear in der Eingabe- und Ausgabegröße, sofern es keine leeren Listen gibt.

Wenn es leere Listen gibt, dann kann konstanter Aufwand entstehen, die nicht durch Ausgabe eine maximalen Paares gedeckt sind. Dies kann den Aufwand um den Faktor $|\Sigma|^2$ erhöhen, da die Anzahl der (nicht künstlichen) Kinder eines Knotens im Suffix-Baum durch $|\Sigma|$ beschränkt ist.

Wenn das Alphabet also groß wird, ist es besser, das Feld als sortierte Liste zu verwalten. Die oben angegebene Aufgaben lassen sich dann wirklich in Zeit $O(|\Sigma| \cdot n + k)$ implementieren. Der Leser möge sich die Details überlegen.

Theorem 3.12 *Sei $t \in \Sigma^n$, dann können alle maximalen Paare in Zeit $O(|\Sigma| \cdot n + k)$ ermittelt werden, wobei k die Anzahl der maximalen Paare ist.*

Auch hier kann man den Algorithmus leicht so modifizieren, dass er nur exakte Paare ausgibt, deren maximalen Repeats eine Mindestlänge aufweisen.

3.1.4 Revers-komplementäre Repeats

In diesem Abschnitt wollen wir uns mit den so genannten revers-komplementären Repeats beschäftigen, die in Genomen auftreten können, wenn das Repeat eigentlich auf dem anderen Strang der DNA-Doppelhelix liegt. Dazu müssen wir erst einmal formal definieren, was wir unter revers-komplementären Repeats verstehen wollen. Wir beginnen mit der Definition von revers-komplementären Sequenzen.

Definition 3.13 *Sei Σ ein Alphabet und sei $\pi \in S(\Sigma)$ eine Permutation auf Σ mit $\pi^2(\sigma) = \sigma$. Für $\sigma \in \Sigma$ ist das über π zugehörige* komplementäre Zeichen $\tilde{\sigma} = \pi(\sigma)$.

Für $w \in \Sigma^*$ ist das über π zugehörige revers-komplementäre Wort $\tilde{w}$ *wie folgt definiert:*

$$\tilde{w} = \begin{cases} \varepsilon & \text{falls} \quad w = \varepsilon, \\ \tilde{v} \cdot \tilde{\sigma} & \text{falls} \quad w = \sigma \cdot v \text{ mit } \sigma \in \Sigma,\ v \in \Sigma^*. \end{cases}$$

Für das biologisch relevante Alphabet $\Sigma = \{A, C, G, T\}$ ist das komplementäre Zeichen wie folgt definiert:

$$\tilde{\sigma} = \pi(\sigma) := \begin{cases} A & \text{falls} \quad \sigma = T, \\ C & \text{falls} \quad \sigma = G, \\ G & \text{falls} \quad \sigma = C, \\ T & \text{falls} \quad \sigma = A. \end{cases}$$

Hierauf basierend können wir nun die so genannten revers-komplementäre Repeats definieren.

Definition 3.14 *Sei $t \in \Sigma^*$. Ein Teilwort s von t heißt* revers-komplementäres Repeat*, wenn es ein Tripel (i, j, ℓ) mit $t_i \cdots t_{i+\ell-1} = s$ und $t_j \cdots t_{j+\ell-1} = \tilde{s}$ gibt.*

Die Lösung zum Auffinden exakter bzw. maximaler revers-komplementärer Repeats ist nun einfach, da wir das Problem auf exakte bzw. maximale Repeats zurückführen können. Sei $t \in \Sigma^*$ die Sequenz in der wir revers-komplementäre Repeats ermitteln wollen. Wir konstruieren zuerst das Wort $t' = ¢t\#\tilde{t}\$$, wobei $¢, \#, \$ \notin \Sigma$ neue Zeichen sind. Ein revers-komplementäres Repeat in t entspricht nun einem normalen Repeat in t'. Dies ist in Abbildung 3.9 schematisch dargestellt.

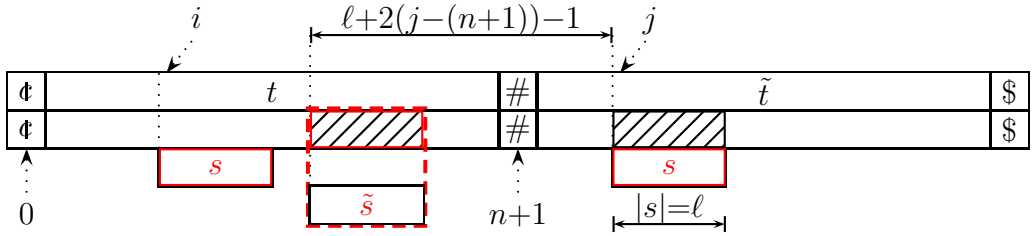


Abbildung 3.9: Skizze: revers-komplementäre Repeats

Wir müssen dabei nur beachten, dass wir auch normale Repeats in t finden (bzw. dann natürlich und nur dann auch in $\tilde{t}$). Wir müssen bei der Ermittlung der Repeats also darauf achten, dass ein Paar (i, j, ℓ) nur dann interessant ist, wenn $i \in [1 : n]$ und $j \in [n + 2 : 2n + 1]$ ist (wobei wir annehmen, dass $t' = t'_0 \cdots t'_{2n+2}$ ist). Als Ausgabe generieren wir dann $(i, j - (\ell + 2(j - (n + 1)) - 1), \ell) = (i, 2n - \ell - j + 3, \ell)$.

Um nicht mit der Überprüfung von Repeats innerhalb von t (bzw. $\tilde{t}$) aufgehalten zu werden, führen wir in den entsprechenden Algorithmen immer zwei Listen mit. Eine mit den Einträge von Positionen aus $[1 : n]$ und eine mit den Einträgen von Positionen aus $[n + 2 : 2n + 1]$. Bei der Ausgabe müssen wir dann immer nur Paare betrachten, bei denen die Positionen aus den jeweils entgegengesetzten Listen stammen.

Des Weiteren wird die Zeichenreihe $\tilde{t}$ natürlich nicht explizit konstruiert, sondern für $k \in [n + 1 : 2n + 1]$ wird dieser mittels $\tilde{t}[k] = \sigma(t[2n + 1 - k])$ jeweils berechnet.

Theorem 3.15 *Sei $t \in \Sigma^*$, dann lassen sich alle Paare, die exakte bzw. maximale revers-komplementäre Repeats darstellen, in Zeit $O(|\Sigma| \cdot n + k)$ ermitteln, wobei k die Anzahl der ausgegebenen Paare ist.*

3.2 Tandem-Repeats und Suffix-Bäume

In diesem Abschnitt wollen wir uns mit so genannten Tandem-Repeats beschäftigen, dass sind kurz gesprochen exakte Repeats, die unmittelbar hintereinander in t vorkommen.

3.2.1 Was sind Tandem-Repeats

Zunächst einmal definieren wir formal, was Tandem-Repeats sind.

Definition 3.16 *Für $t \in \Sigma^*$ heißt ein Paar (i, ℓ)* Tandem-Repeat-Paar *in t, wenn $t_i \cdots t_{i+\ell-1} = t_{i+\ell} \cdots t_{i+2\ell-1}$ gilt. Mit $\mathcal{T}(t)$ bezeichnen wir die Menge aller Tandem-Repeat-Paare von t. Das Wort $t_i \cdots t_{i+2\ell-1}$ heißt dann auch* Tandem-Repeat. *Mit $\overline{\mathcal{T}}(t)$ bezeichnen wir die Menge aller Tandem-Repeats von t. Die* Länge eines Tandem-Repeat-Paares (i, ℓ) *bzw.* eines Tandem-Repeats $\alpha\alpha$ *ist 2ℓ bzw. $2|\alpha|$.*

In Abbildung 3.10 ist schematisch ein Tandem-Repeat dargestellt.

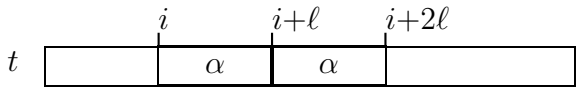


Abbildung 3.10: Skizze: Tandem-Repeat

Wir reden auch manchmal etwas locker von einem Tandem-Repeat anstelle eines Tandem-Repeat-Paares, wenn klar ist, welches zugehörige Tandem-Repeat-Paar hier gemeint ist.

Definition 3.17 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei $t' = t'_0 \cdots t'_{n+1} = ¢t_1 \cdots t_n\$$. Ein Tandem-Repeat-Paar (i, ℓ) von t heißt* rechtsverzweigend *bzw.* linksverzweigend, *wenn $t'_{i+\ell} \neq t'_{i+2\ell}$ bzw. $t'_{i-1} \neq t'_{i+\ell-1}$ gilt.*

Wir sagen auch manchmal etwas locker, dass ein Tandem-Repeat rechtsverzweigend ist, wenn klar ist, welches das zugehörige Tandem-Repeat-Paar ist. Man beachte,

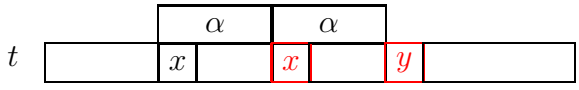


Abbildung 3.11: Skizze: Ein rechtsverzweigendes Tandem-Repeat mit $x \neq y \in \Sigma$

dass es Tandem-Repeats geben kann, für die es sowohl ein Tandem-Repeat-Paar geben kann, das rechtsverzweigend ist, als auch ein anderes, das nicht rechtsverzweigend ist. In Abbildung 3.11 ist schematisch ein rechtsverzweigendes Tandem-Repeat dargestellt.

Definition 3.18 *Sei $t \in \Sigma^*$ und sei (i, ℓ) ein Tandem-Repeat-Paar in t. Ist auch $(i+1, \ell)$ bzw. $(i-1, \ell)$ ein Tandem-Repeat-Paar, so nennen wir das Tandem-Repeat-Paar (i, ℓ) eine* Linksrotation *von $(i+1, \ell)$ bzw.* Rechtsrotation *von $(i-1, \ell)$.*

Mit Hilfe dieser Definition können wir uns beim Auffinden von Tandem-Repeat-Paaren auf rechtsverzweigende (oder auch linksverzweigende) beschränken, wie die folgende Beobachtung zeigt.

Beobachtung 3.19 *Jedes nicht rechts- bzw. linksverzweigende Tandem-Repeat-Paar (i, ℓ) ist eine Linksrotation des Tandem-Repeat-Paares $(i+1, \ell)$ bzw. eine Rechtsrotation des Tandem-Repeat-Paares $(i-1, \ell)$.*

Diese Beobachtung ist in der folgenden Abbildung 3.12 noch einmal illustriert.

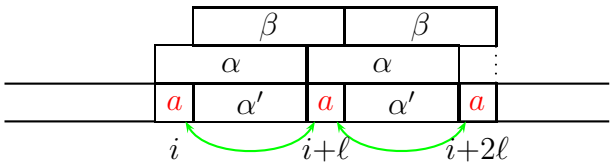


Abbildung 3.12: Skizze: Nicht rechtsverzweigendes Tandem-Repeat $\alpha\alpha$ als Linksrotation eines anderen Tandem-Repeats $\beta\beta$

Der Vollständigkeit halber definieren wir bereits an dieser Stelle noch so genannte primitive Tandem-Repeats.

Definition 3.20 *Eine Zeichenreihe $s \in \Sigma^+$ heißt* primitiv*, wenn aus $s = u^k$ für ein $u \in \Sigma^+$ und ein $k \in \mathbb{N}$ folgt, dass $k = 1$ ist.*

Ein Tandem-Repeat $\alpha\alpha$ heißt primitiv*, wenn α primitiv ist.*

Der Sinn hinter dieser Definition ist es, eine weitere Einschränkung auf interessante Tandem-Repeats zu besitzen. Nichtprimitive Tandem-Repeats bestehen ihrerseits aus einer Konkatenation gleicher Teilwörter und enthalten ihrerseits kürzere primitive Tandem-Repeats, die Anfänge von mehrfachen Wiederholungen sind, wie z.B. α^4. Solche mehrfach gekoppelten Repeats werden in der Literatur oft auch *Tandem-Arrays* genannt.

3.2.2 Eigenschaften von Tandem-Repeats

In diesem Abschnitt wollen wir einige fundamentale Beziehungen von (rechtsverzweigenden) Tandem-Repeats in t und dem zu t gehörigen Suffix-Baum aufstellen. Zunächst einmal wiederholen wir die Definition des im Folgenden wichtigen Begriffs der Worttiefe eines Knotens im Suffix-Baum.

Definition 3.21 *Sei $t \in \Sigma^*$ und sei $T = T(t\$)$ der Suffix-Baum für $t\$$. Für einen Knoten $v \in V(T)$ definieren wir seine* Worttiefe *als $|path(v)|$.*

Für das Folgende ist auch der Begriff der Blattlisten (engl. leaf lists) von fundamentaler Bedeutung, die wir beim Algorithmus zur Erkennung exakter Repeats schon kennen gelernt und dort auch eingesetzt haben, aber bislang noch nicht formal definiert haben.

Definition 3.22 *Sei $t \in \Sigma^*$ und sei $T = T(t\$)$ der Suffix-Baum zu $t\$$. Für ein Blatt $\overline{v} \in V(T)$ ist die* Blattliste $LL(\overline{v})$ *als die einelementige Menge $\{|t\$| - |v| + 1\}$ definiert (d.h. die Indexposition, an der der zugehörige Suffix in t auftritt). Die* Blattliste $LL(\overline{v})$ *eines inneren Knotens $\overline{v} \in V(T)$ ist definiert durch*

$$LL(\overline{v}) := \bigcup_{\substack{\overline{w} \in V(T) \\ (\overline{v},\overline{w}) \in E(T)}} LL(\overline{w})$$

(d.h. die Menge aller Indexposition von Suffixen, deren zugehörigen Blätter sich im Teilbaum von $\overline{v}$ befinden).

Mit Hilfe dieser Begriffe können wir nun eine Charakterisierung von Tandem-Repeats durch Knoten im zugehörigen Suffix-Baum angeben.

Lemma 3.23 *Sei $t \in \Sigma^n$ und sei $i < j \in [1 : n]$ sowie $\ell := j - i$. Dann sind folgende Aussagen äquivalent:*

1. *Das Paar (i, ℓ) ist ein Tandem-Repeat-Paar in t.*
2. *Es existiert ein Konten $\overline{v} \in V(T(t\$))$ mit $|v| \geq \ell$ und $i, j \in LL(\overline{v})$.*

Beweis: 1 ⇒ 2 : Nach Voraussetzung ist also (i, ℓ) ein Tandem-Repeat-Paar in t. Dies bedeutet, dass $t_i \cdots t_{i+\ell-1} = t_{i+\ell} \cdots t_{i+2\ell-1}$ gilt. Dies ist in Abbildung 3.13 illustriert.

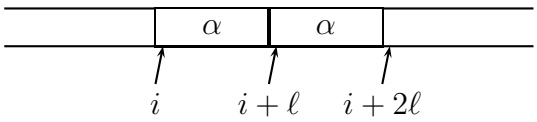


Abbildung 3.13: Skizze: (i, ℓ) ist ein Tandem-Repeat-Paar in t

Sei also $\alpha\alpha$ das zu (i, ℓ) gehörige Tandem-Repeat. Somit beginnt das Suffix an Position i sowie das Suffix an Position $j := i + \ell$ jeweils mit α. Sei weiter β so gewählt, dass $\overline{\alpha\beta}$ der Knoten in $T(t\$)$ ist, an dem sich die Pfade von der Wurzel zu den Blättern von $t_i \cdots t_n\$$ und $t_{i+\ell} \cdots t_n\$$ trennen, siehe hierzu auch Abbildung 3.14.

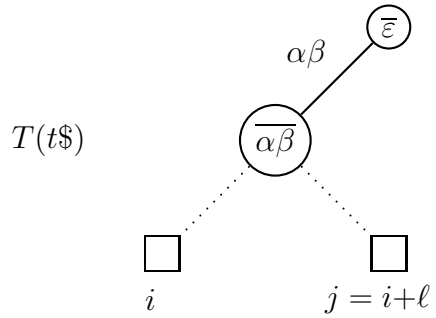


Abbildung 3.14: Skizze: Tandem-Repeat $\alpha\alpha$ und der Suffix-Baum $T(t\$)$

Dann befinden sich jedoch i und $j = i+\ell$ in der Blattliste $LL(\overline{\alpha\beta})$ und die Worttiefe von $\overline{\alpha\beta}$ ist gleich $|\alpha\beta| \geq |\alpha| = \ell$.

2 ⇒ 1 : Sei also v ein Knoten von $T(t\$)$ mit Worttiefe mindestens ℓ so gewählt, dass $i, j \in LL(v)$ gilt. Weiterhin sei $\text{path}(v) = \alpha\beta$ mit $|\alpha| = \ell$ und $\beta \in \Sigma^*$. Dieser Fall ist in Abbildung 3.15 illustriert.

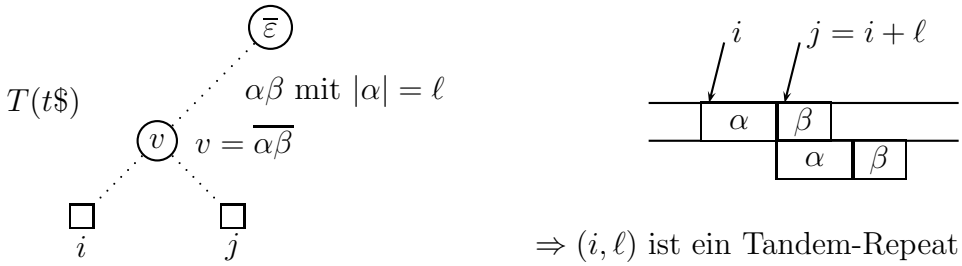


Abbildung 3.15: Skizze: Knoten v mit Worttiefe mindestens ℓ und $i, j \in LL(v)$

Daraus folgt nun sofort, dass $t_i \cdots t_{i+|\alpha\beta|-1} = \alpha\beta$ und $t_j \cdots t_{j+|\alpha\beta|-1} = \alpha\beta$. Da $|\alpha| = \ell$ und $\ell = j - i$ ist, folgt, dass sich die beiden Vorkommen in t überlappen müssen und $\alpha\alpha$ somit ein Tandem-Repeat zum Tandem-Repeat-Paar (i, ℓ) sein muss. Dies ist im

linken Teil der Abbildung 3.15 besonders gut zu erkennen. Damit ist die Behauptung bewiesen. ■

Wir können auch eine analoge Charakterisierung für rechtsverzweigende Tandem-Repeats angeben.

Lemma 3.24 *Sei $t \in \Sigma^n$ und sei $i < j \in [1:n]$ sowie $\ell := j - i > 0$. Dann sind folgende Aussagen äquivalent:*

1. *Das Paar (i, ℓ) ist ein rechtsverzweigendes Tandem-Repeat-Paar in t.*
2. *Es existiert ein Knoten $\overline{v} \in V(T(t\$))$ mit $|v| = \ell$ und $i, j \in LL(\overline{v})$. Weiterhin gilt für alle Knoten $\overline{w} \in V(T(t\$))$ mit $|w| > \ell$, dass nicht sowohl $i \in LL(\overline{w})$ als auch $j \in LL(\overline{w})$ gilt.*

Beweis: Der Beweis ist analog zum Beweis in Lemma 3.23 und bleibt dem Leser zur Übung überlassen. ■

3.2.3 Algorithmus von Stoye und Gusfield

Aus diesem Lemma 3.24 lässt sich sofort der folgende, in Abbildung 3.16 angegebene Algorithmus von Stoye und Gusfield für das Auffinden rechtsverzweigender Tandem-Repeat-Paare herleiten. Dabei werden die rechtsverzweigenden Tandem-Repeat-Paare explizit durch die Abfrage $t_{i+|\text{path}(v)|} \neq t_{i+2|\text{path}(v)|}$ herausgefiltert. Aufgrund der Tatsache, dass die Anfangsposition eines rechtsverzweigendes Tandem-Repeat-Paares nur an einem Knoten aufgefunden werden kann (siehe Lemma 3.24), wird jedes rechtsverzweigendes Tandem-Repeat-Paar genau einmal ausgegeben.

Der Test, ob $i + |\text{path}(v)| \in LL(v)$ gilt, wird effizient mit Hilfe einer DFS-Nummerierung der Blätter durchgeführt. Haben wir zu jedem Blatt j seine DFS-Nummer $\text{DFS}(j)$ und zu jedem inneren Knoten v das DFS-Intervall $\text{DFS_Int}(v)$, das die DFS-Nummer der Blätter in seinem Teilbaum beschreibt, so gilt:

$$j \in LL(v) \quad \Leftrightarrow \quad \text{DFS}(j) \in \text{DFS_Int}(v)$$

Die Korrektheit folgt daher, dass wir einfach nur eine Umbenennung der Knoten vorgenommen haben, so dass die Blattlisten zu Intervallen werden. Es ist nämlich offensichtlich effizienter, zwei Intervallgrenzen abzufragen als eine ganze Liste (als Darstellung der Menge) zu durchlaufen.

Daraus ergibt sich der in Abbildung 3.17 angegeben Algorithmus.

```
TandemRepeats (string t)
begin
    tree T := SuffixTree(t$);
    foreach (v ∈ V(T)) do                                   /* using DFS */
        // after returning from all children
        // creating leaf lists
        LL(v) := ∅;
        foreach ((v, w) ∈ E(T)) do
            LL(v) := LL(v) ∪ LL(w);
        if (LL(v) = ∅) then
            LL(v) := {|t| + 2 − |path(v)|};
        ℓ := |path(v)|;
        foreach (i ∈ LL(v)) do
            if ((i + ℓ ∈ LL(v)) && (t_{i+ℓ} ≠ t_{i+2ℓ})) then
                output (i, ℓ);
end
```

Abbildung 3.16: Algorithmus von Stoye und Gusfield

```
TandemRepeats (string t)
begin
    tree T := SuffixTree(t$);   int dfs := 0;
    foreach (v ∈ V(T)) do                                   /* using DFS */
        if (v is a leaf) then DFS(v) = ++dfs;
        determine DFS_Int(v);
        // after returning from all children
        // creating leaf lists
        LL(v) := ∅;
        foreach ((v, w) ∈ E(T)) do
            LL(v) := LL(v) ∪ LL(w);
        if (LL(v) = ∅) then
            LL(v) := {|t| + 2 − |path(v)|};
        ℓ := |path(v)|;
        foreach (i ∈ LL(v)) do
            if ((DFS(i + ℓ) ∈ DFS_Int(v)) && (t_{i+ℓ} ≠ t_{i+2ℓ})) then
                output (i, ℓ);
end
```

Abbildung 3.17: Algorithmus von Stoye und Gusfield mit DFS-Nummerierung

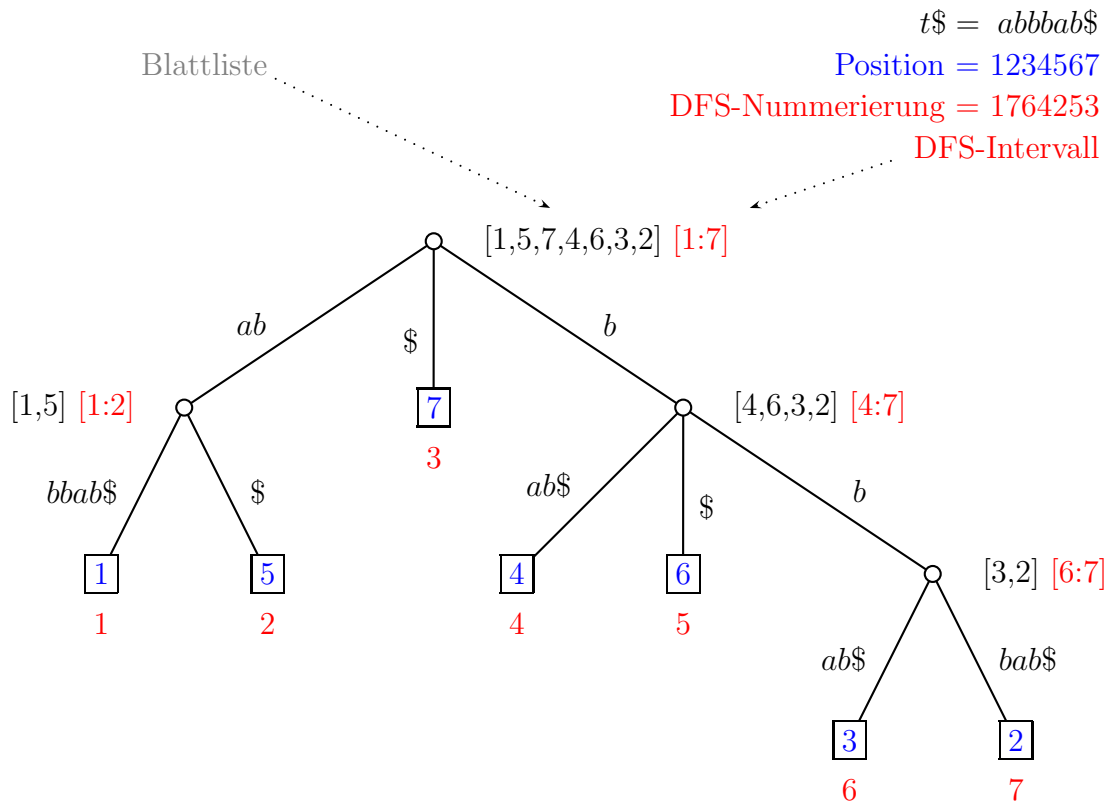


Abbildung 3.18: Beispiel: Auffinden von Tandem-Repeat-Paaren mit dem Algorithmus von Stoye und Gusfield

Ein Beispiel ist hierfür in Abbildung 3.18 angegeben. Wir geben noch ein paar Beispiele zur Bestimmung von $i + |\text{path}(v)| \in LL(v)$ an.

Beispiel 1: Wir betrachten zuerst den Knoten $v = \overline{b}$ und die Abfrage für $i = 4$. Der ausgeführte Test lautet dann: $j = i + |\text{path}(\overline{b})| = 4 + 1 = 5 \stackrel{?}{\in} LL(\overline{b})$. Da $\text{DFS}(5) = 2 \notin [4:7] = \text{DFS_Int}(\overline{b})$, fällt die Antwort negativ aus.

Beispiel 2: Wir betrachten jetzt den Knoten $v = \overline{b}$ und die Abfrage für $i = 2$. Der ausgeführte Test lautet dann: $j = i + |\text{path}(\overline{b})| = 2 + 1 = 3 \stackrel{?}{\in} LL(\overline{b})$. Da $\text{DFS}(3) = 6 \in [4:7] = \text{DFS_Int}(\overline{b})$, fällt die Antwort positiv aus.

Da weiter $t_{i+|\text{path}(v)|} = t_{2+1} = b = t_{2+2} = t_{i+2|\text{path}(v)|}$, handelt es sich um keinen rechtsverzweigenden Tandem-Repeat.

Beispiel 3: Wir betrachten erneut den Knoten $v = \overline{b}$ und die Abfrage für $i = 3$. Der ausgeführte Test lautet dann: $j = i + |\text{path}(\overline{b})| = 3 + 1 = 4 \stackrel{?}{\in} LL(\overline{b})$. Da $\text{DFS}(4) = 4 \in [4:7] = \text{DFS_Int}(\overline{b})$, fällt die Antwort positiv aus.

Da weiter $t_{i+|\text{path}(v)|} = t_{3+1} = b \neq a = t_{3+2} = t_{i+2|\text{path}(v)|}$, handelt es sich um einen rechtsverzweigenden Tandem-Repeat.

3.2.4 Laufzeitanalyse und Beschleunigung

Wir wollen nun die Laufzeit für diesen Algorithmus analysieren. Die Konstruktion des Suffix-Baumes und das Erstellen der Blattlisten (mittels einer Tiefensuche und dem Konkatenieren der Blattlisten der Kinder an den inneren Knoten) benötigt lineare Zeit. Die reine Tiefensuche im zweiten Teil durchläuft $O(n)$ Knoten und für jeden Knoten werden seine Blattlisten durchlaufen. Da die Blattliste maximal $O(n)$ Blätter enthalten kann, beträgt der Zeitaufwand $O(n^2)$. Hierbei haben wir schon ausgenutzt, dass sich der Test $i + |\text{path}(v)| \in LL(v)$ in konstanter Zeit mit Hilfe der DFS-Nummern ausführen lässt.

Theorem 3.25 *Sei $t \in \Sigma^n$. Alle rechtsverzweigenden Tandem-Repeats von t können in Zeit $O(n^2)$ ermittelt werden.*

Berücksichtigen wir die Beobachtung zu Beginn, dass man alle Tandem-Repeats aus rechtsverzweigenden Tandem-Repeats rekonstruieren kann und dass es maximal $O(n^2)$ Tandem-Repeats in einem Wort der Länge n geben kann, erhalten wir das folgende Korollar.

Korollar 3.26 *Sei $t \in \Sigma^n$. Alle Tandem-Repeats von t können in Zeit $O(n^2)$ ermittelt werden.*

19.11.24

Wir wollen jetzt noch einen einfachen Trick vorstellen, mit dem sich die Laufzeit beschleunigen lässt. Wir werden dazu zunächst nicht direkt die Blattliste $LL(v)$ erzeugen, sondern eine disjunkte Vereinigung davon.

Betrachten wir dazu einen Knoten v mit seinen Kinder $v_1, \ldots, v_k$. Sei v' ein Kind von v mit einer längsten Blattliste, d.h. $|LL(v')| \geq |LL(v_i)|$ für $i \in [1 : k]$. Dann definieren wir:

$$LL'(v) := \bigcup_{\substack{i=1 \\ v_i \neq v'}}^{k} LL(v_i).$$

Es gilt dann $LL(v) = LL'(v) \cup LL(v')$ mit $LL'(v) \cap LL(v') = \emptyset$. Dies ist in Abbildung 3.19 illustriert.

Weiter wissen wir aufgrund von Lemma 3.24 für unseren Test $i + \ell \stackrel{?}{\in} LL(v)$ mit $i \in LL(v)$ und $\ell := |\text{path}(v)|$: Gilt $i \notin LL'(v)$ (d.h. $i \in LL(v')$), dann muss $j = i + \ell \in LL'(v)$ gelten.

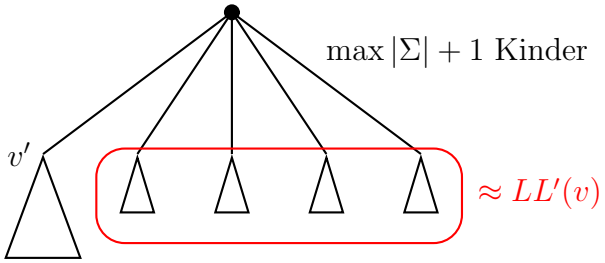


Abbildung 3.19: Skizze: Partitionierung von $LL(v)$ in $LL'(v) \cup LL(v')$

Der im Algorithmus durchgeführte Test, ob $i + \ell \stackrel{?}{\in} LL(v)$ für $i \in LL(v)$ und $\ell := |\text{path}(v)|$ wird dann wie folgt ausgeführt.

Gilt $i \in LL'(v)$: In diesem Fall testen wir ganz normal

$$j = i + \ell \stackrel{?}{\in} LL(v) = LL'(v) \cup LL(v').$$

Gilt $i \notin LL'(v)$: Dann gilt $i \in LL(v')$ und es muss $j = i + \ell \in LL'(v)$ gelten. Wir testen daher umgekehrt für $j \in LL'(v)$, ob gilt:

$$i = j - \ell \stackrel{?}{\in} LL(v').$$

Dadurch müssen wir in der **for**-Schleife nicht mehr über alle Listenelemente der Blattliste von v gehen, sondern es genügt die Liste $LL'(v)$ zu durchlaufen. Der so modifizierte Algorithmus ist in Abbildung 3.20 angegeben. Auch hier werden in der realen Implementierung die Tests auf Enthaltensein in Listen durch Enthaltensein in DFS-Intervallen ersetzt.

3.2.5 Eine einfache Laufzeitanalyse (*)

Wie verhalten sich nun die Längen der Blattlisten zueinander? Wir halten zunächst fest, dass der Knoten v maximal $|\Sigma|+1$ Kinder besitzt und dass die längste Blattliste mindestens so lang sein muss wie eine durchschnittliche. Nach Wahl von v' gilt also:

$$|LL(v')| \geq \frac{|LL(v)|}{|\Sigma| + 1}.$$

Mit $|LL(v)| = |LL(v')| + |LL'(v)|$ folgt sofort:

$$|LL'(v)| = |LL(v)| - |LL(v')| \leq |LL(v)| - \frac{|LL(v)|}{|\Sigma| + 1} = \frac{|\Sigma|}{|\Sigma| + 1} \cdot |LL(v)|.$$

FastTandemRepeats (string t)

```
begin
    tree T := SuffixTree(t$);
    int dfs_num := 0;                    /* used for DFS numbering of leaves */
    foreach (v ∈ V(T)) do                    /* using DFS at bottom-up phase */
        Interval DFS_Int(v) := ∅;
        node v' := v;  /* some node with an empty leaf list (currently) */
        foreach ((v, w) ∈ E(T)) do                        /* w is a child of v */
            if (|DFS_Int(w)| > |DFS_Int(v')|) then
                v' := w;
            DFS_int(v) := DFS_Int(v) ∪ DFS_Int(w);

        ℓ := |path(v)|;
        foreach (x ∈ DFS_Int(v) \ DFS_Int(v')) do      /* for each i ∈ LL'(v) */
            i := iDFS(x);  /* the position in t associated with the leaf */
            if ((DFS(i + ℓ) ∈ DFS_Int(v)) && (t_{i+ℓ} ≠ t_{i+2ℓ})) then
                return (i, ℓ);
            if ((DFS(i − ℓ) ∈ DFS_Int(v')) && (t_i ≠ t_{i+ℓ})) then
                return (i − ℓ, ℓ);
        if (DFS_Int(v') = ∅) then                                  /* v is a leaf */
            p := |t| + 2 − ℓ;          /* the position in t associated with v */
            DFS(p) := ++dfs_num;
            iDFS(dfs_num) := p;
            DFS_Int(v) := [dfs_num, dfs_num];
end
```

Abbildung 3.20: Beschleunigter Algorithmus von Stoye und Gusfield

Wenn also ein Blatt aus $LL'(v)$ an einem Vergleich beteiligt war, befindet es sich anschließend in einer Blattliste, die mindestens um den Faktor $\frac{|\Sigma|+1}{|\Sigma|}$ größer ist. Dies kann nur

$$\log_{\frac{|\Sigma|+1}{|\Sigma|}}(n) = \frac{\log(n)}{\log\left(1+\frac{1}{|\Sigma|}\right)} \approx |\Sigma| \cdot \log(n)$$

oft passieren. Somit kann jedes Blatt nur $O(|\Sigma| \cdot \log(n))$ Vergleiche initiieren. Die Laufzeit aller Tests beträgt also insgesamt: $O(|\Sigma| \cdot n \log(n))$.

Theorem 3.27 *Alle rechtsverzweigenden Tandem-Repeat-Paare (Tandem-Repeats) eines Wortes $t \in \Sigma^n$ können in Zeit $O(|\Sigma| \cdot n \log(n))$ ermittelt werden.*

Durch die obere Schranke der Laufzeit erhalten wir auch sofort eine Beschränkung der Anzahl rechtsverzweigender Tandem-Repeats eines Wortes.

Korollar 3.28 *Ein Wort $t \in \Sigma^n$ besitzt maximal $O(|\Sigma| \cdot n \log(n))$ rechtsverzweigende Tandem-Repeat-Paare (bzw. Tandem-Repeats).*

Korollar 3.29 *Alle Tandem-Repeat-Paare (bzw. Tandem-Repeats) von $t \in \Sigma^n$ können in Zeit $O(|\Sigma| \cdot n \log(n) + k)$ ermittelt werden, wobei k die Anzahl der Tandem-Repeat-Paare (bzw. Tandem-Repeats) in t ist.*

3.2.6 Eine bessere Laufzeitanalyse

Wir können für große Alphabetgrößen noch eine bessere Laufzeitanalyse geben. Wir betrachten dazu wieder einen Knoten v, dessen Kind mit größter Blattliste gerade v' ist. Für ein Blatt $i \in LL'(v)$ betrachten wir den nächsten Vorgänger u von v, so dass $i \in LL'(u)$ gilt. Dies muss nicht notwendigerweise der Elter oder Großelter von v sein. Erst an diesem Knoten u wird das Blatt i das nächste Mal einen Test auslösen. Dies ist in Abbildung 3.21 illustriert.

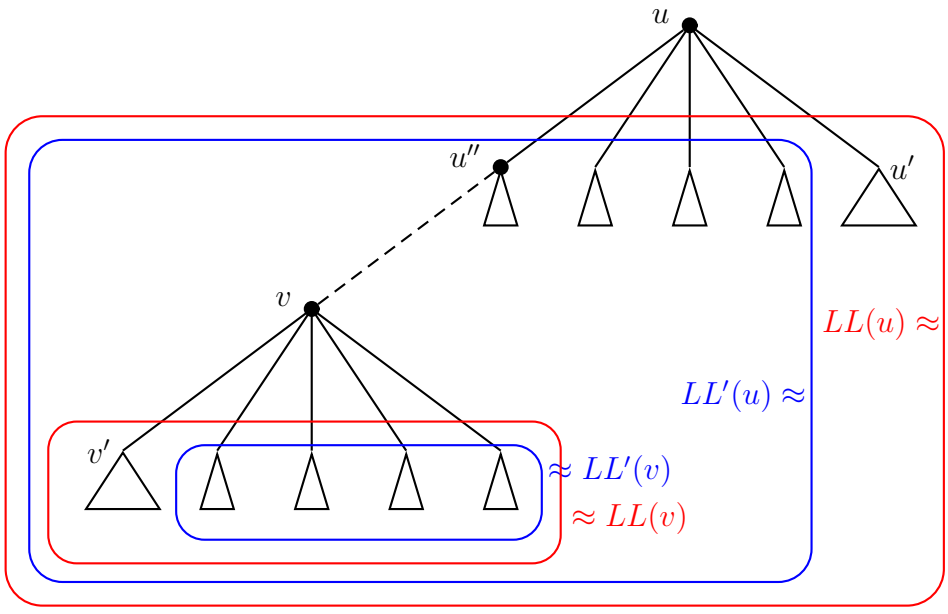


Abbildung 3.21: Skizze: Verhältnis von $LL(v)$ zu $LL'(u)$

Wir fragen uns jetzt, wie verhält sich $|LL(u)|$ zu $|LL(v)|$. Diese Frage ist also zunächst einmal unabhängig von den Blattlisten $LL'(v)$ und $LL'(u)$, deren zugehörige Blätter ja gerade die Tests auslösen. Sei u' das Kind von u mit der größten Blattliste. Dann ist u' kein Vorfahre von v oder v', da wir ja annehmen, dass

$i \in LL'(u)$ und somit $i \notin LL(u')$ gilt. Zunächst einmal gilt nach Wahl von u', dass für das Kind u'' von u, das auch ein Vorfahre von v (oder gleich v) ist:

$$|LL(u')| \geq |LL(u'')| \geq |LL(v)|.$$

Damit können wir weiter zeigen:

$$\begin{aligned} |LL(u)| &= |LL(u')| + |LL'(u)| \\ &\quad \text{da } |LL(u')| \geq |LL(u'')| \geq |LL(v)| \\ &\geq |LL(v)| + |LL'(u)| \\ &\quad \text{da } LL'(u) \supseteq LL(v) \\ &\geq |LL(v)| + |LL(v)| \\ &= 2|LL(v)|. \end{aligned}$$

Löst also ein Blatt i einen Vergleich am Knoten v aus und das nächste Mal erst wieder am Vorfahren u von v, dann hat sich die Länge der Blattliste des zugehörigen Knotens gegenüber dem vorher betrachteten mindestens verdoppelt. Somit kann jedes Blatt nur $O(\log(n))$ Vergleiche initiieren. Die Laufzeit aller Tests beträgt also insgesamt: $O(n \log(n))$.

Theorem 3.30 *Alle rechtsverzweigenden Tandem-Repeat-Paare (Tandem-Repeats) eines Wortes $t \in \Sigma^n$ können in Zeit $O(n \log(n))$ ermittelt werden.*

Durch die obere Schranke der Laufzeit erhalten wir auch sofort eine Beschränkung der Anzahl rechtsverzweigender Tandem-Repeats eines Wortes.

Korollar 3.31 *Ein Wort $t \in \Sigma^n$ besitzt maximal $O(n \log(n))$ rechtsverzweigende Tandem-Repeat-Paare (bzw. Tandem-Repeats).*

Korollar 3.32 *Alle Tandem-Repeat-Paare (bzw. Tandem-Repeats) von $t \in \Sigma^n$ können in Zeit $O(n \log(n)+k)$ ermittelt werden, wobei k die Anzahl der Tandem-Repeat-Paare (bzw. Tandem-Repeats) in t ist.*

3.3 Tandem-Repeats mit Divide-&-Conquer

Wir wollen nun noch ein Verfahren zur Erkennung von Tandem-Repeats vorstellen, das sich auch auf Erkennung ähnlicher Tandem-Repeats erweitern lässt.

3.3.1 Algorithmus von Main und Lorentz

In diesem Abschnitt wollen wir noch eine alternative Methode zur Erkennung von Tandem-Repeats angeben. Dieses Verfahren von Main und Lorentz basiert auf einem Divide-and-Conquer-Ansatz. Wir nehmen wieder an, dass $t \in \Sigma^*$ mit $t = t_1 \cdots t_n$. Der Algorithmus geht wir folgt vor.

0. Schritt: Ist $n \leq 1$, dann enthält t keine Tandem-Repeats und wir sind fertig. Andernfalls berechne $h := \lfloor \frac{n}{2} \rfloor$.

1. Schritt: Finde rekursiv alle Tandem-Repeats in $t_1 \cdots t_h$.

2. Schritt: Finde rekursiv alle Tandem-Repeats in $t_{h+1} \cdots t_n$.

3. Schritt: Finde alle Tandem-Repeats $\alpha\alpha$ der Länge 2ℓ mit $\ell \in [1 : \lfloor n/2 \rfloor]$, die an Position $i \in [h - \ell + 2, h]$ beginnen.

4. Schritt: Finde alle Tandem-Repeats $\alpha\alpha$ der Länge 2ℓ mit $\ell \in [1 : \lfloor n/2 \rfloor]$, die an Position $i \in [h - 2\ell + 2, h - \ell + 1]$ beginnen.

Das Prinzip des Algorithmus ist in der folgenden Abbildung 3.22 noch einmal schematisch dargestellt.

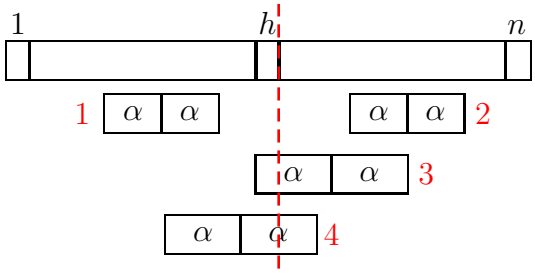


Abbildung 3.22: Skizze: Die vier Schritte des Algorithmus von Main und Lorentz

Die Implementierung der Schritte 0, 1, und 2 sind trivial. Wir müssen uns also nur noch Gedanken um die Implementierung von Schritt 3 und 4 machen. Wie man leicht sieht, sind die Schritte 3 und 4 symmetrisch, so dass wir uns nur einen der beiden Schritte genauer anschauen müssen. Wie wir noch sehen werden, kann der Schritt 3 bzw. 4 in Zeit $O(n)$ realisiert werden (zumindest wenn wir uns auf rechtsverzweigende Tandem-Repeats beschränken). Damit erhalten wir folgende Rekursionsgleichung für die Laufzeit des Algorithmus von Main und Lorentz:

$$T(n) = \begin{cases} O(1) & \text{falls} \quad n \leq 1, \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + O(n) & \text{falls} \quad n \geq 2. \end{cases}$$

Das ist dieselbe Laufzeitanalyse wie für Merge-Sort und die beträgt bekanntlich $O(n \log(n))$. Somit können wir schon das folgende Theorem festhalten.

Theorem 3.33 *Sei $t \in \Sigma^*$ mit $n = |t|$. Mit dem Algorithmus von Main und Lorentz lassen sich alle rechtsverzweigenden Tandem-Repeat-Paare in Zeit $O(n \log(n))$ finden.*

Für den Beweis dieses Satzes müssen wir nur noch zeigen, wie sich Schritt 3 und 4 in linearer Zeit implementieren lassen.

3.3.2 Longest Common Extensions

Für Schritt 3 bzw. 4 im Algorithmus von Main und Lorentz benötigen wir das Konzept der so genannten *longest common extension*.

Definition 3.34 *Sei $t \in \Sigma^*$ mit $n = |t|$ und sei $t' = ¢t\$$. Für $i < j \in [1 : n]$ ist eine* longest common forward extension *bzw.* longest common backward extension *(oder kurz* longest common extension*) von i und j in t definiert als eine Zahl $m \in \mathbb{N}_0$ mit $t_i \cdots t_{i+m-1} = t_j \cdots t_{j+m-1}$ und $t'_{i+m} \neq t'_{j+m}$ (in Zeichen $\mathrm{lce}_f(i,j) = m$) bzw. $t_{i-m+1} \cdots t_i = t_{j-m+1} \cdots t_j$ und $t'_{i-m} \neq t'_{j-m}$ (in Zeichen $\mathrm{lce}_b(i,j) = m$).*

Eine solche longest common (forward) extension ist in der Abbildung 3.23 noch einmal schematisch dargestellt.

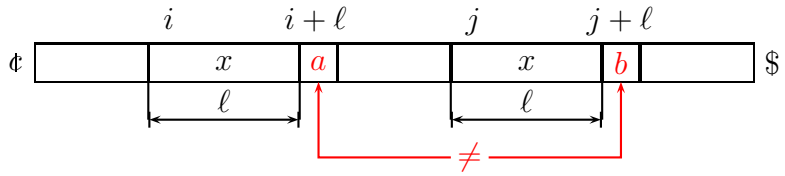


Abbildung 3.23: Skizze: Longest Common (Forward) Extension

Bevor wir uns darum kümmern, wie uns longest common extensions für Schritt 3 bzw. 4 im Algorithmus von Main und Lorentz helfen, werden wir uns erst überlegen, wie man solche longest common extensions effizient berechnen kann.

Dazu betrachten wir zunächst den Suffix-Baum $T = T(t\$)$ für t. In T betrachten wir die beiden Blätter, die zu den Suffixen $t_i \cdots t_n\$$ und $t_j \cdots t_n\$$ gehören. Zu diesen muss es nach Konstruktion des Suffix-Baumes T jeweils einen Pfad von der Wurzel zum entsprechenden Blatt geben. Solange diese beiden Pfade von der Wurzel gemeinsam

verlaufen, verfolgen wir ein gemeinsames Wort, dass sowohl ab Position i als auch ab Position j in t beginnt (common extension). Sobald sich die Pfade trennen, unterscheiden sich der folgende Buchstabe und wir haben an diesem Knoten $v \in V(T)$ die longest common extension $\text{lce}_f(i,j) = \ell = |\text{path}(v)| = |\text{path}(\text{lca}(i,j))|$ gefunden. Dies ist in Abbildung 3.24 noch einmal illustriert.

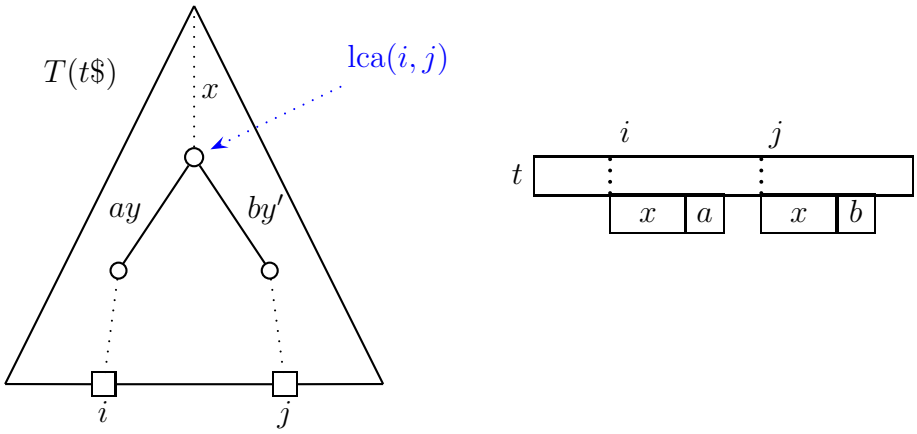


Abbildung 3.24: Skizze: Eine longest common (forward) extension in t und der niedrigste gemeinsame Vorfahre im Suffix-Baum von $t\$$

Wir können also das Problem der longest common extension im Wort t auf eine Anfrage zum niedrigsten gemeinsamen Vorfahren (eng. lowest common ancestor, kurz lca) zweier Blätter im Suffix-Baum für $t\$$ reduzieren.

Wir gehen in Kapitel 4 näher darauf ein, wie man solche lowest common ancestor Anfragen effizient beantworten kann, insbesondere wenn für einen Baum sehr viele Anfragen gestellt werden. Für die weitere Analyse des Algorithmus von Main und Lorentz geben wir hier noch kurz das hierfür interessierende Ergebnis an. Mit einer Vorverarbeitung des Baumes T mit n Knoten in Zeit von $O(n)$, kann jede nachfolgende LCA-Query in konstanter Zeit beantwortet werden.

3.3.3 Conquer-Schritt

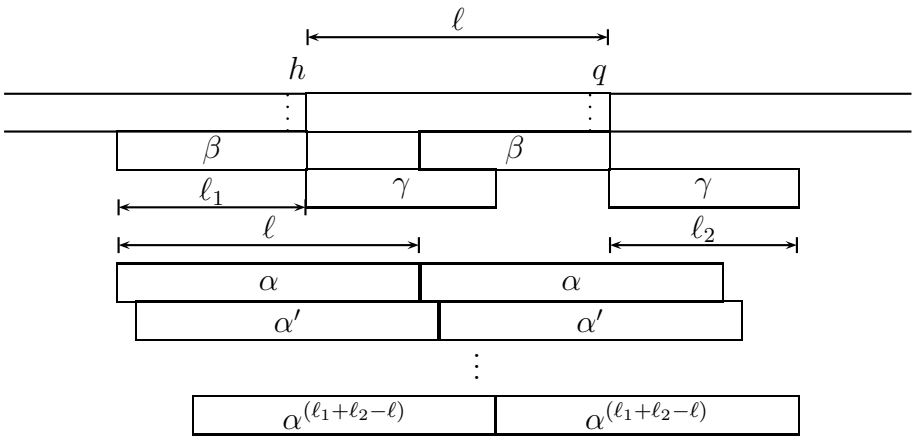
Für die Lösung von Schritt 3 (bzw. analog für Schritt 4) gehen wir wie folgt vor. Wir setzen $h = \lfloor n/2 \rfloor$ und $q = h + \ell$ wobei $\ell \in [1 : \lfloor n/2 \rfloor]$ die Länge einer Hälfte eines Tandem-Repeats ist. Dann ermitteln wir die longest common backward extension in t von den Positionen h und q:

$$\ell_1 := \text{lce}_b(h, q).$$

Weiterhin ermitteln wir die longest common forward extension in t von den Positionen $h+1$ und $q+1$:

$$\ell_2 := \mathrm{lce}_f(h+1, q+1).$$

Gilt für diese beiden longest common extensions $\ell_1 + \ell_2 \geq \ell$, dann überlappen sich diese beiden longest common extensions $\beta = t_{q-\ell_1+1} \cdots t_q = t_{h-\ell_1+1} \cdots t_h$ und $\gamma = t_{h+1} \cdots t_{h+\ell_2} = t_{q+1} \cdots t_{q+\ell_2}$ (bzw. sind unmittelbar aufeinander folgend) im Bereich $t_{h+1} \cdots t_q$. Aufgrund dieser Überlappung beginnen ab Position $h - \ell_1 + 1$ genau $\ell_1 + \ell_2 - \ell + 1$ Tandem-Repeats. Gilt weiter, dass $\ell_1 > 0$, dann beginnt der erste dieser Tandem-Repeats im Wort $t_1 \cdots t_h$, wie für Schritt 3 gefordert. Allerdings gilt für $\ell_1 > \ell$, dass einige der Tandem-Repeats nicht für den Fall 3 sind. Man beachte noch, dass all diese Tandem-Repeats bis auf das letzte nichtrechtsverzweigend sind. Falls $\ell_1 + \ell_2 < \ell$ gilt, dann gibt es offensichtlich kein Tandem-Repeat, das im Teilwort $t_1 \cdots t_h$ beginnt und in das Teilwort $t_{h+1} \cdots t_n$ hineinragt. Wenn nur das letzte rechtsverzweigende Tandem-Repeat-Paar ausgegebenwerden soll, muss $\ell_2 < \ell$ gelten, da ansonsten dieses Tandem-Repeat vollständig in der zweiten Hälfte liegt. Dies ist in Abbildung 3.25 noch einmal illustriert.



$$\ell_1 := \mathrm{lce}_b(h, q) = |\beta| \qquad \ell_2 := \mathrm{lce}_f(h+1, q+1) = |\gamma|$$

Abbildung 3.25: Skizze: Longest common backward bzw. forward extensions in t ab Position h und $q = h + \ell$ bzw. $h+1$ und $q+1$ und die dadurch induzierten Tandem-Repeats

Wir kümmern uns nun noch um die Laufzeit von Schritt 3 (bzw. Schritt 4). Wie noch zu zeigen ist, können wir nach einer linearen Vorverarbeitung jede longest common extension Anfrage in konstanter Zeit bearbeiten. Für ein festes ℓ können somit die zwei LCE-Anfragen in konstanter Zeit beantwortet werden. Die Ausgabe eines rechtsverzweigenden Tandem-Repeat-Paares geht dann ebenfalls in konstanter Zeit, da es maximal eines geben kann. Für alle $\ell \in [1 : \lfloor n/2 \rfloor]$ ist somit Schritt 3 (bzw. Schritt 4) in Zeit $O(n)$ lösbar.

Theorem 3.35 *Der Algorithmus von Main und Lorentz kann alle rechtsverzweigenden Tandem-Repeat-Paare (bzw. alle rechtsverzweigenden Tandem-Repeats) in Zeit $O(n\log(n))$ und Platz $O(n)$ in einem Wort $t \in \Sigma^n$ finden.*

Will man alle Tandem-Repeat-Paare ausgeben, so erhöht sich die Laufzeit um die Anzahl der ausgegeben Tandem-Repeat-Paare, da diese ja nach Beobachtung 3.19 Linksrotationen der rechtsverzweigenden Tandem-Repeat-Paare sind. Somit beträgt die Laufzeit des Algorithmus von Main und Lorentz $O(n\log(n)+k)$, wobei k die Anzahl der Tandem-Repeats in t ist.

Theorem 3.36 *Der Algorithmus von Main und Lorentz kann alle Tandem-Repeat-Paare (bzw. alle Tandem-Repeats) in $t \in \Sigma^n$ in Zeit $O(n\log(n)+k)$ und Platz $O(n)$ finden, wobei k die Anzahl der Tandem-Repeat-Paare (bzw. Tandem-Repeats) ist.*

Wir merken noch Folgendes ohne Beweis an: Die Anzahl primitiver Tandem-Repeat-Paare ist ebenfalls durch $O(n\log(n))$ beschränkt. Die Anzahl so genannter maximaler (verzweigender) Tandem-Repeat-Paare ist jedoch $O(n)$ beschränkt. Wir verweisen hierfür auf die Originalliteratur von R. Kolpakov und G. Kucherov.

3.3.4 Algorithmus von Landau und Schmidt

In diesem Abschnitt wollen wir einen Algorithmus zur Erkennung von Tandem-Repeats mit Fehlern angeben. Dazu zunächst einmal die Definition, welche Fehler wir tolerieren wollen.

Definition 3.37 *Sei $t = t_1 \cdots t_n \in \Sigma^*$. Ein* k-mismatch Tandem-Repeat *der Länge 2ℓ ist ein Paar (i,ℓ), so dass $\delta_H(t_i \cdots t_{i+\ell-1}, t_{i+\ell} \cdots t_{i+2\ell-1}) \leq k$ gilt.*

Hierzu betrachten wir das folgende Beispiel:

$$t = b\,abaaaab\,abbaaa\,b.$$

Das Wort t enthält ein 2-mismatch Tandem-Repeat der Länge 12 mit einem Repeat an Position 2 mit Länge 6. Dabei gibt es jeweils einen Mismatch an den Positionspaaren $(4,10)$ und $(7,13)$.

Wir werden zur Lösung die Idee des Algorithmus von Main und Lorentz wiederverwenden. Wir werden also wieder einen Divide-and-Conquer-Ansatz benutzen. Auch hier müssen wir uns nur um Schritt 3 (bzw. 4) Gedanken machen. Sei also wiederum $t = t_1 \cdots t_n$ und sei $h = \lfloor n/2 \rfloor$ und $q = h + \ell$ für ein festes $\ell \in [1 : \lfloor n/2 \rfloor]$.

Anstatt jetzt an Position h und q nur eine longest common backward extension zu finden, iterieren wir dieses Verfahren jetzt $k+1$ mal. Dazu definieren wir

$$\begin{aligned} \ell_i &:= \mathrm{lce}_b(h - L_{i-1}, q - L_{i-1}) \qquad \text{für} \quad i \in [1 : k+1], \\ L_i &:= \sum_{j=1}^{i} (\ell_i + 1) \qquad \text{für} \quad i \in [0 : k+1]. \end{aligned}$$

Beachte, dass $L_i \geq i$ gilt. Anschaulich ist L_i die Anzahl der Buchstaben, die ab Position h bzw. q nach links gelesen werden muss, bis der i-te Mismatch auftritt. Somit erhalten wir mit

$$(t_{h-L_{k+1}+2} \cdots t_h, t_{q-L_{k+1}+2} \cdots t_q)$$

quasi eine *longest common backward extension with k mismatches*.

Analog führen wir ab den Positionen $h+1$ und $q+1$ ebenfalls $k+1$ longest common forward extension aus. Dazu definieren wir

$$\begin{aligned} \ell'_i &:= \mathrm{lce}_f(h + 1 + L'_{i-1}, q + 1 + L'_{i-1}) \qquad \text{für} \quad i \in [1 : k+1], \\ L'_i &:= \sum_{j=1}^{i} (\ell'_i + 1) \qquad \text{für} \quad i \in [0 : k+1]. \end{aligned}$$

Beachte, dass auch $L'_i \geq i$ gilt. Anschaulich ist L'_i die Anzahl Buchstaben, die ab Position $h+1$ bzw. $q+1$ nach rechts gelesen werden muss, bis der i-te Mismatch auftritt. Somit erhalten wir mit

$$(t_{h+1} \cdots t_{h+L'_{k+1}-1}, t_{q+1} \cdots t_{q+L'_{k+1}-1})$$

quasi eine *longest common forward extension with k mismatches*. Dieser Ansatz zur Lösung von Schritt 3 ist in Abbildung 3.26 noch einmal illustriert.

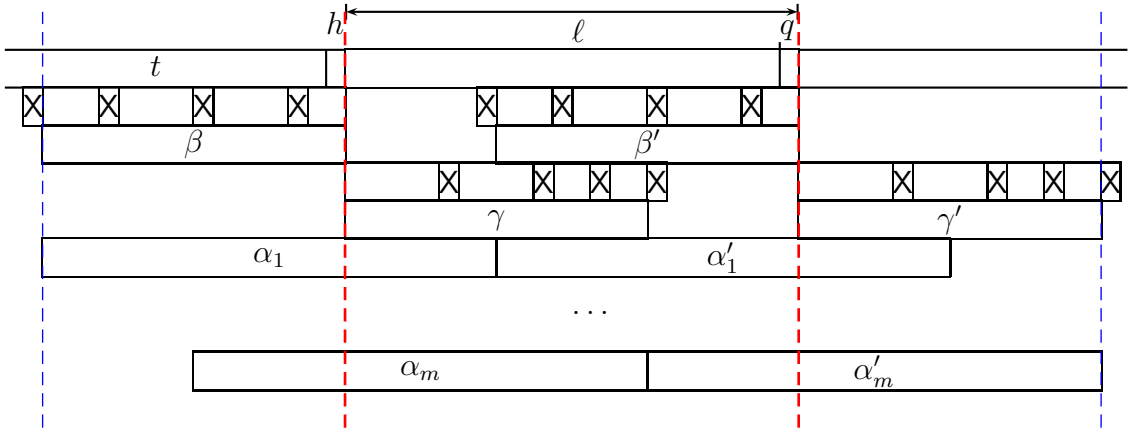


Abbildung 3.26: Skizze: Anwendung von je $k+1$ lce zur Tolerierung $\leq k$ Mismatches

Nur im Intervall $[h-L_{k+1}+2 : q+L'_{k+1}-1]$ kann jetzt ein k-mismatch Tandem-Repeat der Länge 2ℓ auftreten. Damit dieses k-mismatch Tandem-Repeat die Positionen h in der ersten Hälfte beinhaltet (wie im Conquer-Schritt für 3 gefordert), muss es sogar im Intervall $[A : B]$ enthalten sein, wobei $A := \max\{h - L_{k+1} + 2, h - \ell + 2\}$ und $B := \min\{q + L'_{k+1} - 1, h + 2\ell - 1\}$ ist.

Im Gegensatz zum Algorithmus von Main und Lorentz muss jedoch nicht jede in diesem Intervall $[A : B]$ enthaltene Zeichenreihe der Länge 2ℓ ein k-mismatch Tandem-Repeats sein, da sich im Inneren die Mismatches häufen können.

Für $i \in [A : B - 2\ell + 1]$ betrachten wir jetzt für jedes Wort $t_i \cdots t_{i+2\ell-1}$ der Länge 2ℓ die Anzahl $m(i)$ der Mismatches im zugehörigen Tandem-Repeat, d.h.:

$$m(i) := |\{j \in [1 : \ell] \ : \ t_{i+j-1} \neq t_{i+\ell+j-1}\}| = \delta_H(t_i \cdots t_{i+\ell-1}, t_{i+\ell} \cdots t_{i+2\ell-1}).$$

Für $m(i)$ können wir auf jeden Fall festhalten, dass Folgendes gilt:

$$|m(i) - m(i+1)| \leq 1 \qquad \text{für} \quad i \in [A : B - 2\ell + 1].$$

Die Funktion $m(i)$ kann sich auch nur an den Positionen ändern, an denen wir ein explizites Mismatch feststellen. Diese Funktion ist in Abbildung 3.27 illustriert (Es muss hier nicht $m(A) = k$ gelten).

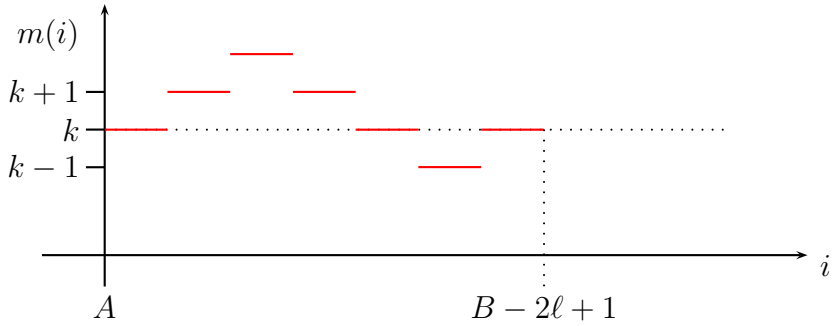


Abbildung 3.27: Skizze: Diagramm der Anzahl Mismatches $m(i)$ bei Teilwörtern der Länge 2ℓ im Kandidaten-Intervall von k-mismatch Tandem-Repeats

22.11.24

Algorithmisch benötigen wir nur die folgenden beiden Listen von Positionen von Mismatches $L = \{h - L_k + 1, \ldots, h - L_1 + 1\}$, die nach vorne herauswandern, und $L' = \{q + L'_1, \ldots, q + L'_k\}$, die von hinten hinzukommen. Somit kann sich $m(i)$ nur dann ändern, wenn wir einen Kandidaten ab einer Position aus

$$\{h - L_k + 1, \ldots, h - L_1 + 1\} \cup \{q + L'_1, \ldots, q + L'_k\}$$

betrachten. Diese beiden Listen sind jeweils aufsteigend sortiert. Sei M die sortierte Vereinigung dieser beiden Listen. Nur an diesen Positionen kann sich $m(i)$ ändern.

Als Tandem-Repeats bzw. Tandem-Repeat-Paar müssen wir also nur für $m \in M$ die Anzahl Mismatches feststellen.

Für das kleinste $m \in M$ kann dies in Zeit in $O(k)$ festgestellt werden. Für die folgenden $m \in M$ muss nur getestet werden, ob vorne ein Mismatch verloren geht oder hinten ein neuer entsteht. Somit ist die Laufzeit für Schritt 3 für ein festes ℓ gerade $O(k)$. Der Conquer-Schritt benötigt also insgesamt Laufzeit $O(kn) + O(z)$, wobei z die Anzahl der ausgegeben k-mismatch Tandem-Repeats (bzw. Paare) ist.

Die Gesamtlaufzeit $T(n)$ des Algorithmus (ohne die Ausgabe der Tandem-Repeats) erfüllt dann die Rekursionsgleichung:

$$T(n) = \begin{cases} O(1) & \text{für} \quad n \leq 2k, \\ T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + O(kn) & \text{für} \quad n > 2k. \end{cases}$$

Die Lösung dieser Rekursionsgleichung lautet $O(kn \log(n/k))$. Somit benötigt der Algorithmus inklusive Ausgabe $O(kn \log(n/k) + z)$, wobei z die Anzahl der k-mismatch Tandem-Repeats ist.

Theorem 3.38 *Alle k-mismatch Tandem-Repeats von $t \in \Sigma^n$ können mit dem Algorithmus von Landau und Schmidt in Zeit $O(kn \log(n/k) + z)$ gefunden werden, wobei z die Anzahl der k-mismatch Tandem-Repeats angibt.*

Wir können die Begriffe links- und rechtsverzweigend auch auf k-mismatch Tandem-Repeat-Paare erweitern.

Definition 3.39 *Sei $k \in \mathbb{N}$ und sei $t = t_1 \cdots t_n \in \Sigma^*$. Ein k-mismatch Tandem-Repeat-Paar $(i, \ell) \in \mathbb{N}^2$ von t heißt* rechtsverzweigend *bzw.* linksverzweigend*, wenn $(i+1, \ell)$ bzw. $(i-1, \ell)$ kein k-mismatch Tandem-Repeat-Paar ist.*

Da wir nun für jedes Intervall maximal $2k$ rechtsverzweigende k-mismatch Tandem-Repeat-Paare ausgeben können, gilt der folgende Satz.

Theorem 3.40 *Alle rechtsverzweigenden k-mismatch Tandem-Repeats von $t \in \Sigma^n$ können mit dem Algorithmus von Landau und Schmidt in Zeit $O(kn \log(n/k))$ gefunden werden.*

R. Kolpakov und G. Kucherov haben einen verbesserten Algorithmus entwickelt, der k-mismatch Tandem-Repeat-Paar in Zeit $O(k \log(k) \cdot n + z)$ auffinden kann. Einige der dazu benötigten Techniken werden im folgenden Abschnitt ebenfalls eingeführt.

G.M. Landau, J.P. Schmidt und D. Sokol haben den Algorithmus auf k-difference Tandem-Repeat-Paare unter der EDIT-Distanz erweitert, wobei die Laufzeit für ihren Algorithmus $O(k \log(k) n \log(n/k) + z)$ beträgt. Dies ist im Wesentlichen eine Verallgemeinerung der hier vorgestellten Technik, wenn auch die Details wesentlich aufwendiger bzw. im angegeben Artikel tatsächlich wohl unvollständig sind. Vermutlich lässt sich das Ergebnis der Laufzeit nur unter enormen Platzaufwand retten, so dass unklar ist, inwieweit dieser Algorithmus praktikabel ist.

3.4 Vokabulare von Tandem-Repeats

In diesem Abschnitt wollen wir noch einen echten Linearzeit-Algorithmus zum Auffinden aller Tandem-Repeats vorstellen, wobei wir als Ausgabe einen mit Tandem-Repeats dekorierten Suffix-Baum generieren werden.

3.4.1 Vokabulare und Überdeckungen

Wir werden zunächst ein paar grundlegende Definitionen angeben, um dann die Idee der Darstellung der Tandem-Repeats und das Prinzip des Algorithmus erläutern zu können.

Definition 3.41 *Sei $t = t_1 \cdots t_n \in \Sigma^*$. Zwei Tandem-Repeat-Paare (i, ℓ) und (i', ℓ') von t sind vom* gleichen Typ*, wenn $\ell = \ell'$ und $t_i \cdots t_{i+\ell-1} = t_{i'} \cdots t_{i'+\ell-1}$ gilt.*

Anschaulich sind zwei Tandem-Repeat-Paare vom gleichen Typ, wenn sie denselben Tandem-Repeat beschreiben.

Wir führen jetzt noch den Begriff des Vokabulars ein, den wir eigentlich schon kennen.

Definition 3.42 *Sei $t \in \Sigma^*$. Die Menge aller Tandem-Repeats $\overline{\mathcal{T}}(t)$ von t bezeichnen wir auch als das* Vokabular $\mathcal{V}(t)$ *von t.*

Wie wir ja bereits gesehen haben, entstehen normale Tandem-Repeats immer durch eine geeignete Anzahl von Linksrotationen aus einem rechtsverzweigenden Tandem-Repeat. Solchen konsekutiven Tandem-Repeats wollen wir zur besseren Orientierung noch einen Namen geben.

Definition 3.43 *Sei $t = t_1 \cdots t_n \in \Sigma^*$. Das Intervall $[i : j] \subseteq [1 : n]$ ist ein* Run von Tandem-Repeats *der Länge 2ℓ von t, wenn $(k, \ell) \in \mathcal{T}(t)$ für alle $k \in [i : j]$.*

Mit Hilfe dieser Runs können wir nun auch beschreiben, wann ein Tandem-Repeat durch mehrere Linksrotationen aus einem anderen Tandem-Repeat hervorgeht.

Definition 3.44 *Sei $t = t_1 \cdots t_n \in \Sigma^*$. Ein Tandem-Repeat-Paar (i, ℓ) von t* überdeckt *ein Tandem-Repeat-Paar (j, ℓ) von t, wenn $[i : j]$ ein Run von Tandem-Repeats der Länge 2ℓ ist.*

Damit können wir sinnvolle Teilmengen der Menge aller Tandem-Repeat-Paare definieren, die uns wieder alle Tandem-Repeat-Paare reproduzieren können.

Definition 3.45 *Sei $t = t_1 \cdots t_n \in \Sigma^*$. Eine Menge $P \subseteq \mathcal{T}(t)$ heißt* Überdeckung *von $\mathcal{T}(t)$, wenn für jedes $w \in \mathcal{V}(t)$ ein $(i, \ell) \in P$ existiert, so dass (i, ℓ) ein Tandem-Repeat-Paar $(j, \ell) \in \mathcal{T}(t)$ mit $w = t_j \cdots t_{j+2\ell-1}$ überdeckt. Gilt zusätzlich für jedes $(j', \ell) \in \mathcal{T}(t)$ mit $w = t_{j'} \cdots t_{j'+2\ell-1}$ die Beziehung $j' \geq j$, dann heißt P eine* linkeste Überdeckung.

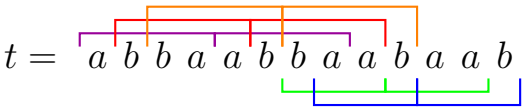


Abbildung 3.28: Beispiel: $t = abbaabbaabaab$

In Abbildung 3.28 sind für die Zeichenreihe $t = abbaabbaabaab$ alle Tandem-Repeats der Länge größer als zwei angegeben. Des Weiteren gilt dort beispielsweise:

- $(1, 4)$ überdeckt $(2, 4)$ und auch $(3, 4)$.
- $(7, 3)$ überdeckt $(8, 3)$.
- $\mathcal{T}(t) = \{(1, 4), (2, 4), (3, 4), (2, 1), (4, 1), (6, 1), (8, 1), (7, 3), (8, 3), (11, 1)\}$.
- $\mathcal{V}(t) = \overline{\mathcal{T}}(t) = \{abbaabba, aa, baabaa, bbaabbaa, bb, aabaab, baabbaab\}$.
- $P = \{(1, 4), (7, 3), (4, 1), (6, 1)\}$ ist eine Überdeckung von $\mathcal{T}(t)$.
- P ist keine linkeste Überdeckung, da das Tandem-Repeat-Paar $(2, 1)$ für bb nicht überdeckt wird.
- $P' = \{(1, 4), (7, 3), (4, 1), (2, 1)\}$ bzw. $P'' = \{(1, 4), (2, 4), (7, 3), (4, 1), (2, 1)\}$ ist jeweils eine linkeste Überdeckung von $\mathcal{T}(t)$, aber auch $\mathcal{T}(t)$ selbst.

Man beachte, dass eine linkeste Überdeckung nicht notwendigerweise eine kleinste Überdeckung der Tandem-Repeat-Paare sein muss. Betrachte dazu Abbildung 3.29. Dort sind für das Wort $t = aabaabbaabaabaa$ alle Tandem-Repeats der Länge größer als zwei dargestellt. Dann ist $P = \{(1,1),(6,1),(8,3),(3,4)\}$ hierfür eine minimale Überdeckung. Diese ist allerdinge keine linkeste Überdeckung, wie man sich leicht überlegt. Des Weiteren ist dort $P' = \{(1,1),(6,1),(1,3),(7,3),(3,4)\}$ beispielsweise eine linkeste Überdeckung.

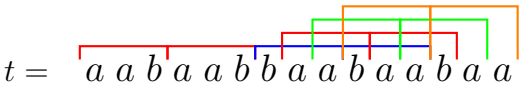


Abbildung 3.29: Beispiel: $t = aabaabbaabaabaa$

Theorem 3.46 *Das Vokabular einer Zeichenreihe der Länge n besitzt maximal $O(n)$ Wörter und kann in einem Suffixbaum repräsentiert werden.*

Der nicht ganz einfache Beweis ist dem Leser als Übungsaufgabe überlassen. Aus dem vorhergehenden Satz folgt sofort das folgenden Korollar.

Korollar 3.47 *Sei $t = t_1 \cdots t_n \in \Sigma^*$. Es gibt eine linkeste Überdeckung der Tandem-Repeat-Paare von t mit Größe $O(n)$.*

Wie schon erwähnt, lässt sich das Vokabular mit Hilfe eines Suffix-Baumes darstellen. Diese Darstellung hat dann offensichtlich eine Platzkomplexität von $O(n)$.

In Abbildung 3.30 ist ein Beispiel für die Darstellung eines Vokabular mit Hilfe eines Suffix-Baumes für $t = abbaabbaabaab$ angegeben. Hierbei sind die Worte des

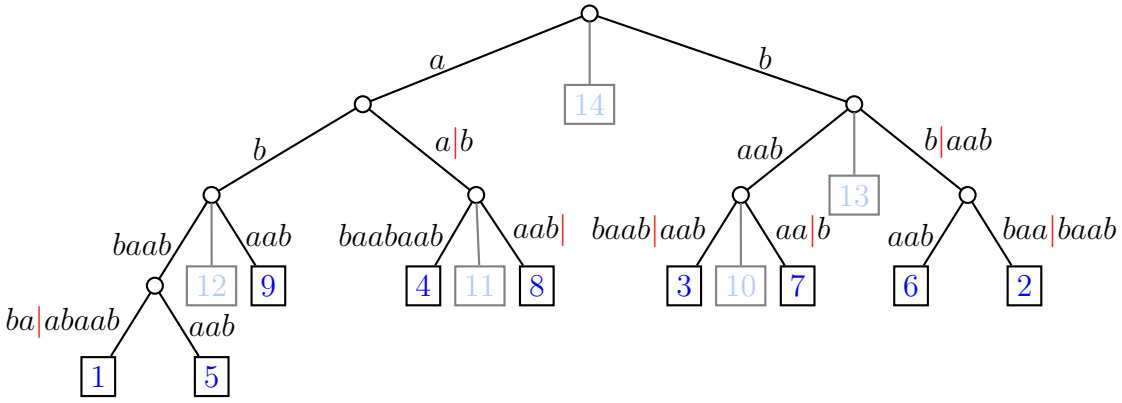


Abbildung 3.30: Beispiel: Der Suffix-Baum zu $abbaabbaabaab$ und seine Dekorierung

Vokabulars als Worte im Suffix-Baum dargestellt, die zu Pfaden von der Wurzel bis zum Trennzeichen | gehen (das auch kurz vor einem Knoten oder Blatt stehen kann). Offensichtlich wird durch den Suffix-Baum dann das folgende Vokabular korrekt dargestellt:

$$\mathcal{V}(t) = \{aa, aabaab, abbaabba, baabaa, baabbaab, bb, bbaabbaa\}.$$

Eigentlich wird dieser Suffix-Baum für $t = abbaabbaabaab\$$ konstruiert, der Übersichtlichkeit halber ist hier bei den Kantenmarkierungen jeweils das \$ am Ende weggelassen worden.

3.4.2 Skizze des Algorithmus von Gusfield und Stoye

Der Algorithmus von Gusfield und Stoye zur Dekorierung eines Suffix-Baumes von t mit dem Vokabular von t wird in drei Phasen vorgehen:

Phase I: Konstruktion einer linkesten Überdeckung P der Tandem-Repeat-Paare $\mathcal{T}(t)$.

Phase II: Konstruktion des Suffix-Baums $T(t\$)$ und Dekoration dieses mit einigen Tandem-Repeats aus der linkesten Überdeckung P.

Phase III: Erweiterung der bisherigen Dekorierung zu einer vollständigen Dekorierung von $T(t)$ mit dem Vokabular von t.

3.4.3 Tandem-Repeats und Lempel-Ziv-Zerlegungen

Bevor wir zur Bestimmung einer linkesten Überdeckung kommen, benötigen wir noch ein Hilfsmittel, die Lempel-Ziv-Zerlegung, sowie einige fundamentale Beziehung von Tandem-Repeats zu diesen.

Notation 3.48 *Sei $t = t_1 \cdots t_n \in \Sigma^*$. Für jede Position $i \in [1:n]$ ist ℓ_i und s_i wie folgt definiert:*

$$\begin{aligned} \ell_i &:= \max\{k \in [0:n-i+1] \,:\, \exists j < i : t_j \cdots t_{j+k-1} = t_i \cdots t_{i+k-1}\}, \\ s_i &:= \min\{j \in [0:i-1] \,:\, t_j \cdots t_{j+\ell_i-1} = t_i \cdots t_{i+\ell_i-1}\}. \end{aligned}$$

Beachte, dass in den Definitionen von ℓ_i bzw. s_i jeweils $k = 0$ bzw. $j = 0$ in den Mengen, aus denen das Maximum bzw. Minimum gebildet wird, enthalten sind. Somit sind die Extrema wohldefiniert.

Anschaulich ist $t_i \cdots t_{i+\ell_i-1}$ die längste Zeichenreihe, die in t ab Position i in t bereits zum zweiten Mal auftritt; s_i gibt dabei an, ab welcher Position diese Zeichenreihe vorher schon einmal aufgetreten ist und ℓ_i gibt deren Länge an. Dabei ist genau dann $s_i = 0$, wenn $\ell_i = 0$ gilt. In Abbildung 3.31 ist dies noch einmal illustriert.

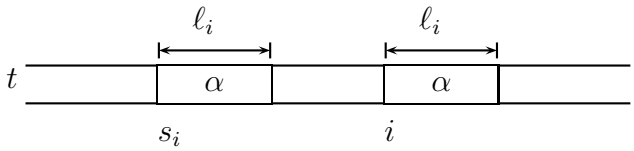


Abbildung 3.31: Skizze: Die Werte ℓ_i und s_i in t

In Abbildung 3.32 sind für das Wort $t = abbaabbaabaab$ noch einmal beispielsweise die Werte für ℓ_i und s_i angegeben.

	a	b	b	a	a	b	b	a	a	b	a	a	b
ℓ_i	0	0	1	1	6	5	4	3	2	4	3	2	1
s_i	0	0	2	1	1	2	3	4	1	3	4	1	2

Abbildung 3.32: Beispiel: Die Werte ℓ_i und s_i für $t = abbaabbaabaab$

Wir halten zunächst die fundamentale Beobachtung fest, dass die Folgenglieder der Folge $(\ell_i)_{i\in\mathbb{N}_0}$ nur um 1 sinken, aber beliebig steigen können.

Beobachtung 3.49 *Für jedes $t \in \Sigma^n$ und jedes $i \in [2 : n]$ gilt, dass $\ell_i \geq \ell_{i-1} - 1$ sowie $\ell_i \leq n - i + 1$ und dass diese Schranken scharf sind.*

Nun können wir die so genannte Lempel-Ziv-Zerlegung einer Zeichenreihe über Σ definieren.

Definition 3.50 *Die* Lempel-Ziv-Zerlegung *einer Zeichenreihe $t = t_1 \cdots t_n \in \Sigma^*$ ist eine Folge von Zahlen $(i_1, \ldots, i_{k+1})$ mit*

$$\begin{aligned} i_1 &:= 1, \\ i_{j+1} &:= i_j + \max(1, \ell_{i_j}) \qquad \text{für} \quad i_j \leq n, \end{aligned}$$

wobei ℓ_i wie oben definiert ist. Das Wort $t_{i_j} \cdots t_{i_{j+1}-1}$ wird als j-ter Block *der Lempel-Ziv-Zerlegung bezeichnet.*

Beachte, dass für eine Lempel-Ziv-Zerlegung eines Wortes $t \in \Sigma^n$ nach Definition immer $i_1 = 1$, $i_2 = 2$ und $i_{k+1} = n + 1$ gilt. Wir geben als Beispiel für das Wort

abbaabbaabaab die Folge $(i_1, \ldots, i_7)$ der Lempel-Ziv-Zerlegung an:

$$\begin{aligned} i_1 &= 1 \\ i_2 &= 1 + \max(1,0) = 2 \\ i_3 &= 2 + \max(1,0) = 3 \\ i_4 &= 3 + \max(1,1) = 4 \\ i_5 &= 4 + \max(1,1) = 5 \\ i_6 &= 5 + \max(1,6) = 11 \\ i_7 &= 11 + \max(1,3) = 14 \end{aligned}$$

In Abbildung 3.33 sind für das Beispiel *abbaabbaabaab* die Lempel-Ziv-Blöcke noch einmal illustriert.

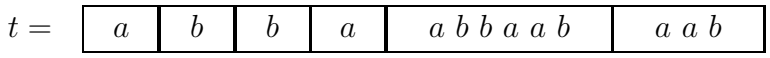


Abbildung 3.33: Beispiel: Lempel-Ziv-Zerlegung von $t =$ *abbaabbabbaab*

26.11.24

Nun können wir einige fundamentale Eigenschaften von Tandem-Repeats in t bezüglich der Lempel-Ziv-Zerlegung von t festhalten.

Lemma 3.51 *Die rechte Hälfte eines Tandem-Repeats kann maximal zwei Lempel-Ziv-Blöcke überlappen.*

Beweis: Für einen Widerspruchsbeweis nehmen wir an, dass die rechte Hälfte eines Tandem-Repeats mindestens drei Lempel-Ziv-Blöcke überlappt. Sei also (i, ℓ) ein Tandem-Repeat-Paar zum Tandem-Repeat $\alpha\alpha$. Dann gibt es ein $j \in [1 : k-1]$, so dass $i_j \in [i+\ell+1, i+\ell-2]$ und $i_{j+1} \in [i+\ell+2, i+\ell-1]$ gilt. Dies ist in Abbildung 3.34 illustriert.

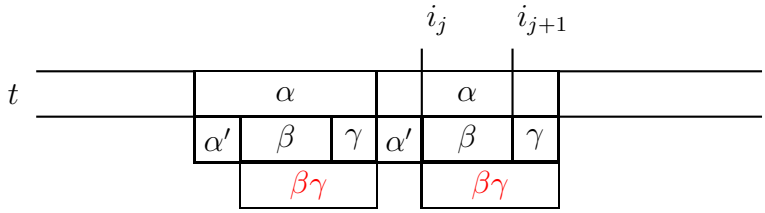


Abbildung 3.34: Skizze: Tandem-Repeats in t und die Lempel-Ziv-Blöcke

Der j-te Block ist dann also β und es gibt $\alpha', \gamma \in \Sigma^+$ mit $\alpha = \alpha'\beta\gamma$. Offensichtlich taucht aber $\beta\gamma$ schon vorher einmal auf, also wäre der j-te Block falsch definiert und wir erhalten den gewünschten Widerspruch. ■

Lemma 3.52 *Das linkeste Vorkommen eines Tandem-Repeats überlappt mindestens zwei Lempel-Ziv-Blöcke.*

Beweis: Für einen Widerspruchsbeweis nehmen wir an, dass es ein linkestes Vorkommen eines Tandem-Repeats gibt, das innerhalb eines Lempel-Ziv-Blocks liegt. Sei $\alpha\alpha$ dieses Tandem-Repeat und sei (i, ℓ) das linkeste Tandem-Repeat-Paar für $\alpha\alpha$ (d.h. mit minimalem i). Sei weiter j so gewählt, dass β der j-te Lempel-Ziv-Block ist, der das Tandem-Repeat-Paar (i, ℓ) enthält. Dies ist auch in Abbildung 3.35 illustriert.

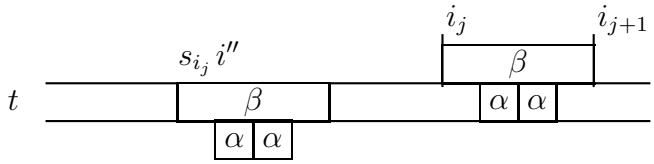


Abbildung 3.35: Skizze: Das linkeste Vorkommen eines Tandem-Repeats in t und die Lempel-Ziv-Zerlegung von t

Nach Definition des j-ten Lempel-Ziv-Blocks muss β ein Teilwort von t sein, dass bereits ab Position $i' < i_j$ schon einmal vorkommt. Somit muss auch $\alpha\alpha$ noch einmal in t an einer Position $i'' < i$ vorkommen, was offensichtlich ein Widerspruch zu der Tatsache ist, dass (i, ℓ) ein linkestes Tandem-Repeat-Paar für $\alpha\alpha$ ist. ■

Für unser weiteres Vorgehen wollen wir noch das Zentrum eines Tandem-Repeats definieren.

Definition 3.53 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ mit der Lempel-Ziv-Zerlegung $(i_1, \ldots, i_{k+1})$ und sei $\alpha\alpha$ ein Tandem-Repeat in t mit $t_i \cdots t_{i+2\ell-1} = \alpha\alpha$ zum Tandem-Repeat-Paar (i, ℓ). Dann ist das* Zentrum *von (i, ℓ) im Block B der Lempel-Ziv-Zerlegung, wenn $i + \ell \in [i_B : i_{B+1} - 1]$.*

Aus den beiden vorhergehenden Lemmata ergibt sich nun der folgende Satz.

Theorem 3.54 *Wenn das linkeste Vorkommen eines Tandem-Repeats $\alpha\alpha$ sein Zentrum im Block B hat, dann gilt:*

- *Entweder ist das linke Ende von $\alpha\alpha$ im Block B und sein rechtes Ende im Block $B + 1$*
- *oder das linke Ende von $\alpha\alpha$ liegt links vom Block B und sein rechtes Ende im Block B oder Block $B + 1$.*

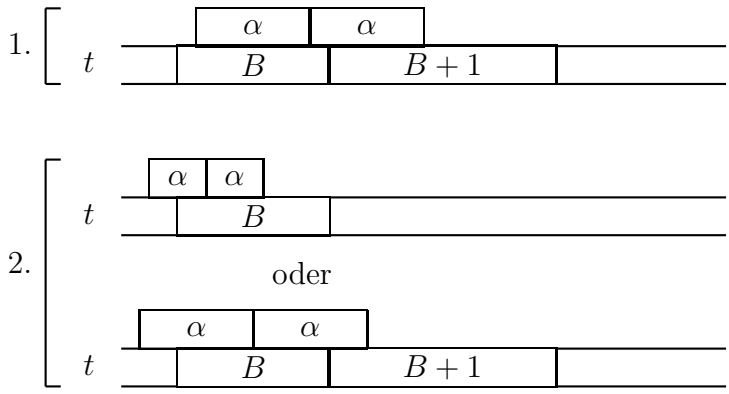


Abbildung 3.36: Skizze: Lage eines Tandem-Repeats $\alpha\alpha$ in t mit Zentrum im Block B zur Lempel-Ziv-Zerlegung von t

Diese verschiedenen Möglichkeiten eines Vorkommens eines Tandem-Repeats ist in der folgenden Abbildung 3.36 noch einmal illustriert.

Wir halten noch als Lemma fest, dass sich für eine Zeichenreihe seine zugehörige Lempel-Ziv-Zerlegung sehr effizient berechnen lässt.

Lemma 3.55 *Sei $t = t_1 \cdots t_n \in \Sigma^*$. Die Lempel-Ziv-Zerlegung von t kann in Zeit $O(n)$ berechnet werden.*

Der Beweis ist dem Leser als Übungsaufgabe überlassen.

3.4.4 Phase I: Bestimmung einer linkesten Überdeckung

In der ersten Phase versuchen wir eine linkeste Überdeckung von $\mathcal{T}(t)$ zu konstruieren. Dazu betrachten wir den Block B der Lempel-Ziv-Zerlegung von t und versuchen alle Tandem-Repeat-Paare, deren Zentrum im Block B liegt und eine der beiden Bedingungen in Theorem 3.54 erfüllt, auszugeben.

Hierzu bezeichne im Folgenden $h := i_B$ bzw. $h' := i_{B+1}$ den Anfang des Blocks B bzw. des Blocks $B + 1$ in der Lempel-Ziv-Zerlegung. Dies ist in Abbildung 3.37 illustriert.

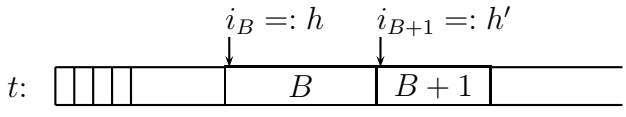


Abbildung 3.37: Skizze: Lempel-Ziv-Blöcke von t

Wir unterscheiden jetzt zwei Fälle, die zu den beiden Charakterisierungen von linkesten Vorkommen von Tandem-Repeats in Theorem 3.54 korrespondieren.

Fall 1: Wir suchen nur nach linkesten Vorkommen von Tandem-Repeats mit Zentrum im Block B, deren linkes Ende sich im Block B und dessen rechtes Ende sich im Block $B+1$ befindet. In diesem Fall gilt für die Halblänge ℓ eines Tandem-Repeats, dass $\ell \in [2 : \ell(B) - 1]$ mit $\ell(B) := i_{B+1} - i_B$. Für $\ell = 1$ kann kein Tandem-Repeat der Länge 2 existieren, dessen Zentrum im Block B und dessen Ende im Block $B+1$ liegt. Weiterhin kann die maximale Länge nur $\ell(B) - 1$ betragen, da sonst ein im Block B beginnendes Tandem-Repeat sein Zentrum bereits im Block $B + 1$ hätte. Weitehin muss die Position h' auf jedem Fall in solch einem Tandem-Repeat sein.

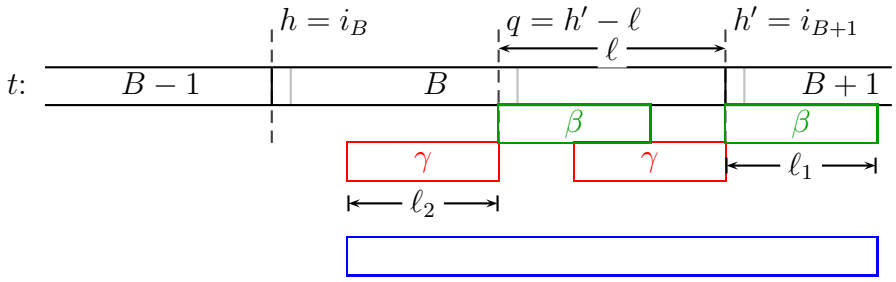


Abbildung 3.38: Skizze: Tandem-Repeats mit Zentrum im Block B, die die erste Bedingung von Theorem 3.54 erfüllen

Dieser Fall ist in Abbildung 3.38 illustriert. Wir setzen $q := h' - \ell$ und bestimmen mit dem Algorithmus für die longest common extensions in konstanter Zeit die folgenden Werte:

$$\begin{aligned} \ell_1 &:= \mathrm{lce}_f(q, h'), \\ \ell_2 &:= \mathrm{lce}_b(q-1, h'-1). \end{aligned}$$

Dabei sei β bzw. γ die longest common forward (bzw. backward) extension ab dem Positionspaar (q, h') bzw. $(q-1, h'-1)$. Aufgrund von Theorem 3.54 wissen wir, dass ein linkestes Tandem-Repeat das Ende von Block B überlappen muss. Es müssen also die folgenden Bedingungen gelten:

$\ell_1 + \ell_2 \geq \ell$: Auch hier gilt wieder, dass wir nur dann einen Tandem-Repeat der Länge 2ℓ gefunden haben, wenn $\ell_1 + \ell_2 \geq \ell$ gilt. Die Tandem-Repeats der Länge 2ℓ befinden sich dann im blauen Block in Abbbildung 3.38.

$\ell_1 > 0$: Gilt weiter $\ell_1 > 0$, dann hat mindestens eines dieser Tandem-Repeats sein rechtes Ende im Block $B + 1$.

$\ell_2 > 0$: Gilt weiter $\ell_2 > 0$, dann gibt es mindestens ein Tandem-Repeat, dessen Zentrum im Block B liegt.

FindLeftmostTRcond1 (block B, LZ $(i_1, \ldots, i_{k+1})$)

```
begin
    for (ℓ ∈ [2 : i_{B+1} − i_B − 1]) do
        h' := i_{B+1};
        q := h' − ℓ;
        ℓ_1 := lce_f(q, h');
        ℓ_2 := lce_b(q − 1, h' − 1);
        if ((ℓ_1 + ℓ_2 ≥ ℓ) && (ℓ_1 > 0) && (ℓ_2 > 0)) then
            return (max{q − ℓ_2, q − ℓ + 1}, ℓ);
end
```

Abbildung 3.39: Algorithmus: Auffinden linkester Tandem-Repeats mit Zentrum im Block B, die die erste Bedingung von Theorem 3.54 erfüllen

Wir geben dann das Tandem-Repeat-Paar $(\max\{q - \ell_2, q - \ell + 1\}, \ell)$ aus. Hierbei ist $(q - \ell_2, \ell)$ ein Tandem-Repeat-Paar, dass alle gefundenen überdeckt (also von diesem Run das linkeste ist). Das Tandem-Repeat-Paar $(q - \ell + 1, \ell)$ ist das erste, das in den Block $B + 1$ hineinragen kann, da

$$(q - \ell + 1) + 2\ell - 1 = q + \ell = (h' - \ell) + \ell = h'$$

gilt. Man beachte, dass aufgrund von $\ell \in [2 : i_{B+1} - i_B - 1]$ das potentielle Tandem-Repeat-Paar $(q - \ell + 1, \ell)$ sein Zentrum im Block B hat, da

$$\begin{aligned}(q - \ell + 1) + \ell &= q + 1 \\ &= h' - \ell + 1 \\ &\in [i_{B+1} - (i_{B+1} - i_B - 1) + 1 : h' - 2 + 1] \\ &= [i_B + 2 : i_{B+1} - 1].\end{aligned}$$

Falls $q - \ell_2 > q - \ell + 1$ ist, gilt aber ebenfalls

$$q - \ell_2 + \ell \leq q - 1 + \ell = h' - \ell - 1 + \ell = h' - 1$$

und somit hat das ausgegebene Tandem-Repeat-Paar sein Zentrum noch im Block B. Der Algorithmus ist in Abbildung 3.39 noch einmal angegeben.

Fall 2: Jetzt suchen wir nach linkesten Vorkommen von Tandem-Repeats mit Zentrum im Block B, deren linkes Ende links vom Block B liegt. In diesem Fall gilt für die Halblänge ℓ des Tandem-Repeats, dass $\ell \in [1 : \ell(B) + \ell(B + 1)]$ mit $\ell(B) = i_{B+1} - i_B$. Weitehin muss die Position h auf jedem Fall in solch einem Tandem-Repeat sein. Dieser Fall ist in Abbildung 3.40 illustriert.

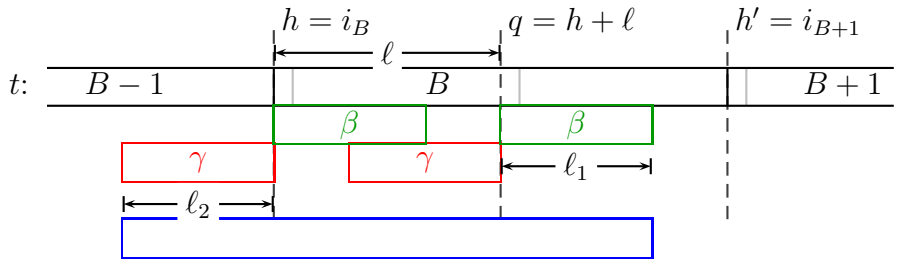


Abbildung 3.40: Skizze: Tandem-Repeats mit Zentrum im Block B, die die zweite Bedingung von Theorem 3.54 erfüllen

Wir setzen $q := h + \ell$ und bestimmen mit dem Algorithmus für die longest common extensions in konstanter Zeit die folgenden Werte:

$$\begin{aligned} \ell_1 &:= \text{lce}_f(h, q), \\ \ell_2 &:= \text{lce}_b(h-1, q-1). \end{aligned}$$

Dabei sei β bzw. γ die longest common forward (backward) extension ab dem Positionspaar (h, q) bzw. $(h-1, q-1)$. Aufgrund von Theorem 3.54 wissen wir, dass ein linkestes Tandem-Repeat den Anfang von Block B überlappen muss. Es müssen also folgende Bedingungen gelten:

$\ell_1 + \ell_2 \geq \ell$: Auch hier gilt wieder, dass es nur dann einen Tandem-Repeat mit Zentrum im Block B geben kann, wenn $\ell_1 + \ell_2 \geq \ell$ gilt. Die Tandem-Repeats der Länge 2ℓ befinden sich dann im blauen Block in Abbbildung 3.40.

$\ell_2 > 0$: Weiter muss $\ell_2 > 0$ sein, damit mindestens ein Tandem-Repeat existiert, dessen linkes Ende links vom Block B liegt.

$\ell_1 \geq 0$: Auch muss $\ell_1 \geq 0$ sein, sonst lässt sich kein Tandem-Repeat mit Zentrum in Block B finden. Dies ist jedoch trivialerweise erfüllt.

$\max\{h - \ell_2, h - \ell\} + \ell < h'$: Damit das Zentrum für das zugehörige Tandem-Repeat wirklich im Block B liegt (hier gilt ja zunächst nur $\ell \leq i_{B+2} - i_B$ und somit $\max\{h - \ell_2, h - \ell\} + \ell \leq i_{B+2} - \min\{\ell_2, \ell\}$).

Als Tandem-Repeat-Paar geben wir dann $(\max\{h - \ell_2, h - \ell\}, \ell)$ aus. Hierbei ist $(h - \ell_2, \ell)$ ein Tandem-Repeat-Paar, dass alle gefundenen überdeckt (also von diesem Run das linkeste ist). Das Tandem-Repeat-Paar $(h - \ell, \ell)$ ist das linkeste, dessen Zentrum gerade noch im Block B liegt, da $(h - \ell) + \ell = h = i_B$. Der Algorithmus ist in Abbildung 3.41 noch einmal angegeben.

FindLeftmostTRcond2 (block B, LZ $(i_1, \ldots, i_{k+1})$)

```
begin
    for (ℓ ∈ [1 : i_{B+2} − i_B]) do
        h := i_B;
        h' := i_{B+1};
        q := h + ℓ;
        ℓ_1 := lce_f(h, q);
        ℓ_2 := lce_b(h − 1, q − 1);
        if ((ℓ_1 + ℓ_2 ≥ ℓ) && (ℓ_2 > 0) && (h − min{ℓ_2, ℓ} < h' − ℓ)) then
            return (max{h − ℓ_2, h − ℓ}, ℓ);
end
```

Abbildung 3.41: Algorithmus: Auffinden linkester Tandem-Repeats mit Zentrum im Block B, die die zweite Bedingung von Theorem 3.54 erfüllen

Kommen wir nun zur Laufzeitabschätzung.

Lemma 3.56 *Die beiden oben angegebenen Algorithmen liefern eine linkeste Überdeckung in Zeit $O(n)$.*

Beweis: Die Korrektheit haben wir bereits bewiesen, da wir nach Theorem 3.54 jedes linkeste Vorkommen eines Tandem-Repeats mit einem Tandem-Repeat-Paar überdeckt haben.

Die Laufzeit zur Bearbeitung eines Blockes B lässt sich mit $c \cdot (\ell(B) + \ell(B+1))$ für eine geeignete Konstante c abschätzen. Für die Laufzeit $T(n)$ gilt dann, wobei $(i_1, \ldots, i_{k+1})$ eine Lempel-Ziv-Zerlegung von t und $\mathcal{B} = \{B_1, \ldots, B_k\}$ die Menge aller Lempel-Ziv-Blöcke von t ist:

$$\begin{aligned} T(n) &\leq \sum_{j=1}^{k-1} c \cdot (\ell(B_j) + \ell(B_{j+1})) + c \cdot (\ell(B_k)) \\ &= \sum_{j=1}^{k-1} c \cdot (i_{j+1} - i_j + i_{j+2} - i_{j+1}) + c \cdot (i_{k+1} - i_k) \\ &= \sum_{j=1}^{k-1} c \cdot (i_{j+2} - i_j) + c \cdot (i_{k+1} - i_k) \\ &= c \cdot (i_{k+1} + i_k - i_2 - i_1) + c \cdot (i_{k+1} - i_k) \end{aligned}$$

$$
\begin{aligned}
&= c \cdot (2i_{k+1} - i_2 - i_1) \\
&\quad \text{da } i_1 = 1,\ i_2 = 2 \text{ und } i_{k+1} = n+1 \\
&\leq c \cdot (2(n+1) - 2 - 1) \\
&\leq 2cn.
\end{aligned}
$$

Damit ist das Lemma bewiesen. ■

Aus diesem Lemma erhalten wir unmittelbar das folgende Korollar.

Korollar 3.57 *Sei $t \in \Sigma^n$. Die konstruierte linkeste Überdeckung von $\mathcal{T}(t)$ hat eine Größe von $O(n)$.*

Im Folgenden führen wir noch eine Partition der Menge P ein, die die angegebenen Algorithmen ohne besonderen Mehraufwand generieren werden.

Notation 3.58 *Sei $t \in \Sigma^n$ und sei $P \subseteq \mathcal{T}(t)$ eine Überdeckung von $\mathcal{T}(t)$. Dann ist*

$$P(i) := \{(i, \ell) \ : \ (i, \ell) \in P\}$$

und es gilt offensichtlich $P = \bigcup_{i=1}^{n} P(i)$.

Mit dieser Notation können wir zeigen, dass die angegebenen Algorithmen auch die Mengen $P(i)$ als sortierte Listen generieren können.

Lemma 3.59 *Der Algorithmus aus Phase I kann so modifiziert werden, dass er die Partition $P(1), \ldots, P(n)$ in Zeit $O(n)$ ausgibt und dabei jedes $P(i)$ als aufsteigend sortierte Liste (nach der Länge der zugehörigen Tandem-Repeats) erzeugt wird.*

Beweis: Wir müssen bei der Ausgabe nur Folgendes beachten: Jedes neu generierte Tandem-Repeat-Paar (i, ℓ) wird an das Ende der Liste $P(i)$ angehängt.

Wir betrachten jetzt ein festes, neu erzeugtes Tandem-Repeat $\alpha\alpha$ und sein zugehöriges Tandem-Repeat-Paar (i, ℓ) mit Zentrum im Block B. Sei $\alpha'\alpha'$ ein weiteres Tandem-Repeat mit zugehörigem Tandem-Repeat-Paar (i, ℓ') in $P(i)$. Liegt dessen Zentrum in einem Block B' links von B, dann muss $\alpha\alpha$ offensichtlich länger als $\alpha'\alpha'$ sein, wie man der Abbildung 3.42 entnehmen kann

Sei nun $\alpha'\alpha'$ ein weiteres Tandem-Repeat mit zugehörigem Tandem-Repeat-Paar $(i, \ell') \in P(i)$ und Zentrum in B. Da in beiden Algorithmen die Längen von Tandem-Repeats in aufsteigender Größe betrachtet werden, folgt die Behauptung sofort. Bei

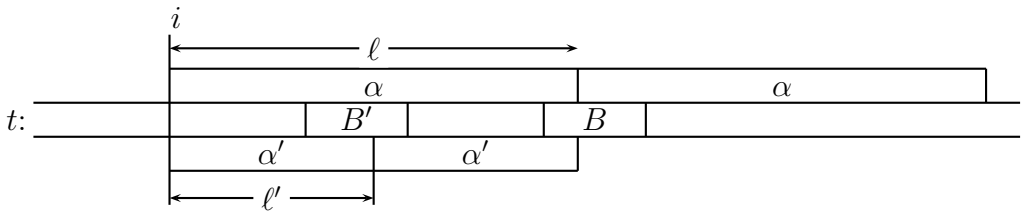


Abbildung 3.42: Skizze: Zwei Tandem-Repeats, die an Position i starten und Zentren in verschiedenen Blöcken besitzen

der Implementierung muss nur darauf geachtet werden, dass die beiden Algorithmus nicht hintereinander, sondern verschränkt ausgeführt werden müssen, d.h. beide Algorithmen müssen in eine Schleife über ℓ integriert werden. ■

Für das Folgende werden wir jedoch benötigen, dass die Listen $P(i)$ absteigend sortiert sind.

Korollar 3.60 *Der Algorithmus aus Phase I kann so modifiziert werden, dass er die Partition $P(1), \ldots, P(n)$ in Zeit $O(n)$ ausgibt und jedes $P(i)$ als absteigend sortierte Liste (nach der Länge der zugehörigen Tandem-Repeats) erzeugt wird.*

Beweis: Die erzeugten Tandem-Repeat-Paare müssen nur am Beginn statt am Ende an die Listen angehängt werden. ■

3.4.5 Phase II: Dekorierung einer Teilmenge (+)

Nun wollen wir einige Tandem-Repeat-Paare aus der Menge der in Phase I konstruierten linkesten Überdeckung P im Suffix-Baum $T(t\$)$ markieren.

Sei $t \in \Sigma^n$ und $T = T(t\$)$ der zugehörige Suffix-Baum. Mit $P(v)$ bezeichnen wir für $v \in V(T)$ eine Liste von Tandem-Repeat-Paaren. Für jedes Blatt $v \in V(T)$ soll dabei $P(v) = P(i)$ für $v = \overline{t_i \cdots t_n\$}$ gelten. Für interne Knoten $v \in V(T)$ hätten wir gerne die folgende Zuordnung:

$$P(v) = \left\{ (j,\ell) \ : \ \exists i. \overline{t_i \cdots t_n\$} \in V(T(v)) : (j,\ell) \in P(i) \wedge 2\ell \leq |\mathrm{path}(v)| \right\},$$

wobei $T(v)$ den am Knoten v gewurzelten Teilbaum von T bezeichnet. Anschaulich beinhaltet dann $P(v)$ alle Tandem-Repeat-Paare aus P, deren zugehörige Tandem-Repeats Präfixe von $\mathrm{path}(v)$ sind. Wir nehmen dabei an, dass die Menge $P(v)$ als Liste dargestellt wird, wobei die Listenelemente absteigend nach der Länge der

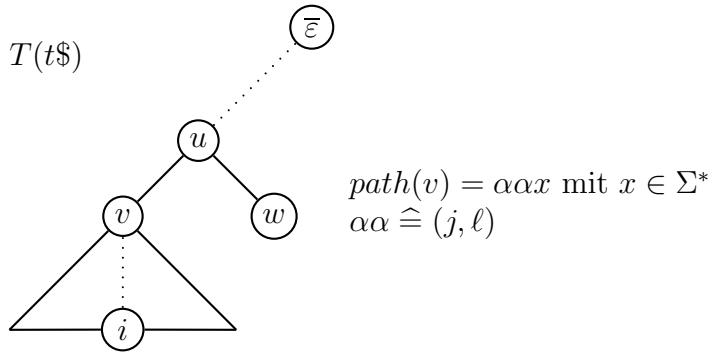


Abbildung 3.43: Skizze: Zur idealen Definition von $P(v)$

Tandem-Repeats sortiert sind. Die Definition ist in Abbildung 3.43 noch einmal illustriert.

Der Algorithmus zur Dekorierung mit den Listen $P(v)$ in $T = T(t\$)$ ist in Abbildung 3.44 angegeben. Hierbei ist zunächst angenommen, dass die Listen $P(v)$ wie oben ideal gegeben sind. Man sieht leicht, dass die Listen $P(v)$ dabei korrekt konstruiert werden, wenn diese aus den Restlisten der Kinder des Knotens v richtig

DecorateTree (tree $T = T(t\$)$)

```
begin
    foreach (v ∈ V(T)) do                                   /* using DFS */
        // after returning from all subtrees
        if (v = t_i ··· t_n$) then                          /* v is a leaf */
            P(v) := P(i);
        else
            generate P(v) (or P'(v));                       /* for details see text */
        u := parent(v);
        while (P(v) ≠ ∅) do
            (i, ℓ) := head(P(v));
            if (2ℓ > |path(u)|) then
                add position mark (2ℓ − |path(u)|) at edge (u, v);
                P(v) := tail(P(v));
            else
                break ;                                     /* exit while-loop */
end
```

Abbildung 3.44: Algorithmus: Dekoration von $T(t\$)$ mit einer Teilmenge der Tandem-Repeats aus der linkesten Überdeckung von Phase I

zusammengemischt werden, so dass die Listen $P(v)$ auch absteigend nach der Länge der korrespondieren Tandem-Repeats sortiert sind.

Für die Dekorierung müssen wir nur die Tandem-Repeat-Paare (i, ℓ) von der Liste $P(v)$ entfernen, für die $2\ell > |\text{path}(u)|$ gilt. Dies sind genau diejenigen, die auf der Kanten (u, v) enden, wobei u der Elter von v ist. Hier wird nun klar, warum die erzeugten Listen aus Phase I absteigend sortiert sein sollten. Wir müssen ja immer die Tandem-Repeat-Paare mit der längsten Halblänge zuerst entfernen.

Idealerweise würde man, nachdem alle Kinder von u abgearbeitet wurden, die restlichen Listen der Kinder von u zur Liste $P(u)$ mischen. Leider kann dies zu einer Gesamtlaufzeit von $O(n^2)$ führen, da das Mischen selbst schon lineare Zeit kosten kann und man im schlimmsten Fall zu oft mischen muss.

Wir werden jetzt folgenden Trick verwenden, um bei einer linearen Laufzeit zu bleiben. Wir wählen als Liste $P'(v)$ für einen inneren Knoten $v \in V(T)$ gerade $P'(v) := P'(w)$, wobei w ein Kind von v ist. Dabei wählen wir das Kind w so aus, dass $T(w)$ das Blatt mit der kleinsten Nummer (Index-Position eines Suffixes von t) enthält. Die Listen $P'(v)$ für die Blätter werden genauso wie bei den Listen $P(v)$ gewählt. Anders ausgedrückt ist die Liste $P'(v)$ eines inneren Knotens genau gleich der Liste des Blattes im Teilbaum von v, dessen zugehörige Indexposition minimal ist.

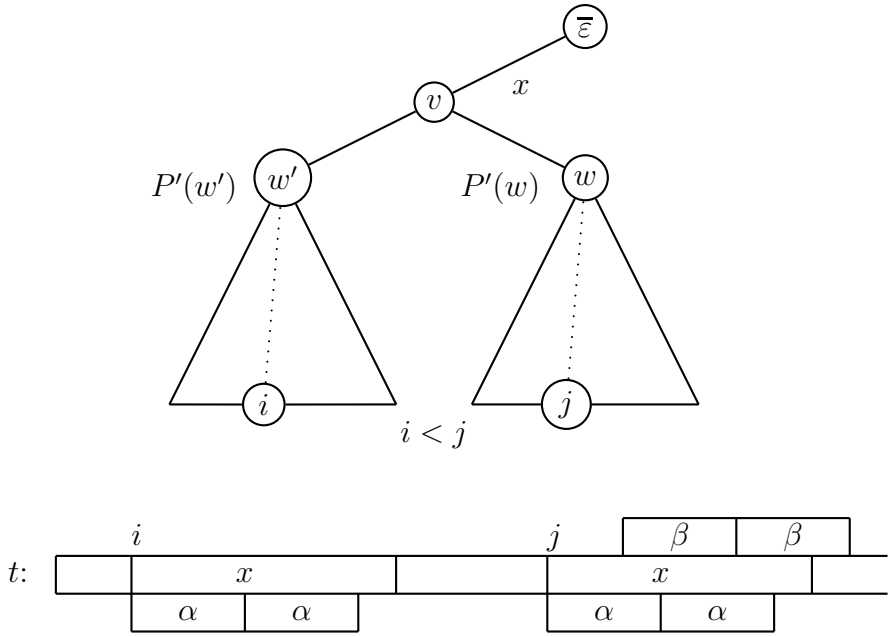


Abbildung 3.45: Skizze: Intuitive Korrektheit der Definition von $P'(u)$ als Liste des linkesten Kindes von u

Warum verschenken wir hierbei keine wertvolle Information? Nehmen wir dazu an, wir wollen $P'(v)$ bestimmen. Seien w und w' zwei Kinder von v, wobei w' das Kind ist, das das Blatt mit dem längsten Suffix enthält. Sei weiter $x = \text{path}(v)$ und $(j, \ell) \in P'(w)$ Dann beschreibt (j, ℓ) einen Tandem-Repeat, der ein Präfix von x ist. Siehe dazu auch Abbildung 3.45.

Da x sowohl als Teilwort von t ab Position i als auch ab Position j vorkommt, wird (j, ℓ) auch von $(i, \ell) \in P(w')$ dargestellt. Damit wird also das Tandem-Repeat zu (j, ℓ) weiterhin dargestellt und (j, ℓ) kann anscheinend ohne Informationsverlust weggelassen werden. Es ist jedoch möglich, dass andere Tandem-Repeat-Paare wie z.B. $\beta\beta$ in Abbildung 3.45, die von (j, ℓ) überdeckt werden, nun von (i, ℓ) nicht mehr überdeckt werden.

Der letzte Fall kann tatsächlich eintreten, wie das folgende Beispiel zeigt. Sei dazu $t = abbabbaabbabbab$, dann ist z.B. $P = \{(1, 3), (9, 3), \ldots\}$ eine linkeste Überdeckung. Hier würde am Knoten $w = \overline{abbabba}$ die Beziehung $(1, 3) \in P'(w)$ gelten und damit würden $abbabb$ an der Kante $(\overline{abba}, \overline{abbabba})$ dekoriert. Das Tandem-Repeat-Paar $(9, 3)$ würde jedoch am Knoten $v = \overline{bbabba}$ verloren gehen, d.h. $(9, 3) \notin P'(w)$. Damit wäre zwar das Tandem-Repeat $abbabb$ dargestellt, aber die Überdeckung $babbab$ vom Tandem-Repeat-Paar $(10, 3)$ durch das Tandem-Repeat-Paar $(9, 3)$ ginge verloren. Dieses Beispiel ist noch einmal in Abbildung 3.46 illustriert.

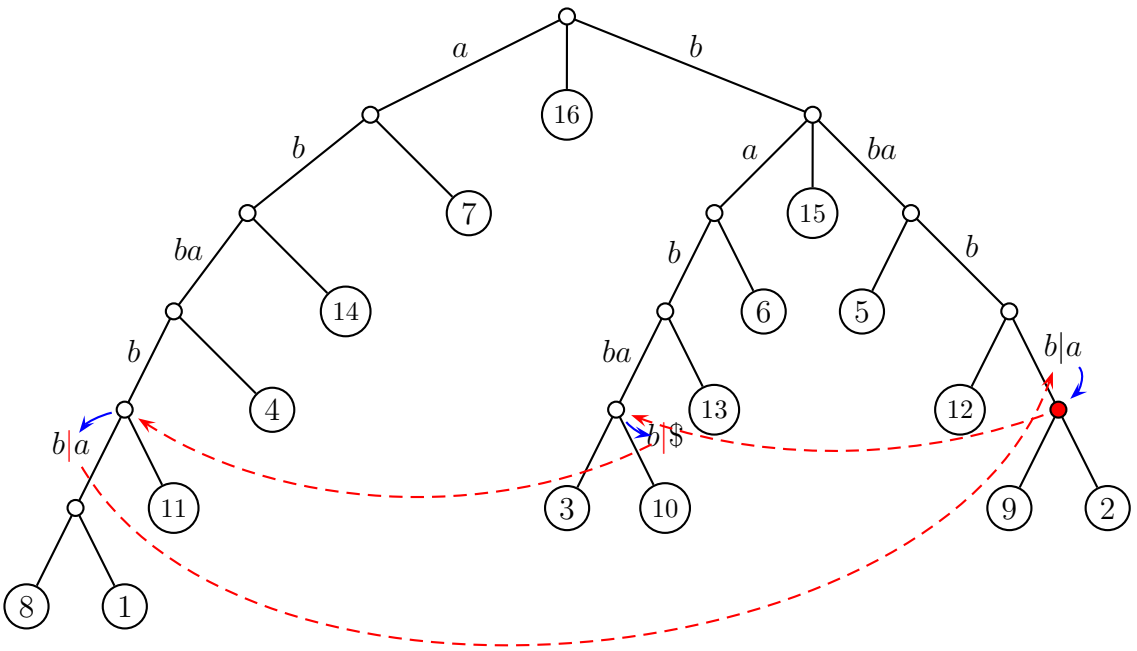


Abbildung 3.46: Beispiel: Auffinden der verlorenen Überdeckung mit Hilfe von Suffix-Link-Walks in $S(abbabbaabbabbab\$)$

Schauen wir uns das Beispiel in Abbildung 3.46 noch einmal an. Wir haben das Tandem-Repeat-Paar $(9, 3)$ in der Liste nicht berücksichtigt und damit die Dekorierung von *babbab* aufgegeben und stattdessen nur das Tandem-Repeat-Paar $(1, 3)$ beibehalten und damit den Tandem-Repeat *abbabb* dekoriert.

Was passiert nun, wenn wir die Rechtsrotation von *abbabb* betrachten, dies wäre das Tandem-Repeat *bbabba*. Wir können dieses im Suffix-Baum finden, indem wir von der Lokation für *abbabb* zuerst dem (eventuell virtuellen) Suffix-Link folgen und müssen dann das Zeichen b noch ablaufen. Im Beispiel in Abbildung 3.46 sind solche (virtuellen) Suffix-Links durch gestrichelte rote Pfeile und das Ablaufen eines Zeichens durch blaue Pfeile dargestellt. Wie man sieht würden wir den Knoten $\overline{bbabba}$ erreichen.

Eine weitere Rechtsrotation liefert das Tandem-Repeat *babbab*. Da auch diese Lokation im Suffix-Baum existiert, haben wir damit auch das nicht überdeckte Tandem-Repeat *babbab* dekoriert. Eine weitere Rechtsrotation führt uns zur Ausgangslokation *abbabb* zurück.

Somit können wir die Hoffnung haben, dass sich ausgehend von den dekorierten Tandem-Repeats durch Rechtsrotationen (Folgen von Suffix-Links von Lokationen und Ablaufen des zugehörigen Zeichens im Suffix-Baum) alle Tandem-Repeats dekorieren lassen. Diese Idee, nicht dekorierte Tandem-Repeats, aufzufinden wollen wir nun noch formalisieren.

28.11.24

Definition 3.61 *Sei $t \in \Sigma^*$ und $T = T(t\$)$ der zugehörige Suffix-Baum. Sei weiter $\mathrm{loc}(aw)$ eine Lokation im Suffix-Baum T mit $a \in \Sigma$ und $w \in \Sigma^*$. Das Ablaufen von $\mathrm{loc}(aw) \to \mathrm{loc}(w) \to \mathrm{loc}(wa)$ im Baum T heißt* Suffix-Link-Walk *in T. Er heißt* erfolgreich*, wenn $\mathrm{loc}(wa)$ existiert, d.h. $wa \sqsubseteq t$, und* erfolglos *sonst. Eine aufeinander folgende Folge von Suffix-Link-Walks heißt* Kette von Suffix-Link-Walks*.*

Nun formalisieren wir Mengen von Tandem-Repeats, die wir in Phase II gerne im Suffix-Baum T dekorieren würden.

Definition 3.62 *Sei $t \in \Sigma^*$ und $T = T(t\$)$ der zugehörige Suffix-Baum. Eine Teilmenge $Q \subseteq \mathcal{V}(t)$ heißt* ausreichend*, wenn für jedes Wort $w \in \mathcal{V}(t)$ die zugehörige Lokation in T durch eine Kette von Suffix-Links-Walks von einer Lokation in T, die zu einem Wort aus Q gehört, erreicht werden kann.*

Haben wir in Phase II eine ausreichende Menge von Tandem-Repeats markiert, dann können wir das Vokabular im Suffix-Baum mit Hilfe von Suffix-Link-Walks

vervollständigen. Wir zeigen jetzt, dass die in Phase II dekorierte Menge tatsächlich ausreichend ist.

Theorem 3.63 *Die dekorierte Teilmenge $Q' \subseteq \mathcal{V}(t)$ aus Phase II ist ausreichend.*

Beweis: Sei $Q \subseteq \mathcal{V}(t)$ die zur linkesten Überdeckung P von $\mathcal{T}(t)$ gehörige Menge von Tandem-Repeats, also $Q = \{t_i \cdots t_{i+2\ell-1} : (i,\ell) \in P\}$. Nach Konstruktion ist Q ausreichend. Sei weiter $P' \subseteq P$ bzw. $Q' \subseteq Q$ die Teilmenge von Tandem-Repeat-Paaren bzw. Tandem-Repeats, die in Phase II im Suffix-Baum T zur Dekoration verwendet bzw. dekoriert wurden, also $Q' = \{t_i \cdots t_{i+2\ell-1} : (i,\ell) \in P'\}$. Zum Schluss sei $Q'' \subseteq \mathcal{V}(t)$ die Teilmenge von Tandem-Repeats, die nicht mit Hilfe von Ketten von Suffix-Link-Walks beginnend an einem Tandem-Repeat aus Q' erreicht werden kann. Für den Beweis des Satzes genügt es dann also zu zeigen, dass $Q'' = \emptyset$.

Für einen Widerspruchsbeweis nehmen wir an, dass $Q'' \neq \emptyset$. Sei wie oben $P' \subseteq P$ die Teilmenge von Tandem-Repeat-Paaren, die genau den Wörtern aus Q' entsprechen, d.h. die Menge der Tandem-Repeat-Paare, die zur Dekoration in Phase II verwendet wurden. Weiter sei $P'' := P \setminus P'$. Die Tandem-Repeats zu den Tandem-Repeat-Paaren aus P'' wurden nicht direkt dekoriert, können aber selbst von Tandem-Repeat-Paaren aus P' dekoriert worden sein. Auf jeden Fall werden nach Konstruktion alle nicht dekorierten Tandem-Repeats aus Q'' von Tandem-Repeat-Paaren aus P'' überdeckt. Beachte hierbei, dass gilt $P'' = \emptyset \Rightarrow Q'' = \emptyset$ und daraus sofort folgt, dass $Q'' \neq \emptyset \Rightarrow P'' \neq \emptyset$.

Sei jetzt $(j,\ell) \in P''$ mit kleinster Startposition j, so dass das zugehörigen Tandem-Repeat $\alpha\alpha = t_j \cdots t_{j+2\ell-1}$ nicht in Phase II dekoriert wurde und auch nicht mit einer Kette von Suffix-Link-Walks aus Q' erreicht werden kann. Wir unterscheiden jetzt zwei Fälle, je nachdem, ob (j,ℓ) das linkeste Auftreten des Tandem-Repeats $\alpha\alpha$ beschreibt oder nicht.

Fall 1: Sei also (j,ℓ) das linkeste Auftreten von $\alpha\alpha$ in t. Wir betrachten den Knoten $v \in V(T)$ mit minimaler Worttiefe, so dass $\alpha\alpha$ ein Präfix von $\text{path}(v)$ ist. Dann beschreibt jedes Blatt in $T(v)$ einen Suffix von t der mit $\alpha\alpha$ beginnt. Also muss j das Blatt mit der kleinsten Index-Position im Teilbaum $T(v)$ sein. Nach Konstruktion ist (j,ℓ) in einer Liste, die in Phase II auf dem Weg vom Blatt, das zum Index j gehört, zum Konten v nicht gelöscht wird. Daher ist $(j,\ell) \in P'(v) \subseteq P'$ und wird daher in T dekoriert. Das liefert den gewünschten Widerspruch.

Fall 2: Sei also (j,ℓ) nicht das linkeste Auftreten von $\alpha\alpha$ in t. Sei also (i,ℓ) mit $i < j$ das linkeste Auftreten von $\alpha\alpha$ in t. Dann muss nach Konstruktion ein $(h,\ell) \in P$ existieren, so dass (h,ℓ) das Paar (i,ℓ) überdeckt. Also gilt $h \leq i < j$. Weiterhin

gilt $(h, \ell) \in P''$, sonst wäre $\alpha\alpha \notin Q''$ (da ansonsten $\alpha\alpha$ durch eine Kette von Suffix-Link-Walks dekoriert wurde dekoriert worden wäre). Dies ist in Abbildung 3.47 noch einmal illustriert.

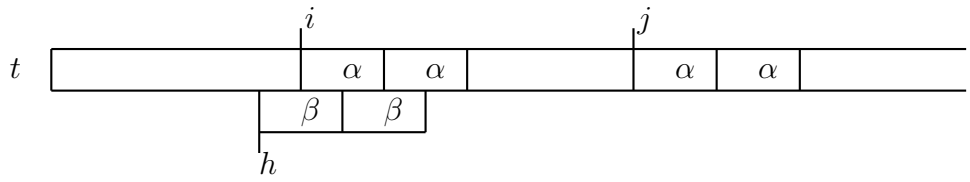


Abbildung 3.47: Skizze: (j, ℓ) beschreibt nicht das linkeste Vorkommen von $\alpha\alpha$ in t

Nach Wahl von j gilt, dass entweder $(h, \ell) \in P'$ oder dass das zu (h, ℓ) gehörigen Tandem-Repeat dekoriert wurde (wegen $h < j$). In beiden Fällen wird das zu (h, ℓ) gehörige Tandem-Repeat letztendlich dekoriert. Da (h, ℓ) aber (i, ℓ) überdeckt und nach Definition dann $[h : i]$ ein Run ist, muss das zu (i, ℓ) gehörige Tandem-Repeat durch eine Kette von Suffix-Link-Walks ausgehend vom zum (h, ℓ) gehörigem Tandem-Repeat, das in Phase II dekoriert wurde, erreicht werden. Damit erhalten wir wieder den gewünschten Widerspruch. ■

3.4.6 Phase III: Vervollständigung der Dekorierung von $\mathcal{V}(t)$ (+)

Im letzten Abschnitt haben wir gezeigt, dass die in Phase II dekorierten Tandem-Repeats in T eine ausreichende Menge darstellen. Wir müssen also in Phase III diese nur noch mit Hilfe von Ketten von Suffix-Link-Walks vervollständigen. Der Algorithmus zur Vervollständigung der Dekorierung von $\mathcal{V}(t)$ geht dabei wie folgt vor.

1. Durchlaufe $T(t\$)$ mittels einer Tiefensuche.
2. An dekorierten Lokation aus Q' in T startet eine Kette von Suffix-Link-Walks bis entweder ein erfolgloser Suffix-Link-Walk auftritt oder der Suffix-Link-Walk auf eine bereits erfolgte Dekorierung aus Phase II oder Phase III (also aus Schritt 2 dieses Algorithmus) trifft.
3. Dabei werden alle neu besuchten Lokationen von Tandem-Repeats an den zugehörigen Kanten dekoriert.

Das folgende Lemma hält nun fest, dass wir mit diesem Algorithmus wirklich alle Tandem-Repeats im Suffix-Baum dekorieren, insbesondere auch dann, wenn wir die Ketten von Suffix-Link-Walks an in Phase III dekorierten Lokationen abbrechen.

Lemma 3.64 *Der Algorithmus aus Phase III vervollständigt eine Dekorierung aus Phase II zum Vokabular der gegebenen Zeichenreihe im zugehörigen Suffix-Baum.*

Beweis: Wir haben bereits gezeigt, dass die Menge Q' ausreichend ist. Endet ein Suffix-Link-Walk erfolglos, dann haben wir alle von diesem Tandem-Repeat durch Suffix-Link-Walks erreichbaren Tandem-Repeats in $\mathcal{V}(t)$ gefunden.

Endet ein Suffix-Link-Walk erfolgreich an einer Lokation, die bereits in Q' ist, können wir auch aufhören, da die folgenden Rechtsrotationen des zuletzt aufgefunden Tandem-Repeats von dieser Lokation in Q' gefunden werden oder wurden.

Dummerweise könnte ein Suffix-Link-Walk auch an einer dekorierten Lokation in T enden, die nicht zu Q' gehört. Wir werden jetzt zeigen, dass dies jedoch nicht passieren kann. Für einen Widerspruchsbeweis nehmen wir an, dass ein Suffix-Link-Walk, gestartet an einem Tandem-Repeat in Q', an einer dekorierten Lokation endet, die kein Tandem-Repeat in Q' beschreibt.

Eine solche Situation kann nur eintreten, wenn wir beim Folgen eines Suffix-Link-Walks eine nachträgliche Dekorierung erwischen, die wir bei einem früheren Suffix-Link-Walk dekoriert hatten. Sei $wawa$ mit $a \in \Sigma$ und $w \in \Sigma^*$ ein Tandem-Repeat, an dessen Lokation in T sich zwei Suffix-Link-Walks treffen. Nach Definition der Suffix-Links, müssen aber beide von der Lokation gekommen sein, die zum Tandem-Repeats $awaw$ gehört hat. Somit können sich zwei Suffix-Link-Walks nur treffen, wenn einer davon einer leerer Suffix-Link-Walk ist und wir erhalten den gewünschten Widerspruch. ■

In den Übungen wird das folgende Lemma bewiesen.

Lemma 3.65 *Sei $t \in \Sigma^n$. An jeder Position $i \in [1:n]$ können in t maximal zwei rechteste Vorkommen eines Tandem-Repeats beginnen.*

Damit erhalten wir sofort eine Aussage über die Größe des Vokabulars einer Zeichenreihe.

Korollar 3.66 *Sei $t \in \Sigma^n$. Dann besitzt t maximal $2n$ viele Tandem-Repeats.*

Weiter erhalten wir eine Aussage über die Anzahl von Dekorierungen pro Kante im zugehörigen Suffix-Baum.

Korollar 3.67 *Sei $t \in \Sigma^*$ und $T = T(t\$)$ der zugehörige Suffix-Baum. Jede Kante $(u, v) \in E(T)$ besitzt maximal zwei Dekorierungen.*

Beweis: Bezeichne $R(v)$ für jeden Knoten $v \in V(T)$ die maximale Startposition eines Suffixes von t, dessen zugehöriges Blatt im Teilbaum $T(v)$ liegt. Betrachten wir jetzt eine Kante $(u, v) \in E(T)$. Das rechteste Vorkommen der dort endenden Tandem-Repeats muss auch ab Position $R(v)$ beginnen. Da dort nach dem vorhergehenden Lemma nur zwei beginnen können, können auf dieser Kante nur zwei Tandem-Repeats enden. ■

Damit können wir nun die Laufzeit des Algorithmus von Gusfield und Stoye analysieren.

Lemma 3.68 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und $T = T(t\$)$ der zugehörige Suffix-Baum. Die Anzahl der traversierten Kanten in T in Phase III durch die Suffix-Link-Walks ist beschränkt durch $O(|\Sigma| \cdot n)$.*

Beweis: Da es nur maximal $2n$ Tandem-Repeats in t geben kann, kann es nur maximal $O(n)$ Suffix-Link-Walks geben, da jeder Suffix-Link-Walk an einer zugehörigen Lokation eines Tandem-Repeat startet und an solchen Lokationen nur jeweils ein Suffix-Link-Walk gestartet wird. Für die Laufzeit müssen wir jedoch analysieren, wie viele Kanten im Suffix-Baum bei einem Suffix-Link-Walk überlaufen werden. Diese können dummerweise pro Suffix-Link-Walk nicht immer durch eine Konstante beschränkt werden (ähnlich wie bei Ukkonens Algorithmus).

Es kann also passieren, dass bei einem Suffix-Link-Walk von $\text{loc}(aw)$ über $\text{loc}(w)$ zu $\text{loc}(wa)$ mehrere Kanten im Suffix-Baum überlaufen werden, und zwar beim Ablaufen von $\text{loc}(aw)$ zu $\text{loc}(w)$ um die kanonische Lokation zu finden (analog wie bei Ukkonens Algorithmus).

Betrachten wir also eine Kante $e \in E(T)$ und ein Zeichen $a \in \Sigma$. Wir fragen uns, wie oft die Kante e bei einem Suffix-Link-Walk übersprungen werden kann, wenn der Suffix-Link-Walk eine Rechtsrotation des Zeichens a beschreibt. Diese Situation ist in Abbildung 3.48 illustriert.

Im Folgenden sagen wir, ein Suffix-Link hat die Markierung $a \in \Sigma$, wenn der Suffix-Link von $\overline{aw}$ nach $\overline{w}$ für ein $w \in \Sigma^*$ ist. Für eine übersprungene Kante $e = (u, v)$ bei einer Rechtsrotation um a ist der gefolgte Suffix-Link derjenige, der als erster auf dem Pfad zur Wurzel in einen Knoten mit Markierung a hineingeht. Wir können also den zum Suffix-Link-Walk gehörenden Suffix-Link eindeutig identifizieren. Dieser Suffix-Link führt also von $\overline{aw}$ nach $\overline{w}$ für ein $w \in \Sigma^*$ (siehe auch Abbildung 3.48).

Weiterhin wissen wir, dass ein Suffix-Link-Walk immer am Ende eines Tandem-Repeats startet. Sei also $aw\alpha$ das auslösende Tandem-Repeat. Dann liegt die Kante e auf dem Pfad von $\overline{w}$ zu $\text{loc}(w\alpha)$.

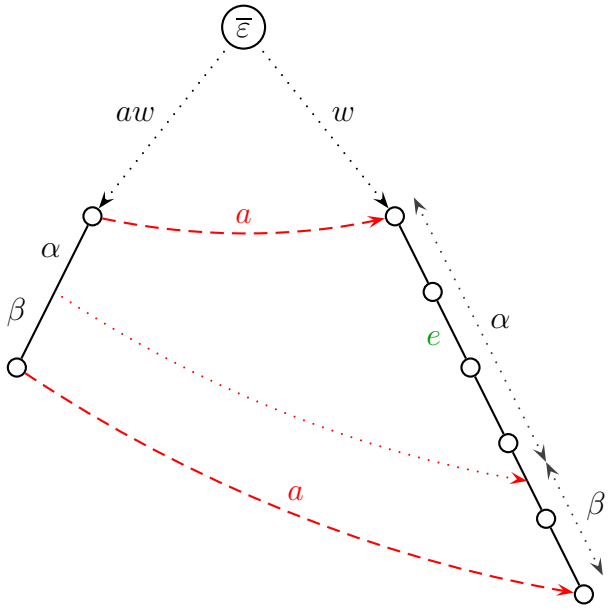


Abbildung 3.48: Skizze: Ein Suffix-Link-Walk von $\mathrm{loc}(aw\alpha)$ zu $\mathrm{loc}(w\alpha a)$

Dies können wir auch umdrehen. Ist α' die Markierung von $\overline{w}$ zum Endpunkt v der Kanten e, dann wissen wir, dass ein Suffix-Link-Walk, der e überspringt, auf der ausgehenden Kante von $\overline{aw}$ begonnen haben muss, dessen Präfix der Kantenmarkierung mit α' beginnt. Es kann also nur eine Kante als Ausgangspunkt des Suffix-Link-Walks zum Überspringen der Kanten e bei einer Rechtsrotation um a in Frage kommen.

Da nach Korollar 3.67 auf jeder Kante nur maximal zwei Tandem-Repeats markiert sind, kann diese Kante für jedes Zeichen $a \in \Sigma$ nur zweimal überspringen werden. Also kann diese Kante insgesamt maximal $2 \cdot |\Sigma|$ mal übersprungen werden. Da im Suffix-Baum T nur $O(n)$ Kanten enthalten sind, folgt die Behauptung. ■

Theorem 3.69 *Das Vokabular $\mathcal{V}(t)$ einer Zeichenreihe $t \in \Sigma^n$ kann im Suffix-Baum $T(t\$)$ in Zeit $O(|\Sigma|n)$ mit Platz $O(n)$ dekoriert werden.*

Lowest Common Ancestors 4

4.1 Algorithmus von Bender und Farach-Colton

In diesem Kapitel wollen wir zwei Algorithmen zur Bestimmung eines niedrigsten gemeinsamen Vorfahren vorstellen. Im ersten Abschnitt werden wir dabei das Problem auf so genannte Range Minimum Queries reduzieren.

Der hier vorgestellte Algorithmus wurde zuerst 1989 von Berkman und Vishkin in einer anderen Darstellung beschrieben und in der hier dargestellten Fassung von Bender und Farach-Colton im Jahre 2000 wiederentdeckt.

4.1.1 Lowest Common Ancestor und Range Minimum Queries

Bevor wir die Problemstellung definieren können, benötigen wir noch den Begriff des niedrigsten gemeinsamen Vorfahren von zwei Knoten in einem Baum.

Definition 4.1 *Sei $T = (V, E)$ ein gewurzelter Baum und seien $v, w \in V$ zwei Knoten von T. Der* niedrigste gemeinsame Vorfahre *von v und w, bezeichnet mit* $\mathrm{lca}(v, w)$ *(engl.* lowest common ancestor*,* nearest common ancestor *oder auch* least common ancestor*), ist der Knoten $u \in V$, so dass u sowohl ein Vorfahre von v als auch von w ist und es keinen echten Nachfahren von u gibt, der ebenfalls ein Vorfahre von v und w ist.*

Damit können wir die Lowest Common Ancestor Queries formalisieren.

Lowest Common Ancestor Query

Eingabe: Ein gewurzelter Baum T und zwei Knoten $v, w \in V(T)$.
Gesucht: Ein Knoten $u \in V(T)$, der der niedrigste gemeinsame Vorfahre von v und w ist.

Wir werden uns nun überlegen, wie wir in einem Baum mit n Knoten ℓ Lowest Common Ancestor Queries in Zeit $O(n + \ell)$ beantworten können. Dazu werden wir das Lowest Common Ancestor Problem auf das Range Minimum Query Problem reduzieren, das wie folgt definiert ist.

RANGE MINIMUM QUERY

Eingabe: Ein Feld F der Länge n von reellen Zahlen und $i \leq j \in [1:n]$.
Gesucht: Ein Index $k \in [i:j]$ mit $F[k] = \min\{F[\ell] \,:\, \ell \in [i:j]\}$.

Man kann Range Minimum Queries natürlich auch allgemeiner für eine total geordnete Menge definieren. Wir werden später zeigen, wie wir mit einer Vorverarbeitung in Zeit $O(n)$ jede solche Anfrage in konstanter Zeit beantworten können. Für die Reduktion betrachten wir die so genannte *Euler-Kontur* oder auch *Euler-Kontur* eines Baumes.

4.1.2 Euler-Kontur eines gewurzelten Baumes

Wir definieren nun, was wir unter einer Euler-Kontur eines gewurzelten Baumes verstehen wollen. Diese Euler-Kontur ist nicht mit Euler-Kreisen oder -Pfaden in Graphen zu verwechseln.

Definition 4.2 *Sei $T = (V, E)$ ein gewurzelter Baum mit Wurzel $r = r(T)$ und seien $T_1, \ldots, T_\ell$ die Teilbäume, die an der Wurzel r hängen. Für $\ell \geq 0$ seien $E_1, \ldots, E_\ell$ mit $E_i = (v_1^{(i)}, \ldots, v_{n_i}^{(i)})$ für $i \in [1:\ell]$ die Euler-Konturen von $T_1, \ldots, T_\ell$. Die* Euler-Kontur *durch T ist eine Liste E' von Knoten des Baumes T der Länge $2|V| - 1$*

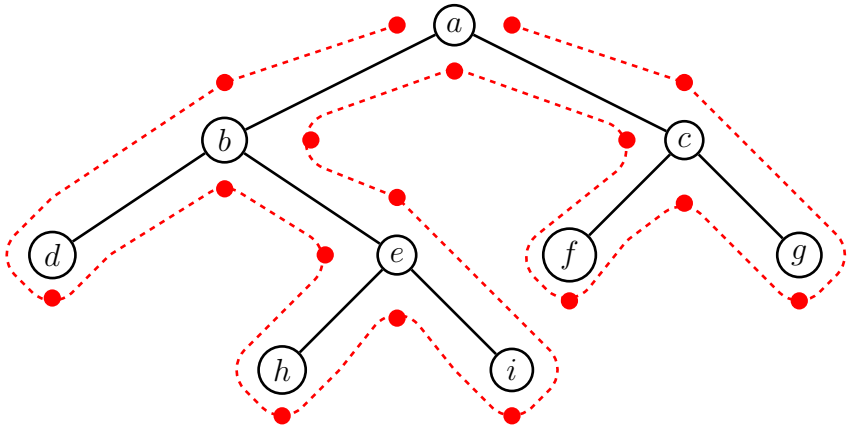
$$E' := (r, v_1^{(1)}, \ldots, v_{n_1}^{(1)}, r, v_1^{(2)}, \ldots, v_{n_2}^{(2)}, r, \ldots, r, v_1^{(\ell)}, \ldots, v_{n_\ell}^{(\ell)}, r).$$

Man beachte, dass in der obigen Definition auch $\ell = 0$ sein kann, d.h. der Baum besteht nur aus dem Blatt r. Dann ist diese Liste durch (r) gegeben. Der Leser sei dazu aufgefordert zu verifizieren, dass die oben definierte Euler-Kontur eines Baumes mit n Knoten tatsächlich eine Liste mit $2n - 1$ Elementen ist.

Euler-Contour (tree $T = (V, E)$)

```
begin
    node v := root(T);
    output v;
    forall ((v, w) ∈ E(T)) do
        EULER-CONTOUR(T(w));  // T(w) denotes the subtree rooted at w
        output v;
end
```

Abbildung 4.1: Algorithmus: Konstruktion einer Euler-Kontur



Euler-Kontur	a	b	d	b	e	h	e	i	e	b	a	c	f	c	g	c	a
Tiefe	0	1	2	1	2	3	2	3	2	1	0	1	2	1	2	1	0

Abbildung 4.2: Beispiel: Euler-Kontur

Die Euler-Kontur kann sehr leicht mit Hilfe einer Tiefensuche in Zeit $O(n)$ berechnet werden. Der Algorithmus hierfür ist in Abbildung 4.1 angegeben. Man kann sich die Euler-Kontur auch bildlich sehr schön als das Abmalen der Bäume anhand ihrer äußeren Kontur vorstellen, wobei bei jedem Antreffen eines Knotens des Baumes dieser in die Liste aufgenommen wird. Dies ist in Abbildung 4.2 anhand eines Beispiels illustriert.

Definition 4.3 *Sei $T = (V, E)$ ein gewurzelter Baum. Die* Tiefe eines Knotens *ist die Anzahl der Kanten auf dem einfachen Weg von der Wurzel zu diesem Knoten. Die maximale Tiefe eines Knotens im Baum T bezeichnet man als die* Tiefe des Baumes*.*

Zusammen mit der Euler-Kontur, d.h. der Liste der abgelaufenen Knoten, betrachten wir zusätzlich noch die Tiefe des entsprechenden Knotens, die bei der Tiefensuche in der Regel mitberechnet werden (siehe auch das Beispiel in der Abbildung 4.2).

4.1.3 Reduktion LCA auf RMQ

Betrachten wir jetzt die Anfrage an zwei Knoten des Baumes i und j. Zuerst bemerken wir, dass diese Knoten, sofern sie keine Blätter sind, in der Euler-Kontur mehrfach vorkommen. Wir wählen jetzt für i und j willkürlich einen der Knoten, der in der Euler-Kontur auftritt, als einen Repräsentanten aus. Weiter stellen wir fest, dass in der Euler-Kontur der niedrigste gemeinsame Vorfahre von i und j in der Teilliste,

die durch die beiden Repräsentanten definiert ist, vorkommen muss, was man wie folgt sieht.

Nehmen wir an, dass i in der Euler-Kontur vor j auftritt. Betrachten wir die Teilliste der Euler-Kontur von einem Repräsentanten i' von i zu einem Repräsentanten j' von j. Sei k der niedrigste gemeinsame Vorfahre von i und j. Da die Tiefensuche von k bereits aktiv sein muss, wenn i' in die Euler-Kontur aufgenommen wird, und k noch aktiv sein muss, wenn j' in die Euler-Kontur aufgenommen wird, kann also in die Teilliste zwischen den Repräsentanten i' und j' kein Repräsentant eines Knoten mit einer Tiefe kleiner oder gleich der Tiefe des Knotens k aufgenommen werden.

Andererseits muss bei der Rückkehr von i auf dem Weg zu j nach der Definition eines niedrigsten gemeinsamen Vorfahrs der Knoten k besucht werden und in die Euler-Kontur aufgenommen werden. Somit befindet sich in der Teilliste der Euler-Kontur zwischen beliebigen Repräsentanten von i und von j der Knoten k mindestens einmal und es ist der einzige Knoten mit minimaler Tiefe in dieser Teilliste. Damit können wir das so erhaltene Zwischenergebnis im folgenden Lemma festhalten.

Lemma 4.4 *Gibt es eine Lösung für das Range Minimum Query Problem, das für die Vorverarbeitung Zeit $O(p(n))$ und für eine Anfrage $O(q(n))$ benötigt, so kann das Problem des niedrigsten gemeinsamen Vorfahren mit einen Zeitbedarf für die Vorverarbeitung in Zeit $O(n+p(2n-1))$ und für eine Anfrage in Zeit $O(q(2n-1))$ gelöst werden.*

Es gilt interessanterweise auch im Wesentlichen die Umkehrung dieses Satzes, den wir im folgenden Lemma festhalten (siehe auch Korollar 4.17). Die wesentliche Argumentation für den Beweis ist im Abschnitt 4.1.8 angegeben.

Lemma 4.5 *Gibt es eine Lösung für das Lowest Common Ancestor Problem, das für die Vorverarbeitung Zeit $O(p(n))$ und für eine Anfrage $O(q(n))$ benötigt, so kann das Range Minimum Query Problem mit einen Zeitbedarf für die Vorverarbeitung in Zeit $O(n+p(n))$ und für eine Anfrage in Zeit $O(q(n))$ gelöst werden.*

Damit können wir uns jetzt ganz auf das Range Minimum Query Problem konzentrieren. Offensichtlich kann ohne eine Vorverarbeitung eine einzelne Anfrage mit $O(j-i+1) = O(n)$ Vergleichen beantwortet werden. Das Problem der Range Minimum Queries ist jedoch insbesondere dann interessant, wenn für ein gegebenes Feld eine Vielzahl von Range Minimum Queries durchgeführt werden. In diesem Fall können mit Hilfe einer Vorverarbeitung die Kosten der einzelnen Queries gesenkt werden.

4.1.4 Ein quadratischer Algorithmus für RMQ

Eine triviale Lösung würde alle möglichen Anfragen vorab berechnen und abspeichern. Dazu könnte eine zweidimensionale Tabelle $Q[i,j]$ für $i \leq j \in [1:n]$ wie folgt angelegt werden:

$$Q[i,j] = \operatorname{argmin}\{F[\ell] \ : \ \ell \in [i:j]\}\,.$$

Dazu würde für jedes Paar (i,j) das Minimum im Bereich $[i:j]$ von F mit $j-i$ Vergleichen bestimmt werden. Dies würde zu einer Laufzeit (entsprechend der Anzahl Vergleiche) für die Vorverarbeitung von

$$\sum_{i=1}^{n}\sum_{j=i}^{n}(j-i) = \Theta(n^3)$$

führen. In der folgenden Abbildung 4.3 ist ein einfacher, auf dynamischer Programmierung basierender Algorithmus angegeben, der diese Tabelle in Zeit $O(n^2)$ (entsprechend der Anzahl Vergleiche) berechnen kann.

RMQ (int $F[]$, int n)

```
begin
    for (i := 1; i ≤ n; i++) do
        Q[i, i] := i;
        for (j := i + 1; j ≤ n; j++) do
            if (F[Q[i, j − 1]] ≤ F[j]) then
                Q[i, j] := Q[i, j − 1];
            else
                Q[i, j] := j;
end
```

Abbildung 4.3: Algorithmus: Vorverarbeitung für Range Minimum Queries

Damit erhalten wir das folgende erste Resultat für die Komplexität des Range Minimum Query Problems.

Theorem 4.6 *Für das Range Minimum Query Problem kann mit Hilfe einer Vorverarbeitung, die mit einem Zeitbedarf von $O(n^2)$ ausgeführt werden kann, jede Anfrage in konstanter Zeit beantwortet werden.*

4.1.5 Eine verbesserte Variante

Eine schnellere Variante erhalten wir, wenn wir nicht alle Werte $Q[i,j]$ berechnen, sondern nur für solche Paare (i,j) deren Differenz $j-i+1$ eine Zweierpotenz ist. Dazu definieren wir eine Tabelle Q' für $i \in [1:n]$ und $k \in [0:\lfloor \log(n) \rfloor]$ wie folgt:

$$Q'[i,k] = \operatorname{argmin}\left\{F[\ell] \ : \ \ell \in [i : i+2^k-1]\right\}.$$

Da die Tabelle nur $\Theta(n\log(n))$ Einträge besitzt, kann diese Tabelle mittels dynamischer Programmierung in Zeit $O(n\log(n))$ berechnet werden, wie es im Algorithmus in Abbildung 4.4 dargestellt ist.

RMQ2 (int $F[]$, int n)

```
begin
    for (i := 1; i ≤ n; i++) do
        Q'[i,0] := i;

    for (k := 0; k < ⌊log(n)⌋; k++) do
        for (i := 1; i + 2^k ≤ n; i++) do
            if (F[Q'[i,k]] ≤ F[Q'[i + 2^k, k]]) then
                Q'[i,k+1] := Q'[i,k];
            else
                Q'[i,k+1] := Q'[i + 2^k, k];
end
```

Abbildung 4.4: Algorithmus: Bessere Vorverarbeitung für Range Minimum Queries

Wie beantworten wir jetzt eine gegebene Anfrage RMQ(i,j)? Wir berechnen zuerst $k := \lfloor \log(j-i+1) \rfloor$. Also gilt $2^k \le j-i+1 < 2^{k+1}$. Dann ermitteln wir die Indizes der Minima in F im Bereich $[i : i+2^k-1]$ bzw. $[j-2^k+1 : j]$ mittels $r := Q'[i,k]$ bzw. $s := Q'[j-2^k+1,k]$. Nach Wahl von k gilt $[i : i+2^k-1] \cup [j-2^k+1 : j] = [i : j]$. Gilt dann $F[r] \le F[s]$, dann ist die Antwort r, ansonsten s. Dies ist auch in Abbildung 4.5 illustriert.

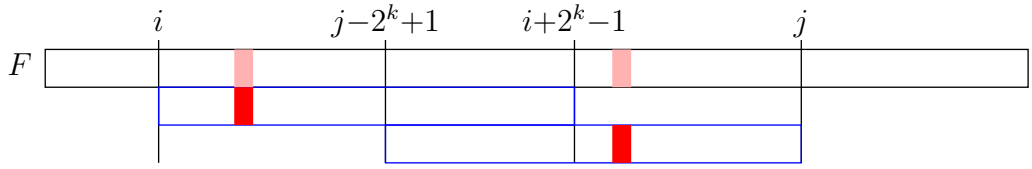


Abbildung 4.5: Skizze: RMQ-Anfrage mittels Q'

Somit erhalten wir den folgenden Satz.

Theorem 4.7 *Für das Range Minimum Query Problem kann mit Hilfe einer Vorverarbeitung, die mit einem Zeitbedarf von $O(n \log(n))$ ausgeführt werden kann, jede Anfrage in konstanter Zeit beantwortet werden.*

03.12.24

4.1.6 Incremental Range Minimum Query (*)

Betrachten wir das Feld F näher, das aus der Euler-Kontur eines Baumes entsteht. Man beobachtet sofort, dass sich die Werte von konsekutiven Feldelementen um genau 1 unterscheiden. Daher wollen wir eigentlich das folgende eingeschränkte Problem lösen.

INCREMENTAL RANGE MINIMUM QUERY

Eingabe: Ein Feld F der Länge n von reellen Zahlen, wobei $|F[i+1] - F[i]| = 1$ für alle $i \in [1 : n-1]$ gilt, und $i \leq j \in [1 : n]$.
Gesucht: Ein (minimaler) Index $k \in [i : j]$ mit $F[k] = \min\{F[\ell] : \ell \in [i : j]\}$.

Wir halten zunächst die folgende fundamentale Eigenschaften fest.

Beobachtung 4.8 *Seien F und G zwei Felder der Länge n mit reellen Einträgen und sei $c \in \mathbb{R}$ eine Konstante. Gilt $F[k] - G[k] = c$ für alle $k \in [1 : n]$, dann gilt $RMQ_F(i, j) = RMQ_G(i, j)$.*

Somit können wir für Incremental Range Minimum Queries annehmen, dass für ein zu verarbeitendes Feld F gilt, dass $F[1] = 0$, indem man von allen Feldelementen $F[1]$ abzieht.

Definition 4.9 *Sei F ein Feld der Länge n mit reellen Einträgen. Das Feld F heißt* inkrementell*, wenn $|F[i+1] - F[i]| = 1$ für alle $i \in [1 : n-1]$. Das Feld F heißt* normalisiert*, wenn $F[1] = 0$.*

Ausgehend von dieser Definition sieht man sofort, dass es nur eine beschränkte Anzahl normalisierter, inkrementeller Felder gibt.

Lemma 4.10 *Es gibt 2^{n-1} verschiedene normalisierte, inkrementelle Felder der Länge n.*

Beweis: Da das Feld F normalisiert ist, gilt $F[1] = 0$. Da das Feld F inkrementell ist, muss man sich für jedes Folgeglied nur merken, ob der Wert um 1 erhöht oder erniedrigt wird. Dies sind insgesamt $n - 1$ Bit-Informationen. ■

4.1.7 Ein optimaler Algorithmus für IRMQ (*)

Für einen optimalen Algorithmus mit linearer Vorverarbeitungszeit unterteilen wir das gegebene Feld F in $\lceil n/k \rceil$ Blöcke der Länge jeweils k (außer eventuell dem letzten) mit $k := \lceil \frac{1}{2} \log(n) \rceil$. Sei dann F' ein Feld der Länge $n' := \lceil n/k \rceil$, wobei gilt:

$$F'[i] = \min \{F[j] \ : \ j \in [(i-1)k + 1 : \min\{i \cdot k, n\}]\}.$$

Anschaulich steht im i-ten Eintrag von Feld F' das Minimum des i-ten Blocks von Feld F. Weiter sei P' ein Feld der gleichen Länge wie F', wobei $P'[i]$ die Position angibt, an welcher Position im i-ten Block von F das Minimum angenommen wird, d.h.

$$P'[i] = \operatorname{argmin} \{F[j] \ : \ j \in [(i-1)k + 1 : \max\{i \cdot k : n\}]\} - (i-1)k.$$

Die Felder F' und P' können offensichtlich in linearer Zeit ermittelt werden. In der Regel wird man das Feld F' nicht abspeichern, da $F'[i] = F[P'[i] + (i-1)k]$ gilt.

Wir können jetzt das Feld F' der Länge $n' = \lceil n/k \rceil$ mit unserem Algorithmus mit Laufzeit $O(n' \log(n'))$ vorverarbeiten, dann lässt sich jede Range Minimum Query auf F' in konstanter Zeit beantworten. Die Laufzeit für diese Vorverarbeitung von F' beträgt

$$O(n' \log(n')) = O\left(\frac{n}{\log(n)} \log\left(\frac{n}{\log(n)}\right)\right) = O(n).$$

Wie beantworten wir jetzt eine Anfrage $\mathrm{RMQ}(i, j)$? Zuerst ermitteln wir die Blöcke i' bzw. j', in denen sich i bzw. j befinden. Gilt $i' = j'$, dann ist die Position $p_i = \mathrm{RMQ}_F(i, j)$ (innerhalb des Blockes $i' = j'$ von F) die Lösung. Gilt andererseits $i' < j'$, dann ermitteln wir die folgenden drei Positionen:

- Die Position $p_i = \mathrm{RMQ}_F(i, k \cdot i')$;
- Falls $i' + 1 < j'$ ist, die Position $p' = \mathrm{RMQ}_F(k \cdot i' + 1, k(j' - 1))$, dabei gilt $p' = (q-1) \cdot k + P'[q]$ mit $q := \mathrm{RMQ}_{F'}(i' + 1, j' - 1)$;
- Die Position $p_j = \mathrm{RMQ}_F(k(j' - 1) + 1, j)$.

Aus diesen drei Werten kann dann die Position mit dem minimalen Element mit zwei weiteren Vergleichen ermittelt werden.

Dabei kann p' nach der Vorverarbeitung von F' in konstanter Zeit ermittelt werden. Wir müssen also nur für p_i bzw. p_j ein effizientes Verfahren angeben. Dabei finden diese Anfragen immer innerhalb eines Blockes der Länge k statt.

Nach Lemma 4.10 gibt es nur

$$2^{k-1} \leq 2^{\lceil \log(n)/2 \rceil - 1} \leq 2^{\log(n)/2} = O\left(\sqrt{n}\right)$$

viele normalisierte, inkrementelle Felder der Länge k. Anstatt für alle $\lceil n/k \rceil$ Teilfelder jeweils eine Lookup-Tabelle für die Incremental Range Minimum Queries zu konstruieren, konstruieren wir für alle möglichen normalisierten, inkrementellen Blöcke jeweils nur eine Tabelle mittels der einfachen dynamischen Programmierung, von denen es ja nur $O(\sqrt{n})$ viele gibt. Die Vorverarbeitungszeit hierfür beträgt offensichtlich:

$$O\left(\sqrt{n} \cdot \log^2(n)\right) = o(n) = O(n).$$

Für jeden der $n' = \lceil n/k \rceil$ Blöcke müssen wir nur noch einen Verweis auf die entsprechende Tabelle angeben, was sich leicht in linearer Zeit bestimmen lässt. Dann können wir eine Incremental Range Minimum Query innerhalb eines Blocks in mit Hilfe der konstruierten Tabellen in konstanter Zeit beantworten.

Theorem 4.11 *Für das Incremental Range Minimum Query Problem kann mit Hilfe einer Vorverarbeitung, die mit einem Zeitbedarf von $O(n)$ ausgeführt werden kann, jede Anfrage in konstanter Zeit beantwortet werden.*

Damit erhalten wir das für uns wichtige Theorem.

Theorem 4.12 *Für das Lowest Common Ancestor Problem kann mit Hilfe einer Vorverarbeitung, die mit einem Zeitbedarf von $O(n)$ ausgeführt werden kann, jede Anfrage in konstanter Zeit beantwortet werden.*

4.1.8 Optimale Lösung für RMQ (*)

In diesem letzten Teil wollen wir noch zeigen, dass wir auch beim allgemeinen Range Minimum Query Problem mit einer linearen Vorverarbeitungszeit jede Anfrage in

konstanter Zeit beantworten können. Hierfür reduzieren wir das Problem lustigerweise auf ein LCA-Problem. Um die Reduktion beschreiben zu können, benötigen wir noch das Konzept eines Kartesischen Baumes.

Definition 4.13 *Sei F ein Feld der Länge n mit reellen Einträgen. Der* Kartesische Baum *des Feldes F ist ein geordneter gewurzelter Baum $T = (V, E)$ mit n Knoten. Die Wurzel ist markiert mit dem Index $k = \operatorname{argmin}\{F[i] : i \in [1:n]\}$. An der Wurzel hängen zwei Teilbäume (die auch leer sein können), wobei der linke bzw. rechte Teilbaum ein Kartesicher Baum für die Teilfelder $F[1:k-1]$ bzw. $F[k+1:n]$ ist.*

Man beachte, dass die Definition nicht eindeutig ist, da es durchaus mehrere Minima in einem Feld geben kann. Wir wählen hier im Folgenden immer das Minimum mit dem kleinsten Index. Die folgende Beobachtung folgt unmittelbar aus der Definition eines Kartesischen Baumes.

Beobachtung 4.14 *Sei F ein Feld der Länge n mit reellen Einträgen und T der zugehörige Kartesische Baum. Eine Inorder-Nummerierung von T ergibt die Folge $1, \ldots, n$.*

Wir zeigen jetzt, wie man einen Kartesischen Baum für ein Feld F in linearer Zeit konstruieren kann. Sei T ein Kartesischer Baum für das Feld F mit den Indizes aus $[1 : n-1]$. Wir werden jetzt T so modifizieren, dass auch das Feldelement $F[n]$ berücksichtigt wird.

Wir betrachten dazu den Pfad vom Knoten v mit Markierung $n-1$ zur Wurzel r des Baumes T. Für die Markierungen $(i_i, \ldots, i_\ell)$ der Knoten auf diesem Pfad mit $n-1 = i_1$ und $i_\ell = r$ gilt nach Konstruktion $F[i_1] \geq \cdots \geq F[i_\ell]$. Wir laufen nun diesen Pfad vom Blatt v zur Wurzel ab und vergleichen, ob $F[n] < F[i_k]$ für jeden Knoten mit Markierung i_k gilt. Sobald diese Bedingung nicht mehr erfüllt ist (also $F[i_k] \leq F[n]$ gilt), erzeugen wir einen neuen Knoten mit Markierung n und machen ihn zum rechten Kind des Knotens mit Markierung i_k. Der Teilbaum mit Wurzel i_{k-1} wird zum linken Teilbaum des neuen Knotens mit Markierung n. Falls wir über die Wurzel hinauslaufen (also eigentlich $k = \ell + 1$ wird), erzeugen wir eine neue Wurzel mit Markierung n die nur einen linken Teilbaum besitzt, nämlich den alten Kartesischen Baum.

Man überlegt sich leicht, dass dieser Algorithmus wirklich einen Kartesischen Baum konstruiert. Wir müssen uns nur noch überlegen, ob dieser auch in linearer Zeit konstruiert werden kann. Hierfür bezeichne $\ell(i)$ die Länge des Pfades (Anzahl der Knoten des Pfades) von der Wurzel zum Knoten mit Markierung i im Kartesischen Baum für $F[1:i]$. Nach Konstruktion gilt $1 \leq \ell(i) \leq \ell(i-1)+1$. Das heißt, dass beim

Einfügen des i-ten Knotens der rechteste Pfad auf eine Länge von eins schrumpfen kann (nämlich dann, wenn der neue Knoten zur Wurzel wird), aber maximal um ein länger werden kann (nämlich dann, wenn der neue Knoten als ein neues Blatt an den rechtesten Pfad angehängt wird).

Die Kosten, um den i-ten Knoten anzuhängen, sind also proportional zur Anzahl der Knoten auf dem rechtesten Pfad des abgehängten Teilbaum plus eins, also $O(\ell(i) - \ell(i-1) + 1)$. Somit ergibt sich für die Gesamtlaufzeit:

$$\sum_{i=1}^{n} O(\ell(i) - \ell(i-1) + 1) = O(\ell(n) - \ell(0) + n) \leq O(n - 0 + n) = O(n).$$

Damit haben wir das folgende Lemma bewiesen.

Lemma 4.15 *Sei F ein Feld der Länge n mit reellen Einträgen. Der Kartesische Baum für F kann in Zeit und Platz $O(n)$ konstruiert werden.*

Für die Lösung unseres Problems behaupten wir jetzt, dass $\mathrm{RMQ}_F(i,j) = \mathrm{lca}_T(i,j)$ gilt, wobei ohne Beschränkung der Allgemeinheit $i \leq j$ gilt. Für den Beweis sei $k = \mathrm{lca}_T(i,j)$.

Wir nehmen zuerst an, dass $k \notin \{i,j\}$. Im Kartesischen Baum T ist auf dem Weg von der Wurzel zu i bzw. j der Knoten k der letzte auf dem gemeinsamen Weg. Sei v bzw. w das linke bzw. rechte Kind von k. Dann muss sich i im Teilbaum von v und j im Teilbaum von w befinden. Sei $\underline{k}$ bzw. $\overline{k}$ die kleinste bzw. die größte Markierung im Teilbaum von v bzw. w Nach Konstruktion des Kartesischen Baumes gilt für alle $x \in [\underline{k} : \overline{k}]$, dass $F[x] \geq F[k]$. Somit ist $k = \mathrm{RMQ}_F(\underline{k}, \overline{k})$. Da $[i:j] \subseteq [\underline{k} : \overline{k}]$ und $k \in [i:j]$, muss auch $k = \mathrm{RMQ}_F(i,j)$ gelten.

Der Fall $k \in \{i,j\}$ ist sehr ähnlich und der Beweis bleibt dem Leser zur Übung überlassen. Somit haben wir den folgenden Satz bewiesen.

Theorem 4.16 *Für das Range Minimum Query Problem kann mit Hilfe einer Vorverarbeitung, die mit einem Zeitbedarf von $O(n)$ ausgeführt werden kann, jede Anfrage in konstanter Zeit beantwortet werden.*

Als Korollar erhalten wir außerdem noch das Ergebnis, dass man auch das RMQ-Problem auf ein LCA-Problem reduzieren kann.

Korollar 4.17 *Gibt es eine Lösung für das Lowest Common Ancestor Problem, das für die Vorverarbeitung Zeit $O(p(n))$ und für eine Anfrage $O(q(n))$ benötigt, so kann das Range Minimum Query Problem mit einen Zeitbedarf für die Vorverarbeitung in Zeit $O(n + p(n))$ und für eine Anfrage in Zeit $O(q(n))$ gelöst werden.*

4.1.9 Eine einfachere optimale Variante nach Alstrup et al.

In diesem Abschnitt stellen wir noch eine einfachere Variante vor, die auf die Arbeit von Alstrup, Gavoille, Kaplan und Rauhe aus dem Jahre 2002 zurückgeht. Hierbei nehmen wir allerdings an, dass einige einfache Bit-Operationen, wie bitweises Und, bitweises Oder sowie Bitscans in konstanter Zeit auf Bit-Vektoren der Länge $\log(n)$ durchgeführt werden können. Intel-Prozessoren stellen diese Operationen beispielsweise zur Verfügung. Darüber hinaus ist zu beachten, dass wir bisher sogar für aufwendigere Operationen, wie Addition, Multiplikation, Logarithmieren für Zahlen, die sich in einem Maschinenwort darstellen lassen, angenommen haben, dass sich diese in konstanter Zeit durchführen lassen. Man beachte, dass auf 64-Bit-Prozessoren so lange Felder (mit 2^{64} Elementen, also mehreren Trillionen) schon gar nicht mehr im Hauptspeicher gehalten werden können.

Für einen optimalen Algorithmus mit linearer Vorverarbeitungszeit unterteilen wir das gegebene Feld F in $\lceil n/k \rceil$ Blöcke der Länge jeweils k (außer eventuell dem letzten) mit $k := \lceil \log(n) \rceil$. Sei dann F' ein Feld der Länge $n' := \lceil n/k \rceil$, das wie folgt definiert ist:

$$F'[i] = \min \{F[j] \ : \ j \in [(i-1)k+1 : \min\{i \cdot k, n\}]\}\,.$$

Anschaulich steht im i-ten Eintrag von Feld F' das Minimum des i-ten Blocks von Feld F. Weiter sei P' ein Feld der gleichen Länge wie F', wobei $P'[i]$ die Position angibt, an welcher Position im i-ten Block von F das Minimum angenommen wird, d.h.

$$P'[i] = \operatorname{argmin} \{F[j] \ : \ j \in [(i-1)k+1 : \min\{i \cdot k : n\}]\} - (i-1)k.$$

Die Felder F' und P' können offensichtlich in linearer Zeit ermittelt werden. In der Regel wird man das Feld F' nicht abspeichern, da $F'[i] = F[P'[i] + (i-1)k]$ gilt.

Wir können jetzt das Feld F' der Länge $n' = \lceil n/k \rceil$ mit unserem Algorithmus mit Laufzeit $O(n' \log(n'))$ vorverarbeiten, dann lässt sich jede Range Minimum Query auf F' in konstanter Zeit beantworten. Die Laufzeit für diese Vorverarbeitung von F' beträgt

$$O(n' \log(n')) = O\left(\frac{n}{\log(n)} \log\left(\frac{n}{\log(n)}\right)\right) = O(n).$$

Wie beantworten wir jetzt eine Anfrage $\mathrm{RMQ}(i,j)$? Zuerst ermitteln wir die Blöcke i' bzw. j' in denen sich i bzw. j befinden:

$$\begin{aligned} i' &= \left\lfloor \frac{i-1}{k} \right\rfloor + 1 = \left\lceil \frac{i}{k} \right\rceil \\ j' &= \left\lfloor \frac{j-1}{k} \right\rfloor + 1 = \left\lceil \frac{j}{k} \right\rceil \end{aligned}$$

Gilt $i' = j'$, dann ist die Position $p_i = \mathrm{RMQ}_F(i, j)$ (innerhalb des Blockes $i' = j'$ von F) die Lösung. Wie wir dies effizient bestimmen sehen wir später.

Gilt andererseits $i' < j'$, dann ermitteln wir die folgenden drei Positionen:

- Die Position $p_i = \mathrm{RMQ}_F(i, k \cdot i')$;
- Falls $i' + 1 < j'$ ist, die Position $p' = \mathrm{RMQ}_F(k \cdot i' + 1, k(j' - 1))$, dabei gilt $p' = (q - 1) \cdot k + P'[q]$ mit $q := \mathrm{RMQ}_{F'}(i' + 1, j' - 1)$;
- Die Position $p_j = \mathrm{RMQ}_F(k(j' - 1) + 1, j)$.

Aus diesen drei Werten kann dann die Position mit dem minimalen Element mit zwei weiteren Vergleichen ermittelt werden (siehe auch Abbildung 4.6).

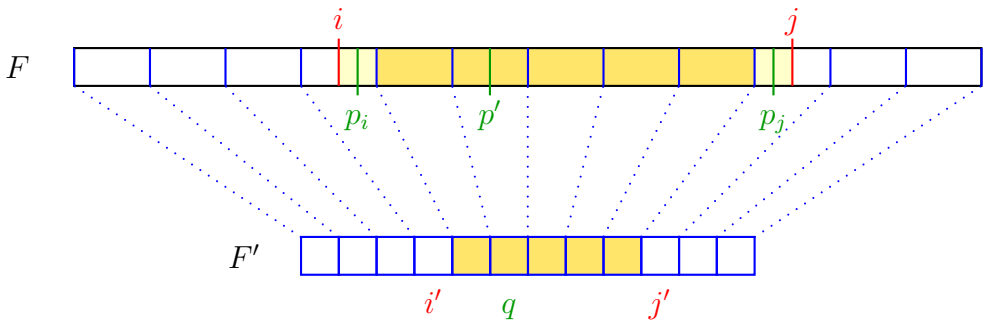


Abbildung 4.6: Skizze: Beantwortung $\mathrm{RMQ}_F(i, j)$ mittels $\mathrm{RMQ}_{F'}(i', j')$

Dabei kann p' nach der Vorverarbeitung von F' in konstanter Zeit ermittelt werden. Wir müssen also nur für p_i bzw. p_j ein effizientes Verfahren angeben. Dabei finden diese Anfragen immer innerhalb eines Blockes der Länge k statt.

Betrachten wir nun einen Block B der Länge k und definieren hierfür k Bit-Vektoren $V^{(B,j)}$ für $j \in [1 : k]$ der Länge jeweils k wie folgt:

$$V_i^{(B,j)} = \begin{cases} 1 & \text{falls } i \leq j \wedge \forall \ell \in [i : j] : B[i] \leq B[\ell], \\ 0 & \text{sonst.} \end{cases}$$

Intuitiv bedeutet dies, dass genau dann $\mathrm{RMQ}_B(i, j) = i$, wenn $V_i^{(B,j)} = 1$. Weiter gilt offensichtlich $V_j^{(B,j)} = 1$ sowie $V_i^{(B,j)} = 0$ für $i > j$.

Angenommen, wir hätten für jeden Block jeweils diese k Bit-Vektoren, dann lässt sich eine RMQ-Anfrage innerhalb dieses Blockes wie folgt bestimmen (Erläuterung folgt)

$$\mathrm{RMQ}_B(i, j) = \texttt{bsf}(V^{(B,j)} \wedge 0^{i-1}1^{k-i+1}),$$

wobei $\wedge$ das bitweise Und ist sowie $\texttt{bsf}(v)$ die Position (aus $[1:k]$) der linkesten 1 eines Bitvektors v liefert (Bit Scan Forward):

$$\texttt{bsf}(v) = i \quad :\Leftrightarrow \quad (v_i = 1 \vee i = k+1) \wedge \forall \ell \in [1:i-1] : v_\ell = 0.$$

Beachte, dass nach Definition $\texttt{bsf}(0^k) = k+1$. Intuitiv ist ein Bitscan nichts anderes als die Berechnung des abgerundeten Zweier-Logarithmus einer natürlichen Zahl.

Mit Hilfe von Abbildung 4.7 wird nun die Berechnung der RMQ-Anfrage klar. Wir suchen das linkeste gesetzte Bit rechts von Position i, da für diese Position i' gilt $\text{RMQ}_B(i', j) = i'$ und diese die linkeste Position im Intervall $[i:j]$ ist.

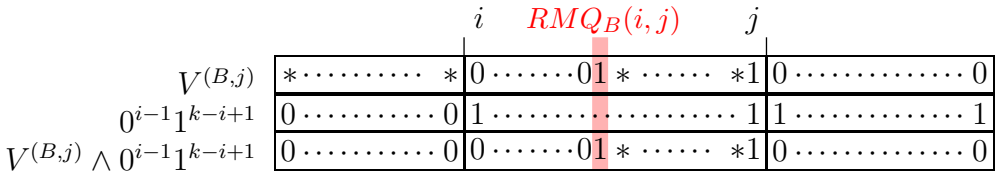


Abbildung 4.7: Skizze: Bit Scan Forward für RMQ-Anfrage

Wie zu Beginn dieses Abschnittes erwähnt, haben wir angenommen, dass diese Operationen in konstanter Zeit durchführbar sind. Somit lässt sie eine RMQ-Anfrage innerhalb eines Blockes mit Hilfe der Bit-Vektoren $V^{(B,j)}$ in konstanter Zeit beantworten.

Wir müssen uns nur noch überlegen, wie man diese Bit-Vektoren $V^{(B,j)}$ bestimmt. Hierzu sei B zunächst ein beliebiger, aber fester Block. Wir bestimmen folgenden Vektor $L^B \in [0:k]^k$, der wie folgt definiert ist:

$$L^B[j] := \max\{i \in [0:j-1] \ : \ B[i] \leq B[j]\} < j.$$

Hierbei nehmen wir an, dass $B[0] = -\infty$ ist, damit die Maximumsbildung über eine nichtleere Menge gebildet wird. Anschaulich ist $L^B[j]$ der größte Index $i < j$, so dass $B[i]$ kleiner oder gleich $B[j]$ ist. Somit ist $L^B[j]$ die größte Bit-Position (kleiner als j) an der der Vektor $V^{(B,j)}$ den Wert 1 annimmt. Daraus ergibt sich die folgende rekursive Definition von $V^{(B,j)}$:

$$V^{(B,j)} = \begin{cases} V^{(B,L^B[j])} \vee 0^{j-1}10^{k-j} & \text{falls } L^B[j] > 0, \\ 0^{j-1}10^{k-j} & \text{falls } L^B[j] = 0. \end{cases}$$

Wir müssen also nur noch den Vektor L^B ermitteln. Als Wert für $L^B[j]$ ist eine Indexposition $i < j$ nur dann interessant, wenn $B[i] \leq B[\ell]$ für alle $\ell \in [i:j]$. Alle solchen Indexpositionen $(i_0, \ldots, i_\nu)$ mit

$$0 = i_0 < i_1 < \cdots < i_\nu \quad \text{und} \quad -\infty = B[i_0] \leq B[i_1] \leq \cdots \leq B[i_\nu]$$

```
constructBitVec (int[] B, int k)
begin
    stack S;
    S.push(0);                                          /* B[0] = −∞ */
    for (j := 1; j ≤ k; j++) do
        while (B[S.top()] > B[j]) do
            S.pop();
        L^B[j] := S.top();
        S.push(j);

    for (j := 1; j ≤ k; j++) do
        if (L^B[j] > 0) then
            V^(B,j) := V^(B,L^B[j]) ∨ 0^(j−1)10^(k−j);
        else
            V^(B,j) := 0^(j−1)10^(k−j);
end
```

Abbildung 4.8: Algorithmus: Erstellung der Bit-Vektoren eines Blockes B

merken wir uns im Algorithmus in Abbildung 4.8 mit Hilfe eines Kellers (der mit dem Wert $i_0 = 0$ initialisiert wird).

Auf dem Keller halten wir daher als Invariante folgende maximale Liste von Indizes $(i_0, \ldots, i_\nu)$ mit $B[i_{m-1}] \leq B[i_m]$ für $m \in [1 : \nu]$. Ist nun $B[j] < B[i_\nu]$, dann entfernt der Algorithmus i_ν vom Keller (und dekrementiert damit implizit ν). Für Indexposition $i \in [i_{\nu-1} + 1 : i_\nu]$ (die ja gar nicht erst auf dem Keller abgelegt worden sind) gilt ohnehin $B[i] \geq B[i_\nu] > B[j]$ und somit $B[i] > B[j]$; diese spielen für $L^B[j]$ somit auch keine Rolle.

Der Algorithmus in Abbildung 4.8 hat eine Laufzeit von $O(k)$, da jeder Index aus $[1 : k]$ nur einmal auf dem Keller abgelegt wird und somit auch nur einmal vom Keller entfernt werden kann. Somit ist die Laufzeit $O(n' \cdot k) = O(\frac{n}{k} \cdot k) = O(n)$ und wir erhalten wiederum den folgenden Satz (siehe auch Theorem 4.16).

Theorem 4.18 *Für das Range Minimum Query Problem kann mit Hilfe einer Vorverarbeitung, die mit einem Zeitbedarf von $O(n)$ ausgeführt werden kann, jede Anfrage in konstanter Zeit beantwortet werden.*

Damit erhalten wir das für uns wichtige folgenden Theorem bereits zum zweiten Mal (siehe auch Theorem 4.12).

Theorem 4.19 *Für das Lowest Common Ancestor Problem kann mit Hilfe einer Vorverarbeitung, die mit einem Zeitbedarf von $O(n)$ ausgeführt werden kann, jede Anfrage in konstanter Zeit beantwortet werden.*

Wir merken an dieser Stelle noch, dass wir mit dieser Methode das allgemeine RMQ-Problem und nicht nur das IRMQ-Problem lösen können. Daher können wir statt der Euler-Kontur auch einfach die Inorder-Liste der Knoten des Baumes verwenden (die der Euler-Kontur allerdings ähnlich ist).

Definition 4.20 *Sei $T = (V, E)$ ein gewurzelter Baum mit Wurzel r und seien $T_1, \ldots, T_\ell$ die Teilbäume, die an den Kindern der Wurzel gewurzelt sind, mit den Inorder-Listen $I_1, \ldots, I_\ell$, wobei $I_j = (v_1^{(j)}, \ldots, v_{n_j}^{(j)})$. Die* Inorder-Liste *von T ist gegeben durch:*

$$\mathit{Inorder}(T) = \begin{cases} (r) & \text{falls } \ell = 0, \\ (v_1^{(1)}, \ldots, v_{n_1}^{(1)}, r) & \text{falls } \ell = 1, \\ (v_1^{(1)}, \ldots, v_{n_1}^{(1)}, r, v_1^{(2)}, \ldots, v_{n_2}^{(2)}, r, \ldots, r, v_1^{(\ell)}, \ldots, v_{n_\ell}^{(\ell)}) & \text{falls } \ell > 1. \end{cases}$$

Wir wollen hier auch noch anmerken, dass der Platzbedarf der hier vorgestellten Varianten für RMQ-Anfragen (und damit auch für LCA-Anfragen) $O(n)$ Maschinenwörter, also $O(n \log(n))$ Bits beträgt. Man kann mit etwas aufwendigeren, aber teilweise ähnlichen Techniken zeigen, dass sich der zusätzliche Platzbedarf (zusätzlich zum betrachteten Feld, auf dem die RMQ-Anfragen gestellt werden) auf $2n+o(n)$ Bits reduzieren lässt, ohne dass man dabei auf die lineare Vorverarbeitungszeit und die konstante Anfragezeit verzichten muss. Für die Details verweisen wir auf die Originalliteratur von Fischer und Heun.

4.2 Algorithmus von Schieber und Vishkin (*)

In diesem Abschnitt behandeln wir eine andere, ältere Variante zur Bestimmung Lowest Common Ancestors. Dieser Abschnitt ist nur als Ergänzung im Skript, da er Teil der Vorlesung im Wintersemester 2003/04 war.

In diesem Abschnitt nehmen wir explizit an, dass sich alle Operationen auf Bit-Strings der Länge $O(\log(n))$ in konstanter Zeit erledigen lassen, wobei n die Anzahl Knoten im untersuchten Baum ist. Dies haben wir bisher in allen Algorithmen auch immer implizit angenommen. Da hier aber explizit scheinbar aufwendigere Operationen auf Bit-Strings vorkommen, wollen wir dies hier explizit anmerken.

4.2.1 LCA-Queries auf vollständigen binären Bäumen

Zuerst wollen wir den einfachen Fall betrachten, dass die Anfragen in einem vollständigen binären Baum stattfinden.

Definition 4.21 *Ein gewurzelter Baum heißt* binär*, wenn jeder Knoten maximal zwei Kinder besitzt. Ein binärer Baum heißt* vollständig*, wenn alle Blätter dieselbe Tiefe besitzen und jeder innere Knoten genau zwei Kinder besitzt.*

Kommen wir nun zur Definition der Inorder-Nummerierung.

Definition 4.22 *Sei $B = (V, E)$ ein binärer Baum mit n Knoten. Sei weiter $I : V \to [1 : n]$ eine bijektive Abbildung. Wenn für jeden Knoten $v \in V$ gilt, dass $I(w) < I(v) < I(w')$ für jeden Nachfahren w des linken Kindes und jeden Nachfahren w' des rechten Kindes, dann ist I die* Inorder-Nummerierung *von B.*

In Abbildung 4.9 ist das Beispiel einer Inorder-Nummerierung für einen beliebigen binären Baum angegeben.

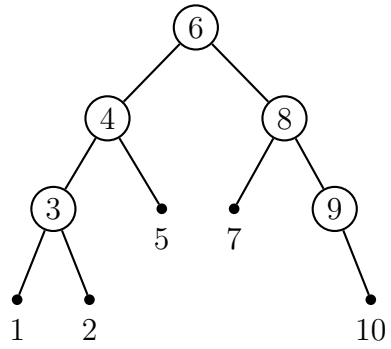


Abbildung 4.9: Beispiel: Inorder-Nummerierung eines binären Baumes

Wir versuchen nun die Anfrage $\text{lca}(i, j)$ für einen vollständigen Baum zu beantworten. Wir unterscheiden zwei Fälle, je nachdem, ob die Knoten Nachfahren voneinander sind oder nicht.

Fall 1: Es gilt $\text{lca}(i, j) \in \{i, j\}$. Wir führen eine Tiefensuche aus und vergeben jedem Knoten seine DFS-Nummer. Für jeden Knoten v bestimmen wir zusätzlich das Intervall von DFS-Nummern aller Knoten, die Nachfahren von v sind. Dann gilt:

$$\text{lca}(i,j) = \begin{cases} i & \text{falls } \text{dfs}(j) \in \text{DFS-Interval}(i), \\ j & \text{falls } \text{dfs}(i) \in \text{DFS-Interval}(j), \\ 0 & \text{sonst.} \end{cases}$$

Wir merken noch an, dass sich die DFS-Nummern und DFS-Intervalle in linearer Zeit mit einer Tiefensuche berechnen lassen und sich dann jede solche Lowest Common Ancestor Anfrage in konstanter Zeit beantworten lässt. Bei Ergebnis 0 gehen wir natürlich in den folgenden Fall 2.

Fall 2: Es gilt $\text{lca}(i, j) \notin \{i, j\}$. Wir vergeben bei einer Tiefensuche die Inorder-Nummern des vollständigen binären Baumes. Ein Beispiel für eine solche Inorder-Nummerierung ist in Abbildung 4.10 angegeben. Hierbei sind die Inorder-Nummern als Bit-Strings dargestellt.

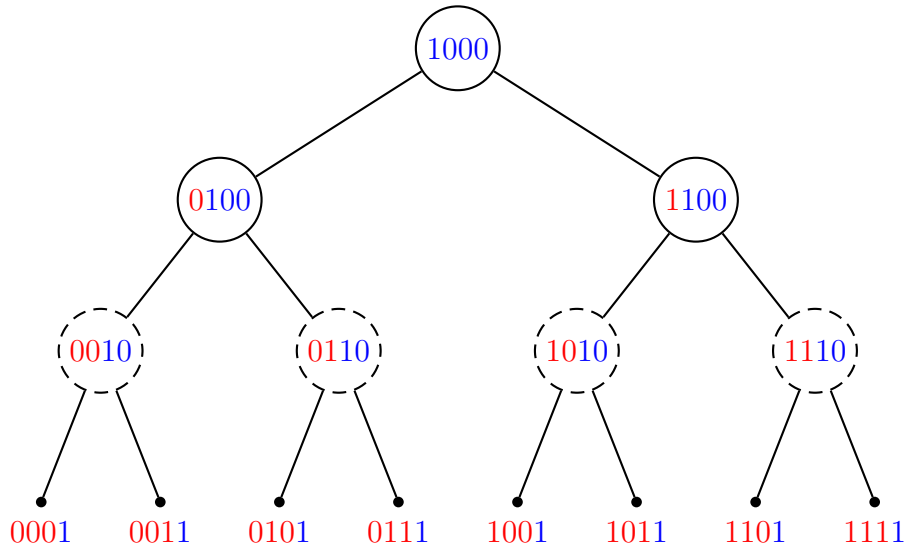


Abbildung 4.10: Beispiel: Inorder-Nummerierung eines vollständigen binären Baumes der Tiefe 3

Für die Inorder-Nummerierung gibt es auch eine zweite Interpretation. Sei dazu v ein Knoten des vollständigen binären Baumes B. Ohne Beschränkung der Allgemeinheit, interpretieren wir für jeden inneren Knoten ein Kind als linkes und ein Kind als rechtes Kind. Dies können wir tun, da in einem vollständigen binären Baum jeder innere Knoten genau zwei Kinder besitzt.

Wir betrachten die folgende Abbildung $\alpha : V(B) \to \{0, 1\}^*$ vermöge der folgenden Zuordnung. Folgen wir dem Pfad von der Wurzel zu einem Knoten v, so setzen wir für jede Verzweigung zu einem linken Kind eine 0, zu einem rechten Kind eine 1. Die so konstruierte Zeichenkette $\alpha(v)$ ordnen wir dem Knoten v zu. Dabei erhält die Wurzel als Zuordnung das leere Wort ε. Dies ist in Abbildung 4.10 ebenfalls illustriert. Die roten Zeichenreihen sind dabei die dem Knoten zugeordnete Zeichenreihe.

Die Inorder-Nummerierung von v erhält man dann als $\alpha(v) \cdot 10^{d-d(v)}$, wobei d die Tiefe des vollständigen binären Baumes d ist und $d(v)$ die Tiefe des Knotens v im

vollständigen binären Baum angibt. Somit haben die zugehörigen Bit-Strings (mit führenden Nullen) eine Länge von $d + 1$. In Abbildung 4.10 sind die angehängten Bit-Strings aus 10^* blau dargestellt. Es sei dem Leser zum Beweis überlassen, dass die obige Beschreibung der Inorder-Nummerierung korrekt ist.

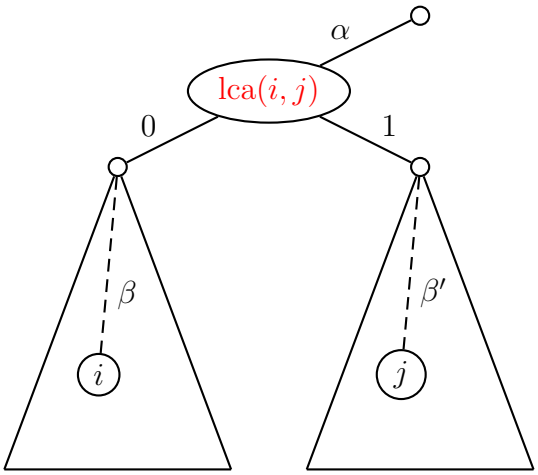


Abbildung 4.11: Skizze: Lowest Common Ancestor von i und j

Wie findet man jetzt mit Hilfe dieser Inorder-Nummerierung $\text{lca}(i, j) \notin \{i, j\}$? Wir betrachten dazu die Skizze in Abbildung 4.11. Wir bemerken, dass dann nach Definition der Pfad von der Wurzel des vollständigen binären Baumes zu i bzw. j bis zum Knoten $\text{lca}(i, j)$ identisch ist. Sei die zugeordnete Zeichenreihe gerade α. Also beginnt die Inorder-Nummerierung von i und j jeweils mit α. Da $\text{lca}(i, j) \notin \{i, j\}$ gilt, müssen sich die Inorder-Nummern von i und j in der darauf folgenden Bit-Position unterscheiden.

Betrachten wir jetzt die Inorder-Nummern als Bit-Strings von i, j und $\text{lca}(i, j)$ wie in Abbildung 4.12. Dann können wir den Bereich von α aus $i \oplus j$ als den führenden 0er Block herauslesen ($\oplus$ bezeichnete hier das exklusive Oder). Mit Hilfe eines Bitscans lässt sich die Position der ersten 1 von links (und somit im Wesentlichen die Länge von α) aus $i \oplus j$ bestimmen. Der Bit-String $i \vee (i \oplus j)$ liefert schon im Wesentlichen den Bit-String für den $\text{lca}(i, j)$. Wir müssen nur noch die letzten Bit-Positionen mittels eines logischen Unds auf 0 setzen.

Auch hier gilt wieder, dass sich die Inorder-Nummerierung mit einer Tiefensuche in linearer Zeit konstruieren kann und dass sich jede anschließende Lowest Common Ancestor Anfrage in konstanter Zeit beantworten lässt. Man beachte, dass diese Methode nur für $\text{lca}(i, j) \notin \{i, j\}$ funktioniert.

Kombiniert man die beiden Methoden in den vorher diskutierten Fällen, erhalten wir als Ergebnis das folgenden Lemma.

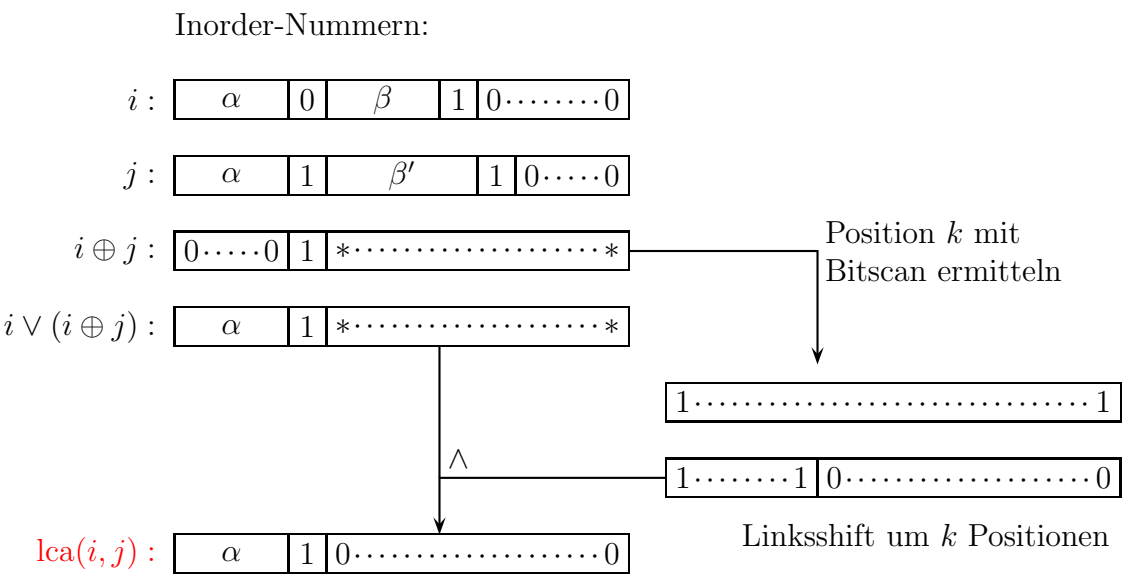


Abbildung 4.12: Skizze: Inorder-Nummern von i, j und $\mathrm{lca}(i, j)$

Lemma 4.23 *Sei B ein vollständiger binärer Baum. Nach einer Vorverarbeitung, die in linearer Zeit ausgeführt werden kann, kann jede Lowest Common Ancestor Query $\mathrm{lca}_B(i, j)$ in konstanter Zeit beantwortet werden.*

Wir wollen hier noch anmerken, dass Bitscans in modernen Prozessoren ein Standardfunktion (z.B. `bsf` und `bsr`), die in konstanter Zeit ausgeführt werden kann. Man überlegen sich, dass eine Implementierung mit Bitkomplexität $O(\mathrm{loglog}(n))$ möglich ist, also nicht aufwendiger ist, als die Addition von zwei Binärzahlen.

4.2.2 LCA-Queries auf beliebigen Bäumen

Wenden wir uns jetzt dem Fall zu, dass wir Lowest Common Ancestor Anfragen in einem beliebigen Baum beantworten wollen. Ohne Beschränkung der Allgemeinheit identifizieren wir im Folgenden die Knoten eines beliebigen Baumes durch seine DFS-Nummer. Ein Beispiel ist in Abbildung 4.13 angegeben.

Notation 4.24 *Sei $k \in \mathbb{N}$. Dann ist $h(k)$ die kleinste Bitposition (von rechts), so dass in der Binärdarstellung von k an dieser Position eine 1 steht.*

Man beachte, dass hierbei die rechteste Bitposition den Index 1 besitzt. Es gilt beispielsweise: $h(8) = 4$, $h(6) = 2$

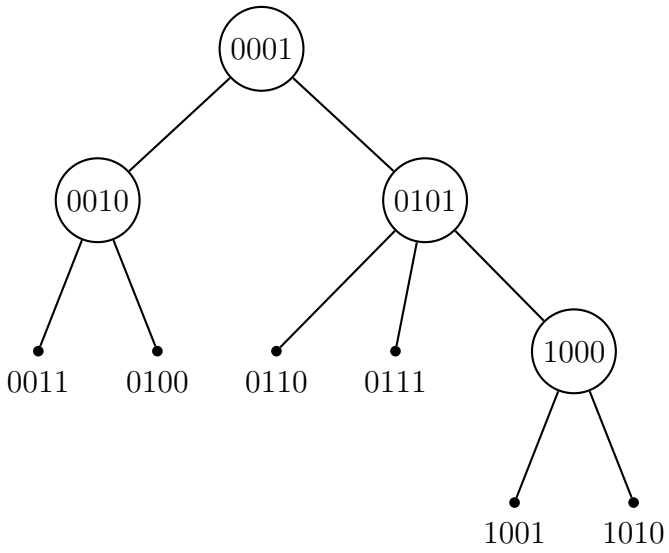


Abbildung 4.13: Beispiel: Ein Baum mit einer DFS-Nummerierung seiner Knoten

Wir wiederholen zuerst die Definition der Höhe eines Knotens in einem Baum und stellen einen Zusammenhang mit seiner Inorder-Nummer in einem vollständigen Baum her.

Definition 4.25 *In einem Baum ist die* Höhe *eines Knotens v die maximale Anzahl der Knoten eines längsten einfachen Pfades von v zu einem Blatt im Teilbaum von v.*

Beobachtung 4.26 *Sei v ein Knoten im vollständigen binären Baum, dann ist seine Höhe gleich $h(inorder(v))$, wobei $inorder(v)$ die Inorder-Nummer von v ist.*

Notation 4.27 *Sei v ein Knoten in einem beliebigen Baum, dann ist $I(v)$ der Knoten w im Teilbaum von v, dessen Wert $h(w)$ maximal ist.*

Beobachtung 4.28 *Ist w ein Nachfahre von v, dann gilt $h(I(w)) \leq h(I(v))$.*

In Abbildung 4.14 sind für einen beliebigen Baum diejenigen Knoten mit gleicher Farbe gekennzeichnet, die denselben Knoten $I(v)$ besitzen. Die schwarzen Knoten sind dabei jeweils als einelementige Mengen zu verstehen.

Lemma 4.29 *Für jeden Knoten $v \in V(T)$ existiert ein eindeutiger Knoten w in $T(v)$, so dass $h(w)$ maximal ist.*

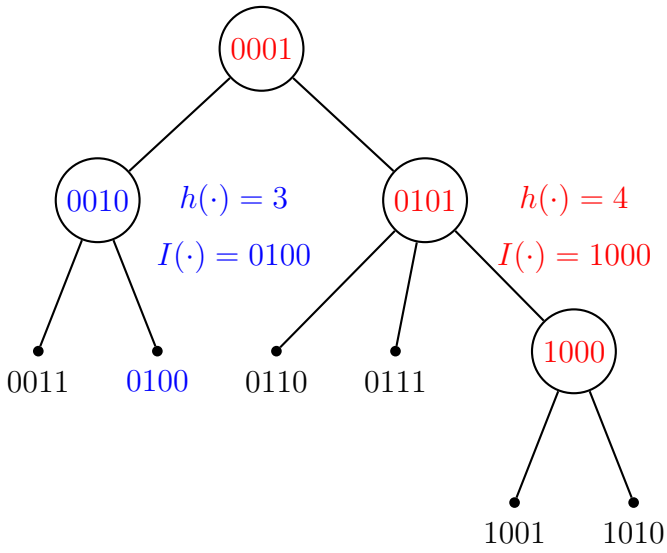


Abbildung 4.14: Beispiel: Knoten im Baum mit gleichem $I(v)$-Wert

Beweis: Für einen Widerspruchsbeweis nehmen wir an, dass $w \neq w' \in V(T(v))$ mit $h(w) = h(w')$ existieren, so dass für alle $u \in V(T(v))$ gilt $h(u) \leq h(w)$. Dann lassen sich w und w' wie folgt mit $\alpha, \alpha', \beta \in \{0,1\}^*$ und $k = h(w) - 1 = h(w') - 1$ schreiben:

$$\begin{aligned} w &= \alpha \cdot 0 \cdot \beta \cdot 10^k, \\ w' &= \alpha' \cdot 1 \cdot \beta \cdot 10^k. \end{aligned}$$

Da w und w' DFS-Nummern sind und sich im Teilbaum von $T(v)$ befinden, muss sich der Knoten $u := \alpha 10^{|\beta|+k+1}$ ebenfalls im Teilbaum $T(v)$ befinden, da $u \in [w : w']$. Da offensichtlich $h(u) > h(w) = h(w')$ gilt, erhalten wir den gewünschten Widerspruch. ■

Damit haben wir auch gezeigt, dass die Abbildung $v \mapsto I(v)$ wohldefiniert ist.

Korollar 4.30 *Sei $v \in V(T)$ und seien w und w' zwei Kinder von v. Dann gilt $|\{h(I(v)), h(I(w)), h(I(w'))\}| \geq 2$.*

Beweis: Für einen Widerspruchsbeweis sei $h(I(v)) = h(I(w)) = h(I(w'))$. Sei weiter $u = I(w)$ und $u' = I(w')$. Nach Definition von I gilt $u \neq u'$. Siehe hierfür auch die Skizze in Abbildung 4.15.

Gilt $h(u) = h(u')$, dann muss $h(I(v)) > h(I(w)) = h(I(w'))$ gelten, sonst hätten wir einen Widerspruch zu Lemma 4.29. Also erhalten wir den gewünschten Widerspruch zu $h(I(v)) = h(I(w)) = h(I(w'))$. ■

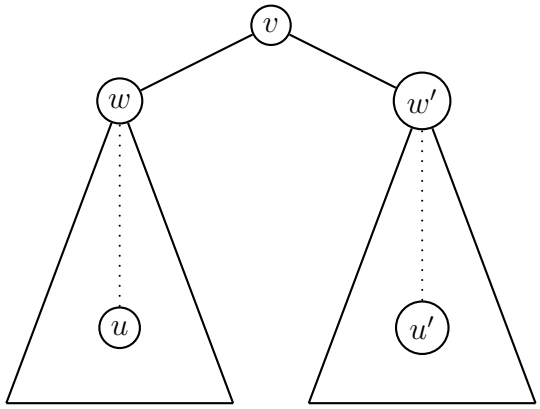


Abbildung 4.15: Skizze: Zum Beweis von Korollar 4.30

Korollar 4.31 *Die Partition der Knoten von T in Mengen mit gleichen $I(v)$-Wert ist eine Partition in Pfade von T.*

Beweis: Aus dem vorhergehenden Korollare folgt, dass nur höchstens ein Kind denselben I-Wert besitzen kann wie sein Elter. ■

Definition 4.32 *Die Pfade in der durch $I(\cdot)$ induzierten Partition heißen* Runs.

Aus Beobachtung 4.28 folgt unmittelbar die folgende Beobachtung:

Beobachtung 4.33 *Für jeden Knoten $v \in V(T)$ ist $I(v)$, der tiefste Knoten des Runs, der v enthält.*

Definieren wir nun noch den Kopf eines Runs.

Definition 4.34 *Der höchste Knoten eines Runs heißt* Kopf des Runs.

Wir werden im Folgenden mit Hilfe der Funktion $I(\cdot)$ die Knoten aus dem beliebigen Baum T auf einen vollständigen binären Baum abbilden. Mit Hilfe der dort schon bekannten Methode wollen wir dann die Lowest Common Ancestor Anfragen in einem beliebigen Baum T auf Lowest Common Ancestor Anfragen in einem vollständigen binären Baum B reduzieren. Dazu betrachten wir die folgende Abbildung:

$$I : V(T) \to V(B) : v \mapsto I(v).$$

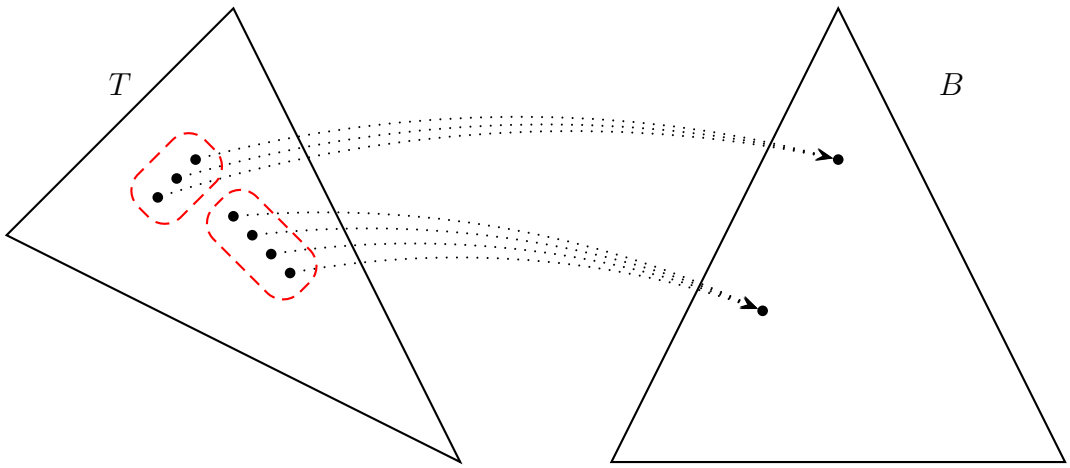


Abbildung 4.16: Skizze: Die Abbildung I von einem Baum in einen vollständigen binären Baum

Hierbei ist v eine DFS-Nummer in T und $I(v)$ wird als der Knoten in B mit Inorder-Nummer $I(v)$ interpretiert. Der vollständige binäre Baum hat dann eine Tiefe von $\lfloor \log(n) \rfloor$, wobei n die Anzahl der Knoten in T bezeichnet. Man beachte, dass die Abbildung I weder injektiv noch surjektiv sein muss. Diese Idee der Abbildung ist in der Skizze in Abbildung 4.16 illustriert.

Diese Abbildung I ist in Abbildung 4.17 noch einmal anhand des expliziten, uns bereits geläufigen Beispiels dargestellt.

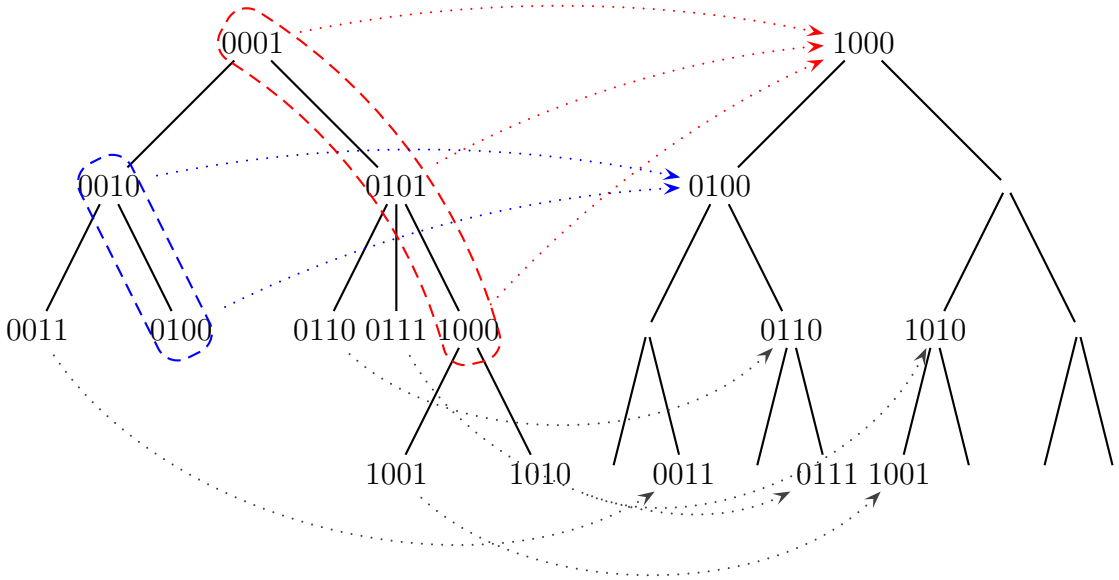


Abbildung 4.17: Beispiel: Die Abbildung I

4.2.3 Vorverarbeitung

Wir geben in diesem Abschnitt den Algorithmus für die Vorverarbeitung für die Lowest Common Ancestor Queries an. Der Algorithmus selbst ist in Abbildung 4.18 angegeben. Hierfür benötigten wir aber erst noch die folgende Notation.

Notation 4.35 *Sei T ein beliebiger Baum mit n Knoten. Für jeden Knoten $v \in V(T)$ ist A_v ein Bit-Feld der Länge $\lfloor \log(n) \rfloor + 1$. Dabei ist $A_v(i)$ genau dann gleich 1, wenn der Knoten v einen Vorfahren in T besitzt, der auf einen Knoten in Höhe i im Baum B abgebildet wird. Ansonsten ist $A_v(i) = 0$. Kurz es gilt:*

$$A_v(i) = \begin{cases} 1 & \textit{falls ein Vorfahre } u \textit{ von } v \textit{ in } T \textit{ existiert mit } h(I(u)) = i, \\ 0 & \textit{sonst.} \end{cases}$$

Für die Definition von A_v betrachten wir noch folgenden Beispiel für $v = (0011)_2 = 3$ aus Abbildung 4.17. Der Knoten $v = 3$ hat dann zwei Vorfahren, nämlich die Runs mit I-Wert $4 = (0100)_2$ und mit I-Wert $8 = (1000)_2$. Somit ist $A_3(4) = A_3(3) = 1$. Weiter gilt $A_3(1) = 1$, da der Knoten v ja ein Vorfahre von sich selbst ist. Da v jedoch keinen Vorfahren u mit $h(I(u)) = 2$ besitzt, ist $A_3(2) = 0$.

Schauen wir uns jetzt an, ob die in Abbildung 4.18 angegebene Vorverarbeitung effizient berechnet werden kann. Die DFS-Nummerierung lässt sich offensichtlich während einer Tiefensuche nebenbei mitberechnen. Der Wert $h(v)$ für jeden Knoten lässt sich ebenfalls pro Knoten in konstanter Zeit mit geeigneten Operationen auf Bit-Strings berechnen. Der Wert $I(v)$ lässt sich ebenfalls leicht berechnen, da nur

```
Preprocessing_LCA (tree T)
begin
    foreach (v ∈ V(T)) do                                   /* using DFS */
        // during descend
        determine dfs(v);                                   /* dfs-number of v */
        // during ascend
        determine h(v) and I(v);
        determine L(v) := w, where w is the head of a run containing v;

    foreach (v ∈ V(T)) do                                   /* using DFS */
        determine A_v;
end
```

Abbildung 4.18: Algorithmus: Vorverarbeitung zur LCA-Query

für den Knoten und seine Kinder derjenige Knoten bestimmt werden muss, der den maximalen h-Wert besitzt.

Für die Ermittlung des Feldes L hilft die folgende Beobachtung: Für jeden Knoten gibt es maximal ein Kind, das zu einem Run gehört, dessen Kopf noch unbekannt ist. Wir müssen also bei der Rückkehr aus der Tiefensuche nur maximal eine Liste von Knoten halten, deren Kopf noch unbekannt ist. Da der Knoten w genau dann ein Kopf eines Runs ist, wenn für seinen Elter u gilt $I(u) \neq I(w)$, lässt sich auch leicht feststellen, wann ein Kind zum Kopf eines Runs wird. Insgesamt kann das Feld L in linearer Zeit bestimmt werden. Diese Situation ist in Abbildung 4.19 skizziert.

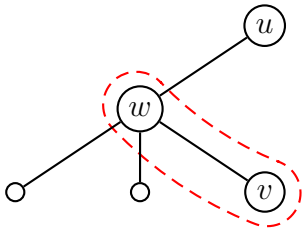


Abbildung 4.19: Skizze: Knoten w mit maximal einem Kind, für den der Kopf seines Runs unbekannt ist

Es stellt sich die Frage, wie sich Bit-Strings A_v effizient berechnen lassen. Wir werden diese Bit-Strings während einer Tiefensuche berechnen. Beim erstmaligen Aufsuchen eines Knotens v übernehmen wir den Bit-String des Elters (bei der Wurzel den Bit-String aus lauter Nullen) und setze eigenes Bit an Position $h(I(v))$. Beim endgültigen Verlassen des Knotens v löschen wir das beim Betreten gesetzte Bit an Position $h(I(v))$. Damit wird A_v für jeden Knoten v nach der Definition korrekt berechnet.

Lemma 4.36 *Sei T eine beliebiger Baum. Die Vorverarbeitung für Lowest Common Ancestor Anfragen in T kann in linearer Zeit implementiert werden.*

Wir werden in den folgend Abschnitten noch sehen, wozu wir die in der Vorverarbeitung berechneten Werte benötigen.

4.2.4 Beziehung zwischen Vorfahren in T und B

In diesem Abschnitt beweisen wir eine im Folgenden wichtige elementare Beziehung zwischen der Vorfahr-Relation in T und B.

Lemma 4.37 *Wenn z ein Vorfahre von x in T ist, dann ist $I(z)$ auch ein Vorfahre von $I(x)$ in B:*

Beweis: Wir unterscheiden zwei Fälle, je nachdem, ob $I(x) = I(z)$ gilt oder nicht.

Fall 1: Sei $I(x) = I(z)$. Da x und z auf denselben Knoten abgebildet werden, ist die Behauptung trivial.

Fall 2: Sei jetzt $I(x) \neq I(z)$. Da der Knoten z ein Vorfahre von x ist, gilt also aufgrund von Beobachtung 4.28, dass $h(I(z)) \geq h(I(x))$. Wäre $h(I(z)) = h(I(x))$, dann müsste $I(x) = I(z)$ sein, da der Knoten mit maximalen h-Wert im Teilbaum von z eindeutig ist. Also erhalten wir einen Widerspruch und es muss $h(I(z)) > h(I(x))$ gelten.

Wir betrachten jetzt die Bit-Strings von $I(z)$ und $I(x)$ wie in Abbildung 4.20. Dabei nehmen wir an, dass sich $I(z)$ und $I(x)$ an der Position k (gezählt von rechts) zum ersten Mal unterscheiden, wenn wir die Bit-Strings von links vergleichen.

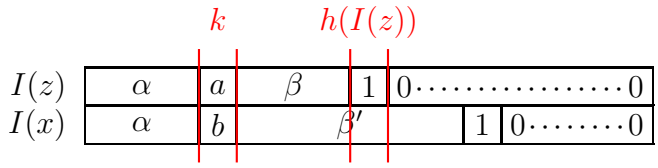


Abbildung 4.20: Skizze: Die Bit-Strings von $I(z)$ und $I(x)$

Wir zeigen jetzt, dass dann $k \leq h(I(z)$ gilt. Für einen Widerspruchsbeweis nehmen wir an, dass $k > h(I(z))$ gilt. Nach Voraussetzung gilt $I(z) \neq I(x)$. Wir unterscheiden jetzt noch zwei Unterfälle, je nachdem, ob $I(z) > I(x)$ oder $I(z) < I(x)$ gilt.

Fall 2a: Es gilt $I(z) > I(x)$. Wir betrachten jetzt den Wert $N := \alpha 10 \cdots 0$. Dann gilt $N \in (I(x) : I(z))$ und N wird in der Tiefensuche von z aus gefunden. Dann wäre aber wieder $h(N) > h(I(z))$, was nicht sein kann.

Fall 2b: Es gilt jetzt $I(z) < I(x)$. Ein analoge Beweisführung liefert denselben Widerspruch.

Wir erhalten also in jedem Fall einen Widerspruch und es muss daher $k \leq h(I(z))$ gelten. Also muss für die Bit-Strings von $I(z)$ und $I(x)$ die in Abbildung 4.21 angegebene Darstellung gelten. Das bedeutet aber nach unserer zweiten Interpretation der

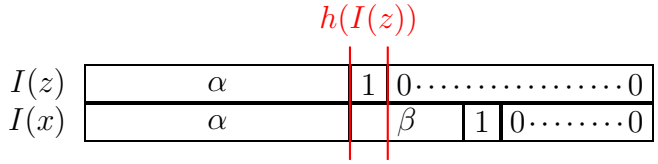


Abbildung 4.21: Skizze: Die Bit-Strings $I(z)$ und $I(x)$ im Falle $k \leq h(I(z))$

Inorder-Nummerierung des vollständigen binären Baumes, dass $I(z)$ ein Vorfahre von $I(x)$ ist. ■

4.2.5 Berechnung einer LCA-Query

Nun überlegen wir uns, wie wir eine Lowest Common Ancestor Query $\text{lca}_T(x, y)$ für den Baum T verarbeiten. Dazu werden wir für x bzw. y die Werte $I(x)$ und $I(y)$ bestimmen. Wie wir gesehen haben, gilt für den niedrigsten gemeinsamem Vorfahren $z = \text{lca}_T(x, y)$, dass $I(z)$ auch ein Vorfahre von $I(x)$ und $I(y)$ im vollständigen binären Baum B ist. Wir bestimmen also $b = \text{lca}_B(I(x), I(y))$ und hoffen, dass wir aus b den Knoten z (bzw. seine DFS-Nummer) rekonstruieren können. Diese Idee ist in Abbildung 4.22 noch einmal bildlich dargestellt.

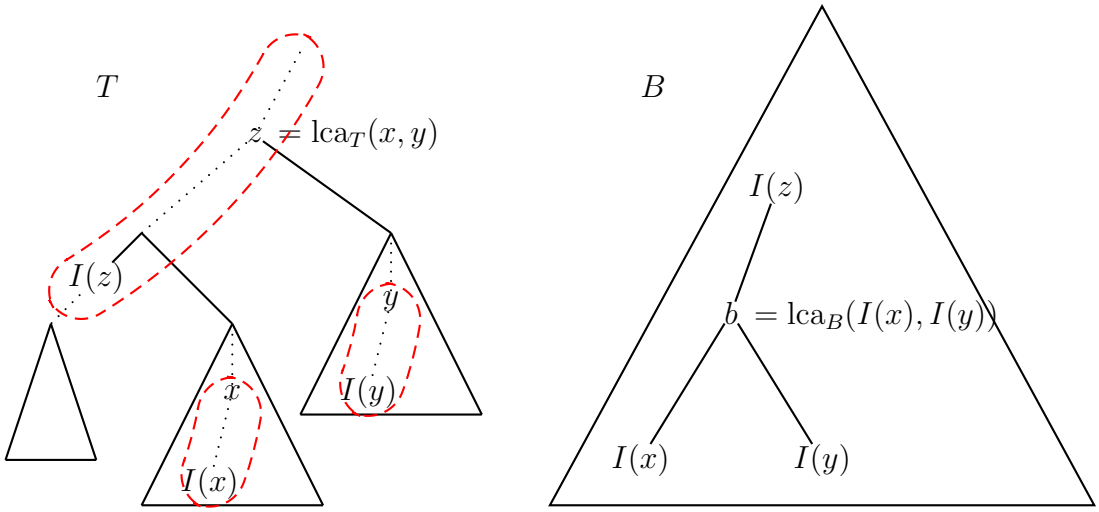


Abbildung 4.22: Skizze: Reduktion einer LCA-Anfrage in T auf eine in B

Der Algorithmus zur Beantwortung von $z = \text{lca}_B(x, y)$ geht dann wie folgt vor:

1. Bestimme $I(x)$ und $I(y)$.
2. Bestimme $b = \text{lca}_B(I(x), I(y))$.
3. Bestimme $h(I(z))$ aus b mit Hilfe des Wertes $h(b)$.
4. Bestimme z aus $I(z)$.

Wir wissen, dass $I(z)$ ein Vorfahre von b ist und wir müssen uns nur den richtigen Vorfahren von b und dessen korrektes Urbild auffinden. Dabei hilft uns das folgende Lemma.

Lemma 4.38 *Sei T ein beliebiger Baum mit n Knoten und seien $x, y, z \in V(T)$ mit $z = \text{lca}_T(x, y)$. Sei B ein vollständiger binärer Baum mit Tiefe $\lfloor \log(n) \rfloor$ und sei $b = \text{lca}_B(I(x), I(y))$. Weiter sei $j = \min\{k \geq h(b) : A_x(k) = A_y(k) = 1\}$. Dann gilt $h(I(z)) = j$.*

Beweis: Sei $k := h(I(z))$. Da z ein Vorfahre von x und y in Baum T ist, gilt $A_x(k) = A_y(k) = 1$ nach Definition von A. Da $I(z)$ ein Vorfahre von $I(x)$ und $I(y)$ ist und somit auch von $b = \text{lca}_B(I(x), I(y))$, gilt also $j \leq k = h(I(z))$.

Da $A_x(j) = 1$, muss ein Vorfahre x' von x in T existieren, so dass $I(x')$ ein Vorfahre von $I(x)$ in B ist und dass $I(x')$ die Höhe $j \geq h(b)$ in B hat. Also ist $I(x')$ ein Vorfahre von b in B auf Höhe j. Analog gibt es einen Vorfahren y' von y in T, so dass $I(y')$ ein Vorfahre von b in B auf Höhe j ist. Also muss $I(x') = I(y')$ gelten. Somit gehören x' und y' zum selben Run in T

Ohne Beschränkung der Allgemeinheit sei x' ein Vorfahre von y'. Dann ist x' ein Vorfahre von x und y. Da $z = \text{lca}_T(x, y)$, muss auch x' ein Vorfahre von z sein. Dann gilt auch $j = h(I(x')) \geq h(I(z)) = k$.

Somit gilt also insgesamt $j = k = h(I(z))$ und die Behauptung ist gezeigt. ■

Lemma 4.39 *Sei T ein beliebiger Baum und $x, y, z \in V(T)$. Sei $z = \text{lca}_T(x, y)$ und sei $h(I(z))$ bekannt, dann kann z in konstanter Zeit ermittelt werden.*

Beweis: Sei $\bar{x}$ bzw. $\bar{y}$ der erste Knoten auf dem Pfad von x bzw. y zu z, der im Run von z liegt. Es gilt dann $z = \bar{x}$ oder $z = \bar{y}$ und somit gilt $z = \min(\bar{x}, \bar{y})$.

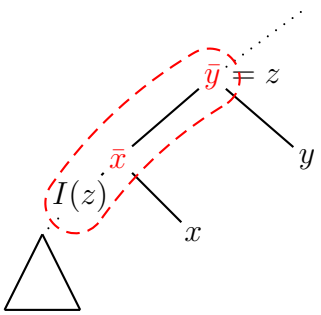


Abbildung 4.23: Skizze: Ermittlung von $z = \text{lca}_T(x, y)$

Um z zu ermitteln, benötigen wir also nur die Knoten $\bar{x}$ und $\bar{y}$. Siehe dazu auch die Abbildung 4.23.

Wie findet man jedoch $\bar{x}$ bzw $\bar{y}$? Sei dazu $j = h(I(z))$. Ist $h(I(x)) = j$, dann befinden sich x und z im selben Run. Also gilt $\bar{x} = x$. Analoges gilt natürlich auch für y.

Betrachten wir also den Fall $\bar{x} \neq x$. Sei w der erste Knoten von $\bar{x}$ zu x der nicht im Run von $\bar{x}$ liegt. Dann gilt $h(I(x)) \leq h(I(w)) < h(I(\bar{x})) = h(I(z))$. Diese Situation ist in Abbildung 4.24 skizziert.

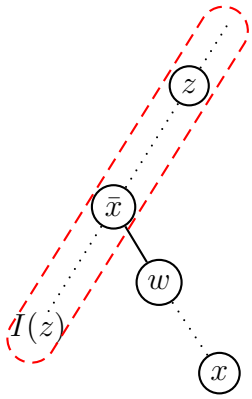


Abbildung 4.24: Skizze: Definition des Knotens w

Weiter gilt, dass $h(I(w))$ auf dem Pfad von x nach w sogar maximal ist. Dann ist $h(I(w)) = \max\{k < j : A_x(k) = 1\}$. Damit lässt sich $h(I(w))$ mit konstant vielen Operationen auf Bit-Strings berechnen. Zuerst müssen alle Bits ab Position j in A_x mittels einer geeigneten Verundung auf Null gesetzt werden. Mit einem anschließenden Bitscan kann dann die linkeste 1 gefunden werden.

Da x ein Nachfahre von w ist, muss auch $I(x)$ ein Nachfahre von $I(w)$ in B sein. Nach der Inorder-Nummerierung in B ergibt sich die folgenden Darstellung der Bit-Strings von $I(w)$ und $I(x)$ wie in Abbildung 4.25 angegeben.

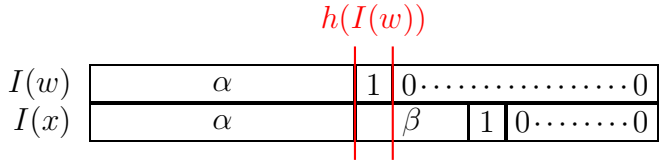


Abbildung 4.25: Skizze: Bit-Strings von $I(w)$ und $I(x)$

Aus $I(x)$ und $h(I(w))$ kann $I(w)$ leicht mit konstant vielen Operationen auf Bit-Strings berechnet werden. Weiterhin kann der Knoten w in T aus $I(w)$ mit Hilfe des des Feldes L bestimmt werden, da ja nach Konstruktion w der Kopf eines Runs

ist. Es gilt also $w = L(I(w))$ und $\bar{x} = \text{parent}(w)$ in T und die Behauptung ist bewiesen. ■

4.2.6 LCA-Query-Algorithmus

Wir wollen nun noch den vollständigen Algorithmus zu Berechnung der Anfrage $z = \text{lca}_T(x, y)$ in Abbildung 4.26 angeben.

Weiterhin wollen wir festhalten, dass der Baum B nur zu Illustration des Verfahrens dient, aber dessen Konstruktion nicht die Auswertung der Anfrage notwendig ist. Wie findet man also $h(b)$ ohne den Baum B zu konstruieren? Wir unterscheiden wieder zwei Fälle, je nachdem, ob $\text{lca}_B(I(x), I(y)) \in \{I(x), I(y)\}$ gilt oder nicht.

LCA (tree T, node x, y)

```
begin
    b := lca_B(I(x), I(y));
    j := min {k ≥ h(b) : A_x(k) = A_y(k) = 1};

    ℓ := min {i : A_x(i) = 1 ist linkeste 1};
    if (ℓ = j) then
        x̄ := x;
    else
        k := max {i < j : A_x(i) = 1};
        u := left(I(x), m, k + 1) · 1 · 0 ··· 0;          /* = I(w) */
        w := L(u);
        x̄ := parent(w);

    ℓ := min {i : A_y(i) = 1 ist linkeste 1};
    if (ℓ = j) then
        ȳ := y;
    else
        k := max {i < j : A_y(i) = 1};
        u := left(I(y), m, k + 1) · 1 · 0 ··· 0;          /* = I(w) */
        w =: L(u);
        ȳ := parent(w);

    z := min{x̄, ȳ};
    return z;
end
```

Abbildung 4.26: Algorithmus: LCA-Query $\text{lca}(x, y)$

Fall 1: Es gilt $\text{lca}_B(I(x), I(y)) \in \{I(x), I(y)\}$. Diese Situation ist in Abbildung 4.27 dargestellt. Man sieht leicht, dass sich $h(b)$ mit einer konstanten Anzahl von Operationen auf Bit-Strings berechnen lässt.

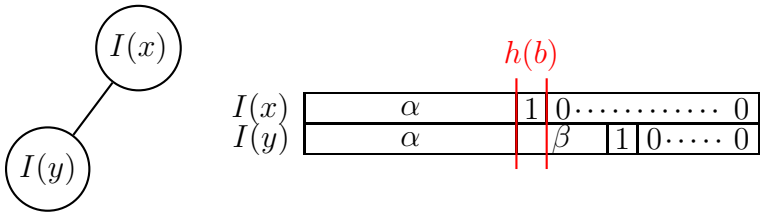


Abbildung 4.27: Skizze: Bestimmung von $h(b)$ im ersten Fall

Fall 2: Es gilt $\text{lca}_B(I(x), I(y)) \notin \{I(x), I(y)\}$. Diese Situation ist in Abbildung 4.28 dargestellt. Man sieht leicht, dass sich $h(b)$ mit einer konstanten Anzahl von Operationen auf Bit-Strings berechnen lässt.

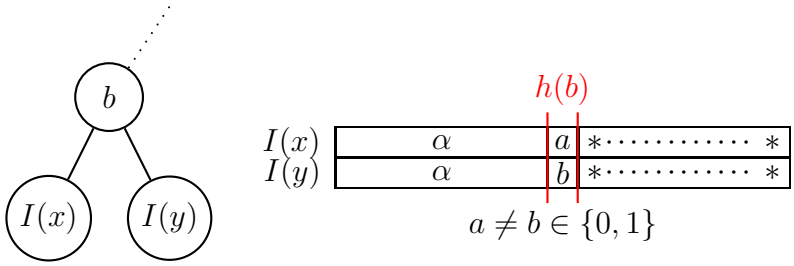


Abbildung 4.28: Skizze: Bestimmung von $h(b)$ im zweiten Fall

Halten wir das fundamentale Ergebnis dieses Abschnittes (den wir bereits anders im vorhergehenden Abschnitt bewiesen haben) im folgenden Satz fest.

Theorem 4.40 *Mit geeigneten Preprocessing, das Zeit $O(n)$ benötigt, kann jede LCA-Query in einem Baum mit n Knoten in konstanter Zeit ausgeführt werden.*

Zum Abschluss geben wir noch ein Beispiel an. Wir wollen den niedrigsten gemeinsamen Vorfahren der Knoten 3 und 7 in Abbildung 4.29 bestimmen.

1. Es gilt $b = 0100$ und damit $h(b) = 3$. Man kann $h(b)$ auch direkt aus 0011 und 0111 ablesen, da an Position 3 zum ersten Mal von links die beiden Bit-Strings differieren.

2. Es gilt $h(I(z)) = j = 4$ nach Lemma 4.38

3. Der Algorithmus in Abbildung 4.26 berechnet folgende Wert:

   $x = 0011$; $\ell = 1$; $k = 2$; $u = 00|10$; $L(u) = 0010$; $\bar{x} = 0001$;
   $y = 0111$; $\ell = 1$; $k = 1$; $u' = 011|1$; $L(u) = 0111$; $\bar{y} = 0101$;
   Dann gilt $z = \min(\bar{x}, \bar{y}) = 0001$.

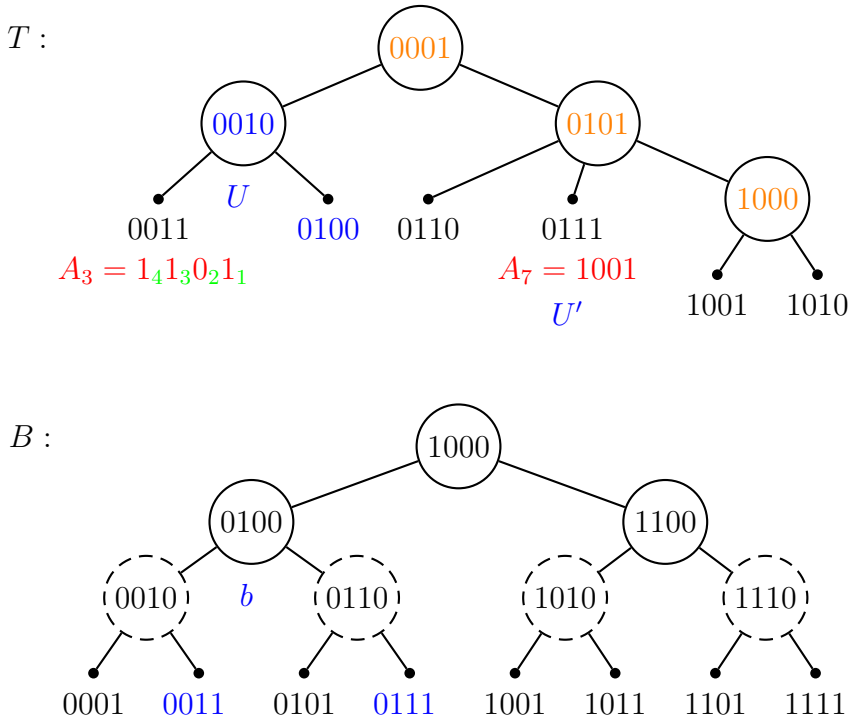


Abbildung 4.29: Beispiel: Die Anfrage $\text{lca}(3, 7) = \text{lca}(0011, 0111)$

Suffix-Arrays 5

5.1 Grundlegende Eigenschaften von Suffix-Arrays

In diesem Abschnitt wollen wir auf eine andere, zum Suffix-Baum ähnliche Datenstruktur, dem Suffix-Array, eingehen. Hauptmerkmal des Suffix-Arrays gegenüber dem Suffix-Baum ist seine speicherplatzsparendere Darstellung.

5.1.1 Was sind Suffix-Arrays?

Zunächst einmal müssen wir uns überlegen, was Suffix-Arrays überhaupt sind.

Definition 5.1 *Sei $t = t_1 \cdots t_n \in \Sigma^*$. Ein* Suffix-Array *für t ist ein Feld der Länge $n{+}1$, welche die Startpositionen aller Suffixe von $t\$$ in geordneter (lexikographischer) Reihenfolge enthält. Dabei gilt $\$ < a \in \Sigma$ mit $\$ \notin \Sigma$.*

Wir geben Beispiel für $t =$ MISSISSIPPI an.

$$
\begin{aligned}
A[0] &= 12 \mathrel{\widehat{=}} \$ \\
A[1] &= 11 \mathrel{\widehat{=}} \text{I\$} \\
A[2] &= 08 \mathrel{\widehat{=}} \text{IPPI\$} \\
A[3] &= 05 \mathrel{\widehat{=}} \text{ISSIPPI\$} \\
A[4] &= 02 \mathrel{\widehat{=}} \text{ISSISSIPPI\$} \\
A[5] &= 01 \mathrel{\widehat{=}} \text{MISSISSIPPI\$} \\
A[6] &= 10 \mathrel{\widehat{=}} \text{PI\$} \\
A[7] &= 09 \mathrel{\widehat{=}} \text{PPI\$} \\
A[8] &= 07 \mathrel{\widehat{=}} \text{SIPPI\$} \\
A[9] &= 04 \mathrel{\widehat{=}} \text{SISSIPPI\$} \\
A[10] &= 06 \mathrel{\widehat{=}} \text{SSIPPI\$} \\
A[11] &= 03 \mathrel{\widehat{=}} \text{SSISSIPPI\$}
\end{aligned}
$$

Man beachte, dass der erste Eintrag immer $n{+}1 \mathrel{\widehat{=}} \$$ ist. Man könnte die Erweiterung um das Endzeichen \$ auch weglassen, wenn man explizit vereinbart, dass ein Präfix

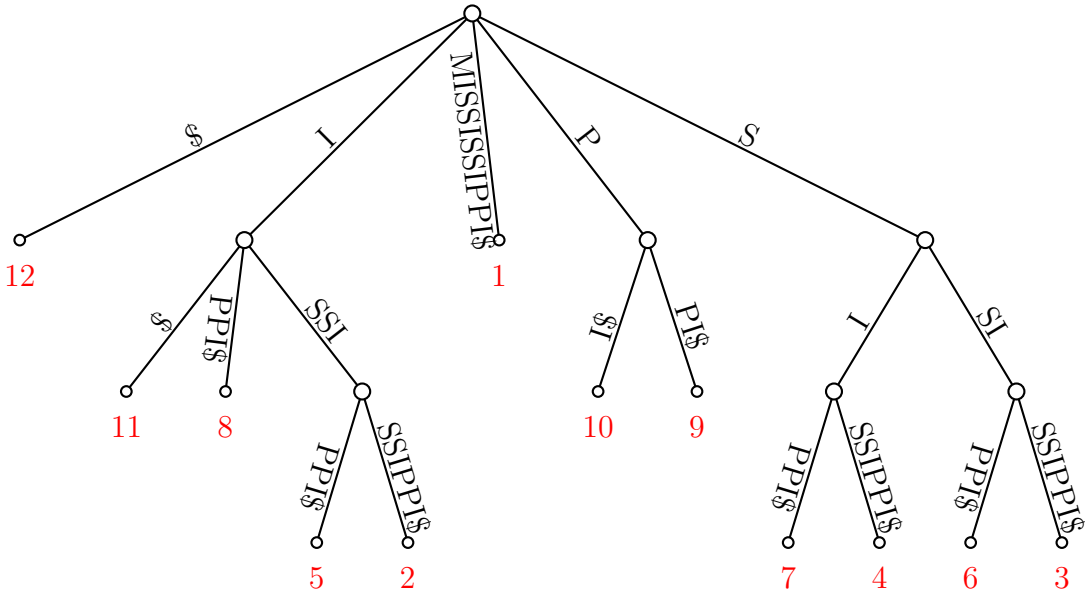


Abbildung 5.1: Beispiel: Suffix-Baum für MISSISSIPPI

eines anderen Wortes immer kleiner als das andere Wort ist. Oft ist dies in der Implementierung nur durch Beachtung von Sonderfällen zu handhaben, so dass die Ergänzung um das Endzeichen \$ oft hilfreich ist. Zum Vergleich ist in Abbildung 5.1 noch einmal der Suffix-Baum zu $t = \text{MISSISSIPPI}$ angegeben. Im Folgenden werden wir uns hauptsächlich damit beschäftigen, wie man solchen Suffix-Arrays effizient konstruiert und wie man ihn als Ersatz für Suffix-Bäume verwenden kann.

5.1.2 Konstruktion aus Suffix-Bäumen

Ein erster einfacher Ansatz zur Konstruktion eines Suffix-Arrays ist die Ableitung aus einem Suffix-Baum. Man konstruiert zunächst den Suffix-Baum und durchläuft diesen dann mit einer Tiefensuche. Berücksichtigt man dabei die Ordnung auf den Kindern eines Knotens, die durch die lexikographische Ordnung der Markierungen der ausgehende Kanten gegeben ist, so werden alle Blätter (bzw. die zugehörigen Suffixe) in lexikographischer Ordnung aufgefunden.

Theorem 5.2 *Ein Suffix-Array für ein gegebenes Wort kann in linearer Zeit konstruiert werden.*

Da der Suffix-Baum jedoch selbst schon relativ viel Platz benötigt, ist man an speicherplatzsparenderen Varianten zur Erstellung von Suffix-Arrays interessiert.

05.12.24

5.1.3 Algorithmus von Manber und Myers

In diesem Abschnitt stellen wir einen ersten direkten Algorithmus zur Konstruktion von Suffix-Arrays vor. Wir werden dabei zunächst auf die Idee des Bucket-Sorts zurückgreifen. Wir initialisieren das Feld A fortlaufend mit den Startpositionen aller Suffixe, absteigend nach ihrer Länge, also $A[i] = i + 1$ für $i \in [0 : n]$. Dann sortieren wir die Zeichenreihen nach dem ersten Zeichen. Eine platzsparende Implementierung mit maximal zwei Feldern der Länge $n + 1$ und einem Feld der Länge $|\Sigma|$ sei dem Leser zur Übung überlassen. Das Ergebnis dieses Bucket-Sorts für $t =$ MISSISSIPPI ist in Abbildung 5.2 illustriert.

	\$	I				M	P		S			
i	0	1	2	3	4	5	6	7	8	9	10	11
$A[i]$	12	2	5	8	11	1	9	10	3	4	6	7

Abbildung 5.2: Beispiel: Bucket-Sort nach dem ersten Zeichen für MISSISSIPPI

Als Ergebnis des Bucket-Sorts nach dem ersten Zeichen erhalten wir eine Menge von Buckets, in denen sich nur Suffixe befinden, die mit demselben Zeichen beginnen. Jetzt müssen wir nach dem zweiten Zeichen sortieren, allerdings nur innerhalb der Buckets. Wir könnten hierfür wieder einen Bucket-Sort verwenden, es gibt aber eine trickreichere, effizientere Variante.

Seien $t^i = t_i \cdots t_n\$$ und $t^j = t_j \cdots t_n\$$ zwei Wörter eines Buckets, dann gilt $t_i = t_j$. Um einen Bucket mit den beiden Suffixe t^i und t^j nach dem zweiten Zeichen zu sortieren, müssen wir t_{i+1} mit t_{j+1} vergleichen. Diesen Vergleich haben wir jedoch schon implizit ausgeführt. Dieser Vergleich hat dasselbe Ergebnis wie der Vergleich von $t^{i+1} = t_{i+1} \cdots t_n\$$ mit $t^{j+1} = t_{j+1} \cdots t_n\$$. Wir müssen also nur in jedem Bucket die Wörter t^i nach dem Ergebnis der Sortierung von t^{i+1} nach dem ersten Zeichen anordnen.

Dabei werden wir uns etwas geschickter anstellen, indem wir alle Indexpositionen des sortierten Feldes A sortiert nach dem ersten Zeichen durchgehen. Ist $A[i] = j$, dann schieben wir den Index $j - 1$ in seinem Bucket an den aktuellen Anfang des Buckets und aktualisieren den Anfang des Buckets. Damit erhalten wir eine neue Einteilung von Buckets, in denen die enthaltenen Suffixe jeweils in den ersten beiden Zeichen übereinstimmen.

Nun sortieren wir innerhalb der neu erhaltenen Buckets nicht nach dem dritten, sondern gleich nach dem dritten und vierten Zeichen. Um diesen Bucket nach dem dritten und vierten Zeichen zu sortieren, müssen wir für t^i und t^j die Wörter $t_{i+2}t_{i+3}$ mit $t_{j+2}t_{j+3}$ vergleichen. Diesen Vergleich haben wir wiederum bereits implizit ausgeführt. Dieser Vergleich hat dasselbe Ergebnis wie der Vergleich von $t^{i+2} = t_{i+2} \cdots t_n\$$

A	0	1	2	3	4	5	6	7	8	9	10	11
	12	2	5	8	11	1	9	10	3	4	6	7
12		11										
2						1						
5									4			
8									4	7		
11							10					
1												
9		11	|8									
10							10	|9				
3		11	|8	|2								
4									4	7	|3	
6		11	|8	|2	5							
7									4	7	|3	6
	12	11	|8	|2	5	1	10	|9	4	7	|3	6
	\$	I	I	I	I	M	P	P	S	S	S	S
		\$	P	S	S	I	I	P	I	I	S	S
			P	S	S	S	\$	I	S	P	I	I

Abbildung 5.3: Beispiel: Sortierung während der ersten Phase nach dem zweiten Zeichen

mit $t^{j+2} = t_{j+2} \cdots t_n\$$. Wir müssen also nur in jedem Bucket die Wörter t^i nach dem Ergebnis der Sortierung von t^{i+2} nach den ersten beiden Zeichen anordnen.

Dies werden wir wieder etwas geschickter machen, indem wir alle Indexpositionen des nach den ersten beiden Zeichen sortierten Feldes A durchgehen. Ist $A[i] = j$, dann schieben wir den Index $j - 2$ in seinem Bucket an den aktuellen Anfang des Buckets und aktualisieren den Anfang des Buckets. Damit erhalten wir eine neue Einteilung von Buckets, die jeweils nach den ersten vier Zeichen sortiert sind.

Somit verdoppelt sich in jeder Phase die Anzahl der Zeichen, nach denen die Suffixe sortiert sind. Nach maximal $\lceil \log(n) \rceil$ Phasen ist das Suffix-Array also fertiggestellt, d.h. wir erhalten $n + 1$ Buckets, die jeweils genau einen Suffix beinhalten.

In Abbildung 5.3 ist die erste Phase noch einmal exemplarisch dargestellt. In der ersten Zeile sind die Buckets aus dem Bucket-Sort gegeben. In jeder Zeile wird das jeweils nächste Element j aus dem bisher sortierten Feld A verwendet. Dann wird der Index $j-1$ innerhalb seines Buckets nach vorne bewegt. In der letzten Zeile sind dann die resultierende Sortierung und die zugehörigen Anfänge der Suffixe dargestellt.

Die grauen Linien geben dabei die alten Bucket-Grenzen an. Wenn zwischen zwei Elementen, die nach vorne geschoben werden, eine alte Bucket-Grenze übersprungen wird, induziert dies auch eine neue Bucket-Grenze in den neu sortierten Buckets.

A	0 12	1 11	2 8	3 2	4 5	5 1	6 10	7 9	8 4	9 7	10 3	11 6
12							10					
11								9				
8											6	
2 5											6	|3
1												
10			8									
9									7			
4 7				2 2	 5							
3 6						1			7	|4		
	12	11	8	2	5	1	10	9	7	|4	6	|3
	$	I $	I P P I $	I S S I S	I S S I P	M I S S I	P I $	P P I $	S I P P I	S I S S I	S S I P P	S S I S S

Abbildung 5.4: Beispiel: Sortierung während der zweiten Phase nach dem dritten und vierten Zeichen

In Abbildung 5.4 ist die zweite Phase noch einmal anhand unseres Beispiels dargestellt. In der ersten Zeile sind die Buckets aus der ersten Phase gegeben. In jeder Zeile wird das jeweils nächste Element j aus dem bisher sortierten Feld A verwendet. Dann wird der Index $j-2$ innerhalb seines Buckets nach vorne bewegt. In der letzten Zeile sind dann die resultierende Sortierung und die zugehörigen Anfänge der Suffixe dargestellt. Wie man Abbildung 5.4 entnimmt, ist das Suffix-Array noch nicht erstellt, da sich t^2 und t^5 noch in einem Bucket befinden. Eine weitere Phase der Sortierung nach dem fünften mit achten Zeichen ist hier ausreichend.

In Abbildung 5.5 ist eine einfache Version des Algorithmus von Manber und Myers in Pseudo-Code angegeben. Hierbei bezeichnet A das Feld der Länge n, in dem am Ende das Suffix-Array stehen soll. Das Feld A' der Länge n ist ein Hilfsfeld, um die Buckets von A zu sortieren. Das Ergebnis wird in A' zwischengespeichert, bevor dann am Ende einer Phase A' wieder auf A kopiert wird. B ist ein Bit-Feld der Länge n, das an Position i genau dann TRUE ist, wenn an Position i ein neuer Bucket startet. Das Feld B' enthält die für A' aktualisierte Version von B'. Das Feld R der Länge n gibt an, an welcher Stelle in A der Index i steht. Es gilt also genau dann $A[i]=j$, wenn $R[j]=i$ ist. Damit gilt insbesondere auch $A[R[i]]=i=R[A[i]]=i$. Das

Manber_Myers (char $t[]$, int n)

```
begin
    // sort all suffixes of t according to their first character
    for (i := 0; i ≤ n; i++) do
        A[i] = i + 1;
    A := bucketsort(t);

    // initialize B
    B[0] := TRUE;
    for (i := 1; i ≤ n; i++) do
        B[i] := (t[A[i]] ≠ t[A[i − 1]]);

    for (k := 0; k ≤ log(n); k++) do
        for (i := 0; i ≤ n; i++) do
            R[A[i]] := i;                                   /* compute R */
            A'[i] := 0;                                     /* initialize A' */
            L[i] := (B[i])?i : L[i − 1];                    /* initialize L */
            B'[i] := B[i];

        for (i := 0; i ≤ n; i++) do
            if (A[i] − 2^k > 0) then
                j := R[A[i] − 2^k];              /* move suffix from position j */
                p := (B[j])?L[j] : L[L[j]];                 /* to position p */
                b := (B[i])?i : L[i];          /* bucket name of current suffix */
                b' := A'[p];                  /* bucket name of previous suffix */
                A'[p] := A[j];          /* move suffix to front of its bucket */
                if (not B[p + 1]) then
                    A'[p + 1] := b;    /* save bucket name of current suffix */
                B'[p] := (B'[p] or (b ≠ b'));                   /* update B' */
                j := (B[j])?j : L[j];
                L[j]++;          /* position for next suffix within bucket */

        // copy A' and B' to A and B, respectively
        for (i := 0; i ≤ n; i++) do
            if (A'[i] > 0) then
                A[i] := A'[i];
                B[i] := B'[i];
end
```

Abbildung 5.5: Alternativer Algorithmus von Manber und Myers

Feld L verweist immer auf die linkeste Position innerhalb seines Buckets. Genauer gesagt gilt $L[i] = \max\{j \leq i : B[j] = \text{TRUE}\}$. Es gilt also genau dann $L[i] = i$, wenn $B[i] = \text{TRUE}$ gilt. Ansonsten ist $L[i] < i$. Weiter verwenden wir hier die C-Notation $(b)\,?\,x : y$ die den Wert x ergibt, wenn der Boolesche Ausdruck b wahr ist und y sonst.

Während des Algorithmus wird jedoch $L[i]$ für $B[i] = \text{TRUE}$ auch dazu missbraucht, um den ersten freien Platz innerhalb seines Bucket anzugeben. Auf diese Position wird dann der nächste Wert innerhalb des Buckets nach vorne geschoben. Weiterhin kann es leere Stellen in A' geben, nämlich genau dann, wenn in A an dieser Position ein Suffix mit Start-Position größer als $n - 2^k$ steht. Da diese Suffixe jedoch eine Länge kleiner als 2^k besitzen, sind diese in A bereits korrekt sortiert und müssen beim Umkopieren von A' nach A nicht mehr berücksichtigt werden. Um solche Positionen in A' leicht feststellen zu können, wurde A' mit 0 (einer nichtexistenten Startposition) initialisiert.

Um die neu entstehenden Bucket-Grenzen bei der Sortierung feststellen zu können, speichern wir hinter der Start-Position des Suffixes den Namen des Buckets, aus dem der Suffix stammte, um den vorhergehenden Suffix an den Anfang des Buckets zu bewegen. Damit können wir leicht feststellen, wann eine neue Bucketgrenze entsteht, nämlich genau dann, wenn zwei aufeinander folgende Suffixe aufgrund von Suffixen aus verschiedenen Buckets entstehen. Dies ist auch in Abbildung 5.6 illustriert.

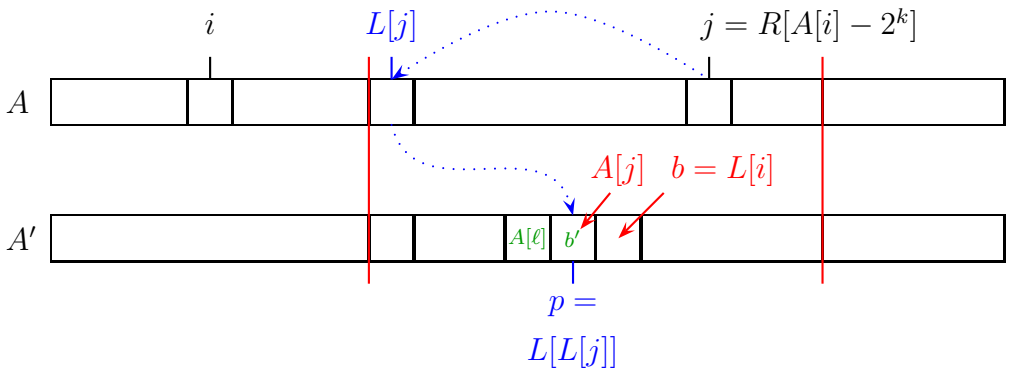


Abbildung 5.6: Skizze: Bucket-Sortierung im Algorithmus von Manber und Myers

Theorem 5.3 *Mit dem Algorithmus von Manber und Myers kann ein Suffix-Array für eine gegebene Zeichenreihe der Länge n in Zeit $O(n \log(n))$ erstellt werden.*

Man kann zeigen, dass man die Implementierung soweit verbessern kann, dass neben dem Eingabwort nur noch zwei Felder der gleichen Länge von Integers benötigt

werden. (Sowie ein Bit-Feld der gleichen Länge, dass jedoch als Vorzeichen in den anderen Integer-Feldern versteckt werden kann.)

Unter geeigneten Annahmen über die Verteilung von Eingabewörtern kann man zeigen, dass man erwarten kann, dass sich alle Suffixe spätestens nach $O(\log(n))$ Zeichen unterscheiden. Im Erwartungswert sind also nur $O(\text{loglog}(n))$ statt $O(\log(n))$ Phasen notwendig. Somit erhalten wir den folgenden Satz.

Theorem 5.4 *Mit dem Algorithmus von Manber und Myers kann ein Suffix-Array für eine gegebene Zeichenreihe der Länge n mit erwarteter Laufzeit $O(n \log\log(n))$ erstellt werden.*

Die erwartete Laufzeit kann mit einem Trick noch weiter auf eine lineare Laufzeit auf Kosten einer Speicherplatzerhöhung um einen konstanten Faktor gesenkt werden. Hierfür verweisen wir den Leser jedoch auf die Originalliteratur von Manber und Myers, da wir im Folgenden noch einen einfachen Linearzeit-Algorithmus zur Konstruktion von Suffix-Arrays vorstellen werden.

5.2 Algorithmus von Ko und Aluru (*)

In diesem Abschnitt stellen wir einen direkten Algorithmus zur Konstruktion von Suffix-Arrays in Linearzeit vor. Dieser Abschnitt ist nur der Vollständigkeit halber Teil des Skripts, da der Stoff Bestandteil der Vorlesung im Wintersemester 2003/04 war.

5.2.1 Typ S und Typ L Suffixe

Zunächst einmal werden wir die Suffixe von $t\$$ in zwei verschiedene Typen klassifizieren.

Definition 5.5 *Sei $t = t_1 \cdots t_n \in \Sigma^*$, dann heißt $t^i := t_i \cdots t_n\$$ vom* Typ S*, wenn $t^i < t^{i+1}$ gilt, und vom* Typ L*, wenn $t^i > t^{i+1}$ gilt. Der Suffix $t^{n+1} = \$$ ist sowohl vom Typ S als auch vom Typ L.*

Man beachte, dass nur der Suffix \$ beiden Typen angehört. Wir zeigen jetzt noch, dass man alle Suffixe von $t\$$ in Linearzeit klassifizieren kann.

Lemma 5.6 *Sei $t = t_1 \cdots t_n \in \Sigma^*$. Alle Suffixe von $t\$$ können in Zeit $O(n)$ in die Typen S und L klassifiziert werden.*

Beweis: Sei t^i ein Suffix von $t\$$. Wir unterscheiden jetzt zwei Fälle, je nachdem, ob $t_i = t_{i+1}$ gilt oder nicht.

Fall 1: Es gilt $t_i \neq t_{i+1}$. Dann ist t^i vom Typ S, wenn $t_i < t_{i+1}$ und andernfalls vom Typ L.

Fall 2: Es gilt $t_i = t_{i+1}$. Sei $j > i$ die kleinste Position mit $t_j \neq t_i$, d.h. es gilt $j = \min\{k \in [i : n+1] : t_k \neq t_i\}$. Dieser Fall ist in Abbildung 5.7 illustriert.

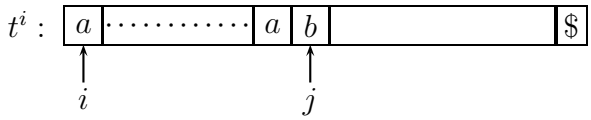


Abbildung 5.7: Skizze: 2. Fall im Beweis der Klassifizierung nach Typ S bzw. Typ L

Gilt nun $t_i < t_j$ dann sind $t^i, \ldots, t^{j-1}$ alle vom Typ S. Gilt andererseits $t_i > t_j$, dann sind $t^i, \ldots, t^{j-1}$ alle vom Typ L.

Offensichtlich lassen sich somit alle Suffixe mittels eines Durchlaufs über t in linearer Zeit in die Typen S und L klassifizieren. ■

Das folgende Lemma liefert uns für die Suffixe, die mit demselben Zeichen beginnen, eine einfache Aussage, ob $t^i < t^j$ gilt oder nicht, die nur auf der Klassifizierung nach den Typen S und L basiert.

Lemma 5.7 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und seien t^i und t^j zwei verschiedene Suffixe von $t\$$. Sei weiter t^i vom Typ L und t^j vom Typ S. Gilt $t_i = t_j$, dann ist $t^i < t^j$.*

Beweis: Wir führen den Beweis durch Widerspruch. Wir nehmen also an, dass $t^i > t^j$ gilt. Sei $c = t_i = t_j \in \Sigma$ und seien $c_1 \neq c_2 \in \Sigma$ die beiden Zeichen, in denen sich die Wörter t^i und t^j das erste Mal (von links nach rechts) unterscheiden. Sei weiter $t^i = c \cdot \alpha \cdot c_1 \cdot \beta$ und $t^i = c \cdot \alpha \cdot c_2 \cdot \beta'$ mit $\alpha, \beta, \beta' \in \Sigma^*$

Fall 1: Es gilt $\alpha \notin \{c\}^*$. Sei c_3 das erste Zeichen in α, das ungleich c ist. Da t^i vom Typ L ist, muss $c > c_3$ gelten, wie man in der folgenden Abbildung 5.8 leicht sieht.

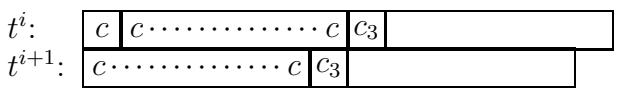


Abbildung 5.8: Skizze: t^i im Fall 1 $\alpha \notin \{c\}^*$

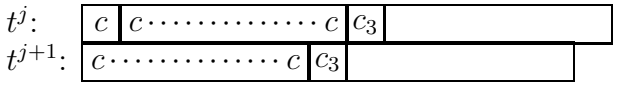


Abbildung 5.9: Skizze: t^j im Fall 1 $\alpha \notin \{c\}^*$

Da t^j vom Typ S ist, muss $c < c_3$ gelten, wie man in der folgenden Abbildung 5.9 leicht sieht. Somit muss sowohl $c < c_3$ als auch $c > c_3$ gelten, was den gewünschten Widerspruch liefert.

Fall 2: Es gilt $\alpha = c^{|\alpha|}$. Da t^i vom Typ L ist, gilt $c > c_1$. Da t^j vom Typ S ist, gilt $c < c_2$ Dies sieht man leicht in der folgenden Abbildung 5.10.

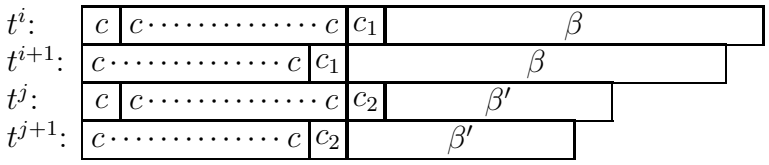


Abbildung 5.10: Skizze: t^i und t^j im Fall 2

Also muss $c_1 < c < c_2$ gelten. Da aber nach Annahme $t_i > t_j$ gilt, muss $c_1 > c_2$ sein. Dies liefert den gewünschten Widerspruch. ■

Als Folgerung dieses Lemmas erhalten wir das folgende Korollar, dass uns Auskunft über die Verteilung der Suffixe innerhalb eines Buckets liefert, wenn wir alle Suffixe zunächst mit einem Bucketsort nach dem ersten Zeichen sortiert haben.

Korollar 5.8 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei $S_c = \{t^i : t_i = c\}$ die Menge aller Suffixe von $t\$$, die mit $c \in \Sigma$ beginnen. Im Suffix-Array von $t\$$ erscheinen alle Suffixe aus S_c vom Typ L vor denjenigen vom Typ S.*

5.2.2 Sortieren der Suffixe mit sortierten Typ S/L Suffixen

Der Algorithmus von Ko und Aluru benötigt im Wesentlichen die folgenden drei Felder:

- Ein Feld A der Länge $n + 1$, das die Startpositionen aller Suffixe von $t\$$ in beliebiger Reihenfolge enthält.
- Ein Feld R der Länge $n + 1$, mit $A[R[j]] = j$.

- Ein Feld C der Länge maximal $n+1$, das die Startpositionen aller Suffixe von $t\$$ vom Typ S enthält und bereits sortiert ist.

Dabei ist das Feld A mehr oder weniger gegeben, das Feld R lässt sich leicht aus dem Feld A in Linearzeit berechnen. Wie wir das Feld C erhalten, werden wir uns später noch genauer überlegen.

Wir werden jetzt zeigen, wie man mithilfe dieser drei Felder das Feld A sortieren kann. Dazu werden im Wesentlichen die folgenden drei Schritte ausgeführt:

1. Führe einen Bucket-Sort von A nach dem ersten Zeichen der Suffixe aus.

   Wie wir schon gesehen haben, kann dies in linearer Zeit durchgeführt werden.

2. Durchlaufe das Feld C von rechts nach links (vom längsten zum kürzesten Suffix) und verschiebe den aktuell betrachteten Suffix $c \in C$ an das aktuelle Ende seines Buckets, indem die Startposition c mit dem Suffix unmittelbar vor dem aktuellen Ende vertauscht wird (siehe Abbildung 5.11). Die schraffierten Bereiche bezeichnen hierbei bereits abgearbeitete Bereiche des Feldes C bzw. die bereits korrekt sortierten Suffixe vom Typ S im aktuellen Bucket.

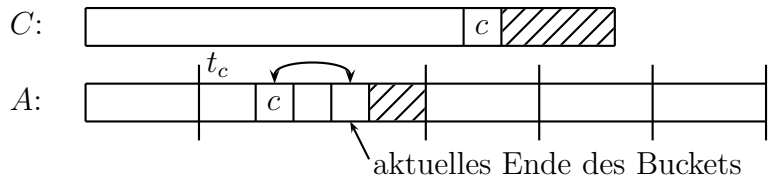


Abbildung 5.11: Skizze: Vertauschen des aktuell betrachteten Suffixes an den Anfang des Ende des Buckets

   Offensichtlich ist die Laufzeit dieses Schritts durch die Länge des Feldes C beschränkt, also ebenfalls in linearer Zeit ausführbar.

3. Durchlaufe das Feld A von links nach rechts (also aufsteigend). Für jedes $A[i]$ mit $t^{A[i]-1}$ vom Typ L verschiebe $A[i]-1$ an den aktuellen Beginn seines Buckets (siehe Abbildung 5.12). Hierbei stellen die schraffierten Bereiche den

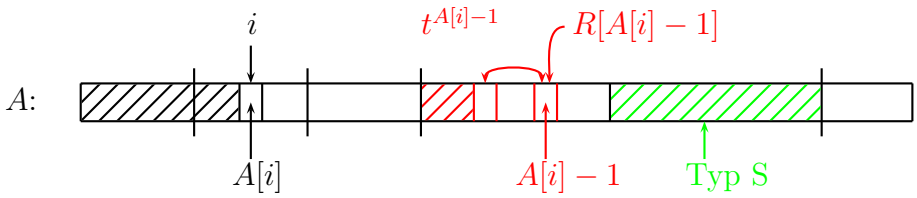


Abbildung 5.12: Skizze: Verschiebung der Typ L Suffixe an den Anfang ihrer Buckets

bereits abgearbeiteten Teil des Feldes A bzw. die bereits korrekt sortierten Suffix vom Typ S und vom Typ L im aktuell betrachteten Bucket dar.

Auch dieser Schritt ist offensichtlich in linearer Zeit durchführbar.

In Abbildung 5.13 ist der Algorithmus anhand eines Beispiels illustriert. Der gesamte Algorithmus hat also offensichtlich eine lineare Laufzeit.

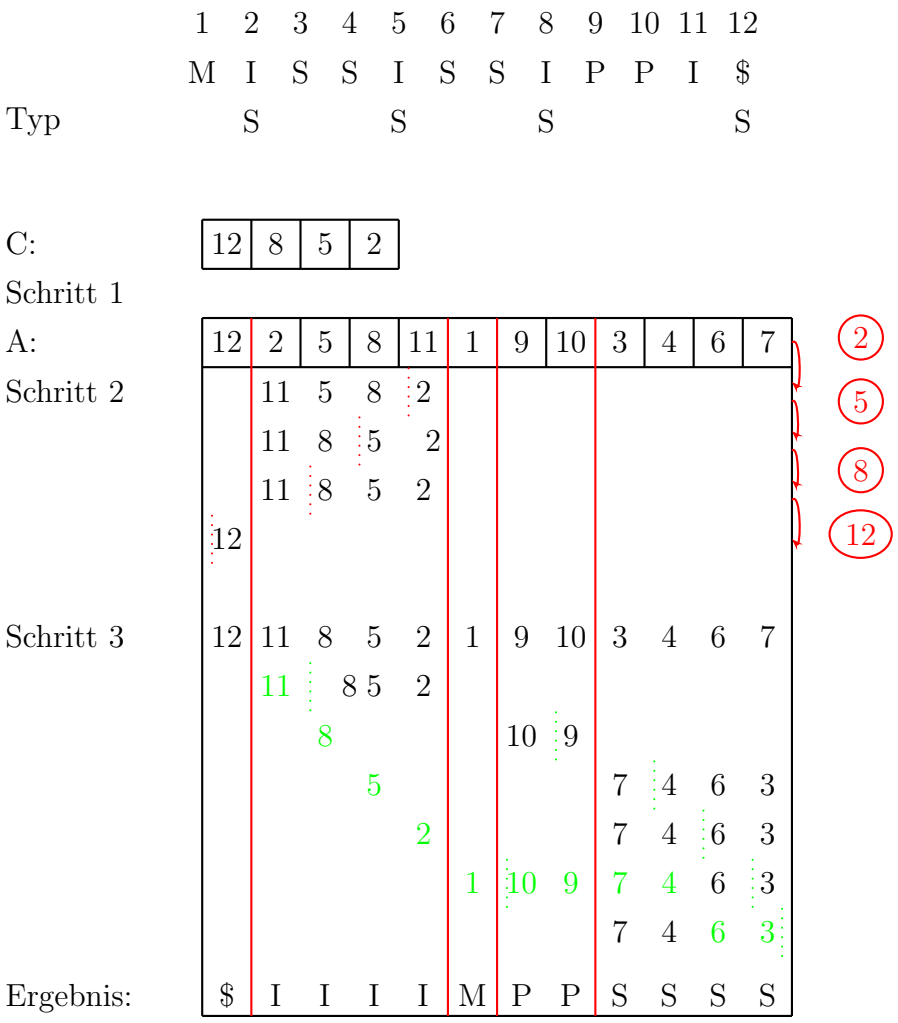


Abbildung 5.13: Beispiel: Der Algorithmus von Ko und Aluru für MISSISSIPPI und gegebenem Feld C

Wir müssen jetzt noch zeigen, dass dieser Algorithmus auch korrekt ist.

Lemma 5.9 *Nach Schritt 3, d.h. wenn die Position i im Feld A erreicht wird, ist der Suffix $t^{A[i]}$ an der korrekten (sortierten) Position in A.*

Beweis: Wir führen den Beweis durch Induktion nach i.

Induktionsanfang ($i = 0$): Das ist korrekt, da nach dem Bucket-Sort nach dem ersten Zeichen $A[0] = |t\$|$ gilt, d.h. $A[0]$ entspricht dem Suffix \$, und dieser ist nach Definition der kleinste.

Induktionsschritt ($i - 1 \to i$): Nach Induktionsvoraussetzung ist $A[0], \ldots, A[i-1]$ bereits korrekt sortiert. Wir führen den Beweis für die Position i jetzt durch einen Widerspruchsbeweis. Dazu nehmen wir an, dass ein $k > i$ mit $t^{A[i-1]} < t^{A[k]} < t^{A[i]}$ existiert. Dabei sei k so gewählt, dass es das i-te Element in der wirklich sortierten Folge ist.

Nach Schritt 2 sind alle Suffixe vom Typ S an der korrekten Position innerhalb des Buckets und damit auch innerhalb des Feldes. Dies folgt aus der Tatsache, dass beim Verschieben an das Ende des Buckets die relative Reihenfolge des sortierten Feldes C in jedem Bucket erhalten bleibt. Somit müssen also die Suffixe $t^{A[i]}$ und $t^{A[k]}$ beide vom Typ L sein.

Da im ersten Schritt das Feld A mittels eines Bucket-Sort nach dem ersten Zeichen sortiert wurde, muss $t_{A[k]} = t_{A[i]} = c$ für ein $c \in \Sigma$ gelten. Sei also $t^{A[i]} = c \cdot \alpha$ und $t^{A[k]} = c \cdot \beta$ mit $\alpha, \beta \in \Sigma^*\$$.

Da $t^{A[k]}$ vom Typ L ist, muss $t^{A[k]} > \beta$ sein. Aus $\beta < t^{A[k]}$ folgt, dass der Suffix β bereits korrekt sortiert worden sein muss, d.h. $\beta \in \{t^{A[1]}, \ldots, t^{A[i-1]}\}$.

Da nach Annahme $t^{A[k]} < t^{A[i]}$ gilt, muss weiter $\beta < \alpha$ gelten. Da $\beta < \alpha$ gilt, muss nach Algorithmenvorschrift $t^{A[k]}$ vor $t^{A[i]}$ an den aktuellen Beginn seines Buckets vertauscht worden sein. Dann würde aber $t^{A[i]}$ nicht vor $t^{A[k]}$ in seinem Bucket stehen. Das liefert den gewünschten Widerspruch. ■

Analog können wir das Feld A auch sortieren, wenn wir die sortierte Reihenfolge der Suffixe vom Typ L kennen, wie der folgenden Algorithmus zeigt:

1. Bucket-Sort von A nach dem ersten Zeichen der Suffixe.

2. Durchlaufe das Feld C von links nach rechts (vom kürzesten zum längsten Suffix) und verschiebe den aktuell betrachteten Suffix $c \in C$ an den aktuellen Beginn seines Buckets, in dem c mit dem Suffix unmittelbar nach dem aktuellen Beginn vertauscht wird.

3. Durchlaufe das Feld A von rechts nach links (also absteigend). Für jedes $A[i]$ mit $t^{A[i]-1}$ vom Typ S verschiebe $A[i]-1$ an das aktuelle Ende seines Buckets.

Der Korrektheitsbeweis ist analog zu dem vom Lemma 5.9. Damit erhalten wir insgesamt den folgenden Satz.

Theorem 5.10 *Sind alle Typ S oder alle Typ L Suffixe von t\$ bereits sortiert, dann können alle Suffixe von t\$ in Zeit und Platz $O(|t|)$ sortiert werden.*

5.2.3 Sortierung der Typ S Suffixe

Wir müssen jetzt also nur noch zeigen, wie man alle Suffixe vom Typ S (bzw. vom Typ L) in linearer Zeit sortieren kann.

Definition 5.11 *Sei $t = t_1 \cdots t_n \in \Sigma^*$. Eine Position $j \in [1 : n+1]$ ist vom* Typ S *(bzw. L), wenn t^j vom Typ S (bzw. L) ist. Ein Teilwort $t_i \cdots t_j \sqsubseteq t\$$ ist vom* Typ S *(bzw. L), wenn i und j Positionen vom Typ S (bzw. L) sind und jede Position $k \in (i : j)$ vom Typ L (bzw. S) ist.*

Betrachten wir als Beispiel das Wort MISSISSIPPI\$. Die roten Positionen sind vom Typ S. Teilwörter vom Typ S sind dann ISSI und IPPI\$. Dies ist in Abbildung 5.14 noch einmal illustriert.

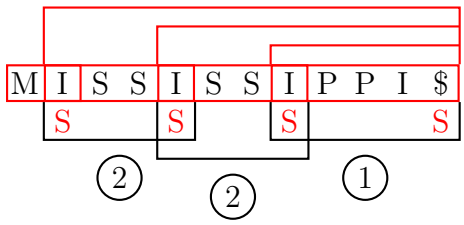


Abbildung 5.14: Beispiel: Typ S Teilwörter von MISSISSIPPI

Im Folgenden nehmen wir ohne Beschränkung der Allgemeinheit an, dass $|\mathcal{S}| \leq |\mathcal{L}|$ gilt. Wie können wir nun die Menge $\mathcal{S}$ sortieren? Wir sortieren dafür zunächst die Menge der Typ S Teilwörter. In unserem Beispiel ist IPPI\$ < ISSI. Um mit dieser sortierten Menge sinnvoll weiterarbeiten zu können, geben wir jedem Typ S Teilwort einen neuen Namen, nämlich ein Zeichen eines neuen Alphabets, so dass diese Zuordnung die Sortierung des Typ S Teilwörter respektiert.

Notation 5.12 *Sei $t \in \Sigma^*$ und sei $\mathcal{S}$ die Menge der Typ S Teilwörter von t\$, dann bezeichne $\pi(s)$ für $s \in \mathcal{S}$ die relative Ordnung in der sortierten Menge, d.h.*

$$\pi(s) := |\{s' \in \mathcal{S} \,:\, s' \leq s\}|.$$

Basierend auf dieser Notation können wir jetzt das Wort t zu einem neuen Wort $\tilde{t}$ wie folgt umbauen, das im Wesentlichen als Suffixe die Typ S Suffixe von t besitzt.

Notation 5.13 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und seien $j_1 < \cdots < j_k$ die Typ S Positionen in $t\$$. Dann ist*

$$\tilde{t} := \pi(t_{j_1} \cdots t_{j_2}) \cdot \pi(t_{j_2} \cdots t_{j_3}) \cdots \pi(t_{j_{k-1}} \cdots t_{j_k}) \in [1 : n+1]^*.$$

Für unser Beispielwort MISSISSIPPI\$ ergibt sich dann $\tilde{t} := 2 \cdot 2 \cdot 1$

Jeder Typ S Suffix in $t\$$ korrespondiert eindeutig zu einem Suffix von $\tilde{t}$. Somit kann das Sortieren von Typ S Suffixen auf das Sortieren von Suffixen in $\tilde{t}$ zurückgeführt werden. Damit auch hier wieder das Wort mit einem kleinsten Symbol endet, wird an dieses Wort noch die Zahl 0 angehängt. Damit können wir die Sortierung der Typ S Suffixe auf das Sortieren der Suffixe eines anderen Wortes zurückfuhren.

Wie ermittelt man aber $\pi(s)$ für alle $s \in \mathcal{S}$? Die entspricht offensichtlich dem Sortieren der Menge $\mathcal{S}$. Dazu benötigen wir zuerst noch die folgende Definition.

Definition 5.14 *Sei $t \in \Sigma^*$. Die* S-Distanz $\sigma(t^i)$ *für ein Suffix t^i von $t\$$ ist definiert durch:*

$$\sigma(t^i) := \min\left\{ j \in [1 : i-1] \;:\; t^{i-j} \text{ ist vom Typ } S \right\},$$

wobei $\min\{\} := 0$ *gilt.*

Anschaulich gibt die S-Distanz an, um wie viele Zeichen man nach links im Wort gehen muss, um die vorherige Typ S Position aufzufinden, sofern es eine solche gibt. Die S-Distanz ist in der folgenden Abbildung 5.15 illustriert.

	M	I	S	S	I	S	S	I	P	P	I	\$
		S			S			S			S	
σ :	0	0	1	2	3	1	2	3	1	2	3	4

Abbildung 5.15: Beispiel: S-Distanz für MISSISSIPPI

Wenn man sich die Typ S Positionen in einem Booleschen Feld merkt, lassen sich die S-Distanzen mit einem Durchlauf über das Feld A in linearer Zeit ohne zusätzlichen Platz berechnen. Die Details bleiben dem Leser zur Übung überlassen.

Jetzt können wir den Algorithmus zum Sortieren der Typ S Teilwörter angeben. Dabei ist A ein Feld mit $n+1$ Elementen, in dem die Suffixe von $t\$$ nach ihrem ersten Zeichen sortiert sind.

1. Sei $m := \max\{\sigma(t^i) : i \in [1:n+1]\}$. Erzeuge m Listen $L_j = \{i : \sigma(t^i) = j\}$ für $j \in [1:m]$, wobei die Reihenfolge innerhalb der Listen der Reihenfolge innerhalb des nach dem ersten Zeichen sortierten Feldes A entspricht.

   Diese lassen sich mit einem einfachen Scan über das Feld für das Wort t erstellen und mit Hilf des Feldes R den entsprechenden Positionen in A zuordnen. Also kann dieser Schritt in linearer Zeit bewerkstelligt werden.

2. Sortiere alle Typ S Teilwörter wie folgt: Führe quasi wiederholte Bucket-Sorts nach dem j-ten Zeichen der Teilwörter aus: Für jedes $j \in [1:m]$ und jedes $i \in L_j$ bewege des Typ S Teilwort t^{i-j} (respektive die zugehörige Anfangsposition) an den aktuellen Anfang seines Buckets.

   Nach jeder Abarbeitung einer Liste L_j müssen die neuen Bucketgrenzen angepasst werden, da jetzt das Feld nach den ersten j Zeichen der Typ S Teilwörter sortiert ist. Da die Summe aller Listenlängen durch $O(n)$ beschränkt ist, kann auch dieser Schritt in Linearzeit ausgeführt werden.

Warum werden mit diesem Algorithmus die Typ S Teilwörter korrekt sortiert? In der Liste L_j befinden sich die Typ S Suffixe in der Reihenfolge, wenn sie nach dem j-ten Zeichen sortiert sind. Mit der Umordnung der Startposition $i-j$ für $i \in L_j$ wird das jeweils kleinste Typ S Teilwort innerhalb seines Buckets nach vorne bewegt, also wird innerhalb der Buckets nach dem j-ten Zeichen sortiert. In Abbildung 5.16 ist dieser Sortierschritt noch einmal illustriert.

In diesem Schritt beachten wir allerdings folgende Modifikation des Bucket-Sorts bzw. der lexikographischen Ordnung: Ist α ein Präfix von β, dann definieren wir $\alpha > \beta$ (im Gegensatz zur üblichen lexikographischen Ordnung). Im Bucket-Sort kann diese Modifikation der Ordnung leicht implementiert werden. Warum wir diese Modifikation benötigen, wird im folgenden Lemma klar. Wir weisen allerdings darauf hin, dass diese Modifikation beim Sortieren von Typ L Teilwörtern nicht erforderlich ist.

Lemma 5.15 *Sei $t \in \Sigma^*$. Seien t^i und t^j zwei Typ S Suffixe von $t\$$ und seien $\tilde{t}^{i'}$ und $\tilde{t}^{j'}$ zwei Suffixe von $\tilde{t} \in \mathbb{N}_0^+$, die jeweils zueinander korrespondieren. Dann gilt $t^i \leq t^j \Leftrightarrow \tilde{t}^{i'} \leq \tilde{t}^{j'}$.*

Beweis: $\Rightarrow$: Es gilt nach Voraussetzung $t^i \leq t^j$. Sei also $t^i = \alpha a \beta$ und $t^j = \alpha b \gamma$ mit $\alpha, \beta, \gamma \in \Sigma^*$ und $a < b \in \Sigma$. Diese Situation ist mit $\alpha = \alpha' \alpha''$ in Abbildung 5.17

i :	1	2	3	4	5	6	7	8	9	10	11	12
t_i :	M	I	S	S	I	S	S	I	P	P	I	\$
Type		S			S			S				S
$\sigma(t^i)$:	0	0	1	2	3	1	2	3	1	2	3	4

i :	1	2	3	4	5	6	7	8	9	10	11	12
$A[i]$:	12	2	5	8	11	1	9	10	3	4	6	7
$t_{A[i]}$:	\$	I	I	I	I	M	P	P	S	S	S	S
$\sigma(t^{A[i]})$:	4	0	3	3	3	0	1	2	1	2	1	2

$$
\begin{aligned}
L_1 &= [9, 3, 6] \\
L_2 &= [10, 4, 7] \\
L_3 &= [5, 8, 11] \\
L_4 &= [12]
\end{aligned}
$$

12:\$	2: ISSI	5: ISSI	8: IPPI\$
12:\$	8: IPPI\$	5: ISSI	2: ISSI
12:\$	8: IPPI\$	2: ISSI	5: ISSI

Abbildung 5.16: Beispiel: Sortieren des Typ S Teilwörter von MISSISSIPPI

illustriert. Beachte hierbei, dass in der Illustration die Bedingung $i < j$ vollkommen willkürlich gewählt wurde (es könnte auch $i > j$ gelten) und dass sich die beiden Bereiche α auch überlappen können.

Wir zerlegen jetzt die Zeichenreihe α in $\alpha = \alpha'\alpha''$, so dass in α' die Typ S Teilwörter in den Suffixen t^i und t^j gleich sind. Sollte α'' nicht das leere Wort sein, so kann dies nur passieren, wenn $\alpha'' = c^{|\alpha''|}$ für ein $c \in \Sigma$ gilt. Der einfache Beweis hierfür sei dem Leser überlassen. In der Abbildung 5.17 sind die Typ S Teilwörter im Bereich α durch die roten Balken schematisch dargestellt.

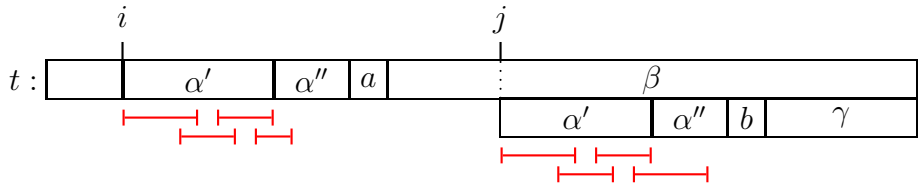


Abbildung 5.17: Skizze: $t^i < t^j$

Ist $\alpha'' = \varepsilon$, dann sind die Typ S Teilwörter in t^i und t^j im Bereich α gleich. Das erste sich unterscheidende Teilwort vom Typ S muss das Zeichen a in t^i bzw. das Zeichen b in t^j enthalten. Nach unsere Sortierung gilt dann offensichtlich auch $\tilde{t}^{i'} < \tilde{t}^{j'}$.

Sei jetzt also $\alpha'' \neq \varepsilon$. Der Zerfall der Positionen in α'' von t^i bzw. t^j in Typ S bzw. Typ L Positionen hängt dann von dem folgenden Zeichen ungleich c in t^i bzw. in t^j und dessen Beziehung zu c ab. Auf jeden Fall sind alle Positionen in α'' vom jeweils gleichen Typ. Sind sowohl in t^i als auch in t^j alle Positionen in α'' vom selben Typ, dann beinhaltet auch α'' sowohl in t^i als auch in t^j dieselbe Zerlegung in Typ S Teilwörter und wir erhalten einen Widerspruch zur Wahl von α''.

Da $t^i < t^j$ gilt, müssen alle Positionen in α'' in t^i vom Typ L und in t^j vom Typ S sein. Den ebenfalls nicht allzu schweren Beweis überlassen wir dem Leser. Dann sind in $\tilde{t}$ die Zeichen ab Position i' bzw. j' gleich bis es zur Übersetzung der Typ S Teilwörter in α'' kommt. In $\tilde{t}^{i'}$ folgt als nächstes Zeichen die Bucketnummer von $c^k d$ für ein $1 < k \in \mathbb{N}$ und ein $c \neq d \in \Sigma$. In $\tilde{t}^{j'}$ folgt als nächstes Zeichen die Bucketnummer von cc. In diesem Fall ist also cc ein Präfix von $c^k d$ und nach unserer modifizierten Sortierreihenfolge gilt dann $\tilde{t}^{i'} < \tilde{t}^{j'}$.

$\Leftarrow$: Sei jetzt $\tilde{t}^{i'} < \tilde{t}^{j'}$ mit $\tilde{t}^{i'} = \mu \cdot m \cdot \rho$ und $\tilde{t}^{j'} = \mu \cdot m' \cdot \rho'$ mit $\mu, m, m', \rho, \rho' \in \mathbb{N}$ und $m < m'$.

Sei α das Teilwort von $t\$$ das zu μ korrespondiert. Seien weiter β und β' Teilwörter von t, die zu m und m' korrespondieren. Diese müssen also Typ S Teilwörter von $t\$$ sein. Dies ist in Abbildung 5.18 illustriert

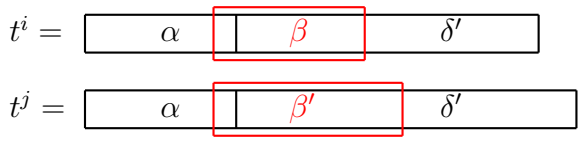


Abbildung 5.18: Skizze: Korrespondierende t^i und t^j zu $\tilde{t}^{i'} < \tilde{t}^{j'}$

Zunächst halten wir fest, dass β kein echtes Präfix von β' sein kann. Dies folgt aus Sortierung der Typ S Teilwörter. Dort gilt: ist β ein echtes Präfix von β', dann ist $\pi(\beta) > \pi(\beta')$. Wir unterscheiden jetzt zwei Fälle, je nachdem, ob β' ein Präfix von β ist oder nicht.

Fall 1: Es gilt, dass β' kein Präfix von β ist. Dann gilt aufgrund von $m < m'$, dass $\beta < \beta'$ und somit $t^i < t^j$.

Fall 2: Jetzt gilt, dass β' ein echtes Präfix von β ist. Siehe hierzu auch Abbildung 5.19.

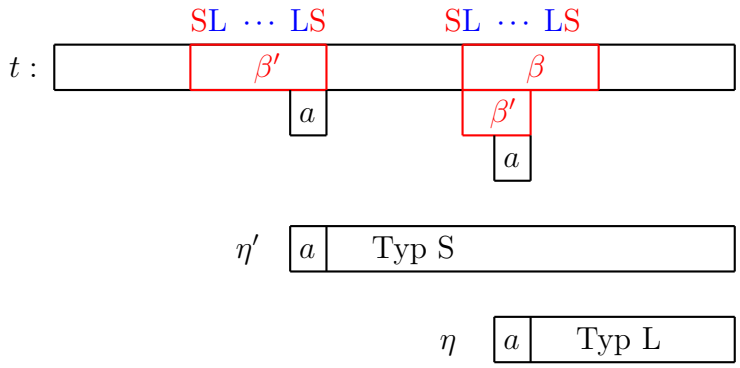


Abbildung 5.19: Skizze: β' ist ein echtes Präfix von β

Sei, wie in Abbildung 5.19 angegeben, a das letzte Zeichen von β'. Im Bereich β' muss sich a an einer Typ S Position befinden. Das weitere Vorkommen als letztes Zeichen von β' als Präfix von β muss eine Typ L Position sein. Betrachten wir die beiden Suffixe η und η', die an an diesen Positionen beginnen. Da beide mit demselben Zeichen beginnen, muss nach Lemma 5.7 $\eta < \eta'$ gelten. Damit folgt sofort nach der lexikographischen Ordnung, dass dann auch $t^i < t^j$ gilt. ■

5.2.4 Der Algorithmus von Ko und Aluru und Laufzeitanalyse

Wir wollen jetzt noch den vollständigen Algorithmus von Ko und Aluru angeben:

1. Bestimmung der Typ S-/L Positionen von t. Es gelte im Folgenden $|\mathcal{S}| \leq |\mathcal{L}|$.
2. Sortierung der Typ S Teilwörter (Bestimmung des Feldes C):
   - Bucket-Sort nach dem ersten Zeichen (von allen Typ S Suffixen).
   - Bestimmung der S-Distanzen und Generierung der Listen $L_1, \ldots, L_m$.
   - Verfeinerung der Buckets mit Hilfe der Listen $L_1, \ldots, L_m$.
3. Konstruktion von $\tilde{t} \in [1 : n+1]^+$ aus $t \in \Sigma^*$.
   - Ersetzung der Typ S Teilwörter in t durch ihren Rang (mit Hilfe von Schritt 2)
4. Rekursives Sortieren der Suffixe von $\tilde{t}$. Es gilt: $|\tilde{t}| \leq \frac{n+2}{2}$.
5. Aus 4. folgt Sortierung der Typ S Suffixe von t.

6. Sortierung der Suffixe von t mit Hilfe der relativen Ordnung der Suffixe vom Typ S.

   - Bucket-Sort nach dem ersten Zeichen der Suffixe.
   - Verschiebe die Typ S Suffixe innerhalb seines Bucket ans Ende mit Hilfe der bekannten Sortierung der Typ S Suffixe.
   - Verschiebe die Typ L Suffixe innerhalb seines Buckets an den Anfang mit Hilfe des bereits teilweise sortierten Feldes A.

Wir kommen jetzt zur Laufzeitanalyse des Algorithmus von Ko und Aluru. Wie bisher bezeichne:

$$\begin{aligned} \mathcal{S} &:= \left\{t^j \,:\, t^j \text{ ist vom Typ S}\right\} \\ \mathcal{L} &:= \left\{t^j \,:\, t^j \text{ ist vom Typ L}\right\} \end{aligned}$$

Gilt nun $|\mathcal{S}| \leq |\mathcal{L}|$ dann sortieren wir alle Suffixe vom Typ S rekursiv (bzw. die neu konstruierten Zeichenreihe $\tilde{t}$), also die Menge $\mathcal{S}$, und anschließend das Feld A mit Hilfe der sortierten Suffixe vom Typ S. Ansonsten sortieren wir alle Suffixe vom Typ L rekursiv (bzw. die neu konstruierten Zeichenreihe $\tilde{t}$), also die Menge $\mathcal{L}$, und anschließend das Feld A mit Hilfe der sortierten Suffixe vom Typ L. Für die Laufzeit gilt dann:

$$T(n) \leq T\left(\frac{n+2}{2}\right) + O(n).$$

Bekanntermaßen hat diese Rekursionsgleichung die Lösung $T(n) = O(n)$.

Theorem 5.16 *Ein Suffix-Array für $t \in \Sigma^n$ kann mit Hilfe des Algorithmus von Ko und Aluru in Zeit und Platz $O(n)$ konstruiert werden.*

Zum Abschluss noch einige Anmerkungen:

- Bei geschickter Implementierung benötigt man nur zwei zusätzliche Felder der Länge n und einige Felder von Booleschen Werten. Für Details sei der Leser auf die Originalliteratur von Ko und Aluru verwiesen.
- Bei der Rekursion findet ein Alphabetwechsel von $\Sigma \cup \{\$\}$ auf $[0 : n+1]$ statt (0 für das neue Endzeichen). Dies stellt jedoch keine Einschränkung dar, da man für ein Wort der Länge $n+1$ über einem geordneten Alphabet Σ ohne Weiteres das Alphabet Σ selbst als Teilmenge von $[1 : n+1]$ und \$ als 0 interpretieren kann.

5.3 Skew-Algorithmus von Kärkkäinen und Sanders

In diesem Abschnitt wollen wir noch einen einfacheren Algorithmus zur Konstruktion von Suffix-Arrays vorstellen, der jedoch in der Praxis nicht ganz so platzsparend wie der Algorithmus von Ko und Aluru ist.

5.3.1 Tripel und Verkürzung der Zeichenreihe

Sei wiederum $t \in \Sigma^*$ das Wort, für das das Suffix-Array erstellt werden soll. Hier nehmen wir für eine einfachere Darstellung an, dass $t\$$ von 0 mit n indiziert ist, also es gilt $t = t_0 \cdots t_{n-1}$.

Zuerst konstruieren wir alle dreibuchstabigen Teilwörter, so genannte *Tripel* von $t\$$, d.h. $t_i \cdot t_{i+1} \cdot t_{i+2}$ für $i \in [0 : n+1]$. Damit alle Tripel eine Länge von genau 3 Zeichen haben, nehmen wir der Einfachheit halber an, dass das Wort mit 4 Dollarzeichen endet. Für das weitere Vorgehen benötigen wir nur die Tripel ab einer Position $i \in [0 : n+1]$ mit $i \bmod 3 \neq 0$. Wir sortieren all diese Tripel mithilfe eines Bucket-Sorts und nummerieren sie nach ihrer Ordnung. Ein Beispiel ist in Abbildung 5.20 angegeben.

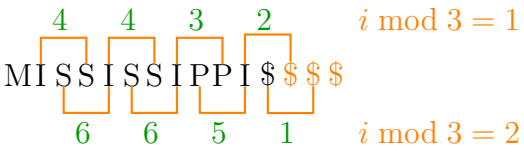


Abbildung 5.20: Beispiel: sortierte Tripel von MISSISSIPPI

Im zweiten Schritt erzeugen wir eine neue Zeichenreihe $\tilde{t}$ wie folgt. Für das Teilwort $t_1 \cdots t_{3\lceil n/3 \rceil}$ ersetzen wir die Tripel $t_i \cdot t_{i+1} \cdot t_{i+2}$ mit $i \bmod 3 = 1$ durch ihre Ordnungsnummer und erhalten somit die Zeichenreihe $t^{(1)}$. Dieselbe Prozedur führen wir für das Teilwort $t_2 \cdots t_{3\lceil n/3 \rceil + 1}$ durch. Auch hier ersetzen wir die Tripel $t_i \cdot t_{i+1} \cdot t_{i+2}$ mit $i \bmod 3 = 2$ durch ihre Ordnungsnummer und erhalten somit die Zeichenreihe $t^{(2)}$. Nun setzen wir $\tilde{t} := t^{(1)} \cdot t^{(2)}$.

Im Beispiel MISSISSIPPI erhalten wir dann $\tilde{t} = 4 \cdot 4 \cdot 3 \cdot 2 \cdot 6 \cdot 6 \cdot 5 \cdot 1$, wobei die Zeichenreihe $t^{(1)}$ bzw. $t^{(2)}$ rot bzw. blau dargestellt ist. Für die Länge von $\tilde{t}$ gilt offensichtlich $|\tilde{t}| = 2\lceil \frac{n}{3} \rceil$.

5.3.2 Rekursives Sortieren der verkürzten Zeichenreihe

Da wir nun eine neue, kürzere Zeichenreihe erhalten haben, können wir unseren Sortieralgorithmus rekursiv mit dieser Zeichenreihe aufrufen. Dies liefert uns das

$A'[0]$	=	8	$\widehat{=}$	0	$\widehat{=}$ ε
$A'[1]$	=	7	$\widehat{=}$	10	$\widehat{=}$ \$\$\$
$A'[2]$	=	3	$\widehat{=}$	266510	$\widehat{=}$ I\$\$ $\cdot\, t_2 \cdots t_{3\lceil n/3 \rceil}$
$A'[3]$	=	2	$\widehat{=}$	3266510	$\widehat{=}$ IPP I\$\$ $\cdot\, t_2 \cdots t_{3\lceil n/3 \rceil}$
$A'[4]$	=	1	$\widehat{=}$	43266510	$\widehat{=}$ ISS IPP I\$\$ $\cdot\, t_2 \cdots t_{3\lceil n/3 \rceil}$
$A'[5]$	=	0	$\widehat{=}$	443266510	$\widehat{=}$ ISS ISS IPP I\$\$ $\cdot\, t_2 \cdots t_{3\lceil n/3 \rceil}$
$A'[6]$	=	6	$\widehat{=}$	510	$\widehat{=}$ PPI \$\$\$
$A'[7]$	=	5	$\widehat{=}$	6510	$\widehat{=}$ SSI PPI \$\$\$
$A'[8]$	=	4	$\widehat{=}$	66510	$\widehat{=}$ SSI SSI PPI \$\$\$

Abbildung 5.21: Beispiel: Das Feld A' für MISSISSIPPI

Feld A' mit den sortierten Suffixen (respektive ihrer Startpositionen) für $\tilde{t}$. Für unser Beispiel MISSISSIPPI ist das Ergebnis in Abbildung 5.21 dargestellt, wobei neben dem eigentlichen Ergebnis, auch die Suffixe in $\tilde{t} \cdot 0$ sowie die korrespondierenden Suffixe von t angegeben sind.

Wie man leicht sieht, liefert dieser Schritt bereits eine Sortierung aller Suffixe t^i mit $i \bmod 3 \neq 0$. Die Suffixe, die an Position $i \bmod 3 = 2$ beginnen, werden dabei ja so berücksichtigt, wie wir es wollen. Die Suffixe, die an einer Position $i \bmod 3 = 1$ beginnen, werden noch durch einige Zeichen verlängert. Die Verlängerung wird jedoch durch eine Zeichenreihe aus einigen Dollarzeichen getrennt. Da das Dollarzeichen jedoch das kleinste Zeichen des Alphabets hat, spielt es quasi die Rolle eines Leerzeichens und die Präfixe werden korrekt sortiert. Man muss dabei nur beachten, dass sich nach Voraussetzung die Suffixe spätestens beim ersten Auftreten eines Dollarzeichens unterscheiden müssen.

Wie erhalten wir nun eine Indizierung der Elemente von t anstelle von $\tilde{t}$ (also der Tripletts in t)? Zuerst beachten wir, dass $A'[0] = 2\lceil \frac{n}{3} \rceil$ und somit das Sonderzeichen 0 in $\tilde{t}$ beschreibt, was der leeren Zeichenreihe in t entspricht. Dieser Array-Eintrag kann also ignoriert werden bzw. muss aus dem Array entfernt werden. Wir müssen nur berücksichtigen, dass die Array-Werte aus $[0 : \lceil \frac{n}{3} \rceil - 1]$ bzw. $[\lceil \frac{n}{3} \rceil : 2\lceil \frac{n}{3} \rceil - 1]$ in $\tilde{t}$ die Tripletts aus $t^{(1)}$ bzw. $t^{(2)}$ beschreiben. Also rechnen wir die Information wie folgt um:

$$A_{12}[i-1] = \begin{cases} 1 + 3 \cdot (A'[i]) & \text{falls } A'[i] < \lceil \frac{n}{3} \rceil, \\ 2 + 3 \cdot (A'[i] - \lceil \frac{n}{3} \rceil) & \text{sonst.} \end{cases}$$

Hierbei ist $i \in [1 : 2\lceil \frac{n}{3} \rceil]$. Somit entspricht A_{12} dem Array, dass die Startpositionen der sortierten Suffixe aus $\{t^i \,:\, i \bmod 3 \neq 0\}$ enthält.

Für unser Beispiel MISSISSIPPI ist das Ergebnis dies in Abbildung 5.22 dargestellt ($n = 11$).

$$
\begin{array}{lcllcl}
A_{12}[0] & = & 2+3\cdot(\mathbf{7}-4) & = & 11 & \widehat{=}\ \$ \\
A_{12}[1] & = & 1+3\cdot(\mathbf{3}) & = & 10 & \widehat{=}\ \text{I\$} \\
A_{12}[2] & = & 1+3\cdot(\mathbf{2}) & = & 7 & \widehat{=}\ \text{IPPI\$} \\
A_{12}[3] & = & 1+3\cdot(\mathbf{1}) & = & 4 & \widehat{=}\ \text{ISSIPPI\$} \\
A_{12}[4] & = & 1+3\cdot(\mathbf{0}) & = & 1 & \widehat{=}\ \text{ISSISSIPPI\$} \\
A_{12}[5] & = & 2+3\cdot(\mathbf{6}-4) & = & 8 & \widehat{=}\ \text{PPI\$} \\
A_{12}[6] & = & 2+3\cdot(\mathbf{5}-4) & = & 5 & \widehat{=}\ \text{SSIPPI\$} \\
A_{12}[7] & = & 2+3\cdot(\mathbf{4}-4) & = & 2 & \widehat{=}\ \text{SSISSIPPI\$}
\end{array}
$$

Abbildung 5.22: Beispiel: Das Feld A_{12} für MISSISSIPPI

5.3.3 Sortieren der restlichen Suffixe

Im nächsten Schritt sortieren wir alle Suffixe, die an einer durch 3 teilbaren Position beginnen, d.h. wir betrachten nur Suffixe t^i mit $i \bmod 3 = 0$. Es gilt offensichtlich: $t^i = t_i \cdot t^{i+1}$ mit $(i+1) \bmod 3 = 1$. Da uns ja die relative Ordnung der Suffixe t^{i+1} mit $(i+1) \bmod 3 = 1$ durch das Feld A_{12} bereits bekannt ist, kann uns das Feld A_{12} für das Sortieren helfen.

Wir führen zuerst einen einfachen Bucket-Sort in der Menge $\{t^i : i \bmod 3 = 0\}$ nach dem erstem Zeichen durch. Mittels eines Scans über A_{12} von links nach rechts, verschieben wir beim Betrachten von $A_{12}[i]$ den Suffix $t^{A_{12}[i]-1}$ an den aktuellen Beginn seines Buckets, sofern $A_{12}[i] - 1 \bmod 3 = 0$ bzw. $A_{12}[i] \bmod 3 = 1$ gilt. Dies liefert uns das sortierte Feld A_0 für $\{t^i : i \bmod 3 = 0\}$. Dies ist in Abbildung 5.23 noch einmal für unser Beispiel illustriert.

A_0	0	1	2	3
	0	9	3	6
11				
10		9		
7			6	
4			6	3
1	0			
8				
5				
2				
	0	9	6	3
	M	P	S	S
	I	I	I	I
	S	\$	P	S

Abbildung 5.23: Beispiel: Sortierung von A_0 mithilfe von A_{12}

5.3.4 Mischen von A_{12} und A_0

Jetzt müssen wir nur noch A_{12} und A_0 zusammenmischen. Damit die Vergleiche beim Mischen nicht zu aufwendig werden, beachten wir noch Folgendes. Wenn wir t^i mit t^j mit $i \bmod 3 = 0$ und $j \bmod 3 \in [1:2]$ vergleichen, gilt:

Fall 1: Sei $j \bmod 3 = 1$. Dann gilt:

$$\begin{aligned} t^i &= t_i \cdot t^{i+1}, \text{ wobei } (i+1) \bmod 3 = 1, \\ t^j &= t_j \cdot t^{j+1}, \text{ wobei } (j+1) \bmod 3 = 2. \end{aligned}$$

Wir vergleichen also nur t_i mit t_j. Bei Gleichheit greifen wir auf das Resultat in A_{12} zurück. Andernfalls kennen wir das Ergebnis des Vergleichs ja schon.

Fall 2: Sei $j \bmod 3 = 2$. Dann gilt:

$$\begin{aligned} t^i &= t_i \cdot t_{i+1} \cdot t^{i+2}, \text{ wobei } (i+2) \bmod 3 = 2, \\ t^j &= t_j \cdot t_{j+1} \cdot t^{j+2}, \text{ wobei } (j+2) \bmod 3 = 1. \end{aligned}$$

Wir müssen also nur zuerst t_i mit t_j vergleichen, bei Gleichheit dann auch noch t_{i+1} mit t_{j+1}. Ergeben beide Vergleiche eine Gleichheit, dann können wir das Ergebnis mittels eines Vergleichs von t^{i+2} mit t^{j+2} ebenfalls wieder aus A_{12} ablesen.

10.12.25

Für den Vergleich der beiden Werte aus A_{12} benötigen wir in Wirklichkeit wieder das inverse Feld R_{12} mit $A_{12}[R_{12}[i]] = i$. Der vollständige Algorithmus von Kärkkäinen und Sanders ist in Abbildung 5.24 angegeben. Beachte, dass in Schritt 1 die verwendeten Mengen als einfache Mengen (und nicht als Multimengen) zu verstehen sind.

1. Sortiere $\mathcal{T} = \{t_i \cdot t_{i+1} \cdot t_{i+2} : i \bmod 3 \neq 0\}$ mithilfe eines Bucket-Sorts und sei $\pi(i) := |\{t_j \cdot t_{j+1} \cdot t_{j+2} \leq t_i \cdot t_{i+1} \cdot t_{i+2} : j \bmod 3 \neq 0\}|$ für $i \bmod 3 \neq 0$.
2. Setze $t^{(1)} := \pi(1) \cdot \pi(4) \cdots \pi(3\lceil n/3 \rceil - 2)$ und $t^{(2)} := \pi(2) \cdot \pi(5) \cdots \pi(3\lceil n/3 \rceil - 1)$. Setze $\tilde{t} := t^{(1)} \cdot t^{(2)}$.
3. Sortiere die Suffixe von $\tilde{t} \cdot 0$ rekursiv. Sei A' das zugehörige Suffix-Array.
4. Berechne A_{12} aus A' (Sortierung aller Suffixe t^i mit $i \bmod 3 \neq 0$).
5. Sortiere $\{t^i : i \bmod 3 = 0\}$ nach dem ersten Zeichen mithilfe eines Bucketsorts und sortiere dann die Buckets mithilfe des Feldes A_{12} in das Feld A_0.
6. Mische A_{12} mit A_0 mithilfe von A_{12} in das Feld A.

Abbildung 5.24: Algorithmus: Konstruktion vom Suffix-Arrays nach Kärkkäinen und Sanders

5.3.5 Laufzeitanalyse

Zum Abschluss untersuchen wir noch die Laufzeit des Algorithmus von Kärkkäinen und Sanders. Der Algorithmus gehorcht offensichtlich der folgenden Rekursionsgleichung:

$$T(n) = T\left(2\left\lceil\frac{n}{3}\right\rceil\right) + O(n)$$

Bekanntermaßen hat diese Rekursionsgleichung die Lösung $T(n) = O(n)$.

Theorem 5.17 *Sei $t = t_0 \cdots t_{n-1} \in \Sigma^*$. Der Algorithmus von Kärkkäinen und Sanders kann in Zeit und Platz von $O(n)$ das zugehörige Suffix-Array konstruieren.*

5.3.6 Aktuelle Verfahren

Seit 2003 sind viele verschiedene Verfahren zur Linearzeit-Konstruktion von Suffix-Arrays entwickelt worden, fast alle (bis auf einen neueren Ansatz) basieren dabei ebenfalls auf dem Divide-and-Conquer-Ansatz. Viele der Algorithmen sind Eingabesensitiv, d.h. ihre Laufzeit bzw. ihr Speicherplatzverbrauch hängt von der Eingabe ab. Dabei sind einige Verfahren besonders schnell oder platzsparend, wenn es in der Eingabesequenz viele bzw. lange repetitive Bereiche gab, andere sind gerade auf solchen Eingaben besonders langsam oder verbrauchen viel Speicherplatz. Aktuell ist das Verfahren *SAIS* von Nong, Zhang und Chan über viele Eingaben gesehen das schnellste und speicherplatzsparendste. Dieser ähnelt im Divide-Schritt dem Algorithmus von Ko und Aluru. Y. Mori hat hierzu eine effiziente Implementierung in verschiedenen Programmiersprachen zur allgemeinen Benutzung zur Verfügung gestellt. Für Details verweisen wir den Leser auf die Literatur.

5.4 Suchen in Suffix-Arrays

Eine einfache Anwendung von Suffix-Bäumen ist das schnelle Auffinden von Wörtern in einem vorverarbeiteten Text. In diesem Abschnitt wollen wir zeigen, dass man auch in Suffix-Arrays relativ effizient nach Teilwörtern suchen kann. In diesem Abschnitt nehmen wir an, dass das Suffix-Array zu $t = t_1 \cdots t_n \in \Sigma^*$ gehört und wir nach $s = s_1 \cdots s_m \in \Sigma^*$ suchen wollen.

5.4.1 Binäre Suche

Ein erster simpler Ansatz stellt die Implementierung mithilfe einer binären Suche dar. Wie sieht die Laufzeit dafür aus? Wie bei der binären Suche in einem Feld der

Länge $n+1$ üblich, müssen wir insgesamt $O(\log(n))$ Vergleiche auf Zeichenreihen vornehmen. Man beachte jedoch, dass die Zeichenreihen in einem Suffix-Array im Mittel sehr lang sein können. Im schlimmsten Fall müssen wir jeweils bis zu $O(m)$ Zeichen Vergleichen, um einen Vergleich auf Zeichenreihen beantworten zu können (wenn auch im Mittel deutlich weniger).

Theorem 5.18 *Sei $t = t_1 \cdots t_n\$ \in \Sigma^n\$$ und $s = s_1 \cdots s_m \in \Sigma^*$. Eine Suche nach s in t mithilfe der binären Suche im Suffix-Array von $t\$$ lässt sich in Zeit $O(m \log(n))$ erledigen.*

Das ist im Vergleich zum Suffix-Baum mit einer Laufzeit von $O(|\Sigma| \cdot m)$ asymptotisch deutlich langsamer. Man beachte aber, das $\log(n)$ für viele Zwecke kleiner gleich 40 ist.

5.4.2 Verbesserte binäre Suche

Man kann die Laufzeit bei der binären Suche mit dem folgenden Hilfsmittel jedoch noch beschleunigen.

Definition 5.19 *Sei $t = t_1 \cdots t_n \in \Sigma^n$ und A das zugehörige Suffix-Array. Dann ist der* längste gemeinsame Präfix *(engl.* longest common prefix*) von den Positionen $i < j \in [1:n]$ der Wert $\mathrm{lcp}(i,j) = k$, wenn $t_{A[i]} \cdots t_{A[i]+k-1} = t_{A[j]} \cdots t_{A[j]+k-1}$ und wenn $t_{A[i]+k} < t_{A[j]+k}$.*

Im Prinzip kennen wir die Notation der längsten gemeinsamen Präfixe schon, da hier gilt: $\mathrm{lcp}(i,j) = \mathrm{lce}_f(A[i], A[j])$. Wir nehmen im Folgenden an, dass wir die längsten gemeinsamen Präfixe für das Positionspaar $i < j \in [1:n]$ in einem Feld $\mathrm{LCP}[i,j]$ gespeichert hätten.

Bezeichne $[L:R]$ das aktuelle Intervall der binären Suche im Suffix-Array A. Für dieses Intervall und die zu suchende Zeichenreihe s gilt dann $t^{A[L]} < s < t^{A[R]}$. Mit ℓ und r bezeichnen wir die Anzahl von Zeichenvergleichen, die wir mit $t^{A[L]}$ bzw. $t^{A[R]}$ und s erfolgreich ausgeführt haben, d.h. es gilt

$$\begin{aligned} s_1 \cdots s_\ell &= t_{A[L]} \cdots t_{A[L]+\ell-1} \quad \wedge \quad s_{\ell+1} \neq t_{A[L]+\ell}, \\ s_1 \cdots s_r &= t_{A[R]} \cdots t_{A[R]+r-1} \quad \wedge \quad s_{r+1} \neq t_{A[R]+r}. \end{aligned}$$

Zu Beginn testen wir, ob $s > t^{A[n]}$ ist (wobei wir annehmen, dass genau k Zeichenvergleiche ausgeführt werden). Falls $s > t^{A[n]}$, ist s kein Teilwort von t. Falls s ein Präfix von $t^{A[n]}$ ist, haben wir s gefunden und sind fertig. Andernfalls setzen wir

$L = 0$ und $\ell = 0$ sowie $R = n$ und $r = k-1$. Beachte, dass nach Definition für jedes Wort $s \in \Sigma^+$ die Beziehung $s > \$$ gilt.

Wir wählen dann gemäß der Strategie der binären Suche $M := \lceil \frac{L+R}{2} \rceil$ und versuchen effizient zu entscheiden, ob sich s im Bereich $[L : M]$ oder $[M : R]$ befindet. Wir überlegen uns jetzt, wie wir bei einer binären Suche geschickter vorgehen können. Dazu unterscheiden wir einige Fälle.

Fall 1: Es gilt $\ell = r$. Wir vergleichen s mit $t^{A[M]}$ ab Position $\ell + 1 = r + 1$ bis wir einen erfolglosen Zeichenvergleich an Position k durchgeführt haben (oder s gefunden haben, dann sind wir ja fertig).

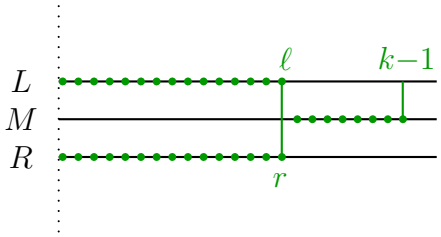


Abbildung 5.25: Skizze: Binäre Suche in $[L : R]$ bei $\ell = r$

Anschließend aktualisieren wir entweder L und ℓ oder R und r. War

$$s_{\ell+1} \cdots s_k > t_{A[M]+\ell} \cdots t_{A[M]+k-1},$$

dann setzen wir $L = M$ und $\ell = k - 1$ und sonst $R = M$ und $r = k - 1$. Dies ist in Abbildung 5.25 noch einmal illustriert.

Fall 2: Es gelte jetzt $\ell > r$. Wir unterscheiden jetzt noch drei Unterfälle.

Fall 2a: Wir nehmen an, dass zusätzlich $\mathrm{lcp}(L, M) > \ell$ gilt. Nach dem Prinzip der binären Suche gilt $s > t^{A[L]}$ und somit aufgrund des längsten gemeinsamen Präfixes auch $s > t^{A[M]}$. Wir setzen dann $L := M$ und lassen ℓ, R sowie r unverändert. Dies ist auch in Abbildung 5.26 illustriert. Der schraffierte Bereich zeigt dabei die Übereinstimmung von $t^{A[L]}$ mit $t^{A[M]}$

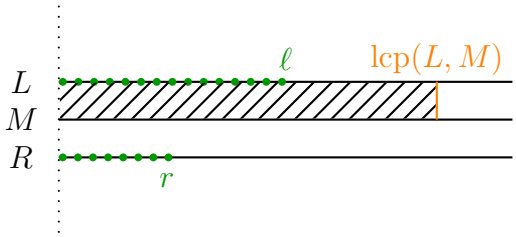


Abbildung 5.26: Skizze: Binäre Suche in $[L : R]$ mit $\ell > r$ und $\mathrm{lcp}(L, M) > \ell$

Fall 2b: Es gilt $\ell = \text{lcp}(L, M)$. Jetzt vergleichen wir $t^{A[M]}$ mit s ab Position $\ell + 1$ bis wir einen erfolglosen Zeichenvergleich an Position k durchgeführt haben (oder s gefunden haben, dann sind wir ja fertig). Dies ist auch in Abbildung 5.27 illustriert. Der schraffierte Bereich zeigt dabei die Übereinstimmung von $t^{A[L]}$ mit $t^{A[M]}$.

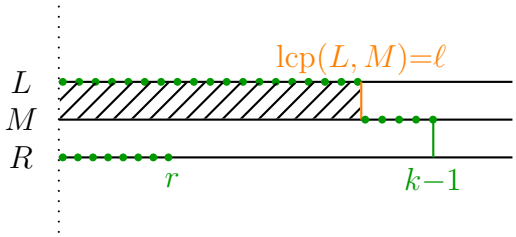


Abbildung 5.27: Skizze: Binäre Suche in $[L : R]$ mit $\ell > r$ und $\text{lcp}(L, M) = \ell$

Anschließend aktualisieren wir entweder L und ℓ oder R und r. War

$$s_{\ell+1} \cdots s_k > t_{A[M]+\ell} \cdots t_{A[M]+k-1},$$

dann setzen wir $L = M$ und $\ell = k - 1$ und sonst $R = M$ und $r = k - 1$.

Fall 2c: Es gilt $\ell > \text{lcp}(L, M)$. Zunächst halten wir fest, dass $\text{lcp}(L, M) \geq r$ gelten muss. Da $t^{A[L]}$ und $t^{A[R]}$ aufgrund der bisherigen Suche nach s in den ersten $\min(\ell, r)$ Positionen übereinstimmen müssen, muss aufgrund der Sortierung des Suffix-Arrays auch $\text{lcp}(L, M) \geq \min(\ell, r)$ gelten. Nach Voraussetzung des Falles 2c gilt dann auch $\text{lcp}(L, M) \geq \min(\ell, r) = r$.

Weiter gilt nach Voraussetzung $s > t^{A[L]}$. Da das Suffix-Array die Suffixe in sortierter Reihenfolge enthält, muss $t^{A[M]} > t^{A[L]}$ und somit muss nach Definition des längsten gemeinsamen Präfixes gelten, dass $t_{A[M]+\text{lcp}(L,M)} > t_{A[L]+\text{lcp}(L,M)} = s_{\text{lcp}(L,M)}$. Somit liegt s im Intervall $[L : M]$ und wir aktualisieren $R := M$ und $r := \text{lcp}(L, M)$ und lassen L und ℓ unverändert. Dies ist auch in Abbildung 5.28 illustriert. Der schraffierte Bereich zeigt dabei die Übereinstimmung von $t^{A[L]}$ mit $t^{A[M]}$

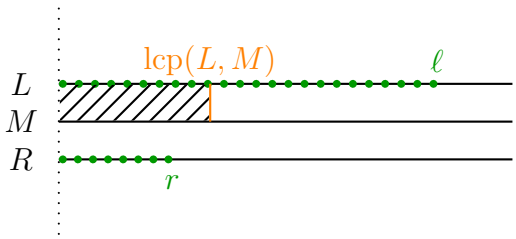


Abbildung 5.28: Skizze: Binäre Suche in $[L : R]$ mit $\ell > r$ und $\text{lcp}(L, M) < \ell$

Fall 3: Nun gilt $\ell < r$. Dieser Fall ist jedoch symmetrisch zum Fall 2.

Damit haben wir den folgenden Satz bewiesen.

Theorem 5.20 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei $s = s_1 \cdots s_m \in \Sigma^*$. Mit Hilfe des Suffix-Arrays und den LCP-Werten von t kann s in t mittels der verbesserten binären Suche in Zeit $O(m + \log(n))$ gefunden werden.*

Beweis: Die Korrektheit haben wir bereits gezeigt, wir müssen uns nur noch um die Laufzeit kümmern. Wir zählen hierzu die erfolgreichen und erfolglosen Zeichenvergleiche getrennt.

Für jeden erfolgreichen Zeichenvergleich wird r oder ℓ erhöht. Da r und ℓ nie erniedrigt werden und da $\ell, r \in [0 : m]$ gilt, kann es maximal $O(m)$ erfolgreiche Zeichenvergleiche geben.

Jeder erfolglose Zeichenvergleich liefert eine Intervallhalbierung. Daher kann es maximal $O(\log(n))$ erfolglose Zeichenvergleiche geben.

Da der Gesamtzeitbedarf proportional zur Anzahl der Zeichenvergleiche ist, beträgt der Gesamtzeitbedarf $O(m + \log(n))$. ■

Wir müssen uns nur noch überlegen, woher wir die benötigten LCP-Werte bekommen. Im schlimmsten Falle können $\Theta(n^2)$ verschiedene LCP-Anfragen gestellt werden. Man überlegt sich jedoch leicht, dass bei einer binärer Suche für einen festen Text t (aber variables Suchwort s) nicht alle LCP-Anfragen gestellt werden können. Insbesondere kann man sich überlegen, dass es unabhängig von Anfragezeichenreihe s nur $O(n)$ verschiedene mögliche LCP-Anfragen geben kann.

12.12.24

Somit können diese $O(n)$ Anfragen vorab in einem Feld in linearer Zeit vorberechnet werden. Für die effiziente Berechnung (mithilfe von RMQ-Anfragen) wird insbesondere auch noch das folgende Lemma benötigt. Die Beweis- und Implementierungsdetails überlassen wir dem Leser als Übungsaufgabe.

Lemma 5.21 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A das zugehörige Suffix-Array. Dann gilt für $i < j \in [0 : n]$, dass*

$$\mathrm{lcp}(i, j) = \min \{\mathrm{lcp}(k - 1, k) \ : \ k \in [i + 1 : j]\} .$$

Beweis: Übungsaufgabe. ■

Des Weiteren kann man sich überlegen, wie man mithilfe der LCP-Tabelle auch das Intervall ermitteln kann, in dem sich alle Suffixe befinden, die mit dem Suchwort s beginnen.

Theorem 5.22 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei $s = s_1 \cdots s_m \in \Sigma^*$. Mit Hilfe des Suffix-Arrays und den LCP-Werten von t kann für s in t mittels der verbesserten binären Suche in Zeit $O(m + \log(n) + k)$ ein Intervall $[L : R]$ angegeben werden, in dem alle Suffixe mit s beginnen, wobei $k = R - L + 1$ ist.*

Beweis: Übungsaufgabe. ■

5.4.3 Effiziente Berechnung der LCP-Tabelle

Für die effiziente Berechnung der allgemeinen LCP-Tabelle, wie sie für die binäre Suche benötigt wird, hilft eine spezielle LCP-Tabelle (siehe Lemma 5.21), für die wir jetzt ein effizientes Konstruktionsverfahren angeben wollen. Da wir diese LCP-Tabelle auch im nächsten Abschnitt noch benötigen werden, stellt sich diese Tabelle als relativ universell im Gebrauch mit Suffix-Arrays heraus.

Definition 5.23 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A das zugehörige Suffix-Array. Dann ist das Feld L der Länge n die zu A gehörige* LCP-Tabelle*, wenn $L[i] = \text{lcp}(i-1, i)$ für $i \in [1 : n]$.*

Zur Konstruktion arbeiten wir die Suffixe von $t\$$ in absteigender Länge ab. Sei wiederum A das Suffix-Array von t und R das zugehörige inverse Suffix-Array, d.h. $A[R[i]] = i$. Dabei beachten wir, dass die folgende elementare Beziehung gilt

$$\text{lcp}(R[i], R[i] - 1) \geq \text{lcp}(R[i-1], R[i-1] - 1) - 1.$$

Zuerst halten wir fest, dass $t^i = t^{A[R[i]]}$ ein um ein Zeichen kürzeres Suffix von $t^{i-1} = t^{A[R[i-1]]}$ ist. Weiterhin ist $s = t^{A[R[i-1]-1]+1}$ ein Suffix von t, das zudem auch noch an den ersten $\text{lcp}(R[i-1], R[i-1] - 1) - 1$ Positionen mit t^i übereinstimmt und für das $s < t^i$ gilt (außer wenn $lcp(R[i-1], R[i-1] - 1) = 0$, aber dann gibt es für die weitere Diskussion keine besonderen Voraussetzungen mehr). Wir bezeichnen diesen Teil mit s'. Somit muss entweder $t^{A[R[i]-1]} = s$ sein oder es müssen zumindest s und $t^{A[R[i]-1]}$ an den ersten $\text{lcp}(R[i-1], R[i-1]) - 1$ Positionen übereinstimmen (andernfalls wäre A falsch sortiert). Diese Beziehung ist auch in Abbildung 5.29 noch einmal schematisch dargestellt.

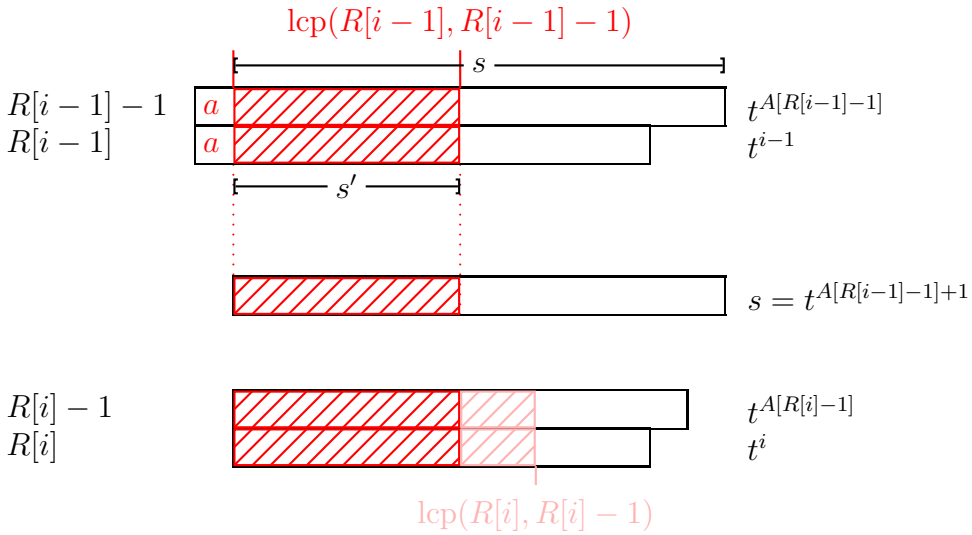


Abbildung 5.29: Skizze: Beziehung zwischen $L[i-1]$ und $L[i]$

Daraus können wir sofort den sehr einfachen, in Abbildung 5.30 dargestellten Algorithmus ableiten und den folgenden Satz festhalten.

Theorem 5.24 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A das zugehörige Suffix-Array. Die zugehörige LCP-Tabelle L kann in Zeit $O(n)$ konstruiert werden.*

Beweis: Die Korrektheit haben wir schon gezeigt. Wir müssen uns nur noch die Laufzeit überlegen. Die äußere for-Schleife ohne die innere while-Schleife wird genau

```
LCP (int A[]; int R[]; char t[]; int n)
begin
    k := 0;
    for (i := 1; i ≤ n; i++) do
        if (R[i] > 0) then        /* always true, since R[i] = 0 ⇔ i = n + 1 */
            j := A[R[i] − 1];                          /* t^j is in front of t^i */
            while (t_{i+k} = t_{j+k}) do
                k++;
            // Now t_{i+k} ≠ t_{j+k}
            L[R[i]] := k;
            k := max(0, k − 1);
end
```

Abbildung 5.30: Algorithmus: Berechnung der LCP-Tabelle L

n Mal durchlaufen. Somit ist Laufzeit ohne die Inkrementierungen von k und die Anzahl der Zeichenvergleiche innerhalb von t durch $O(n)$ beschränkt.

Wie oft kann k überhaupt inkrementiert werden? Da $k \in [0:n]$ gilt und k maximal n mal dekrementiert werden kann (wenn k nicht sowieso schon 0 ist), kann k maximal $2n$ mal inkrementiert werden. Somit wird der Test der while-Schleife insgesamt über alle Durchläufe der for-Schleife maximal $2n + n = 3n$ Mal ausgeführt. Also ist die Gesamtlaufzeit $O(n)$. ■

Dieser Algorithmus benötigt neben dem Platz für den eigentlichen Text t , das Suffix-Array A und die LCP-Tabelle L noch den Platz für das inverse Suffix-Array R. Bei einer normalen Implementierung in der Praxis bedeutet dies für einen Text der Länge $n < 2^{32}$ eine Platzbedarf von $13n$ Bytes.

Mit einem Trick kann man auf das Feld für das inverse Suffix-Array R verzichten und kommt mit nur $9n$ Bytes aus, was bei normalen Anwendungen optimal ist. Für Details verweisen wir den Leser auf die Originalliteratur von Manzini. Dort sind auch weitere Platzreduktionen angebeben, wenn beispielsweise anschließend das Suffix-Array selbst auch nicht mehr benötigt wird.

Wir wollen auch noch anmerken, dass sich das LCP-Array mit $2n+o(n)$ Bits abspeichern lässt, wobei jeder Wert der LCP-Tabelle in konstanter Zeit ermittelt werden kann. Hierfür verweisen wir auf die Originalliteratur von Sadakane.

5.5 Enhanced Suffix-Arrays (*)

In diesem Abschnitt wollen wir jetzt noch zeigen, wie man die Algorithmen für Suffix-Bäume auf Suffix-Arrays übertragen kann. Dabei wird die LCP-Tabelle eine bedeutende Rolle spielen. In diesem Abschnitt nehmen wir jetzt jedoch an, dass $\$ \notin \Sigma$ jetzt das größte Zeichen ist, d.h. es gilt $a < \$$ für alle $a \in \Sigma$.

Dieser Abschnitt ist nur noch der Vollständigkeit halber im Skript enthalten, da dieser Abschnitt bis zum WS 2004/05 gelesen wurde. Die im nächsten Abschnitt vorgestellten Extended Suffix Arrays können Suffix-Bäume wesentlich einfacher simulieren und können darüber hinaus auch sehr leicht in äußert platzsparende Varianten transformiert werden.

5.5.1 LCP-Intervalle

Zunächst definieren die so genannten LCP-Intervalle, die grundlegend für das Folgende sind. Wir werden später auch Intervalle der Länge 1 zulassen.

Definition 5.25 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A bzw. L das zugehörige Suffix-Array bzw. die zugehörige LCP-Tabelle. Ein Intervall $[i : j]$ mit $i < j \in [0 : n]$ heißt ein* LCP-Intervall vom Typ ℓ *oder kurz ein* ℓ-Intervall, *wenn*

1. $L[i] < \ell$,
2. $L[k] \geq \ell$ *für alle* $k \in [i+1 : j]$,
3. $L[k] = \ell$, *für mindestens ein* $k \in [i+1 : j]$ *und*
4. $L[j+1] < \ell$.

Hierbei gelte $L[0] = L[n+1] = -1$.

Notation 5.26 *ℓ-$[i : j]$ bezeichnet das ℓ-Intervall $[i : j]$.*

In Abbildung 5.31 sind die ℓ-Intervalle für unser Beispielwort MISSISSIPPI noch einmal illustriert.

i	$A[i]$	$L[i]$	$C[i]$ down	$C[i]$ up	$C[i]$ next	$t^{A[i]}$
0	8	−1	4			IPPI$
1	5	1	2		3	ISSIPPI$
2	2	4				ISSISSIPPI$
3	11	1		2		I$
4	1	0		1	5	MISSISSIPPI$
5	10	0	6		7	PI$
6	9	1				PPI$
7	7	0	9	6	11	SIPPI$
8	4	2				SISSIPPI$
9	6	1	10	8		SSIPPI$
10	3	3				SSISSIPPI$
11	12	0		9		$

Abbildung 5.31: Beispiel: Suffix-Array, LCP-Tabelle, Child-Tabelle und alle ℓ-Intervalle für MISSISSIPPI

Lemma 5.27 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A bzw. L das zugehörige Suffix-Array bzw. die zugehörige LCP-Tabelle. Für jedes $k \in [1 : n]$ mit $L[k] = \ell$ existiert ein ℓ-Intervall $[i : j]$ mit $k \in [i + 1 : j]$*

Beweis: Wir definieren zunächst einmal $i := \max\{p < k : L[p] < L[k] = \ell\}$ und $j := \min\{p \geq k : L[p + 1] < L[k] = \ell\}$. Diese existieren nach der erweiterten Definition von L immer ($L[0] = L[n + 1] = -1$).

Wir behaupten jetzt, dass $[i : j]$ das gesuchte ℓ-Intervall ist. Nach Definition von i gilt $L[i] < \ell$, also gilt Bedingung 1 der Definition eines LCP-Intervalls. Analog gilt nach Definition von j gilt $L[j + 1] < \ell$, also gilt Bedingung 4 der Definition eines LCP-Intervalls. Weiterhin gilt $k \in [i + 1 : j] \neq \emptyset$ und $L[k] = \ell$, also Bedingung 3 der Definition. Da wir i maximal und j minimal unter der Bedingung $L[p] < L[k] = \ell$ wählen, gilt $L[r] \geq L[k] = \ell$ für alle $r \in [i + 1 : j]$ und somit die Bedingung 2. ■

Jetzt definieren wir noch so genannte ℓ-Indizes

Definition 5.28 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A bzw. L das zugehörige Suffix-Array bzw. die zugehörige LCP-Tabelle. Wenn $[i : j]$ ein ℓ-Intervall ist und wenn $L[k] = \ell$ für ein $k \in [i + 1 : j]$ ist, dann heißt k ein* ℓ-Index*. Die Menge aller ℓ-Indizes eines ℓ-Intervalls $[i : j]$ wird mit $I_\ell[i : j]$ bezeichnet.*

Definition 5.29 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A bzw. L das zugehörige Suffix-Array bzw. die zugehörige LCP-Tabelle. Ein m-Intervall $[i' : j']$ ist in einem ℓ-Intervall $[i : j]$* enthalten*, wenn $i \leq i' < j' \leq j$ und $m > \ell$ gilt.*

Wenn $[i' : j']$ ein in $[i : j]$ enthaltenes Intervall ist und es kein anderes ℓ'-Intervall gibt, das in $[i : j]$ enthalten ist und das $[i' : j']$ enthält, dann heißt $[i' : j']$ ein Kind-Intervall *von $[i : j]$. Umgekehrt heißt $[i : j]$ das* Elter-Intervall *von $[i' : j']$.*

Mit Hilfe dieser Enthaltensein-Relation können wir zu den LCP-Intervallen auch eine Baumstruktur generieren. Betrachten wir zunächst noch einmal den Suffix-Baum für MISSISSIPPI in Abbildung 5.32. Konstruieren wir für das zugehörige Suffix-Array und den zugehörigen LCP-Werten den so genannten *LCP-Intervall-Baum*, der auf der Enthaltensein-Relation der LCP-Intervall definiert ist. Beachte dabei, dass $[0 : n]$ immer ein 0-Intervall ist. Weiter ergänzen wir den so konstruierten LCP-Intervall-Baum um die so genannten *Singletons*, das sind die einelementigen Mengen von $[0 : n]$. In der Regel werden wir auch Singletons als Kind-Intervalle zulassen. In Abbildung 5.33 ist der LCP-Intervall-Baum für unser Beispielwort MISSISSIPPI angegeben.

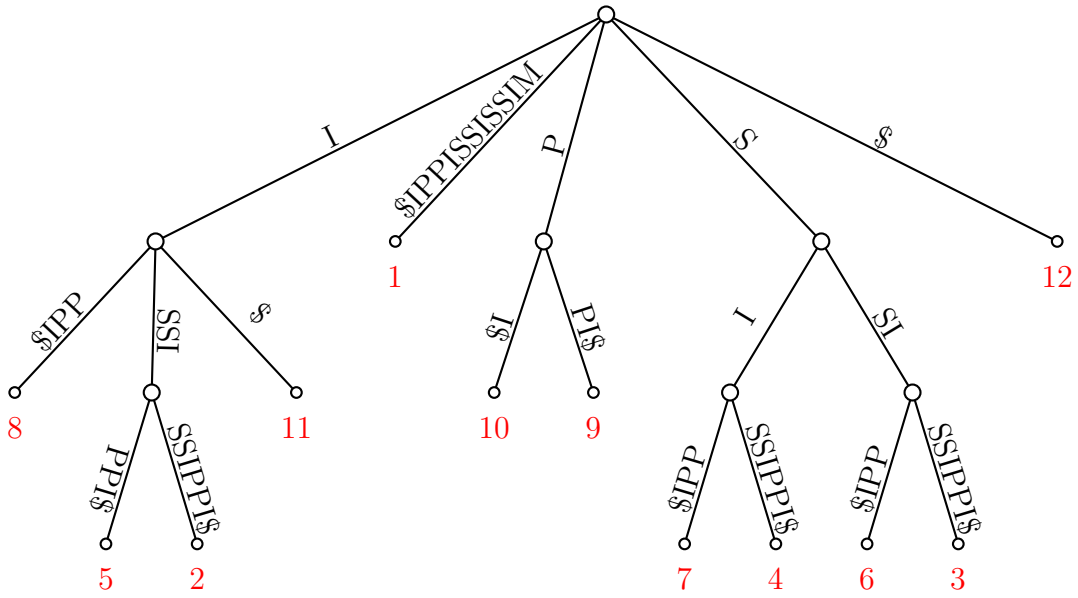


Abbildung 5.32: Beispiel: Suffix-Baum für MISSISSIPPI

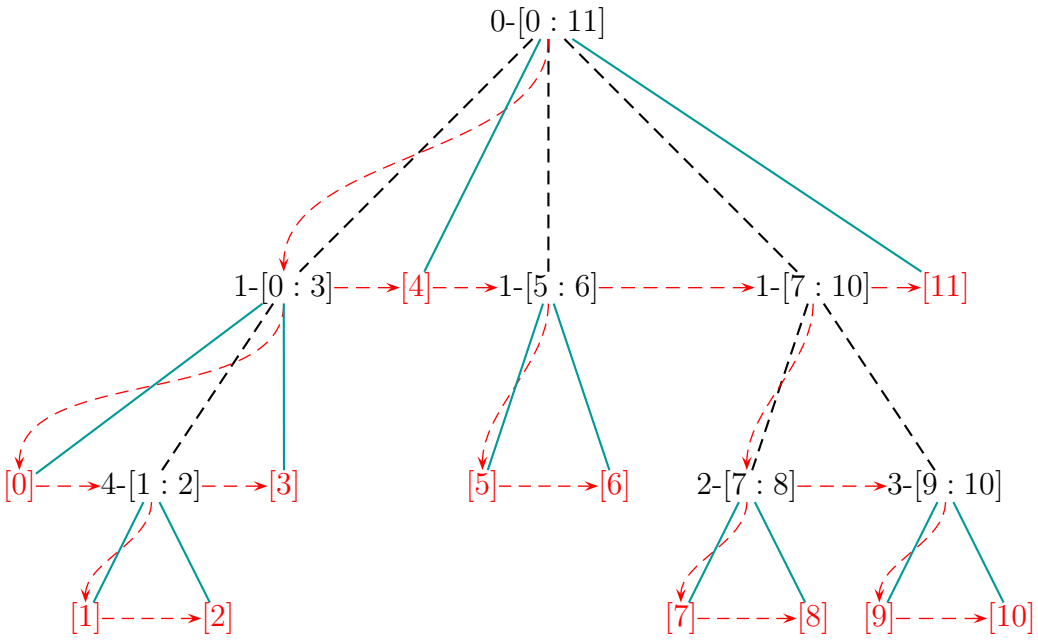


Abbildung 5.33: Beispiel: LCP-Intervall-Baum für MISSISSIPPI

Man sieht sofort die Ähnlichkeit zum entsprechenden Suffix-Baum. In den LCP-Intervallen sind also bereits alle wesentlichen Informationen des Suffix-Baumes (ohne die Blätter) enthalten. In Abbildung 5.33 entsprechen die schwarzen Kanten der Enthaltensein-Relation der Suffix-Bäume und die türkisen Kanten den Kanten zu den Blätter, die nach Definition ja keine ℓ-Intervalle sind. Die roten Kanten entsprechen den Kanten auf das älteste Kind bzw. auf das nächstjüngere Geschwister gemäß der Definition in der Darstellung eines Suffix-Baumes. Im Folgenden müssen wir nur versuchen diese roten Kanten aus den LCP-Intervallen zu rekonstruieren.

Das folgende Lemma zeigt, wie man für ein gegebenes LCP-Intervall die zugehörigen Kind-Intervalle bestimmen kann.

Lemma 5.30 *Sei $[i : j]$ ein ℓ-Intervall und seien $i_1 < i_2 < \cdots < i_k$ die ℓ-Indizes des ℓ-Intervalls $[i : j]$, dann sind die Kind-Intervalle von $[i : j]$ gerade $[i : i_1 - 1]$, $[i_1 : i_2 - 1]$, $\ldots$, $[i_k : j]$.*

Wir merken hier noch an, dass hierbei $[i_p : i_{p+1} - 1]$ für $p \in [1 : k - 1]$ durchaus ein Singleton sein kann.

Beweis: Sei $[r : s]$ eines der Intervalle $[i : i_1 - 1]$, $[i_1 : i_2 - 1]$, $\ldots [i_k : j]$.

Ist $[r : s]$ ein Singleton (d.h. $r = s$), dann ist $[r : s]$ nach unserer Konvention ein Kind-Intervall.

Gelte also jetzt $r < s$. Wir definieren zunächst $m = \min\{L[x] : x \in [r + 1 : s]\}$ sowie $x' = \operatorname{argmin}\{L[x] : x \in [r + 1 : s]\}$. Nach Definition des ℓ-Intervalls gilt $m > \ell$. Nach Lemma 5.27 existiert ein m-Intervall I, das die Position x' umfasst. Nach Definition der ℓ-Indizes des ℓ-Intervalls $[i : j]$ und nach Wahl von m muss diese Intervall I gerade gleich $[r : s]$ sein.

Weiterhin ist $[r : s]$ in $[i : j]$ enthalten. Da weiterhin nach Definition der ℓ-Indizes $L[i_1] = L[i_2] = \cdots = L[i_k] = \ell$ gilt, kann kein anderes Intervall in $[i : j]$ enthalten sein, welches $[r : s]$ enthält.

Offensichtlich sind die Intervalle $[i : i_1 - 1]$, $[i_1 : i_2 - 1]$, $\ldots [i_{k-1} : i_k - 1]$, $[i_k : j]$ alle möglichen Kind-Intervalle, die $[i : j]$ besitzen kann. Damit ist der Beweis des Lemmas abgeschlossen. ■

5.5.2 Die Child-Tabelle und ihre Eigenschaften

Um jetzt das Suffix-Array genauso verwenden zu können wie den entsprechenden Suffix-Baum, müssen wir die Menge der ℓ-Indizes eines LCP-Intervalls in linearer Zeit berechnen können. Dazu dient die folgende Notation der Child-Tabelle.

Definition 5.31 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A bzw. L das zugehörige Suffix-Array bzw. die zugehörige LCP-Tabelle. Dann ist die* Child-Tabelle *wie folgt definiert:*

$$\begin{aligned} C[i].up &= \min\{q < i \,:\, L[q] > L[i] \wedge \forall k \in [q+1 : i-1] : L[k] \geq L[q]\}\,; \\ C[i].down &= \max\{q > i \,:\, L[q] > L[i] \wedge \forall k \in [i+1 : q-1] : L[k] > L[q]\}\,; \\ C[i].next &= \min\{q > i \,:\, L[q] = L[i] \wedge \forall k \in [i+1 : q-1] : L[k] > L[q]\}\,. \end{aligned}$$

Hierbei gilt $\min\emptyset = \max\emptyset = \perp$, wobei $\perp$ für undefiniert steht.

Im Wesentlichen ist das Suffix-Array zusammen mit der LCP-Tabelle und der Child-Tabelle das *Enhanced-Suffix-Array*. In Abbildung 5.34 sind die LCP-Werte zur Definition der Child-Tabelle noch einmal illustriert. Der grüne Bereich gibt dabei den Bereich des LCP-Wertes von $L[q+1]$ bzw. $L[q-1]$ an.

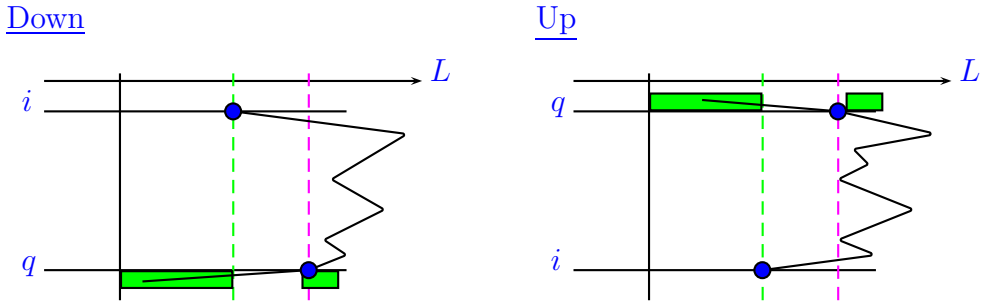


Abbildung 5.34: Skizze: Die L-Gebirge in der Definition von $C[i]$.down und $C[i]$.up

Lemma 5.32 *Für jedes ℓ-Intervall $[i : j]$ gelten die folgenden Aussagen:*

1) $C[j+1].up \in [i+1 : j]$ oder $C[i].down \in [i+1 : j]$.

2) $C[j+1].up$ speichert den ersten ℓ-Index von $[i : j]$, wenn $C[j+1].up \in [i+1 : j]$.

3) $C[i].down$ speichert den ersten ℓ-Index von $[i : j]$, wenn $C[i].down \in [i+1 : j]$.

Beweis: zu 1.) Sei $\ell' := L[j+1]$. Da $[i : j]$ ein ℓ-Intervall ist, gilt $\ell' < \ell$. Weiter gilt $L[k] \geq \ell$ für alle $k \in [i+1 : j]$.

Gilt $L[i] < \ell'$, dann ist $C[j+1].\text{up} \in [i+1 : j]$ nach Definition von $C[j+1].\text{up}$. Das L-Gebirge im Intervall $[i : j+1]$ ist für diesen Fall Abbildung 5.35 a) noch einmal illustriert.

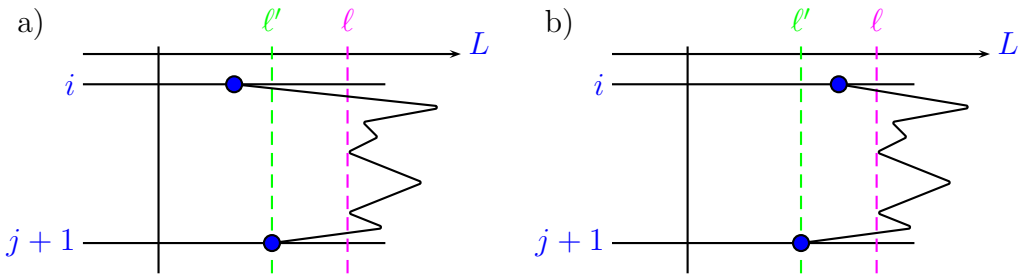


Abbildung 5.35: Skizze: Die L-Gebirge in Behauptung 1

Gilt andererseits $L[i] \geq \ell'$, dann ist $C[i].\text{down} \in [i+1 : j]$ nach Definition von $C[i].\text{down}$. Beachte, dass nach Definition des ℓ-Intervalls $[i : j]$ in jedem Falle $L[i] < \ell$ gelten muss. Das L-Gebirge im Intervall $[i : j+1]$ ist für diesen Fall Abbildung 5.35 b) noch einmal illustriert.

zu 2.) Wenn $C[j+1].\text{up} \in [i+1 : j]$, dann gilt:

$$\begin{aligned} & C[j+1].\text{up} \\ &= \min\{q \in [i+1 : j] \ : \ L[q] > L[j+1] \wedge \forall k \in [q+1 : j] : L[k] \geq L[q]\} \end{aligned}$$

Da $[i : j]$ ein ℓ-Intervall ist und da $L[q] \geq \ell > L[j+1]$ für $q \in [i+1 : q]$ sein muss, erhalten wir weiter

$$\begin{aligned} &= \min\{q \in [i+1 : j] \ : \ \forall k \in [q+1 : j] : L[k] \geq L[q]\} \\ &= \min I_\ell[i : j]. \end{aligned}$$

zu 3.) Sei i_1 der erste ℓ-Index des ℓ-Intervalls $[i : j]$. Dann gilt $L[i_1] = \ell > L[i]$ und für alle $k \in [i+1 : i_1 - 1]$ gilt $L[k] > \ell = L[i_1]$. Weiter gilt für alle $q \in [i_1 + 1 : j]$, dass $L[q] \geq \ell > L[i]$, aber nicht $L[i_1] > L[q]$. Somit wird das Maximum an der Position i_1 angenommen. ■

Korollar 5.33 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A, L bzw. C das zugehörige Suffix-Array, die zugehörige LCP-Tabelle bzw. die zugehörige Child-Tabelle. Für jedes LCP-Intervall $[i : j]$ gilt:*

$$\text{lcp}(i,j) = \begin{cases} L[C[j+1].up] & \text{falls } C[j+1].up \in [i+1 : j], \\ L[C[i].down] & \text{sonst.} \end{cases}$$

Beweis: Die Korrektheit folgt unmittelbar aus dem vorherigen Lemma zusammen mit Lemma 5.21. ■

Mit Hilfe dieser fundamentalen Eigenschaften der Child-Tabelle können wir den in Abbildung 5.36 angegeben Algorithmus GETNEXTCHILDINTERVAL zum Auffinden eines Kind-Intervalls entwerfen. Als Eingabe erhält der Algorithmus ein ℓ-Intervall $[i : j]$ und einen Parameter $k \in \{i\} \cup I_\ell[i : j]$. Ist $k = i$ so liefert der Algorithmus den ersten ℓ-Index von $[i : j]$, ansonsten den kleinsten ℓ-Index, der größer als k ist.

Somit können wir zu einem ℓ-Intervall alle Kinder-Intervalle in konstanter Zeit pro Intervall ermitteln. Dazu werden wir später noch zeigen, dass wir die Child-Tabelle in linearer Zeit konstruieren können.

Lemma 5.34 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A bzw. C das zugehörige Suffix-Array bzw. die zugehörige Child-Tabelle. Für ein gegebenes ℓ-Intervall $[i : j]$ von A und einen Index $k \in \{i\} \cup I_\ell[i : j]$ kann der kleinste ℓ-Index von ℓ-$[i : j]$, der größer als k ist, in konstanter Zeit bestimmt werden.*

5.5.3 Optimale Suche in Enhanced Suffix-Arrays

Basierend auf dem Algorithmus zum Auffinden aller Kind-Intervalle eines ℓ-Intervalls geben wir in diesem Abschnitt einen optimalen Algorithmus zum Suchen nach Teilwörtern basierend auf Suffix-Arrays an. Der Algorithmus ist in Abbildung 5.36 angegeben.

Die Invariante im Algorithmus ist die folgende: Während der Suche nach einem Wort $s = s_1 \cdots s_m$ gibt die Boolesche Variable `prefix` an, ob ein Präfix der Länge p von s ein in t enthalten ist oder nicht. Weiterhin ist $[i : j]$ ein p-Intervall (außer im Falle $p = m$), dass nur Suffixe von t enthält, die mit dem Präfix $s_1 \cdots s_p$ beginnen. Halten wir das Ergebnis im folgenden Satz fest.

Theorem 5.35 *In einem Suffix-Array zusammen mit den Tabellen L und C kann in Zeit $O(|\Sigma| \cdot m)$ festgestellt werden, ob $s \in \Sigma^m$ in $t \in \Sigma^*$ enthalten ist.*

Beweis: Die Korrektheit folgt aus den Zeichenvergleichen. Die Laufzeit folgt aus der Tatsache, dass alle anderen Funktionen in konstanter Zeit durchgeführt werden können und die Laufzeit somit proportional zur Anzahl der Zeichenvergleiche ist, also $O(m)$ beträgt. ■

Find (char $s[]$, int n)

```
begin
    interval [i : j] := [0 : n];
    int p := 0;
    bool prefix := TRUE; /* prefix=TRUE iff s_1 ··· s_p is a prefix of t^{A[i]} */
    while (([i : j] ≠ ⊥) && (p < m) && (prefix)) do
        if (i < j) then                                   /* a Child-Interval */
            k := min(LCP(i, j), m);
            prefix := (t_{A[i]+p} ··· t_{A[i]+k-1} = s_{p+1} ··· s_k);
            p := k;
        else if (i = j) then                                  /* a Singleton */
            prefix := (t_{A[i]+p} ··· t_{A[i]+m-1} = s_{p+1} ··· s_m);
            p := m;
        if (prefix) then [i : j] := getChildIntervalByChar(i, j, p, s_{p+1});
    output (prefix)?[i : j] : ⊥;
end
```

getChildIntervalByChar (int i, j, p; char c)

```
begin
    interval [i' : j'] := getNextChildInterval(i, j, i);
    while ((t_{A[i']+p} ≠ c) && (j' < j)) do
        [i' : j'] := getNextChildInterval(i, j, j' + 1);
    return (t_{A[i']+p} = c)?([i' : j']):(⊥);
end
```

getNextChildInterval (int i, j, k)

```
begin
    if (k = i) then
        return (C[j + 1].up ∈ [i + 1 : j])?[i : C[j + 1].up − 1]:[i : C[i].down − 1];
    else
        return (C[k].next ≠ ⊥)?[k : C[k].next − 1]:[k : j];
end
```

LCP (int i, j)

```
begin
    return (C[j + 1].up ∈ [i + 1 : j])?L[C[j + 1].up]:L[C[i].down];
end
```

Abbildung 5.36: Algorithmus: Suche s im Enhanced Suffix-Array von t

5.5.4 Berechnung der Child-Tabelle

Wir müssen uns jetzt nur noch überlegen, wie wir die Child-Tabelle in linearer Zeit konstruieren können. Die hierfür benötigten Algorithmen sind in der Abbildung 5.37 angegeben.

Sowohl im Algorithmus ComputeUpDownTable als auch im Algorithmus ComputeNextTable gilt die folgende Invariante. Sind $0, i_1, \ldots, i_k$ die Element auf dem Stack S, dann gilt zum einen $0 < i_1 < \cdots < i_k$ und zum anderen $L[i_1] \leq \cdots \leq L[i_k]$. Weiterhin gilt für alle $k \in [i_j + 1 : i_{j+1} - 1]$ für zwei aufeinander folgende Stackelemente $i_j < i_{j+1}$ mit $L[i_j] < L[i_{j_1}]$, dass $L[k] > L[i_{j+1}]$.

Wir zeigen mithilfe der beiden folgenden Lemmata die Korrektheit der beiden Algorithmen.

Lemma 5.36 *Der Algorithmus ComputeUpDownTable ermittelt die korrekten Up- und Down-Werte der Child-Tabelle in linearer Zeit.*

Beweis: Wenn $C[S.\text{top}()].\text{down}$ gesetzt wird, gilt $L[k] \leq L[S.\text{top}()] < L[\text{lastIdx}]$ sowie $S.\text{top}() < \text{lastIdx} < k$. Es gilt also $\text{lastIdx} < k$ und $L[\text{lastIdx}] > L[S.\text{top}()]$ sowie aufgrund der Invariante $L[p] > L[\text{lastIdx}]$ für alle $p \in [S.\text{top}()+1 : \text{lastIdx}-1]$. Somit befindet sich lastIdx in der Menge, deren Maximum gerade $C[S.\text{top}()].\text{down}$ ist. Angenommen lastIdx wäre nicht das Maximum, sondern $q' \in [\text{lastIdx}+1 : k-1]$. Nach Definition von $C[S.\text{top}()].\text{down}$ muss $L[\text{lastIdx}] > L[q']$ sein. Somit muss lastIdx vom Stack entfernt worden sein, als der Index q' betrachtet wurde, was den gewünschten Widerspruch liefert.

Wenn $C[k].\text{up}$ gesetzt wird, dann gilt insbesondere $L[S.\text{top}()] \leq L[k] < L[\text{lastIdx}]$ und $S.\text{top}() < \text{lastIdx} < k$. Es gilt also $\text{lastIdx} < k$ und $L[\text{lastIdx}] > L[k]$ sowie aufgrund der Invariante $L[p] \geq L[\text{lastIdx}]$ für alle $p \in [\text{lastIdx} + 1 : i - 1]$. Somit befindet sich lastIdx in der Menge, deren Minimum gerade $C[k].\text{up}$ ist. Angenommen lastIdx wäre nicht das Minimum, sondern $q' \in [S.\text{top}() + 1 : \text{lastIdx} - 1]$. Nach Definition von $C[k].\text{up}$, muss $L[\text{lastIdx}] \geq L[q'] > L[k] \geq L[S.\text{top}()]$ gelten. Somit gilt $q \in [S.\text{top}() + 1 : \text{lastIdx} - 1]$, was den gewünschten Widerspruch liefern. ■

Die Korrektheit des folgenden Lemmas folgt direkt aus der Inspektion des Algorithmus in Abbildung 5.37.

Lemma 5.37 *Der Algorithmus ComputeNextTable ermittelt die korrekten Next-Werte der Child-Tabelle in linearer Zeit.*

ComputeUpDownTable

```
begin
    int lastIdx := −1;
    stack S := empty();
    S.push(0);

    for (k := 1; k ≤ n; k++) do
        while (L[k] < L[S.top()]) do
            lastIdx := S.pop();
            if ((L[k] ≤ L[S.top()]) && (L[S.top()] < L[lastIdx])) then
                C[S.top()].down := lastIdx;

        // Now L[k] ≥ L[S.top()]
        if (lastIdx ≠ −1) then
            C[k].up := lastIdx;
            lastIdx := −1;
        S.push(k);
end
```

ComputeNextTable

```
begin
    stack S := empty();
    S.push(0);

    for (k := 1; k ≤ n; k++) do
        while (L[k] < L[S.top()]) do
            S.pop();
        if (L[k] = L[S.top()]) then
            lastIdx := S.pop();
            C[lastIdx].next := k;
        S.push(k);
end
```

Abbildung 5.37: Algorithmen: Berechnung der Up- und Down- sowie Next-Werte

5.5.5 Komprimierte Darstellung der Child-Tabelle

Wir wollen jetzt noch zeigen, dass die Child Tabelle in einem Feld mit n statt $3n$ Einträgen gespeichert werden kann.

Lemma 5.38 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A bzw. C das zugehörige Suffix-Array bzw. die zugehörige Child-Tabelle. Für jedes $i \in [0 : n]$ gilt, dass $C[i].next = \bot$, wenn $C[i+1].up \neq \bot$ gilt.*

Beweis: Ist $C[i+1].\text{up} \neq \bot$, dann ist insbesondere $L[i] > L[i+1]$. Dann kann aber $C[i].\text{next}$ nicht definiert sein. ∎

Somit kann das Up-Feld im entsprechenden Next-Feld gespeichert werden. Gilt bei einer Anfrage $C[i].\text{next} > i$, dann wird tatsächlich das Next-Feld gespeichert, andernfalls das Up-Feld an Position $i+1$.

Wir werden jetzt noch zeigen, dass auch das Down-Feld im Next-Feld gespeichert werden kann. Wir bemerken zuerst, dass wir für ein ℓ-Intervall $[i : j]$ das Feld $C[i].\text{down}$ nur dann gespeichert werden muss, wenn $C[j+1].\text{up} \notin [i+1 : j]$.

Lemma 5.39 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A bzw. C das zugehörige Suffix-Array bzw. die zugehörige Child-Tabelle. Für jedes ℓ-Intervall $[i : j]$, für das $C[j+1].up \notin [i+1 : j]$ gilt, gilt $C[i].next = \bot$ und $C[i+1].up = \bot$.*

Beweis: Wenn $[i : j]$ ein ℓ-Intervall ist, gilt $L[i] < \ell$, $L[j+1] < \ell$ sowie $L[k] \geq \ell$ für $k \in [i+1 : j]$. Wenn weiter $C[j+1].\text{up} \notin [i+1 : j]$ gilt, dann muss $L[i] > L[j+1]$. Dann gilt $L[k] \geq \ell > L[i]$ für alle $k \in [i+1 : j]$ und $L[j+1] < L[i]$. Also muss $C[i].\text{next} = \bot$ gelten.

Da $L[i] < \ell$ und $L[i+1] \geq \ell$ gilt, muss $C[i+1].\text{up} = \bot$ sein. ∎

Somit können wir auch den Wert $C[i].\text{down}$, wenn er denn nicht bereits durch $C[j+1].\text{up}$ gegeben ist, in $C[i].\text{next}$ speichern. Wir müssen jetzt nur noch unterscheiden, ob $C[i].\text{next}$ den Next- oder ein Up-Eintrag enthält, wenn $C[i].\text{next} > i$ ist. Gilt $L[i] = L[C[i].\text{next}]$, dann wird der Next-Wert gespeichert, gilt $L[i] < L[C[i].\text{next}]$, dann wird der zugehörige Down-Wert gespeichert.

Für unser Beispiel ist dies in Abbildung 5.38 noch einmal veranschaulicht.

Definition 5.40 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A, L bzw. C das zugehörige Suffix-Array, die zugehörige LCP-Tabelle bzw. die zugehörige Child-Tabelle. Dann ist (A, L, C) das zu t gehörige* Enhanced-Suffix-Array.

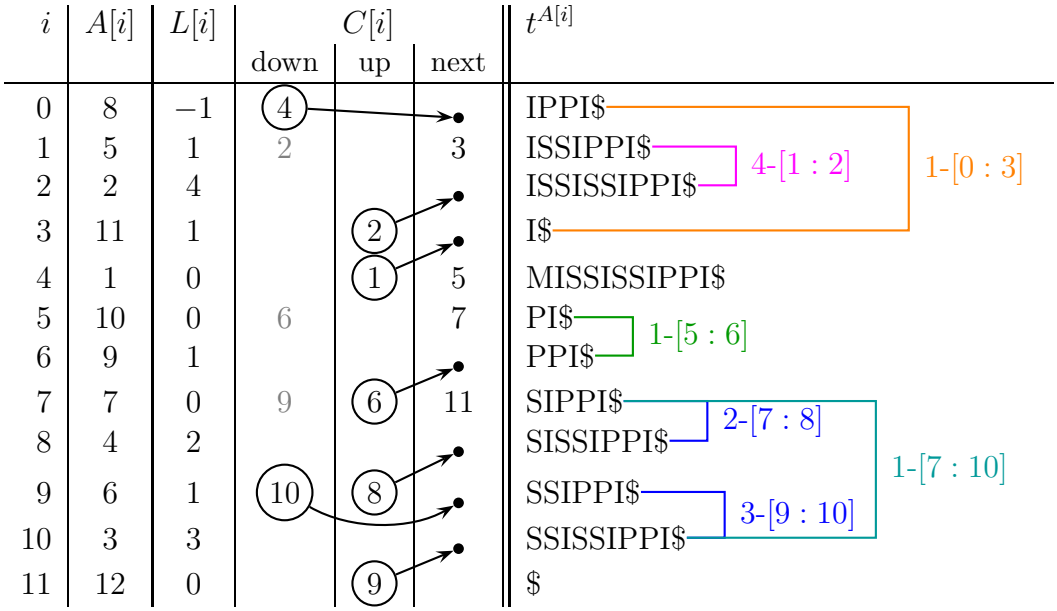


Abbildung 5.38: Beispiel: Suffix-Array, LCP-Tabelle, komprimierter Child-Tabelle und alle ℓ-Intervalle für MISSISSIPPI (außer dem 0-Intervall)

Wie man leicht sieht, kann das Enhanced Suffix-Array für $n < 2^{32}$ mit $12n$ Bytes realisiert werden. Da die LCP-Tabelle hauptsächlich kleine Werte enthält, kann diese auch durch ein Feld mit n Bytes realisiert werden. Analog kann auch die Child-Tabelle mit einem Feld der Größe von n Bytes realisiert werden, wenn statt der tatsächlichen Indexpositionen die relativen Offsets zur Position i gespeichert werden.

Für Werte außerhalb des Intervalls $[0 : 254]$ bzw. $[-127 : 127]$, wird im Feld der Wert 255 bzw. -128 gespeichert und in einem Extrafeld das Paar aus Indexposition und Wert gespeichert, auf die dann mit einer binären Suche im Ausnahmefall schnell zugegriffen werden kann. Dann benötigt das Enhanced-Suffix-Array etwa 6n Bytes plus den Text t sowie Platz für die übergroßen Werte (also außerhalb des Intervalls $[0 : 254]$ bzw. $[-127 : 127]$).

Die Suche in der Tabelle mit den übergroßen Werten kann mithilfe einer binären Suche relativ effizient ausgeführt werden. Die worst-case Laufzeit sinkt zwar bei dieser speicherplatzsparenden Darstellung, macht in der Praxis aber nicht allzuviel aus, wenn die zugehörigen Texte nur wenige übergroße Werte implizieren.

5.5.6 Simulation von Suffix-Baum-Algorithmen auf Suffix-Arrays

Somit lassen sich leicht DFS-Traversierungen auf einem Suffix-Array vornehmen, wie beispielsweise in Abbildung 5.39 angegeben. Auch BFS-Traversierung können ähnlich durchgeführt werden, die Details seien dem Leser zur Übung überlassen.

traverse (interval $[i : j]$)

```
begin
    // traverse suffix-array recursively in a depth-first-search
       manner

    [i' : j'] := getNextChildInterval(i, j, i);
    while ([i' : j'] ≠ ⊥) do
        traverse([i' : j']);
        [i' : j'] := getNextChildInterval(i, j, j' + 1);
end
```

Abbildung 5.39: Algorithmus: Traversierung des Suffix-Arrays

Es bleibt noch zu überlegen, wie man Suffix-Links in die Enhanced-Suffix-Arrays integrieren kann.

Definition 5.41 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A das zugehörige Suffix-Array. Für $t^{A[i]} = aw\$$ mit $a \in \Sigma$ und $w \in \Sigma^*$ ist $\sigma(i) = j$ mit $t^{A[j]} = w\$$ der* Suffix-Link an Position i.

Wie man leicht sieht, gilt die folgende Beobachtung.

Beobachtung 5.42 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A bzw. R das zugehörige Suffix-Array bzw. das zugehörige inverse Suffix-Array. Dann gilt $\sigma(i) = R[A[i]+1]$, sofern $A[i] \leq n$*

Wir haben somit nur Suffix-Links für Singletons definiert. Diese zeigen selbst jedoch wieder auf Singletons, namentlich das Intervall $[R[A[i] + 1] : R[A[i] + 1]]$, sofern $A[i] \neq n + 1$ ist. Ist $A[i] = n + 1$, so beschreibt das Singleton das Suffix \$, dessen Suffix-Link das Wurzel-Intervall 0-$[0 : n]$ ist. Für LCP-Intervalle können wir das folgende Lemma beweisen.

Lemma 5.43 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A das zugehörige Suffix-Array. Zu jedem ℓ-Intervall $[i : j]$ mit $\ell > 0$ existiert genau ein kleinstes $(\ell - 1)$-Intervall $[r : s]$ mit $r \leq \sigma(i) < \sigma(j) \leq s$.*

Beweis: Da $[i : j]$ ein ℓ-Intervall ist, gilt offensichtlich für $k \in [\sigma(i) + 1 : \sigma(j)]$, dass $L[k] \geq \ell - 1$. Da $[i : j]$ ein ℓ-Intervall ist, muss es weiter einen ℓ-Index $k \in [i + 1 : j]$ mit $L[k] = \ell$ geben. Deshalb muss es auch einen Index $k' \in [\sigma(i) + 1 : \sigma(j)]$ mit

$L[k'] = \ell - 1$ geben. Nach Lemma 5.27 existiert dann ein $(\ell-1)$-Intervall $[r : s]$, das nach Konstruktion $[\sigma(i) : \sigma(j)]$ umfasst. ■

Mit diesem Existenz-Satz können wir nun die folgende Definition für Suffix-Links von LCP-Intervallen wagen.

Definition 5.44 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A das zugehörige Suffix-Array. Zu jedem ℓ-Intervall $[i : j]$ mit $\ell > 0$ ist das kleinste $(\ell-1)$-Intervall $[r : s]$ mit $r \leq \sigma(i) < \sigma(j) \leq s$ das* Suffix-Link-Intervall *von $[i : j]$.*

Der Beweis des folgenden Korollars ergibt sich sofort aus einer genauen Inspektion des Beweises von Lemma 5.43.

Korollar 5.45 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A das zugehörige Suffix-Array. Sei $[i : j]$ ein ℓ-Intervall mit $\ell > 0$ und sei $[r : s]$ sein Suffix-Link-Intervall. Dann existiert ein $(\ell-1)$-Index $k \in I_{\ell-1}[r : s]$ und $k \in [\sigma(i)+1 : \sigma(j)]$.*

Sei nun $[i : j]$ ein ℓ-Intervall und $[r : s]$ das zugehörige Suffix-Link-Intervall. Dann gilt $r \leq \sigma(i) < \sigma(j) \leq s$ und es existiert ein $k \in [\sigma(i)+1 : \sigma(j)]$ mit $L[k] = \ell - 1$. Dieses k lässt sich mithilfe einer Range Minimum Query Anfrage $\mathrm{RMQ}_L(\sigma(i)+1, \sigma(j))$ auf dem Feld L in konstanter Zeit ermitteln (nach einer linearen Vorverarbeitungszeit). Wenn wir an jedem ℓ-Index die Intervallgrenzen des zugehörigen ℓ-Intervalls notiert haben, können wir diese in konstanter Zeit nachschlagen.

Wie speichern wir uns nun für jedes ℓ-Intervall seine Intervallgrenzen an seinen ℓ-Indizes? Dazu traversieren wir den LCP-Intervall-Baum mit einer Breitensuche (die Implementierung einer Breitensuche sei dem Leser zur Übung überlassen). Jedes Mal wenn wir ein neues ℓ-Intervall bearbeiten, tragen wir an seinen ℓ-Indizes die Intervall grenzen ein. Die Laufzeit ist also proportional zur Anzahl aller ℓ-Indizes (für alle ℓ-Intervalle und für alle ℓ) und ist somit linear in $|t|$.

Theorem 5.46 *Sei E ein Enhanced-Suffix-Array, dann kann dieses in linearer Zeit mit linearem Platz so erweitert werden, dass für jedes LCP-Intervall von E dessen Suffix-Link-Intervall in konstanter Zeit aufgefunden werden kann.*

5.6 Extended Suffix Arrays

In diesem Abschnitt wollen wir eine einfache Variante vorstellen, wie man die Algorithmen für Suffix-Bäume auf Suffix-Arrays übertragen kann. Dabei wird die LCP-Tabelle wiederum eine bedeutende Rolle spielen. In diesem Abschnitt nehmen wir an, dass $\$ \notin \Sigma$ jetzt das kleinste Zeichen ist, d.h. es gilt $\$ < a$ für alle $a \in \Sigma$.

5.6.1 LCP-Intervalle

Zunächst definieren die so genannten LCP-Intervalle, die grundlegend für das Folgende sind.

Definition 5.47 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A bzw. L das zugehörige Suffix-Array bzw. die zugehörige LCP-Tabelle. Ein Intervall $[i : j]$ mit $i < j \in [0 : n]$ heißt ein* LCP-Intervall vom Typ ℓ *oder kurz ein* ℓ-Intervall, *wenn*

1. $L[i] < \ell$,
2. $L[k] \geq \ell$ *für alle* $k \in [i+1 : j]$,
3. $L[k] = \ell$, *für mindestens ein* $k \in [i+1 : j]$ *und*
4. $L[j+1] < \ell$.

Hierbei gelte $L[0] = L[n+1] = -1$.

Notation 5.48 *ℓ-$[i : j]$ bezeichnet das ℓ-Intervall $[i : j]$.*

In Abbildung 5.40 sind die ℓ-Intervalle für unser Beispielwort MISSISSIPPI noch einmal illustriert.

Wir werden später auch Intervalle der Länge 1 als LCP-Intervalle bezeichnen, die dann keinen expliziten Typ besitzen.

i	$A[i]$	$L[i]$	$t^{A[i]}$
0	12	−1	\$
1	11	0	I\$
2	8	1	IPPI\$
3	5	1	ISSIPPI\$
4	2	4	ISSISSIPPI\$
5	1	0	MISSISSIPPI\$
6	10	0	PI\$
7	9	1	PPI\$
8	7	0	SIPPI\$
9	4	2	SISSIPPI\$
10	6	1	SSIPPI\$
11	3	3	SSISSIPPI\$

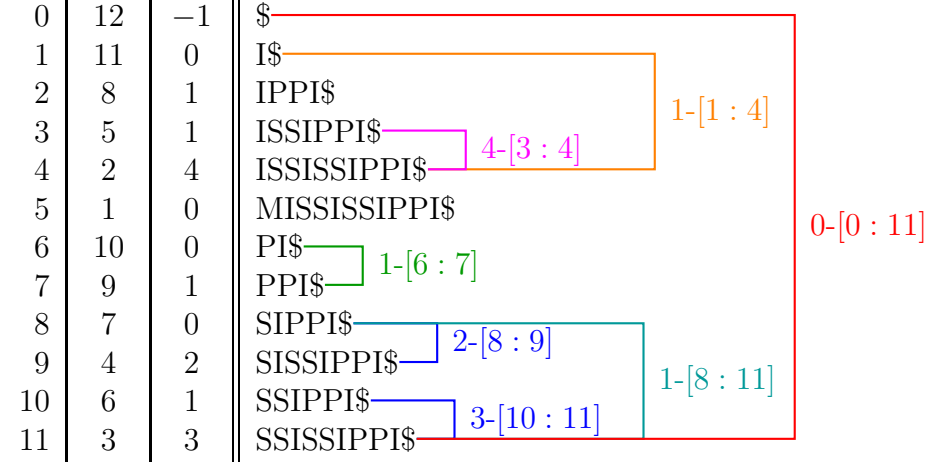


Abbildung 5.40: Beispiel: Alle ℓ-Intervalle für MISSISSIPPI

Lemma 5.49 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A bzw. L das zugehörige Suffix-Array bzw. die zugehörige LCP-Tabelle. Für jedes $k \in [1 : n]$ mit $L[k] = \ell$ existiert ein ℓ-Intervall $[i : j]$ mit $k \in [i + 1 : j]$*

Beweis: Wir definieren zunächst einmal $i := \max\{p < k : L[p] < L[k] = \ell\}$ und $j := \min\{p \geq k : L[p+1] < L[k] = \ell\}$. Diese existieren nach der erweiterten Definition von L immer ($L[0] = L[n+1] = -1$). Wir behaupten jetzt, dass $[i : j]$ das gesuchte ℓ-Intervall ist. Nach Definition von i gilt $L[i] < \ell$, also gilt Bedingung 1 der Definition eines LCP-Intervalls. Analog gilt nach Definition von j gilt $L[j+1] < \ell$, also gilt Bedingung 4 der Definition eines LCP-Intervalls. Weiterhin gilt $k \in [i + 1 : j] \neq \emptyset$ und $L[k] = \ell$, also Bedingung 3 der Definition. Da wir i maximal und j minimal unter der Bedingung $L[p] < L[k] = \ell$ wählen, gilt $L[r] \geq L[k] = \ell$ für alle $r \in [i + 1 : j]$ und somit die Bedingung 2. ■

Wir halten noch fest, dass diese LCP-Intervalle entweder ineinander enthalten oder disjunkt sind.

Lemma 5.50 *Seien $[i : j]$ und $[i' : j']$ zwei LCP-Intervalle vom Typ ℓ bzw. ℓ' mit $\ell \geq \ell'$. Dann gilt entweder $[i : j] \subseteq [i'; j']$ oder $[i : j] \cap [i'; j'] = \emptyset$.*

Beweis: Übungsaufgabe. ■

Definition 5.51 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A bzw. L das zugehörige Suffix-Array bzw. die zugehörige LCP-Tabelle. Ein m-Intervall $[i' : j']$ ist in einem ℓ-Intervall $[i : j]$* enthalten*, wenn $i \leq i' < j' \leq j$ und $m > \ell$ gilt.*

Wenn $[i' : j']$ ein in $[i : j]$ enthaltenes Intervall ist und es kein anderes ℓ''-Intervall gibt, das in $[i : j]$ enthalten ist und das $[i' : j']$ enthält, dann heißt $[i' : j']$ ein Kind-Intervall *von $[i : j]$. Umgekehrt heißt $[i : j]$ das* Elter-Intervall *von $[i' : j']$.*

Mit Hilfe dieser Enthaltensein-Relation können wir zu den LCP-Intervallen auch eine Baumstruktur generieren. Betrachten wir zunächst noch einmal den Suffix-Baum für MISSISSIPPI in Abbildung 5.41. Konstruieren wir für das zugehörige Suffix-Array und den zugehörigen LCP-Werten den so genannten *LCP-Intervall-Baum*, der auf der Enthaltensein-Relation der LCP-Intervall definiert ist. Beachte dabei, dass $[0 : n]$ immer ein 0-Intervall ist (da immer $L[1] = 0$ gilt). Weiter ergänzen wir den so konstruierten LCP-Intervall-Baum um die so genannten *Singletons*, das sind die einelementigen Mengen von $[0 : n]$. In Abbildung 5.42 ist der LCP-Intervall-Baum für unser Beispielwort MISSISSIPPI angegeben.

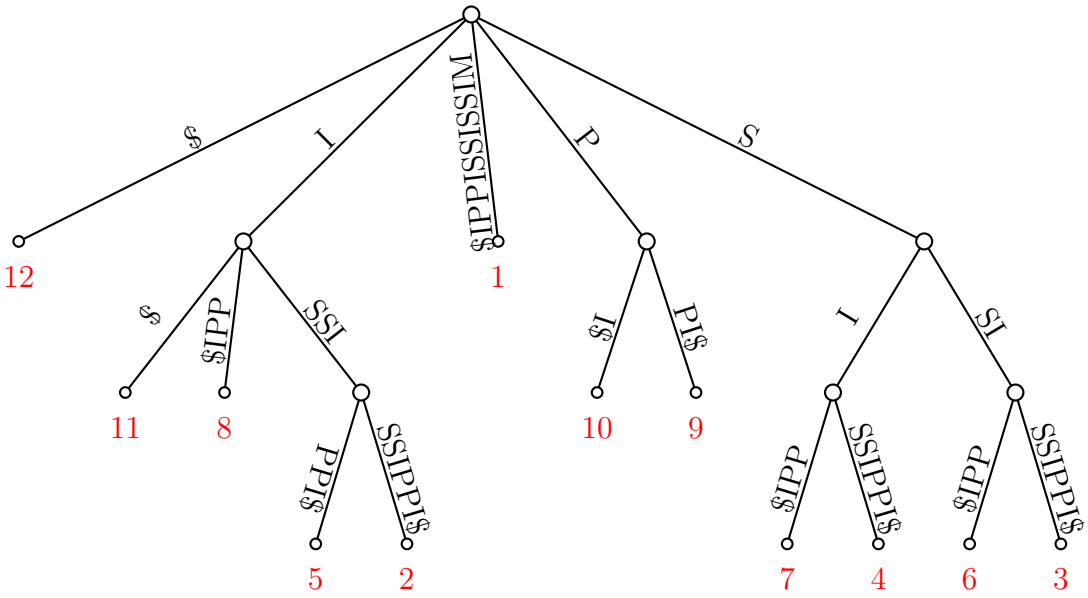


Abbildung 5.41: Beispiel: Suffix-Baum für MISSISSIPPI

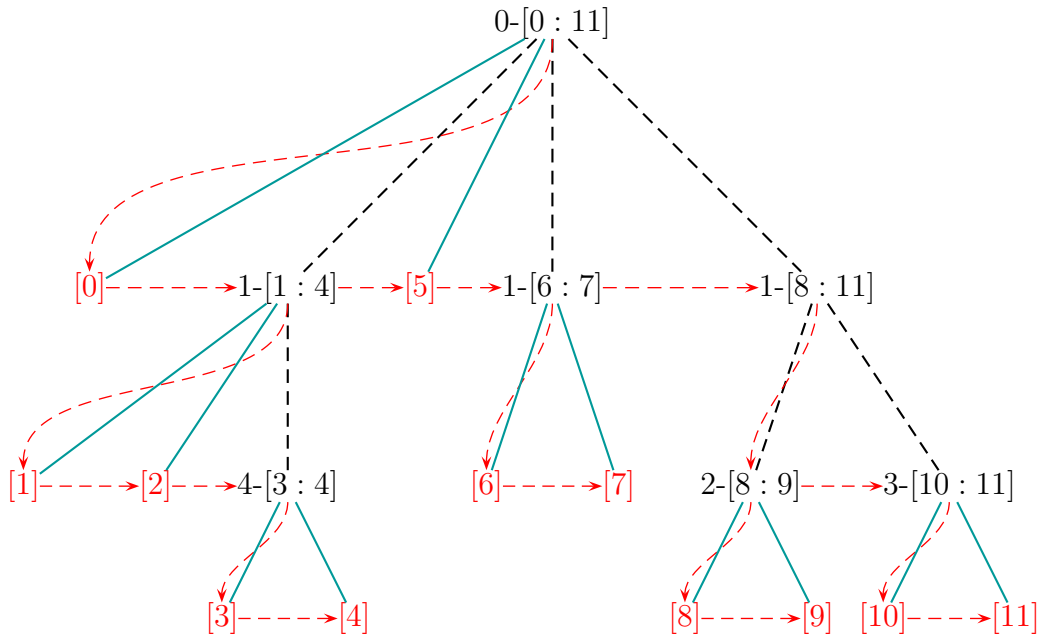


Abbildung 5.42: Beispiel: LCP-Intervall-Baum für MISSISSIPPI

Man sieht sofort die Ähnlichkeit zum entsprechenden Suffix-Baum. In den LCP-Intervallen sind also bereits alle wesentlichen Informationen des Suffix-Baumes (ohne die Blätter) enthalten. In Abbildung 5.42 entsprechen die schwarzen Kanten der Enthaltensein-Relation-der Suffix-Bäume und die türkisen Kanten den Kanten zu den Blätter, die nach Definition ja keine ℓ-Intervalle sind. Die roten Kanten entsprechen den Kanten auf das älteste Kind bzw. auf das nächstjüngere Geschwister gemäß der Definition in der Darstellung eines Suffix-Baumes. Im Folgenden müssen wir nur versuchen diese roten Kanten aus den LCP-Intervallen zu rekonstruieren.

Jetzt definieren wir noch so genannte ℓ-Indizes

Definition 5.52 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A bzw. L das zugehörige Suffix-Array bzw. die zugehörige LCP-Tabelle. Wenn $[i : j]$ ein ℓ-Intervall ist und wenn $L[k] = \ell$ für ein $k \in [i+1 : j]$ ist, dann heißt k ein* ℓ-Index. *Die Menge aller ℓ-Indizes eines ℓ-Intervalls $[i : j]$ wird mit $I_\ell[i : j]$ bezeichnet.*

17.12.24

Damit können wir die Kinder-Inetrvalle eines LCP-Intervalls auffinden.

Lemma 5.53 *Sei $[i : j]$ ein ℓ-Intervall und seien $i_1 < i_2 < \cdots < i_k$ die ℓ-Indizes des ℓ-Intervalls $[i : j]$, dann sind die Kind-Intervalle von $[i : j]$ gerade $[i : i_1 - 1]$, $[i_1 : i_2 - 1]$, $\ldots$, $[i_k : j]$.*

Wir merken hier noch an, dass $[i_p : i_{p+1} - 1]$ für $p \in [1 : k-1]$ durchaus ein Singleton sein kann.

Beweis: Sei $[r : s]$ eines der Intervalle $[i : i_1 - 1]$, $[i_1 : i_2 - 1]$, $\ldots [i_k : j]$.

Ist $[r : s]$ ein Singleton (d.h. $r = s$), dann ist $[r : s]$ nach unserer Konvention ein LCP-Intervall und auch ein Kind-Intervall von $[i : j]$.

Gelte also jetzt $r < s$. Wir definieren zunächst

$$\begin{aligned} m &:= \min\{L[x] \,:\, x \in [r+1 : s]\}, \\ x' &:= \operatorname{argmin}\{L[x] \,:\, x \in [r+1 : s]\}. \end{aligned}$$

Nach Definition des ℓ-Intervalls gilt $m > \ell$ (da $x' \in [i+1 : j] \setminus I_\ell[i : j]$). Nach Lemma 5.49 existiert ein m-Intervall I, dass die Position x' umfasst. Nach Definition der ℓ-Indizes des ℓ-Intervalls $[i : j]$ und nach Wahl von m muss diese Intervall I gerade gleich $[r : s]$ sein (siehe die Konstruktion des LCP-Intervalls im Beweis von Lemma 5.49), also ist $[r : s]$ ebenfalls ein LCP-Intervall.

In beiden Fällen ist $[r : s]$ in $[i : j]$ enthalten. Weiterhin gilt nach Definition der ℓ-Indizes, dass $L[i_1] = L[i_2] = \cdots = L[i_k] = \ell$. Also kann es kein ℓ'-Intervall $[i' : j']$ mit $\ell' > \ell$ geben, für das $[r : s] \subsetneq [i' : j'] \subsetneq [i : j]$ gilt, da es dann ein ℓ-Index $k \in [i' + 1 : j']$ mit $L[k] = \ell < \ell'$ geben würde.

Offensichtlich sind die Intervalle $[i : i_1 - 1]$, $[i_1 : i_2 - 1]$, $\ldots [i_k : j]$ alle möglichen Kind-Intervalle, die $[i : j]$ besitzen kann. ■

5.6.2 Navigation im Extended Suffix-Array

Wie findet man nun das älteste Kind eines ℓ-Intervalls? Sei ℓ-$[i : j]$ das betrachtete ℓ-Intervall. Das erste Kind-Intervall muss die Form ℓ'-$[i : k]$ mit $\ell' > \ell$ haben. Dabei gilt offensichtlich, dass der erste ℓ-Index der minimale Wert in der LCP-Tabelle im Bereich $[i + 1 : j]$ sein muss, also gilt für das Ende des ersten Kind-Intervalls:

$$k = \mathrm{RMQ}_L(i + 1, j) - 1.$$

Hierbei nutzen wir aus, dass die RMQ-Anfrage bei mehrdeutigen Ergebnissen immer den linkesten Index zurückliefert. Beachte, dass $L[k+1] = \ell$ gelten muss. Wie findet man nun den Wert ℓ', sofern $i < k$? Wir suchen einfach im ersten gefundenen Kind-Intervall nach dem ersten ℓ'-Index respektive nach dem zugehörigen LCP-Wert:

$$\ell' = L[\mathrm{RMQ}_L(i + 1, k)].$$

Und wie findet man nun das nächstjüngere Geschwister eines ℓ-Intervalls? Dies lässt sich ebenso leicht mit einer RMQ-Anfrage erledigen, vorausgesetzt, das Elter-Intervall ist bekannt. Sei also ℓ'-$[i' : j' - 1]$ ein Kind von ℓ-$[i : j]$ mit $j' \leq j$. Dann gilt offensichtlich für das benachbarte, nächstjüngere Geschwister ℓ''-$[j' : k]$, dass

$$k = \mathrm{RMQ}_L(j' + 1, j) - 1,$$

sofern $L[k + 1] = \ell$. Wenn $j' = k$ haben wir ein Singleton gefunden und für $j' < k$ ist der Typ des LCP-Intervalls gegeben durch:

$$\ell'' = L[\mathrm{RMQ}_L(j' + 1, k)].$$

Andernfalls (also $L[k + 1] \neq \ell$ und somit $L[k] > \ell$) haben wir das letzte Kind des Elter-Intervalls ℓ-$[i : j]$ gefunden und setzen $k = j$, d.h. das gesuchte Kind-Intervall ist $[j' : j]$, den Typ ℓ'' von $[j' : j]$ ermitteln wir dann wie oben (sofern $[j' : j]$ kein Singleton ist).

Damit können wir alle Algorithmen auf Suffix-Bäumen, die mit einer Tiefen- oder Breitensuche implementiert sind, auf Extended Suffix-Arrays übertragen. Wir werden später noch sehen, welche zusätzlichen Datenstrukturen man braucht, um den Elter eines ℓ-Intervalls auffinden zu können.

Wir können also die Navigation durch die ℓ-Intervalle mit Hilfe einer Datenstruktur für RMQ auf der LCP-Tabelle L effizient erledigen. Wie im Kapitel 4 angemerkt, kann eine solche Datenstruktur für n Werte mit $2n + o(n)$ zusätzlichen Bits auskommen.

5.6.3 Optimale Suche in Extended Suffix-Arrays

Basierend auf dem Algorithmus zum Auffinden aller Kind-Intervalle eines ℓ-Intervalls geben wir in diesem Abschnitt einen Algorithmus zum Suchen nach Teilwörtern basierend auf Suffix-Arrays an. Dieser Algorithmus orientiert sich an der Suche nach Wörtern mithilfe von Suffix-Bäumen und ist in Abbildung 5.43 angegeben.

Die Invariante im Algorithmus ist die folgende: Während der Suche nach einem Wort $s = s_1 \cdots s_m$ gibt die Boolesche Variable `prefix` an, ob ein Präfix der Länge p von s ein Teilwort t enthalten ist oder nicht. Weiterhin ist $[i : j]$ ein ℓ-Intervall mit $\ell \geq p$, dass nur Suffixe von t enthält, die mit dem Präfix $s_1 \cdots s_p$ beginnen. Dabei wird in der while-Schleife jeweils überprüft, ob jeder Suffix im LCP-Intervall $[i : j]$ mit den ersten k Zeichen von s übereinstimmt, wobei bereits bekannt ist, dass es eine Übereinstimmung der ersten p Zeichen gibt. Dies entspricht im Suffix-Baum der Überprüfung, ob das Kantenlabel der eingehenden Kante des zum LCP-Intervall $[i : j]$ korrespondierenden Knotens mit dem entsprechende Teilwort von s übereinstimmt. Im positiven Fall, wird dann das Kind-Intervall ausgewählt, das der ausgehenden Kanten entspricht, deren Kantenlabel mit dem Zeichen s_{p+1} beginnt. Halten wir das Ergebnis im folgenden Satz fest.

Theorem 5.54 *In einem Extended Suffix-Array kann in Zeit $O(|\Sigma| \cdot m)$ festgestellt werden, ob $s \in \Sigma^m$ in $t \in \Sigma^*$ enthalten ist.*

Beweis: Die Korrektheit folgt aus den Zeichenvergleichen. Die Laufzeit folgt aus der Tatsache, dass bis auf die Auswahl der korrekten Kind-Intervalls alle anderen Funktionen in konstanter Zeit durchgeführt werden können und die Laufzeit somit proportional zur Anzahl der Zeichenvergleiche multipliziert mit der Maximalanzahl der Kinder-Intervalle eines LCP-Intervalls ist, also somit höchstens $O(|\Sigma| \cdot m)$ betragen kann. ■

5.6.4 Auffinden des Elters

Überlegen wir uns zunächst, wie man das Elter-Intervall eines ℓ-Intervalls ℓ-$[i : j]$ ermitteln kann. Zunächst überlegt man sich leicht, dass das Elter-Intervall ein ℓ'-Intervall mit $\ell' = \max\{L[i], L[j + 1]\}$ ist. Dies folgt aus der Tatsache, dass jedes

Find (char $s[]$, int m)

```
begin
    interval [i : j] := [0 : n];
    int p := 0;
    bool prefix := TRUE;
    // prefix=TRUE iff s_1 ··· s_p is a prefix of t^{A[k]} for all k ∈ [i : j]

    while (([i : j] ≠ ∅) && (p < m) && (prefix)) do
        if (i < j) then                                  /* [i : j] is a LCP-Interval */
            k := min(L[RMQ_L(i + 1, j)], m);               /* k = min(lcp(i, j), m) */
        else                                             /* [i : j] is a Singleton */
            k := m;
        prefix := (t_{A[i]+p} ··· t_{A[i]+k−1} = s_{p+1} ··· s_k);
        p := k;
        if (prefix && (p < m)) then
            [i : j] := getChildIntervalByChar(i, j, s_{p+1});
    output (prefix)?[i : j] : ∅;
end
```

getChildIntervalByChar (int i, j; char c)

```
begin
    int ℓ := L[RMQ_L(i + 1, j)];
    interval [i' : j'] := getNextChildInterval(i, j, i);
    while ((t_{A[i']+ℓ} ≠ c) && (j' < j)) do
        [i' : j'] := getNextChildInterval(i, j, j' + 1);
    if (t_{A[i']+ℓ} = c) then return [i' : j'];
    else return ∅;
end
```

getNextChildInterval (int i, j, k)

```
begin
    ℓ := L[RMQ_L(i + 1, j)];                          /* [i : j] is an ℓ-interval */
    if (k < j) then
        j' := RMQ_L(k + 1, j);                        /* j' might be an ℓ-index */
    if ((k = j) || (L[j'] > ℓ)) then return [k : j];
    else return [k : j' − 1];
end
```

Abbildung 5.43: Algorithmus: Suche s im Extended Suffix-Array von t

LCP-Intervall mindestens zwei Kind-Intervalle umfassen muss und somit zumindest auf einer Seite des Kind-Intervalls ein ℓ'-Index Stehen muss. Weiterhin definieren wir zwei Felder P (wie previous) und N (wie next), die so genannte Previous- bzw. Next-Tabelle.

Definition 5.55 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A bzw. L das zugehörige Suffix-Array bzw. die zugehörige LCP-Tabelle. Dann sind die* Previous- *und* Next-Tabelle *gegeben durch:*

$$\begin{aligned} P[i] &= \max\{j \in [0:i] \,:\, L[j] < L[i]\}, \\ N[i] &= \min\{j \in [i:n+1] \,:\, L[j] < L[i]\}. \end{aligned}$$

In der Literatur werden diese Previous- bzw. Next-Tabellen oft auch mit *PSV* bzw. *NSV* für *Previous Smaller Values* bzw. *Next Smaller Values* bezeichnet.

Wir halten zunächst das folgende Lemma fest, wobei der Beweis ähnlich zu dem Beweis zur Konstruktion des Vektors L^B bei der Vorverarbeitung zu RMQ ist (siehe Seite 152).

Lemma 5.56 *Sei $t \in \Sigma^n$ und S bzw. L das zugehörige Suffix-Array bzw. die zugehörige LCP-Tabelle. Die Felder P und N können in linearer Zeit erstellt werden.*

Beweis: Übungsaufgabe. ■

Mit Hilfe der Felder P und N kann dann das Elter-Intervall ℓ'-$[i' : j']$ des Kind-Intervalls ℓ-$[i : j]$ leicht wie folgt angegeben werden:

$$\begin{aligned} \ell' &= \max\{L[i], L[j+1]\}, \\ k &= \operatorname{argmax}\{L[i], L[j+1]\}, \\ i' &= P[k], \\ j' &= N[k] - 1. \end{aligned}$$

Der Beweis hierfür orientiert sich im Wesentlichen an Lemma 5.49 bzw. an dessen Beweis. Wie oben schon bemerkt, muss jedes LCP-Intervall mindestens zwei LCP-Intervalle umfassen, somit ist Position i oder $j+1$ ein ℓ'-Index (oder beide). Wenn dort einer kein ℓ'-Index ist, ist der Wert kleiner als der zum ℓ'-Index gehörige LCP-Wert.

19.12.24

5.6.5 Suffix-Links

Es bleibt noch zu überlegen, wie man Suffix-Links in die Extended-Suffix-Arrays integrieren kann.

Definition 5.57 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A das zugehörige Suffix-Array. Für $t^{A[i]} = aw\$$ mit $a \in \Sigma$ und $w \in \Sigma^*$ ist $\sigma(i) = j$ mit $t^{A[j]} = w\$$ der* Suffix-Link an Position i.

Wie man leicht sieht, gilt die folgende Beobachtung.

Beobachtung 5.58 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A bzw. R das zugehörige Suffix-Array bzw. das zugehörige inverse Suffix-Array. Dann gilt $\sigma(i) = R[A[i]+1]$, sofern $A[i] \leq n$*

Wir haben somit nur Suffix-Links für Singletons definiert. Diese zeigen selbst jedoch wieder auf Singletons, namentlich das Intervall $[R[A[i]+1] : R[A[i]+1]]$, sofern $A[i] \neq n+1$ ist. Ist $A[i] = n+1$, so beschreibt das Singleton das Suffix \$, dessen Suffix-Link das Wurzel-Intervall 0-$[0:n]$ ist.

Für LCP-Intervalle können wir das folgende Lemma beweisen.

Lemma 5.59 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A bzw. L das zugehörige Suffix-Array bzw. die zugehörige LCP-Tabelle. Zu jedem ℓ-Intervall $[i:j]$ mit $\ell > 0$ existiert genau ein $(\ell-1)$-Intervall $[r:s]$ mit $r \leq \sigma(i) < \sigma(j) \leq s$.*

Beweis: Da $[i:j]$ ein ℓ-Intervall ist, gilt offensichtlich für alle $k \in [i+1:j]$, dass $L[k] \geq \ell$. Somit gilt auch für alle $k \in [\sigma(i)+1:\sigma(j)]$, dass $L[k] \geq \ell - 1$.

Da $[i:j]$ ein ℓ-Intervall ist, muss es einen ℓ-Index $k \in [i+1:j]$ mit $L[k] = \ell$ geben, d.h. die beiden Suffixe $t^{A[k]}$ und $t^{A[k-1]}$ stimmen genau in der ersten ℓ Positionen überein. Somit stimmen auch die beiden Suffixe $t^{A[k]+1} = t^{A[\sigma(k)]}$ und $t^{A[k-1]+1} = t^{A[\sigma(k-1)]}$ genau in den ersten $\ell - 1$ Positionen überein und es gilt

$$t^{A[\sigma(k-1)]} = t^{A[k-1]+1} < t^{A[k]+1} = t^{A[\sigma(k)]}.$$

Somit muss ein Index $k' \in [\sigma(k-1)+1 : \sigma(k)] \subseteq [\sigma(i)+1 : \sigma(j)]$ den Wert $\ell - 1$ besitzen, d.h. $L[k'] = \ell - 1$.

Nach Lemma 5.49 existiert dann ein $(\ell-1)$-Intervall $[r:s]$, das nach Konstruktion $[\sigma(i):\sigma(j)]$ umfasst. ∎

Mit diesem Existenz-Satz können wir nun die folgende Definition für Suffix-Links von LCP-Intervallen wagen.

Definition 5.60 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A bzw. L das zugehörige Suffix-Array bzw. die zugehörige LCP-Tabelle. Zu jedem ℓ-Intervall $[i : j]$ mit $\ell > 0$ ist das $(\ell-1)$-Intervall $[r : s]$ mit $r \leq \sigma(i) < \sigma(j) \leq s$ das* Suffix-Link-Intervall *von $[i : j]$.*

In Abbildung 5.44 sind die Suffix-Links noch einmal illustriert. Alle nicht eingezeichneten Suffix-Links zeigen auf das 0-Intervall 0-$[0 : n]$.

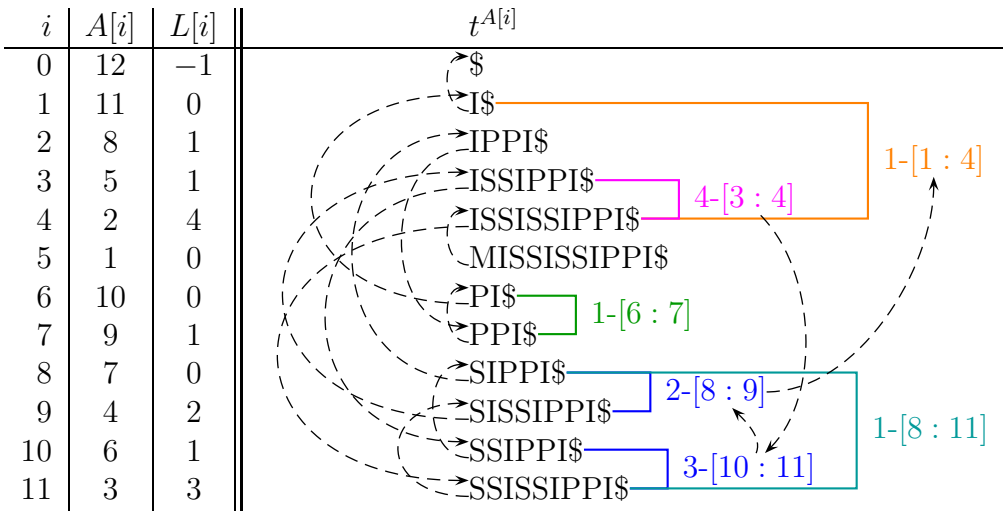


Abbildung 5.44: Beispiel: Suffix-Array, LCP-Tabelle und alle ℓ-Intervalle für das Wort MISSISSIPPI (außer dem 0-Intervall) mitsamt Suffix-Links (außer denen auf das 0-Intervall)

Der Beweis des folgenden Korollars ergibt sich sofort aus einer genauen Inspektion des Beweises von Lemma 5.59.

Korollar 5.61 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A bzw. L das zugehörige Suffix-Array bzw. die zugehörige LCP-Tabelle. Sei weiter $[i : j]$ ein ℓ-Intervall mit $\ell > 0$ und sei $[r : s]$ sein Suffix-Link-Intervall. Dann existiert ein $(\ell-1)$-Index $k \in I_{\ell-1}[r : s]$ mit $k \in [\sigma(i)+1 : \sigma(j)]$.*

Sei nun $[i : j]$ ein ℓ-Intervall und $[r : s]$ das zugehörige Suffix-Link-Intervall. Dann gilt $r \leq \sigma(i) < \sigma(j) \leq s$ und es existiert ein $k \in [\sigma(i)+1 : \sigma(j)]$ mit $L[k] = \ell - 1$. Dieses k lässt sich mithilfe einer Range Minimum Query Anfrage $k = \mathrm{RMQ}_L(\sigma(i)+1, \sigma(j))$

auf dem Feld L in konstanter Zeit ermitteln (nach einer linearen Vorverarbeitungszeit). Dann kann auch das Suffix-Link-Intervall $[r : s]$ wie folgt bestimmt werden:

$$\begin{aligned} k &= \mathrm{RMQ}_L(\sigma(i)+1, \sigma(j)), \\ r &= P[k], \\ s &= N[k]-1. \end{aligned}$$

Theorem 5.62 *Sei E ein Extended Suffix Array, dann kann dieses in linearer Zeit mit linearem Platz so erweitert werden, dass für jedes LCP-Intervall von E dessen Suffix-Link-Intervall in konstanter Zeit aufgefunden werden kann.*

5.6.6 LCA-Queries in Extended Suffix-Arrays

Wir wollen nun für zwei LCP-Intervalle $[i : j]$ und $[i' : j']$ mit $i < i'$ (oder $i = i'$ und $j > j'$) das zum kleinsten gemeinsamen Vorfahren gehörige LCP-Intervall bestimmen. Gilt $j' \leq j$ (also $[i' : j'] \subseteq [i : j]$), dann ist offensichtlich $[i : j]$ ein Vorfahre von $[i' : j']$ und somit ist $[i : j]$ das LCP-Intervall, das den niedrigsten gemeinsamen Vorfahren repräsentiert. Gelte nun also $j < j'$, dann muss auch $j < i'$ gelten, da sich LCP-Intervalle nicht überlappen können.

Für das niedrigste LCP-Intervall ℓ-$[r : s]$ von $[i : j]$ und $[i' : j']$ gilt, dass ein ℓ-Index in der LCP-Tabelle im Intervall $[j + 1 : i']$ stehen muss. Dieser ℓ-Index lässt sich dann wie folgt bestimmen:

$$k = \mathrm{RMQ}_L(j+1, i').$$

Das zugehörige LCP-Intervall ist dann durch $[r : s]$ mit

$$\begin{aligned} r &= P[k], \\ s &= N[k]-1. \end{aligned}$$

gegeben. Der Beweis hierfür orientiert sich wiederum an Lemma 5.49 bzw. an dessen Beweis. Dies gilt auch, wenn die LCP-Intervalle Singletons sind.

5.6.7 Speicherplatzbedarf

In diesem Abschnitt fassen wir noch einmal kurz den Speicherplatzbedarf für die hier vorgestellte Variante des Extended-Suffix-Arrays zusammen. Siehe hierzu auch die Tabelle 5.45. Dort ist die verwendete Datenstruktur genannt und für welche Zwecke diese Datenstruktur verwendet wird. Auf die RMQ-Information zur LCP-Tabelle

Datenstruktur	Feld	Verwendung
Text	t	Immer
Suffix-Array	A	Immer
LCP-Tabelle	L	Immer
RMQ-Info.	Q	(Fast)† Immer
Prev/Next-Array	P, N	Parent, Suffix-Link, LCA
Inv. Suffix-Array	R	Suffix-Link

Abbildung 5.45: Tabelle: Verwendung Extended Suffix-Array (für † siehe Text)

kann verzichtet werden, wenn der Algorithmus nur auf einem Bottom-Up-Traversal des Suffix-Baums besteht. Für die Details verweisen wir auf die Originalarbeit von Kasai, Lee, Arimura, Arikawa und Park.

In der die Tabelle 5.46 ist der für unsere Implementierung benötigte Platz sowie für die momentan beste bekannte Implementierung verzeichnet. Der Platzbedarf ist dabei jeweils in Bit-Komplexität angegeben.

Wir gehen zuerst auf den Platzbedarf für unsere Implementierung etwas genauer ein. Für den gegebenen Text wird die Zeichenreihe t abgespeichert, dabei wird in der Praxis angenommen, dass jedes Zeichen durch ein Byte dargestellt wird (d.h. $|\Sigma| = 256$). Das Suffix-Array (bzw. das inverse Suffix-Array) speichert ja eine Permutation der Werte aus $[1 : n + 1]$ (bzw. $[0 : n]$) und benötigt daher pro Eintrag $\log(n)$ Bits, in der Praxis 4 Bytes ($n < 2^{32}$) bzw. 8 Bytes ($n < 2^{64}$) je nach Architektur. Die LCP-Tabelle beträgt normalerweise (außer bei starken Wiederholungen im Text) nur kleine Werte, so dass Werte aus $[0 : 254]$ in einem Byte in einem Feld gespeichert werden. Sind die Werte größer, wird in der LCP-Tabelle der Wert 255 gespeichert und der korrekte Wert einer Hash-Tabelle außerhalb des Feldes. Sind die LCP größtenteils klein, so bleibt auch die Hash-Tabelle vernachlässigbar klein. Die hier vorgestellte Methode für RMQ benötigt $8n$ Bytes. Für zusätzliche Funktionalitäten wir explizite Elter-Information (die bei normalen Suffix-Baum-Traversierungen gar

Datenstruktur	Platz (gezeigt)	Platz (state-of-the-art)
Text	$n \log(\|\Sigma\|)$	$n \log(\|\Sigma\|)$
Suffix-Array	$n \log(n)$	$n \log(\|\Sigma\|)^\dagger$
LCP-Tabelle	$n \log(n)$	$2n + o(n)$
RMQ-Info.	$2n \log(n)$	$2n + o(n)$
Prev/Next-Array	$2n \log(n)$	$2n + o(n)^\ddagger$
Inv. Suffix-Array	$n \log(n)$	†

Abbildung 5.46: Tabelle: Platzbedarf in Bitkomplexität des Extended Suffix-Array (für †,‡ siehe Text)

nicht benötigt wird) werden noch die Previous- und Next-Tabelle sowie das inverse Suffix-Array benötigt. Speichert man in den Previous- und Next-Tabelle auch nicht die absoluten Indexwerte, sondern relative Offsets, so sind auch diese oft klein und es kann derselbe Trick wie beim Speichern der LCP-Tabelle angewendet werden.

Zum Abschluss gehen wir noch kurz auf den aktuellen Stand der Forschung ein. Für das Suffix-Array selbst gibt es bereits speicherplatzeffiziente Varianten, wie beispielsweise kompakte oder komprimierte Suffix-Arrays (siehe zum Beispiel die Originalarbeit von Sadakane oder das Survey von Navarro und Mäkinen). Diese kommen mit einem Platzbedarf von $n \log(|\Sigma|)$ oder sogar teilweise $H_0 \cdot n + o(n)$ aus, wobei der Zugriff auf ein Element des Suffix-Arrays dann allerdings polylogarithmische Zeit kostet. Dafür wird im selben Platz oft auch der Zugriff auf das inverse Suffix-Array mit der gleichen Zeitschranke ermöglicht. Hierbei ist $H_0 \in [1, \log(|\Sigma|)]$ die (nullte) empirische Entropie des zugrunde liegenden Textes t. Für Details verweisen wir auf das Survey von Navarro und Mäkinen.

Es gibt auch so genannte Self-Indices, wobei dann auf eine explizite Speicherung von t verzichtet werden kann. Sowohl die LCP-Tabelle als auch die RMQ-Datenstruktur kann mit jeweils $2n + o(n)$ Bits implementiert werden. Für die Prev- bzw. Next-Tabelle gibt es Implementierungen, so dass diese zusammen mit der LCP-Tabelle und der RMQ-Datenstruktur in $6n + o(n)$ benötigen. Eine Implementierung der Prev- bzw. Next-Tabelle allein ist in $2.544n + o(n)$ Bits möglich (siehe die Originalarbeit von Fischer). Für die weiteren Details verweisen wir auf die Originalliteratur. Insgesamt ist eine Implementierung eines Textes mit n Zeichen in $4n$ Bytes möglich (für große n).

5.7 Burrows-Wheeler-Transformation und FM-Index

In diesem Abschnitt wollen wir eine andere effiziente Suche in einem Text mittels eines so genannten Volltext-Indexes vorstellen. Die hierin verwendeten Ideen werden auch bei den besonders platzsparenden kompakten oder komprimierten Suffix-Arrays verwendet.

5.7.1 Burrows-Wheeler-Transformation

Zuerst definieren wir formal die Burrows-Wheeler-Transformation.

Definition 5.63 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A das zugehörige Suffix-Array. Dann ist $\hat{t} = t_{A[0]-1} \cdot t_{A[1]-1} \cdots t_{A[n]-1}$ die* Burrows-Wheeler-Transformation *von t, wobei hier $t_0 = t_{n+1} = \$$ angenommen wird.*

i	$A[i]$	$t_{A[i]-1}$	$t^{A[i]}\ldots$
0	12	I	$MISSISSIPPI
1	11	P	I$MISSISSIPP
2	8	S	IPPI$MISSISS
3	5	S	ISSIPPI$MISS
4	2	M	ISSISSIPPI$M
5	1	$	MISSISSIPPI$
6	10	P	PI$MISSISSIP
7	9	I	PPI$MISSISSI
8	7	S	SIPPI$MISSIS
9	4	S	SISSIPPI$MIS
10	6	I	SSIPPI$MISSI
11	3	I	SSISSIPPI$MI

Abbildung 5.47: Beispiel: Burrows-Wheeler-Transformation für MISSISSIPPI

Ein Beispiel für die Burrows-Wheeler-Transformation für unser übliches Beispielwort MISSISSIPPI ist in Abbildung 5.47 angegeben.

Oft wird die Burrows-Wheeler-Transformation ohne das Sonderzeichen $ am Ende definiert. Dann wird in der Regel noch die Position des Buchstabens angegeben, das eigentlich an der letzten Position steht, siehe auch Abbildung 5.48. Hier wäre dann das Ergebnis der Burrows-Wheeler-Transformation: (PSSMIPISSII,5). Im Falle der Verwendung des Sonderzeichens $ ist diese Angabe der Zeile nicht nötig, da diese Position gerade durch das $-Zeichen gekennzeichnet ist. Man beachte dabei, dass die beiden Transformierten nicht gleich sind, aber im Wesentlich dieselben Eigenschaften haben.

i	$A[i]$	$t_{A[i]-1}$	$t^{A[i]}\ldots$
1	11	P	IMISSISSIPP
2	8	S	IPPIMISSISS
3	5	S	ISSIPPIMISS
4	2	M	ISSISSIPPIM
5	1	I	MISSISSIPPI
6	10	P	PIMISSISSIP
7	9	I	PPIMISSISSI
8	7	S	SIPPIMISSIS
9	4	S	SISSIPPIMIS
10	6	I	SSIPPIMISSI
11	3	I	SSISSIPPIMI

Abbildung 5.48: Beispiel: Burrows-Wheeler-Transformation für MISSISSIPPI

Die Reihenfolge des Suffixe ist hier zufälligerweise im Wesentlichen die gleiche, dies muss in der Regel nicht sein, spielt aber für die Anwendungen keine bedeutende Rolle. Wäre beispielsweise M der größte Buchstabe, wäre im Beispiel in Abbildung 5.48 die erste Zeile als vierte Zeile nach hinten gerutscht.

Früher wurde die Burrows-Wheeler-Transformation normalerweise über so genannte zyklische Rotationen definiert.

Definition 5.64 *Sei $t =\in \Sigma^*$ und sei $t = uv$ mit $u, v \in \Sigma^*$. Dann ist vu k-te* zyklische Rechtsrotation*, bzw. k'-te* zyklische Linksrotation*, wobei $k = |u|$ bzw. $k' = |v|$.*

Wenn man jetzt alle solchen zyklischen Rechtsrotationen von $t\$$ bzw. t (also für alle $k \in [0 : n]$ bzw. $k \in [0 : n-1]$) durchführt und diese lexikographisch sortiert, dann ist die Konkatenation der jeweils letzten Zeichen dieser lexikographischen sortierten zyklischen Rechtsrotationen die zugehörige Burrows-Wheeler-Transformation, siehe auch Abbildung 5.47 bzw. Abbildung 5.48.

Die Burrows-Wheeler-Transformation wurde in den 1990ern von Michael Burrows und David Wheeler zur Datenkompression verwendet und wurde in den 1980ern bereits von David Wheeler entdeckt. Daher wurde früher auch in der Implementierung tatsächlich die zyklischen Rechtsrotationen sortiert, meist mithilfe eines modifizierten Quicksorts. Seit dem Bekanntwerden effizienter Algorithmen zur Konstruktion von Suffix-Arrays werden nun diese verwendet. Halten wird das als Ergebnis noch fest.

Lemma 5.65 *Die Burrows-Wheeler-Transformierte $\hat{t}$ für einen Text $t \in \Sigma^n$ kann in Zeit $O(n)$ berechnet werden.*

Da das Suffix-Array von t eine Permutation von $[0 : n]$ darstellt, bilden auch die für die Burrows-Wheeler-Transformation verwendeten Indexpositionen eine Permutation von $[0 : n]$. Somit ist die Burrows-Wheeler-Transformation eine spezielle Permutation der Zeichen in der gegebenen Zeichenreihe $t\$$.

Die Burrows-Wheeler-Transformation hat für normale Texte die schöne Eigenschaft, dass sie gleiche Symbole des Alphabets hintereinander gruppiert (bzw. nur durch einzelne, wenig verschiedene Symbole trennt). Ein Beispiel hierfür ist für die Phrase *Fischers Fritz fischt frische Fische* (der besseren Illustration wegen ohne Rücksicht auf Groß- und Kleinschreibung) in Abbildung 5.49 dargestellt. Dies folgt im Wesentlichen aus der Tatsache, dass in solchen Texten die Wahrscheinlichkeit für das Auftretens eines Zeichens von seinem Kontext abhängt. Im konkreten Beispiel steht vor der Phrase CH immer ein S. Daher steht nach einer lexikographischen Sortierung der zyklischen Rechtsrotationen in den Zeilen, die mit CH beginnen am Ende immer

CHERS_FRITZ_FISCHT_FRISCHE_FISCHE_FIS
CHE_FISCHERS_FRITZ_FISCHT_FRISCHE_FIS
CHE_FISCHE_FISCHERS_FRITZ_FISCHT_FRIS
CHT_FRISCHE_FISCHE_FISCHERS_FRITZ_FIS
ERS_FRITZ_FISCHT_FRISCHE_FISCHE_FISCH
E_FISCHERS_FRITZ_FISCHT_FRISCHE_FISCH
E_FISCHE_FISCHERS_FRITZ_FISCHT_FRISCH
FISCHERS_FRITZ_FISCHT_FRISCHE_FISCHE_
FISCHE_FISCHERS_FRITZ_FISCHT_FRISCHE_
FISCHT_FRISCHE_FISCHE_FISCHERS_FRITZ_
FRISCHE_FISCHE_FISCHERS_FRITZ_FISCHT_
FRITZ_FISCHT_FRISCHE_FISCHE_FISCHERS_
HERS_FRITZ_FISCHT_FRISCHE_FISCHE_FISC
HE_FISCHERS_FRITZ_FISCHT_FRISCHE_FISC
HE_FISCHE_FISCHERS_FRITZ_FISCHT_FRISC
HT_FRISCHE_FISCHE_FISCHERS_FRITZ_FISC
ISCHERS_FRITZ_FISCHT_FRISCHE_FISCHE_F
ISCHE_FISCHERS_FRITZ_FISCHT_FRISCHE_F
ISCHE_FISCHE_FISCHERS_FRITZ_FISCHT_FR
ISCHT_FRISCHE_FISCHE_FISCHERS_FRITZ_F
ITZ_FISCHT_FRISCHE_FISCHE_FISCHERS_FR
RISCHE_FISCHE_FISCHERS_FRITZ_FISCHT_F
RITZ_FISCHT_FRISCHE_FISCHE_FISCHERS_F
RS_FRITZ_FISCHT_FRISCHE_FISCHE_FISCHE
SCHERS_FRITZ_FISCHT_FRISCHE_FISCHE_FI
SCHE_FISCHERS_FRITZ_FISCHT_FRISCHE_FI
SCHE_FISCHE_FISCHERS_FRITZ_FISCHT_FRI
SCHT_FRISCHE_FISCHE_FISCHERS_FRITZ_FI
S_FRITZ_FISCHT_FRISCHE_FISCHE_FISCHER
TZ_FISCHT_FRISCHE_FISCHE_FISCHERS_FRI
T_FRISCHE_FISCHE_FISCHERS_FRITZ_FISCH
Z_FISCHT_FRISCHE_FISCHE_FISCHERS_FRIT
_FISCHERS_FRITZ_FISCHT_FRISCHE_FISCHE
_FISCHE_FISCHERS_FRITZ_FISCHT_FRISCHE
_FISCHT_FRISCHE_FISCHE_FISCHERS_FRITZ
_FRISCHE_FISCHE_FISCHERS_FRITZ_FISCHT
_FRITZ_FISCHT_FRISCHE_FISCHE_FISCHERS

SSSSHHH____CCCCFFRFRFFEIIIIRIHTEEZTS

Abbildung 5.49: Beispiel: Sortierte zyklische Rotationen von *Fischers Fritz fischt frische Fische* und in der letzten Zeile die zugehörige Burrows-Wheeler-Transformation

ein S und bilden somit einen Block. Für Texte, deren Buchstabenverteilung stark kontextabhängig ist, kommen solche Block-Bildungen also oft vor. Dies gilt beispielsweise für natürliche Sprachen (wie Deutsch oder Englisch), aber auch für Quelltexte von Programmen oder Aminosäuresequenzen.

Für die Textkomprimierung kann man dann einfach eine Run-Length-Codierung darauf loslassen, die einen Block von k Vorkommen des Zeichens a durch eine geeignete Codierung von $k * a$ ersetzt, wobei man sich da für die Details einer geschickten Implementierung noch Gedanken machen muss.

In bzip2 wird nach der Burrows-Wheeler-Transformation noch eine Move-to-Front-Codierung eingesetzt, die den transformierten Text geschickt (und umkehrbar) in einen Text über einem neuen Alphabet $\Sigma' = [1 : |\Sigma|]$ übersetzt. Sofern die Burrows-Wheeler-Transformierte hinreichend große Blöcke gleicher Zeichen konstruiert, hat der Text nach der Move-to-Front-Codierung die folgende schöne Eigenschaft, dass kleinere Zeichen (bzgl. $<$) eine deutlich höhere Auftrittswahrscheinlichkeit besitzen. Für solche Texte mit einer solch schiefen Verteilung der Auftrittswahrscheinlichkeiten liefert die Huffman-Codierung eine sehr gute Kompressionsrate (nahe am informationstheoretischen Optimum). Dies nutzt bzip2 in geschickter Weise aus.

Man sollte dabei beachten, dass es für jedes echte Kompressionsverfahren, also eines, das mindestens einen Text verkürzt, es immer mindestens einen Text gibt, der nach der Komprimierung länger als der ursprüngliche Text (über dem gleichen Alphabet) ist. In der Regel sind solche Texte aber nicht in der Teilmenge der Texte enthalten, die man komprimieren will, da beispielsweise nicht alle (bzw. sogar die wenigsten) Zeichenfolgen über einem Alphabet einen sinnvollen deutschen Text oder einen Java-Quelltext ergeben.

5.7.2 Inverse der Burrows-Wheeler-Transformation

Als nächstes wollen wir uns damit beschäftigen, wie man aus der Burrows-Wheeler-Transformation $\hat{t}$ wieder die ursprüngliche Zeichenreihe t rekonstruieren kann. Damit wird auch klar, dass die Burrows-Wheeler-Transformation keine willkürliche Permutation der Zeichen aus t ist. Hierfür definieren wir die Funktion LF, die für eine Array-Position i angibt, an welcher Position in A das um ein Zeichen längere Suffix ab Position $A[i] - 1$ (also $t^{A[i]-1}$) gespeichert ist.

Definition 5.66 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei A das zugehörige Suffix-Array. Dann ist die Funktion* $\mathrm{LF} : [0 : n] \to [0 : n]$ *wie folgt definiert:*

$$\mathrm{LF}(i) = j \quad :\Longleftrightarrow \quad A[j] = A[i] - 1.$$

Hierbei ist bei den Werten von $A[\cdot]$ ggf. 0 mit $n + 1$ zu identifizieren.

i	$A[i]$	LF(i)	$t^{A[i]}\ldots$
0	12	1	\$MISSISSIPPI
1	11	6	I\$MISSISSIPP
2	8	8	IPPI\$MISSISS
3	5	9	ISSIPPI\$MISS
4	2	5	ISSISSIPPI\$M
5	1	0	MISSISSIPPI\$
6	10	7	PI\$MISSISSIP
7	9	2	PPI\$MISSISSI
8	7	10	SIPPI\$MISSIS
9	4	11	SISSIPPI\$MIS
10	6	3	SSIPPI\$MISSI
11	3	4	SSISSIPPI\$MI

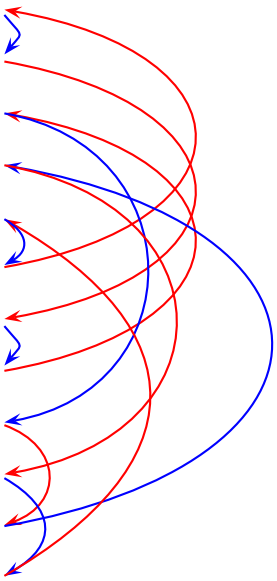


Abbildung 5.50: Beispiel: LF-Werte für die Burrows-Wheeler-Transformation für MISSISSIPPI

Anschaulich liefert für das i-te Zeichen $\hat{t}_i$ in der Burrows-Wheeler-Transformation, das dem Zeichen t_k im ursprünglichen Text t entspricht, die Funktion LF die Position $j = \text{LF}(i)$ in der Burrows-Wheeler-Transformation $\hat{t}$, an dem das Zeichen $\hat{t}_j$ steht, das unmittelbar vor diesem im ursprünglichen Text t steht, also t_{k-1}. In Abbildung 5.50 ist die LF-Funktion durch Pfeile angegeben, die Farben dienen dabei nur der besseren Unterscheidbarkeit der einzelnen Pfeile.

Ein naiver Ansatz zur Berechnung der LF-Funktion ist in Abbildung 5.51 angegeben. Diese ist offensichtlich in linearer Zeit durchführbar, allerdings ist der Platzbedarf relativ hoch (obwohl linear) und insbesondere wird der ursprüngliche Text t benötigt.

computeLF (char $t[]$, int n)

```
begin
    int[n + 1] A := SuffiArray(t, n);
    int[n + 1] R := InverseSuffixArray(t, n);
    for (int j := 0; j ≤ n; j++) do
        int i := R[(A[j] mod (n + 1)) + 1];
        LF[i] := j;
end
```

Abbildung 5.51: Algorithmus: Naiver Ansatz zur Berechnung der LF-Funktion

Wir werden später platzeffizientere Versionen zur Berechnung der LF-Funktionen kennen lernen, die als Eingabe wirklich nur die Burrows-Wheeler-Transformierte benötigen.

Wir zeigen jetzt noch, wie man mithilfe der LF-Funktion aus der Burrows-Wheeler-Transformierten tatsächlich den ursprünglichen Text rekonstruieren kann. Zu Beginn kennen wir ja das letzte Zeichen von $t\$ = t_1 \cdots t_n\$$, wir müssen dazu in $\hat{t}$ nur die Position des Dollarzeichens bestimmen (in der alternativen Fassung wird uns das mit der Position explizit mitgeteilt). Darüber hinaus wissen wir sogar, dass der Suffix, der mit dem Dollar-Zeichen beginnt, im Suffix Array an Position 0 stehen muss. Von daher steht an Position $A[0]-1$ in t das letzte echte Zeichen von t, also gilt $t_n = t_{A[0]-1} = \hat{t}_0$. Beachte hierbei, dass immer $A[0] = n+1$ gilt.

Nehmen wir nun an, dass wir $t_k \cdots t_n$ bereits rekonstruiert hätten und dass wir für die Bestimmung von t_k die Beziehung $t_k = t_{A[i]-1} = \hat{t}_i$ verwendet haben. Hierbei und im Folgenden ist zu beachten, dass die Berechnung der Array-Position modulo $n+1$ zu betrachten ist, d.h. falls $A[i]-1 = 0$ ist, interpretieren wir diesen Wert als $n+1$ (da die Indizierung von t bzw. $t\$$ bei 1 beginnt).

07.01.25

Welches Zeichen in t steht nun unmittelbar vor t_k? Hier war $t_k = t_{A[i]-1} = \hat{t}_i$ das letzte Zeichen der virtuellen Zeichenkette $t_{A[i]} \cdots t_n\$t_1 \cdots t_{A[i]-1}$ der Länge $n+1$, die im Suffix-Array an Position i gespeichert ist. Das gesuchte Zeichen $t_{k-1} = t_{A[i]-2}$ ist dann das letzte Zeichen der virtuellen Zeichenkette $t_{A[i]-1} \cdots t_n\$t_1 \cdots t_{A[i]-2}$ der Länge $n+1$, die im Suffix-Array an Position j gespeichert ist. Diese Position ist aber gerade durch $A[j] = A[i]-1$ bestimmt und lässt sich mit $j = \mathrm{LF}(i)$ ermitteln. Somit ist dann $t_{k-1} = t_{A[j]-1} = \hat{t}_j = \hat{t}_{\mathrm{LF}[i]}$.

Beachte hierbei, dass wir für die Rekonstruktion des ursprünglichen Textes nur die Burrows-Wheeler-Transformierte und die Funktion LF benötigen. Wir werden später sehen, wie wir die Funktion LF platzeffizient ohne Kenntnis des Suffix-Arrays bzw. des inversen Suffix-Arrays und auch ohne den ursprünglichen Text t bestimmen können.

Der Name der Funktion LF kommt von *Last-to-First-Mapping*, da die Funktion gerade die Permutation der Zeichen (inklusive der Positionen in $t\$$) der letzten auf die erste Spalte in den sortierten zyklischen Rotationen von $t\$$ darstellt.

In Abbildung 5.52 ist als Beispiel die Rekonstruktion von t aus der Burrows-Wheeler-Transformierten unseres Beispielwortes MISSISSIPPI angegeben.

Wir können hierbei natürlich mit Hilfe eines Scans zuerst die Position i des Zeichens \$ in der Burrows-Wheeler-Transformierten $\hat{t}$ von t bestimmen. Die Kenntnis dieser Position i ist aber nicht wirklich nötig, da zum einen das letzte Zeichen von $t\$$ immer das Dollarzeichen ist und da zum anderen der Wert von $\mathrm{LF}[i] = 0$ ist, egal welchen Wert i hat. Somit kann man bei der Rekonstruktion de Einfachheit halber immer

i	0	1	2	3	4	5	6	7	8	9	10	11
$\hat{t}_i$	I	P	S	S	M	\$	P	I	S	S	I	I

i	0	1	2	3	4	5	6	7	8	9	10	11
LF(i)	1	6	8	9	5	0	7	2	10	11	3	4

k	i	LF(i)	$\hat{t}_{\mathrm{LF}(i)}$	$t_k \cdots t_n$\$
11	(5)	0	I	I\$
10	0	1	P	PI\$
9	1	6	P	PPI\$
8	6	7	I	IPPI\$
7	7	2	S	SIPPI\$
6	2	8	S	SSIPPI\$
5	8	10	I	ISSIPPI\$
4	10	3	S	SISSIPPI\$
3	3	9	S	SSISSIPPI\$
2	9	11	I	ISSISSIPPI\$
1	11	4	M	MISSISSIPPI\$

Abbildung 5.52: Beispiel: Rekonstruktion aus der Burrows-Wheeler-Transformierten von MISSISSIPPI

gleich mit $i = 0$ beginnen (weswegen die Indexposition 5 zu Beginn des Beispiels in Abbildung 5.52 eingeklammert ist).

5.7.3 Berechnung der LF-Funktion

Zuerst stellen wir einige Definitionen zusammen, bevor wir dann eine effiziente Berechnung der Funktion LF vorstellen.

Definition 5.67 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei $\hat{t} = \hat{t}_0 \cdots \hat{t}_n$ die zugehörige Burrows-Wheeler-Transformierte. Dann ist $C : \Sigma \cup \{\$\} \to [0:n]$ eine Funktion, so dass für $a \in \Sigma \cup \{\$\}$:*

$$C(a) = \#\left\{i \in [0:n] \ : \ \hat{t}_i < a\right\}.$$

Hierbei ist $<$ eine totale Ordnung auf $\Sigma \cup \{\$\}$ und $t_{n+1} = \$$.

Weiter ist $Occ : \Sigma \cup \{\$\} \times [0:n] \to [0:n]$ eine Funktion, so dass für $a \in \Sigma \cup \{\$\}$ und $i \in [0:n]$:

$$Occ(a, i) = \#\left\{k \in [0:i] \ : \ \hat{t}_k = a\right\}.$$

In der Definition von $C(\cdot)$ können wir sowohl $\hat{t}$ als auch $t\$$ verwenden, da ja $\hat{t}$ nur eine Permutation von $t\$$ ist und es nur auf die Häufigkeiten der Buchstaben und nicht auf deren Reihenfolgen ankommt.

Nun stellen wir noch eine Notation vor, damit wir für ein Symbol in einem total geordneten Alphabet auch dessen unmittelbaren Nachfolger in der totalen Ordnung einfach bezeichnen können.

Notation 5.68 *Sei Σ ein Alphabet mit einer totalen Ordnung $<$ und sei $\$ \notin \Sigma$ mit $\$ < a$ für alle $a \in \Sigma$. Für $a \in \Sigma \cup \{\$\}$ bezeichnet $a+1$ den Buchstaben, der in der totalen Ordnung für $\Sigma \cup \{\$\}$ unmittelbar auf a folgt. Falls a das größte Zeichen in Σ ist, dann bezeichnen wir mit $a+1 = \top$ ein virtuelles Zeichen, das größer als alle Zeichen in $\Sigma \cup \{\$\}$ ist.*

Es gilt dann immer für ein Wort t, dass $C(\$) = 0$ und $C(\top) = |t| + 1$ (das Dollarzeichen wird ja immer implizit mitverwendet).

Lemma 5.69 *Sei $t = t_1 \cdots t_n \in \Sigma^*$ und sei $\hat{t} = \hat{t}_0 \cdots \hat{t}_n$ die zugehörige Burrows-Wheeler-Transformierte. Für die zugehörige* LF*-Funktion gilt dann*

$$\mathrm{LF}(i) = C(\hat{t}_i) + Occ(\hat{t}_i, i-1).$$

Beweis: Betrachten man nun für ein i den Eintrag $A[i]$ im Suffix-Array. Es wird also das Suffix $t_{A[i]} \cdots t_n\$$ bzw. die zyklische Rotation $w := t_{A[i]} \cdots t_n \$ t_1 \cdots t_{A[i]-1}$ von $t\$$ betrachtet. Von dieser zyklischen Rotation w ist allerdings nur der erste und letzte Buchstabe bekannt. Der letzte Buchstabe ist nach Definition der Burrows-Wheeler-Transformation gerade $\hat{t}_i$. Die jeweils ersten Buchstaben im Suffix-Array sind sortiert, da das Suffix-Array ja lexikographisch sortiert ist. Da dort genau die ersten Buchstaben aller Suffixe von $t\$$ stehen, sind dies auch genau die Buchstaben, die in $t\$$ bzw. $\hat{t}$ auftreten. Daher kann man diesen Buchstaben mit der Funktion C, die sich dann aus $\hat{t}$ bestimmen lässt, wie folgt ermitteln. Der erste Buchstabe von w ist genau dann $a \in \Sigma \cup \{\$\}$, wenn $C(a) \leq i < C(a+1)$.

Gesucht wird nun nach der LF-Funktion die Position j, so dass $A[j]$ den Suffix $t_{A[i]-1} \cdots t_n\$$ bzw. die zyklische Rotation $w' = t_{A[i]-1} \cdots t_n \$ t_1 \cdots t_{A[i]-2}$ enthält. Wo steht nun w' im Suffix-Array (bzw. der Teil vor dem Dollar-Zeichen). Offensichtlich gilt wieder, dass

$$j \in [C(t_{A[i]-1}), C(t_{A[i]-1}+1) - 1] = [C(\hat{t}_i), C(\hat{t}_i + 1) - 1],$$

wobei nach Definition der Burrows-Wheeler-Transformation $t_{A[i]-1} = \hat{t}_i$ gilt (siehe auch den roten Bereich in Abbildung 5.53).

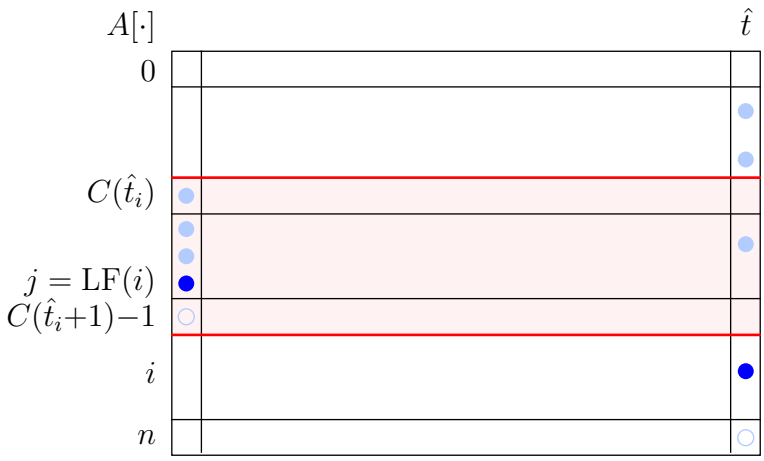


Abbildung 5.53: Skizze: Bestimmung der Position von LF[i]

Wir müssen jetzt nur noch die relative Position von $w' = t_{A[i]-1} \cdots t_n \$ t_1 \cdots t_{A[i]-2}$ (bzw. von dem Suffix bis zum Dollarzeichen) im Interval $[C(\hat{t}_i), C(\hat{t}_i{+}1){-}1]$ bestimmen. Hierfür nutzen wir wieder einmal die bekannte Sortierung für die Einordnung von $w = t_{A[i]} \cdots t_n \$ t_1 \cdots t_{A[i]-1}$ aus. Diese endet mit dem Buchstaben $t_{A[i]-1} = \hat{t}_i$ Für die relative Position sind jetzt nur die Worte interessant, die in der letzten Spalte (also $\hat{t}$) ein $\hat{t}_i$ stehen haben. Mit $Occ(\hat{t}_i, i{-}1)$ bekommen wir die Anzahl dieser Worte, die lexikographisch kleiner sind. Somit sind $C(t_{A[i]-1}){+}Occ(\hat{t}_i, i{-}1)$ Suffixe im Suffix-Array lexikographisch kleiner als das gesuchte. Da die Indizierung des Suffix-Arrays bei 0 beginnt, steht das gesuchte Wort also an Position $j = C(\hat{t}_i) + Occ(\hat{t}_i, i-1)$ (siehe auch den dunkelblauen Punkt in Abbildung 5.53). ■

Wir können also die LF-Funktion aus der Burrows-Wheeler-Transformierten $\hat{t}$ direkt berechnen. Die Funktion $C(\cdot)$ lässt sich mittels einfachen Zählens der Vorkommen der einzelnen Buchstaben und geschickter Addition in Zeit $O(n)$ ermitteln und als Feld speichern. Die Funktion $Occ(\cdot,\cdot)$ kann naiv in einer Tabelle der Größe $O(|\Sigma|n)$ in Zeit $O(|\Sigma|n)$ bestimmt und gespeichert werden. Wir werden später noch sehen, wie man mittels einer platzeffizienteren Datenstruktur die Funktion immer in konstanter Zeit ermitteln kann.

5.7.4 FM-Index

Wir wollen jetzt eine Suche nach einem Suchwort s in einem Text t basierend auf der Burrows-Wheeler-Transformirten $\hat{t}$ beschreiben. Es gilt auch hier wiederum, dass das Suffix-Array (das wir für die Suche nicht benötigen werden), die lexikographische Ordnung der zyklischen Rotation $t_{A[i]} \cdots t_n \$ t_1 \cdots t_{A[i]-1}$ von $t\$$ beschreibt. Auch hier

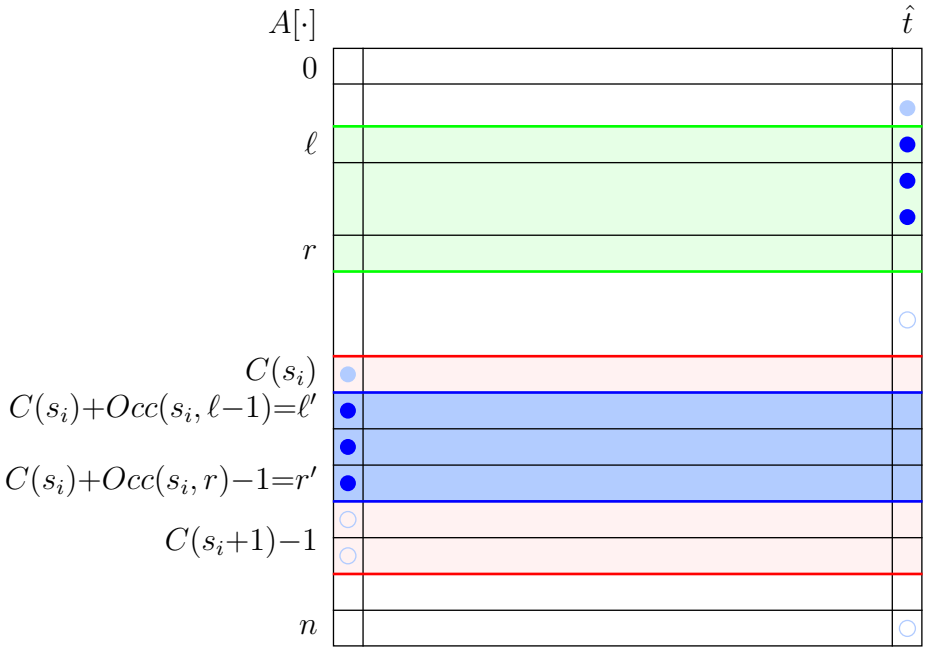


Abbildung 5.54: Skizze: Bestimmung des Intervalls bei der FM-Rückwärtssuche

ist wiederum die Burrows-Wheeler-Transformierte $\hat{t}$ die Konkatenation der letzten Buchstaben der lexikographischen Sortierung aller zyklischen Rotationen von $t\$$.

Wir nehmen im Folgenden an, dass wir das Intervall $[\ell : r]$ kennen, in dem alle Suffixe von t mit $s_{i+1} \cdots s_m$ beginnen. Wir wollen dann das Intervall $[\ell' : r']$ bestimmen, in dem alle Suffixe mir $s_i \cdots s_m$ beginnen. Zu Beginn ist $i = m$ und somit $s_{m+1} \cdots s_m = \varepsilon$ und $[\ell : r] = [0 : n]$.

Als erstes halten wir wiederum (wie im Falle der LF-Funktion) fest, dass

$$[\ell' : r'] \subseteq [C(s_i), C(s_i + 1) - 1]$$

gilt, da ja das Suffix $s_i \cdots s_m$ des Suchworts mit dem Buchstaben s_i beginnt, siehe auch die schematische Darstellung in Abbildung 5.54. Wir wissen weiter, dass alle Suffixe im Intervall $[\ell : r]$ des Suffix-Arrays mit $s_{i+1} \cdots s_m$ beginnen und diese natürlich lexikographisch sortiert sind. Wir müssen wie im vorigen Beweis der LF-Funktion nur wieder wissen, wieviele Suffixe, die mit s_i beginnen, echt vor $s_i \cdot s_{i+1} \cdots s_m$ liegen bzw. mit $s_i \cdot s_{i+1} \cdots s_m$ beginnen. Dies lässt sich wieder mit $Occ(s_i, \ell - 1)$ bzw. $Occ(s_i, r) - 1$ ermitteln. Somit gilt

$$[\ell' : r'] = [C(s_i) + Occ(s_i, \ell - 1) : C(s_i) + Occ(s_i, r) - 1].$$

Dies ist auch noch einmal in Abbildung 5.54 schematisch dargestellt.

```
backwardSearch (char s[], int m)
begin
    int ℓ := 0;
    int r := n;
    // s_{i+1} ··· s_m is always a prefix of t^{A[k]} for all k ∈ [ℓ : r]
    // at the beginning, this is true for i = m and [ℓ : r] = [0 : n]
    for (int i := m; i > 0; i--) do
        ℓ := C(s_i) + Occ(s_i, ℓ − 1);
        r := C(s_i) + Occ(s_i, r) − 1;
        if (ℓ > r) then
            return ∅;
    return [ℓ, r];
end
```

Abbildung 5.55: Algorithmus: Rückwärtssuche im FM-Index

Damit erhalten wir den in Abbildung 5.55 angeben Algorithmus zur Rückwärtssuche in Texten, die tatsächlich nur die Burrows-Wheeler-Transformierte und die Funktionen $C(\cdot)$ und $Occ(\cdot,\cdot)$ benötigt (die beispielsweise als ein Feld C und als eine Tabelle Occ mit Platzbedarf von $O(|\Sigma| \cdot n \log(n))$ Bits abgespeichert werden können, eine platzeffiziente Datenstruktur für Occ werden wir später noch kennenlernen).

Theorem 5.70 *Sei $t \in \Sigma^n$ und sei $\hat{t}$ die zugehörige Burrows-Wheeler-Transformierte. Ein Suchwort $s \in \Sigma^m$ kann in Zeit $O(m)$ mit der Rückwärtssuche in t gefunden werden, wenn die Funktionen $C(\cdot)$ und $Occ(\cdot,\cdot)$ zur Verfügung stehen.*

Beweis: Die Korrektheit haben wir für die Rückwärtssuche bereits gezeigt. Die Laufzeit ist offensichtlich $O(m)$, wenn nur das Intervall der Positionen erwünscht wird. Sollen auch noch die tatsächlichen Positionen von s in t ausgegeben werden, wird darüber hinaus noch das Suffix-Array von t benötigt. ■

Volltext-Indizes, die auf der Burrows-Wheeler-Transformierten und einigen weiteren Hilfsdatenstrukturen aufbauen, nennt man *FM-Index*. Nach der Journal-Version von Ferragina und Manzini steht das FM für *Full-Text Index in Minute Space*, hierbei bedeutet *minute* dem Sinne nach spärlich. Allerdings deuten die beiden Buchstaben auch auf die beiden Autoren hin, die diese Art eines Volltext-Indizes entwickelt haben.

In Abbildung 5.56 ist ein Beispiel für eine solche Rückwärtssuche nach dem Wort ISS im FM-Index für unser Beispielwort MISSISSIPPI angegeben. In der Abbildung ist

i	$A[i]$	$t_{A[i]-1} = \hat{t}_i$	$t^{A[i]} \dots$
0	12	I	\$MISSISSIPPI
1	11	P	I\$MISSISSIPP
2	8	S	IPPI\$MISSISS
3	5	S	ISSIPPI\$MISS
4	2	M	ISSISSIPPI\$M
5	1	\$	MISSISSIPPI\$
6	10	P	PI\$MISSISSIP
7	9	I	PPI\$MISSISSI
8	7	S	SIPPI\$MISSIS
9	4	S	SISSIPPI\$MIS
10	6	I	SSIPPI\$MISSI
11	3	I	SSISSIPPI\$MI

$Occ(\cdot,\cdot)$	\$	I	M	P	S
0	0	1	0	0	0
1	0	1	0	1	0
2	0	1	0	1	1
3	0	1	0	1	2
4	0	1	1	1	2
5	1	1	1	1	2
6	1	1	1	2	2
7	1	2	1	2	2
8	1	2	1	2	3
9	1	2	1	2	4
10	1	3	1	2	4
11	1	4	1	2	4

	\$	I	M	P	S
$C(\cdot)$	0	1	5	6	8

$[\ell : r] = [0 : 11]$
$i = 3,\ s_i = \mathrm{S}$
$\ell' = C(\mathrm{S}) + Occ(\mathrm{S}, -1) = 8 + 0 = 8$
$r' = C(\mathrm{S}) + Occ(\mathrm{S}, 11) - 1 = 8 + 4 - 1 = 11$

$[\ell : r] = [8 : 11]$
$i = 2,\ s_i = \mathrm{S}$
$\ell' = C(\mathrm{S}) + Occ(\mathrm{S}, 7) = 8 + 2 = 10$
$r' = C(\mathrm{S}) + Occ(\mathrm{S}, 11) - 1 = 8 + 4 - 1 = 11$

$[\ell : r] = [10 : 11]$
$i = 1,\ s_i = \mathrm{I}$
$\ell' = C(\mathrm{I}) + Occ(\mathrm{I}, 9) = 1 + 2 = 3$
$r' = C(\mathrm{I}) + Occ(\mathrm{I}, 11) - 1 = 1 + 4 - 1 = 4$

$[\ell : r] = [3 : 4]$.

Abbildung 5.56: Beispiel: Suche nach $s = \mathrm{ISS}$ im FM-Index für MISSISSIPPI

oben noch einmal das Suffix-Array dargestellt, von dem nur die letzte Spalte als Burrows-Wheeler-Transformation tatsächlich bekannt ist. Rechts daneben ist eine Tabelle angegeben, die die zugehörigen Werte von $Occ(\cdot,\cdot)$ beinhaltet. Darunter ist ein Feld mit den zugehörigen Werten $C(\cdot)$ angegeben. Ganz unten ist dann für jeden Buchstaben s_i von $s = s_1 \cdots s_3 = \mathrm{ISS}$ jeweils die Aktualisierung der Intervalle von $[\ell : r]$ nach $[\ell' : r']$ für das Durchlaufen der Werte $i = 3, 2, 1$ angegeben.

5.7.5 Rank-Select-Datenstrukturen

Für die kompakte Darstellung der Funktion $Occ(\cdot)$ benötigen wir zunächst noch eine so genannte Rank-Select-Datenstruktur. Wir definieren zuerst, welche Funktionen eine solche Datenstruktur unterstützen soll.

Definition 5.71 *Sei $B = B_1 \cdots B_n \in \{0,1\}^n$ ein Feld der Länge n mit binären Einträgen, auch kurz* Bit-Feld *oder* Bit-Vektor *genannt.*

Die Rank-Funktion $\text{rank}_x^B(i)$ *auf B liefert die Anzahl der Symbole $x \in \{0,1\}$ im Teilfeld $B_1 \cdots B_i$, also $\text{rank}_x^B(i) := \#\{k \in [1:i] : B[k] = x\}$.*

Die Select-Funktion $\text{select}_x^B(i)$ *auf B liefert die Position j mit $B[j] = x$, so dass $\text{rank}_x^B(j) = i$ gilt, oder $n+1$, wenn kein solches j existiert.*

Wir werden in diesem Abschnitt allerdings nur die Rank-Funktion genauer darstellen, da wir die Select-Funktion nicht benötigen werden. Für binäre Alphabete benötigen wir nur die Rank-Funktion nur für ein Zeichen des Alphabets.

Lemma 5.72 *Sei $B = B_1 \cdots B_n \in \{0,1\}^n$ und $i \in [1:n]$. Dann gilt*

$$\begin{aligned} \text{rank}_0^B(i) &= i - \text{rank}_1^B(i), \\ \text{rank}_1^B(i) &= i - \text{rank}_0^B(i). \end{aligned}$$

Beweis: Übungsaufgabe. ■

Wir werden für die Burrows-Wheeler-Transformation $\hat{t}$ für $t \in \Sigma^n$ für jedes Zeichen in $a \in \Sigma \cup \{\$\}$ ein Bit-Array B_a der Länge $n+1$ erstellen, wobei $B_a[i]$ genau dann den Wert 1 enthält, wenn $\hat{t}_i = a$, und andernfalls den Wert 0. In Abbildung 5.57 sind diese Bit-Arrays für unser Beispielwort MISSISSIPPI angegeben.

$\hat{t}$	I	P	S	S	M	$	P	I	S	S	I	I
$B_\$$	0	0	0	0	0	1	0	0	0	0	0	0
B_I	1	0	0	0	0	0	0	1	0	0	1	1
B_M	0	0	0	0	1	0	0	0	0	0	0	0
B_P	0	1	0	0	0	0	1	0	0	0	0	0
B_S	0	0	1	1	0	0	0	0	1	1	0	0

Abbildung 5.57: Beispiel: Bit-Arrays für die Burrows-Wheeler-Transformation von MISSISSIPPI

Damit erhalten wir sofort, dass $Occ(a, i) = \text{rank}_1^{B_a}(i)$ für das Bit-Array B_a. Wir müssen also nur noch eine effiziente Datenstruktur finden, die konstante Rank-Anfragen an ein Bit-Array beantworten kann.

Wir legen zuerst ein paar Konstanten fest (wobei wir ab jetzt der Einfachheit annehmen, dass das Bit-Array die Länge n hat):

$$\begin{aligned} s' &:= \left\lfloor \frac{\log(n)}{2} \right\rfloor, \\ s &:= (s')^2. \end{aligned}$$

Wir stellen uns jetzt das Feld unterteilt in Superblöcke der Länge s vor, weiter unterteilen wir jeden Superblock in genau s' Blöcke der Länge s', siehe auch Abbildung 5.58.

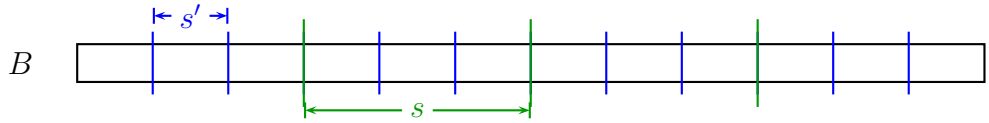


Abbildung 5.58: Skizze: Block-Einteilung der Bit-Arrays

In einer Vorverarbeitung werden wir jetzt die Rank-Anfragen vorberechnen, die genau am Ende eines Superblocks enden (also für Vielfache von s), d.h. wir erstellen eine Tabelle R für $i \in [1 : \lfloor n/s \rfloor]$:

$$R[i] = \text{rank}_1^B(i \cdot s).$$

Die Erstellung dieses Feldes ist mittels eines Scans über B offensichtlich in Zeit $O(n)$ möglich.

Weiterhin bestimmen wir für Anfragen, die an genau einer Blockgrenze enden (also für Vielfache von s'), die Anzahl der Einsen vom Beginn des Superblocks, in dem der Block liegt, bis zum Ende des angegeben Blocks, d.h. wir erstellen eine Tabelle R' für $i \in [1 : \lfloor n/s' \rfloor]$:

$$\begin{aligned} R'[i] &= \text{rank}_1^B(i \cdot s') - \text{rank}_1^B((\lceil i \cdot s'/s \rceil - 1) \cdot s) \\ &= \text{rank}_1^B(i \cdot s') - R[\lceil i \cdot s'/s \rceil - 1]. \end{aligned}$$

Alternativ erhalten wir unter Verwendung von $s = (s')^2$:

$$\begin{aligned} R'[i] &= \text{rank}_1^B(i \cdot s') - \text{rank}_1^B((\lceil i/s' \rceil - 1) \cdot s) \\ &= \text{rank}_1^B(i \cdot s') - R[\lceil i/s' \rceil - 1]. \end{aligned}$$

Die Erstellung dieses Feldes ist mittels eines Scans über B offensichtlich auch wieder in Zeit $O(n)$ möglich.

Weiterhin berechnen wir für alle möglichen Bitfelder $B'' \in \{0,1\}^{s'}$ der Länge s' die Rank-Anfragen für $i \in [1:s']$ in einer Tabelle R'' vor:

$$R''[B'', i] = \text{rank}_1^{B''}(i).$$

Da es $2^{s'}$ verschiedenen Bitfelder B'' der Länge s' gibt und jeweils für alle $i \in [1:s']$ der Rank-Wert zu i ermittelt werden muss, ist dies in Zeit (und Platz in Maschinenwörtern)

$$O\left(2^{s'} \cdot s'\right) = O\left(2^{\log(n)/2} \cdot \log(n)\right) = O\left(\sqrt{n} \cdot \log(n)\right) = o(n)$$

möglich. Dabei kann man mit einem Scan über ein Bitfeld jeden Wert in konstanter Zeit berechnen und die Erzeugung aller Bitfelder der Länge s' kann mit geschickter Implementierung im Mittel in konstanter Zeit pro Bitfeld erledigt werden. Somit hat dieser Schritt sogar einen sublinearen Zeitbedarf.

Damit können wir eine Rank-Anfrage $\text{rank}_1^B(i)$ mit drei Anfragen an die Tabellen R, R und R'' stückweise ermittelt werden, siehe hierzu auch die Skizze in Abbildung 5.59. Die Anzahl der 1en im Teilfeld $B[1:i]$ setzt sich aus den Einsen im Bereich $[1:(\lceil i/s \rceil - 1) \cdot s]$ (dunkelgelb in Abbildung 5.59), im Bereich $[(\lceil i/s \rceil - 1) \cdot s + 1 : (\lceil i/s' \rceil - 1) \cdot s']$ (hellgelb in Abbildung 5.59) und im Bereich $[(\lceil i/s' \rceil - 1) \cdot s' + 1 : i]$ (hellrot in Abbildung 5.59) zusammen.

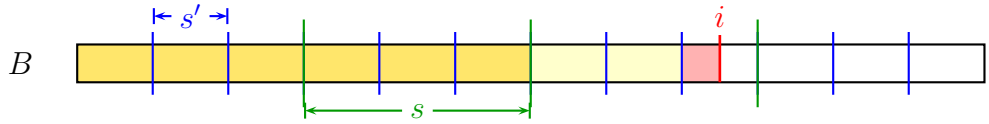


Abbildung 5.59: Skizze: Beantwortung einer Rank-Anfrage

Für die Bestimmung von $\text{rank}_1^B(i)$ gilt dann:

$$\begin{aligned}
\text{rank}_1^B(i) &= \text{rank}_1^B((\lceil i/s \rceil - 1) \cdot s) \\
&\quad + \left[\text{rank}_1^B((\lceil i/s' \rceil - 1) \cdot s') - \text{rank}_1^B((\lceil i/s \rceil - 1) \cdot s)\right] \\
&\quad + \text{rank}_1^{B^i}(i - \lceil i/s' \rceil \cdot s') \\
&= R[\lceil i/s \rceil - 1] + R'[\lceil i/s' \rceil - 1] + R''[B^{(i)}, i - (\lceil i/s' \rceil - 1) \cdot s'],
\end{aligned}$$

wobei $B^{(i)}$ das Bit-Array ist, das in B von Position $(\lceil i/s' \rceil - 1) \cdot s' + 1$ mit $\lceil i/s' \rceil \cdot s'$ steht also $B^{(i)} = B[(\lceil i/s' \rceil - 1) \cdot s' + 1 : \lceil i/s' \rceil \cdot s']$. Man überlege sich, wie sich dieses in konstanter Zeit aus dem Bitfeld B extrahieren lässt (unter Verwendung standardmäßig verfügbarer Bit-Operationen).

09.01.25

Wie viel Platz benötigen wir zusätzlich. Für das Feld R benötigen wir $O(n/s)$ Einträge für Werte aus $[0:n]$, also ist der Platzbedarf in Bits

$$O\left(\frac{n}{s}\cdot\log(n)\right) = O\left(\frac{n}{\log^2(n)}\cdot\log(n)\right) = O\left(\frac{n}{\log(n)}\right) = o(n).$$

Für das Feld R' benötigen wir $O(n/s')$ Einträge für Werte aus $[0:s]$, also ist der Platzbedarf in Bits

$$O\left(\frac{n}{s'}\cdot\log(s)\right) = O\left(\frac{n}{\log(n)}\cdot\log(\log(n))\right) = o(n).$$

Für die Tabelle R'' benötigen wir $O(2^{s'}\cdot s')$ Einträge für Werte aus $[0:s']$, also ist der Platzbedarf in Bits

$$\begin{aligned} O\left(2^{s'}\cdot s'\cdot\log(s')\right) &= O\left(2^{\log(n)/2}\cdot\log(n)\cdot\log(\log(n))\right) \\ &= O\left(\sqrt{n}\cdot\log(n)\cdot\log(\log(n))\right) \\ &= o(n). \end{aligned}$$

Theorem 5.73 *Für ein Bit-Array $B \in \{0,1\}^n$ der Länge n kann eine Hilfsdatenstruktur mit zusätzlichem Platzbedarf $o(n)$ Bits erzeugt werden, so dass alle Rank- und Select-Anfragen in konstanter Zeit beantwortet werden können.*

Den Beweis für die Rank-Anfragen haben wir geliefert, für die Select-Anfragen, die wir hier konkret nicht benötigen verweisen wir auf die Originalliteratur oder auf das Survey von Navarro und Mäkinen.

Damit erhalten wird das folgende Resultat für den FM-Index.

Theorem 5.74 *Sei $t \in \Sigma^n$ und sei $\hat{t}$ die zugehörige Burrows-Wheeler-Transformierte. Ein Suchwort $s \in \Sigma^m$ kann in Zeit $O(m)$ mit der Rückwärtssuche basierend auf $C(\cdot)$ und $Occ(\cdot,\cdot)$ in t gefunden werden, wobei der Platzbedarf (in Bit-Komplexität) für den FM-Index wie folgt abgeschätzt werden kann:*

$$n\cdot\log(|\Sigma|) + |\Sigma|\cdot\log(n) + |\Sigma|\cdot n + o(|\Sigma|\cdot n).$$

Hierbei ist zu beachten, dass für die Bestimmung der Positionen von s in t zusätzlich noch das Suffix-Array benötigt wird.

Beweis: Für die Burrows-Wheeler-Transformierte ist der Platzbedarf naturgemäß $n\log(|\Sigma|)$.

Für das Feld C genügen $|\Sigma| \log(n)$ Bits, da der Wert $C(\$) = 0$ nicht gespeichert werden muss.

Für die Rank-Select-Datenstrukturen für jedes $a \in \Sigma$ beträgt der Platzbedarf insgesamt $|\Sigma| \cdot n + o(|\Sigma| \cdot n)$. Das Bit-Feld für das Dollarzeichen werden wir einfach als eine Zahl bzgl. dessen Indexposition in $\hat{t}$ speichern. ■

5.7.6 Wavelet-Trees

Wir zeigen jetzt noch exemplarisch, wie man mit so genannten *Wavelet-Trees* den Platzbedarf der benötigten Rank-Select-Datenstrukturen von $n \cdot |\Sigma| + o(n \cdot |\Sigma|)$ auf $n \log(|\Sigma|) + o(n \log(|\Sigma|))$ senken kann.

Wir konstruieren zunächst einen fast balancierten binären Baum (die Tiefe von zwei Blättern unterscheidet sich um maximal eins), deren Blätter eindeutig mit den Symbolen aus $\Sigma \cup \{\$\}$ markiert sind. Der Baum hat also genau $|\Sigma|+1 = |\Sigma \cup \{\$\}|$ Blätter. Dazu teilen wir rekursiv an jedem inneren Knoten das zugehörige Alphabet in zwei etwa gleich große Hälften. Sei Γ mit $|\Gamma| > 1$ das dem inneren Knoten zugeordnete Alphabet (zu Beginn ist dann an der Wurzel $\Gamma = \Sigma \cup \{\$\}$). Das linke Kind erhält die Menge $\Gamma' \subseteq \Gamma$ der $\lceil \frac{|\Gamma|}{2} \rceil$ kleinsten Zeichen aus Γ und das rechte Kind erhält die Menge $\Gamma'' \subseteq \Gamma$ der $\lfloor \frac{|\Gamma|}{2} \rfloor$ größten Zeichen aus Γ.

In Abbildung 5.60 ist ein solcher Baum für unser Beispielwort MISSISSIPPI dargestellt. Die aufgeteilten Alphabete sind dabei jeweils an den Kanten zu den Kindern notiert und die Blätter sind mit den eindeutigen Zeichen aus dem Alphabet notiert (hier mit der entsprechenden Vielfachheiten der Vorkommen der Buchstaben des zugrunde liegenden Alphabets im Beispielwort MISSISSIPPI\$).

Nun ordnen wir jedem inneren Knoten zuerst eine Teilfolge des ursprünglichen Wortes $\hat{t}$ zu. Dabei werden alle Buchstaben gelöscht, die in dem dem Knoten zugeordneten Alphabet nicht auftreten, also wird dem Knoten v mit zugeordnetem Alphabet Γ das Wort $\hat{t}|_\Gamma$ zugewiesen. Für jeden inneren Knoten v wird nun für dieses Wort w_v ein Bit-Array erstellt, wobei jeder Buchstabe aus dem Alphabet Γ' (der ersten Hälfte von Γ) eine 0 und jedem Buchstaben aus dem Alphabet Γ' (der zweiten Hälfte von Γ) eine 1 zugeordnet wird.

Für ein Beispiel wollen wir $Occ(I, 9)$ ermitteln (wir nehmen hier der Einfachheit halber an, dass wir im Beispielwort bei 1 beginnen zu indizieren). Zuerst stellen wir an der Wurzel fest, dass I zur ersten Hälfte des Alphabets $\{\$, I, M, P, S\}$ gehört. Wir ermitteln daher $\text{rank}_0^{w_r}(9) = 4$. Das bedeutet, dass es genau 4 kleinere Zeichen (also aus dem ersten Teilalphabet) gibt, die in den ersten 9 Zeichen des Wortes an der Wurzel enthalten sind. Nun betrachten wir das linke Kind v', auch hier ist wieder I

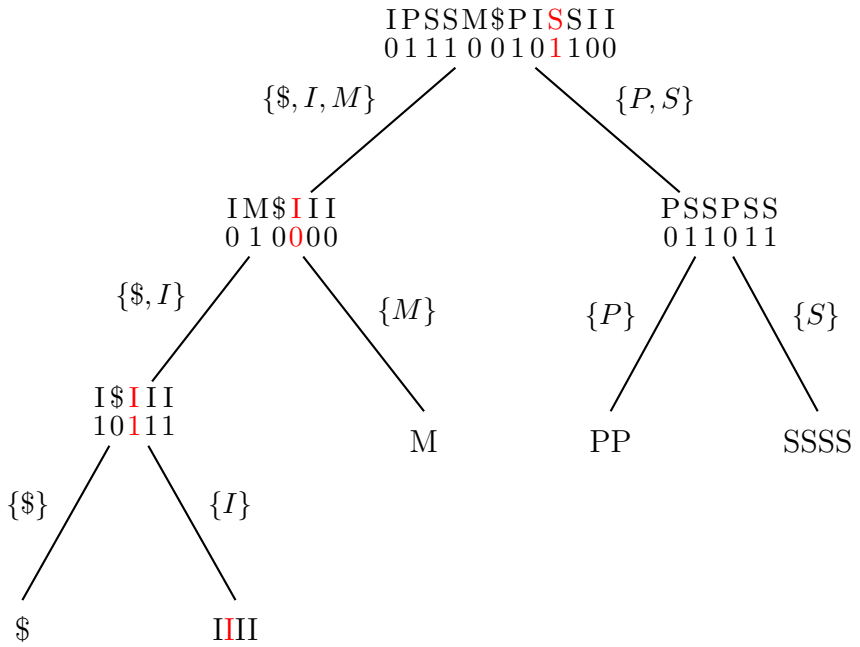


Abbildung 5.60: Beispiel: Wavelet-Tree für MISSISSIPPI

in der kleineren Hälfte des Alphabets enthalten. Hier gibt es jetzt $\text{rank}_0^{w_{v'}}(4) = 3$ kleinere Zeichen in $w_{v'}$ vor der 4-ten Position. Für das weitere linke Kind v'' ist nun I in der größeren Hälfte des Alphabets. Daher bestimmen wir nun $\text{rank}_1^{w_{v''}}(3) = 2$. Wir wissen nun, dass genau zwei größere Zeichen bis zur Position 3 in $w_{v''}$ vorkommen. Da wir nun an einem Blatt (zwangsweise mit dem Symbol I) angekommen sind, wissen wir, dass insgesamt 2 Zeichen I im ursprünglichen Wort IPSSM$PISSII (der Burrows-Wheeler Transformierten von MISSISSIPPI$) bis zur Position 9 vorkommen.

Für dieses Vorgehen müssen wir nur für jeden Knoten die Datenstrukturen für Rank-Select-Anfragen aufbauen. In jedem Level des Baumes sind dabei genau so viele Bit-Zeichen vorhanden, wie das ursprüngliche Wort lang ist, da wir die Buchstaben der ursprünglichen Burrows-Wheeler-Transformierten ja jeweils nur auf verschiedene Knoten eines Levels verteilen (mit Ausnahme der untersten Levels). Damit beträgt der gesamte Platzbedarf $n \log(|\Sigma|) + o(n \log(|\Sigma|))$ Bits.

Damit erhalten wird das folgende Resultat für den FM-Index.

Theorem 5.75 *Sei $t \in \Sigma^n$ und sei $\hat{t}$ die zugehörige Burrows-Wheeler-Transformierte. Ein Suchwort $s \in \Sigma^m$ kann in Zeit $O(\log|\Sigma| \cdot m)$ mit der Rückwärtssuche in t gefunden werden, wobei der Platzbedarf für den FM-Index basierend auf dem Wavelet-Tree für $\hat{t}$ wie folgt in Bit-Komplexität abgeschätzt werden kann:*

$$2n \cdot \log(|\Sigma|) + |\Sigma| \cdot \log(n) + o(n \cdot \log(|\Sigma|)).$$

Hierbei ist zu beachten, dass für die Bestimmung der Positionen von s in t ebenfalls noch das Suffix-Array benötigt wird.

Beweis: Für die Burrows-Wheeler-Transformierte ist der Platzbedarf naturgemäß $n \log(|\Sigma|)$.

Für das Feld C genügt $|\Sigma| \log(n)$, da der Wert $C(\$) = 0$ nicht gespeichert werden muss.

Für die Rank-Select-Datenstrukturen mit Hilfe des Wavelet-Trees sind insgesamt $n \cdot \log(|\Sigma|) + o(n \log(|\Sigma|))$ Bits nötig.

Die Suchzeit erhöht sich um den Faktor $\log(|\Sigma|)$, da für eine Abfrage von $Occ(a, i)$ im Wavelet-Tree das Blatt mit dem Buchstaben a gefunden werden muss und die Tiefe des Wavelet-Trees $\log(|\Sigma|)$ beträgt. ■

Hierbei ist zu beachten, dass wir bei den Wavelet-Trees für die Rank-Anfragen tatsächlich die Anfragen nach 0 und nach 1 benötigen (im Gegensatz zu der einfachen Anwendung im letzten Abschnitt) . Die ist nach Lemma 5.72 allerdings kein weiteres Problem.

Für die Implementierung muss nicht für jeden Knoten ein Bit-Array mit Rank-Anfragen angelegt zu werden, dies kann in einem Bit-Array für alle Knoten eines Level des Wavelet-Trees oder sogar für alle Knoten des Wavelet-Trees angelegt werden. Aufgrund der regelmäßigen Struktur des Wavelet-Trees kann das benötigt Teilfeld leicht bestimmt werden und mithilfe von höchstens zwei Rank-Anfragen kann der richtige Wert auch leicht ermittelt werden.

Zum Schluss verweisen wir noch die *Succinct Data Structure Library (SDSL)*, in der viele succincte Datenstrukturen implementiert sind und leicht verwendet werden können. Dieses ist im GitHub-Repository verfügbar und das Projekt ist im Paper von Gog, Beller, Moffat und Petri genauer beschrieben.

Genome Rearrangements 6

6.1 Modellbildung

In diesem Abschnitt wollen wir uns mit so genannten Genome Rearrangements beschäftigen. Es hat sich herausgestellt, dass die DNA-Sequenzen von Genen nah verwandter Spezies sehr ähnlich sind, aber sich die Anordnung der Gene auf dem Genom doch sehr erheblich unterscheiden kann. Aus diesem Grunde betrachten wir nun die DNA nicht als Folge von Basenpaaren, sondern als lineare Folge von Genen bzw. Bereiche von Genen und wollen die Änderungen auf diesem Niveau studieren.

6.1.1 Rearrangements und zugehörige Mutationen

Wir verstehen also im Folgenden unter einem Genom die lineare Abfolge der kodierenden Gene auf diesem. In Abbildung 6.1 ist ein künstliches Beispiels von zwei Genomen gezeigt, in der die Gene oder auch zusammengefasste Bereiche von Genen mit den Buchstaben von A bis J markiert sind.

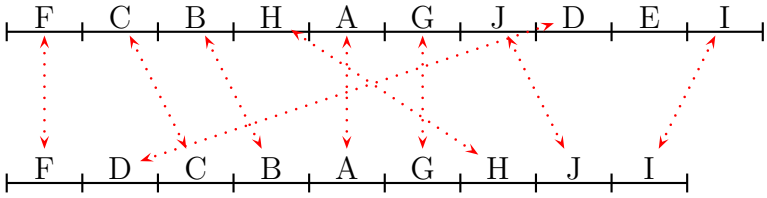


Abbildung 6.1: Beispiel: Genome Rearrangement

Wir wollen uns zunächst einmal überlegen, welche biologischen Mutationen solche Umordnungen der Gene hervorrufen können.

Deletion, Insertion: Wie auf Basen-Ebene können hier jetzt jedoch ganze Gen-Gruppen eliminiert oder eingefügt werden. Siehe hierzu auch Abbildung 6.2.

Abbildung 6.2: Skizze: Deletion bzw. Insertion

Duplikation: Hierbei werden ganze Gruppen von Genen dupliziert. Siehe hierzu auch Abbildung 6.3.

Abbildung 6.3: Skizze: Duplikation

Inversion, Reversion: Die Ordnung der Gene wird in einem Teilbereich umgedreht. Für den biologischen Begriff Inversion verwenden wir den mathematisch korrekten Begriff der Reversion. Siehe hierzu auch Abbildung 6.4.

Abbildung 6.4: Skizze: Reversion

Transposition: Bei einer Transposition werden zwei benachbarte Bereiche von Gengruppen miteinander vertauscht. Siehe hierzu auch Abbildung 6.5.

Abbildung 6.5: Skizze: Transposition

Translokation: Bei einer Translokation werden von jeweils zwei Chromosomen die jeweiligen Enden vertauscht. Siehe hierzu auch Abbildung 6.6.

Abbildung 6.6: Skizze: Translokation

Fission: Hierbei wird ein Chromosom in zwei neue Chromosomen aufgesplittet. Siehe hierzu auch Abbildung 6.7.

Abbildung 6.7: Skizze: Fission

Fusion: Bei der Fusion werden zwei Chromosome zu einem neuen Chromosom verschmolzen. Siehe hierzu auch Abbildung 6.8.

Abbildung 6.8: Skizze: Fusion

Bei Genomen von mehreren Chromosomen kann eine Duplikation nicht nur einen Bereich sondern auch ein ganzes Chromosom duplizieren. In Abbildung 6.9 ist ein Beispiel für ein Genome Rearrangement angegeben. Man sollte sich überlegen, dass es zumindest eine kürzere Folge von Operationen für dieses Beispiel gibt.

F	C	B	H	A	G	J	D	E	I	Deletion
F	C	B	H	A	G	J	D	I		Reversal
F	D	J	G	A	H	B	C	I		Reversal
F	D	C	B	H	A	G	J	I		Transposition
F	D	C	B	A	G	H	J	I		

Abbildung 6.9: Beispiel: Ein Genom Rearrangement

6.1.2 Permutationen

Für die folgenden Untersuchungen betrachten wir folgende Einschränkungen:

Keine Deletionen/Insertionen: Diese sind jedoch sehr leicht festzustellen, da man nur überprüfen muss, welche Gruppen nur auf einem Genom vorhanden sind und welche nicht.

Keine Duplikate/Duplikationen: Diese machen momentan die mathematische Analyse noch sehr schwierig, deshalb konzentrieren wir uns zunächst auf den einfacheren Fall ohne Duplikate.

Mono-chromosomale Genome: Dies ist eine vorläufige Einschränkung, um diesen Fall zuerst behandeln zu können.

Aufgrund des Ausschluss von Duplikaten, können wir die Gen-Gruppen in einem der beiden Genome von 1 bis n in der auftretenden Reihenfolge durchnummerieren. Da auch keine Insertionen und Deletionen erlaubt sind, muss das andere Genom dann eine Permutation der Zahlen von 1 bis n darstellen. Siehe hierzu auch das Beispiel in Abbildung 6.10. Damit kann das eine Genom als Permutation von $[1 : n]$ dargestellt

1	2	3	4	5	6	7	8	9
F	C	B	H	A	G	J	D	I
F	D	C	B	A	G	H	J	I
1	8	2	3	5	6	4	7	9

Abbildung 6.10: Beispiel: Nummerierung der Gen-Gruppen

werden, während das andere Genom dann ohne Beschränkung der Allgemeinheit als Identität beschrieben werden kann.

Definition 6.1 *Sei $n \in \mathbb{N}$, dann heißt*

$$S_n = \{(\pi_1, \ldots, \pi_n) \ : \ \{\pi_1, \ldots, \pi_n\} = [1 : n]\}$$

die symmetrische Gruppe *und seine Elemente* Permutationen. *Die zugehörige Operation $\circ$ ist die Hintereinanderausführung von Permutationen, die durch $(\pi \circ \sigma)_i = \sigma_{\pi_i}$ definiert ist.*

Wie allseits bekannt ist, ist $(S_n, \circ)$ eine Gruppe. Mit $\mathrm{id} = (1, \ldots, n)$ bezeichnen wir die identische Permutation.

Damit stellt sich nun die folgende Frage: Wie viele (und welche) Elementaroperationen (wie Reversionen oder Transpositionen) sind nötig, um eine gegebene Permutation $\pi \in S_n$ in die Identität zu überführen (d.h. zu *sortieren*).

Rearrangement Distance

Eingabe: Eine Permutation $\pi \in S_n$.
Gesucht: Eine kürzeste Folge von Elementaroperationen $\rho_1, \ldots, \rho_k$ vorgegebenen Typs mit $\pi \circ \rho_1 \circ \cdots \circ \rho_k = \mathrm{id}$.

Der minimale Abstand ist ein Maß für den evolutionären Abstand. In Wirklichkeit kann der Abstand auch größer sein, aber der tatsächliche Abstand ist logischerweise nur aus den beiden Anordnungen in den Genomen nicht mehr rekonstruierbar. Es ist auch mehr als plausibel, dass der von der Natur gewählte Weg relativ nah am kürzest möglichen Weg liegt, da sinnlose Umwege eher unwahrscheinlich sind.

Reversionen oder Transpositionen sind Generatoren der symmetrischen Gruppe, d.h. für jedes $\pi \in S_n$ existieren $\rho_1, \ldots, \rho_n$ mit $\pi = \rho_1 \circ \cdots \circ \rho_k$. Auch Reversionen und Transpositionen können wir in Form von Permutationen (im Sinne von Abbildungen) angeben:

$$\mathrm{Rev}(i,j) = \begin{pmatrix} 1 \cdots i{-}1 & i & i{+}1 & \cdots & j{-}1 & j & j{+}1 \cdots n \\ 1 \cdots i{-}1 & j & j{-}1 & \cdots & i{+}1 & i & j{+}1 \cdots n \end{pmatrix}$$

$$\mathrm{Tpos}(i,j,k) = \begin{pmatrix} 1 \cdots i{-}1 & i & \cdots & j & j{+}1 & \cdots & k & k+1 \cdots n \\ 1 \cdots i{-}1 & k{-}j{+}i & \cdots & k & i & \cdots & i{+}k{-}j{-}1 & k+1 \cdots n \end{pmatrix}$$

Man beachte hierbei, dass in dieser Darstellung die Einfärbung der grünen und blauen Bereiches bei einer Transposition nicht notwendigerweise korrekt dargestellt sind und eine korrekte Darstellung auch immer von den konkreten Werten i, j und k abhängt.

Weiterhin muss man auch beachten, dass man Permutationen auf zwei Arten interpretieren kann: zum einen als aktuelle Umordnung der Elemente und zum anderen als

Permutation als Umordnung	Permutation als Abbildung
$\pi = (2,3,1,5,4)$	$\pi' = \left(\begin{smallmatrix}1&2&3&4&5\\3&1&2&5&4\end{smallmatrix}\right)$
Reversion $\rho_1 = (2,4)$	$\rho_1' = \left(\begin{smallmatrix}1&2&3&4&5\\1&4&3&2&5\end{smallmatrix}\right)$
$\pi \cdot \rho_1 = (2,5,1,3,4)$	$\pi' \circ \rho_1' = \left(\begin{smallmatrix}1&2&3&4&5\\3&1&4&5&2\end{smallmatrix}\right)$
Reversion $\rho_2 = (1,4)$	$\rho_2' = \left(\begin{smallmatrix}1&2&3&4&5\\4&3&2&1&5\end{smallmatrix}\right)$
$\pi \cdot \rho_1 \cdot \rho_2 = (3,1,5,2,4)$	$\pi' \circ \rho_1' \circ \rho_2' = \left(\begin{smallmatrix}1&2&3&4&5\\2&4&1&5&3\end{smallmatrix}\right)$
Transposition $\tau = (1,2,4)$	$\tau' = \left(\begin{smallmatrix}1&2&3&4&5\\3&4&1&2&5\end{smallmatrix}\right)$
$\pi \cdot \rho_1 \cdot \rho_2 \cdot \tau = (5,2,3,1,4)$	$\pi' \circ \rho_1' \circ \rho_2' \circ \tau' = \left(\begin{smallmatrix}1&2&3&4&5\\4&2&3&5&1\end{smallmatrix}\right)$

Abbildung 6.11: Beispiel: Rechnen mit Permutationen

Abbildungsvorschrift. Der letztere Fall hat den Vorteil, dass man dann auch die Elementaroperationen als Permutationen auffassen kann und die Verknüpfung der symmetrischen Gruppe genau der Ausführung dieser Elementaroperationen entspricht. Um die beiden Darstellungsarten unterscheiden zu können, wird die Umordnung $(2,3,1,5,4)$, interpretiert als Abbildung, durch $\left(\begin{smallmatrix}1&2&3&4&5\\3&1&2&5&4\end{smallmatrix}\right)$ notiert. In der Abbildung bedeutet dies beispielsweise, dass das erste bzw. vierte Element an die dritte bzw. fünfte Stelle kommt. Genau dort steht auch die 1 bzw. die 4 in der Umordnung. In der Abbildung 6.11 ist ein etwas umfangreicheres Beispiel mit Anwendungen von Reversionen und Transpositionen angegeben.

Man beachte, dass wir im Folgenden der Einfachheit halber oft die Schreibweise für Umordnung auch für Abbildungen verwenden werden.

6.2 Sorting by Reversals

In diesem Abschnitt wollen wir uns nun mit dem Genome Rearrangement beschäftigen, wenn nur Reversionen als Elementaroperationen verwendet werden dürfen.

Sorting by Reversals (Min-SBR)

Eingabe: Eine Permutation $\pi \in S_n$.
Gesucht: Eine kürzeste Folge von Reversionen $\rho_1, \ldots, \rho_k$ mit $\pi \circ \rho_1 \circ \cdots \circ \rho_k = \mathrm{id}$.

6.2.1 Komplexität von Min-SBR

In diesem Abschnitt wollen wir kurz ohne Beweise auf die Komplexität von Min-SBR eingehen. Man kann zeigen, dass Min-SBR ein $\mathcal{NP}$-hartes Optimierungsproblem ist. Viel schlimmer noch, Min-SBR ist sogar $\mathcal{APX}$-hart.

Definition 6.2 *Ein Optimierungsproblem P gehört zur Klasse $\mathcal{APX}$, wenn es einen polynomiell zeitbeschränkten Algorithmus A sowie ein $r \geq 1$ gibt, so dass A eine r-Approximation für P ist.*

Hierbei bedeutet r-Approximation, dass der Algorithmus eine Lösung liefert, die höchstens um den Faktor r von der optimalen Lösung abweicht.

Ein $\mathcal{APX}$-hartes Optimierungsproblem gehört somit zu den schwierigsten Problemen der Klasse, die sich bis auf einen konstanten Faktor approximieren lassen. Eine weitere Folgerung ist, dass $\mathcal{APX}$-harte Probleme kein polynomielles Approximationsschema besitzen können (außer wenn $\mathcal{P} = \mathcal{NP}$).

Ein Optimierungsproblem gehört zur Klasse $\mathcal{PTAS}$ (Polynomial Time Approximation Scheme), wenn es ein Polynom p und einen Algorithmus A gibt, der für jedes $\varepsilon > 0$ das Problem P in Zeit $p(x)$ für die Eingabe x mit Approximationsrate $1 + \varepsilon$ löst. Hierbei kann das Polynom p aber durchaus exponentiell in $1/\varepsilon$ sein (und wird es in der Regel auch sein). Eine Folgerung aus der Komplexitätstheorie für $\mathcal{APX}$-harte Probleme ist, dass sie nicht zur Klasse $\mathcal{PTAS}$ gehören können (außer wenn $\mathcal{P} = \mathcal{NP}$).

Mit Hilfe des PCP-Theorems über probabilistisch verifizierbare Beweise konnten Berman und Karpinski sogar zeigen, dass jeder polynomielle Approximationsalgorithmus für Min-SBR eine Approximationsrate schlechter als 1.008 haben muss, außer wenn $\mathcal{P} = \mathcal{NP}$ gilt.

Theorem 6.3 *Jeder polynomielle Approximationsalgorithmus für Min-SBR hat eine Approximationsgüte von 1.008 oder schlechter, außer wenn $\mathcal{P} = \mathcal{NP}$.*

6.2.2 2-Approximation für Min-SBR

Nach den schlechten Nachrichten aus dem vorherigen Abschnitt wollen wir jetzt noch zeigen, dass es zumindest einen einfachen Algorithmus gibt, der Min-SBR in polynomieller Zeit bis auf einen Faktor 2 löst.

Definition 6.4 *Sei $\pi \in S_n$, dann heißt $(0, \pi_1, \ldots, \pi_n, n+1)$ die* erweiterte Permutation *von $\pi \in S_n$.*

Beachte, dass wir bei dieser Definition von erweiterten Permutationen die Indizes ab 0 (statt 1) laufen lassen. Wir werden im Folgenden nur erweiterte Permutationen betrachten. Dabei dürfen die neuen Werte 0 und $n+1$ nie von einer Elementaroperation verändert werden.

Definition 6.5 *Sei $\pi \in S_n$ eine erweiterte Permutation. Ein* Breakpoint *von π ist ein Paar $(i, i+1)$ mit $i \in [0:n]$, so dass $|\pi_i - \pi_{i+1}| \neq 1$. Dann bezeichnet $b(\pi)$ die Anzahl der Breakpoints von π.*

In $\pi = 0|432|7|1|56|89$ sind die Breakpoints mit blauen Strichen markiert. Offensichtlich gilt $b(\pi) = 5$.

Lemma 6.6 *Sei $\pi \in S_n$ eine erweiterte Permutation, dann sind mindestens $\lceil \frac{b(\pi)}{2} \rceil$ viele Reversionen nötig um π in die Identität zu überführen.*

Beweis: Die Identität ist die einzige erweiterte Permutation mit $b(\pi) = 0$. Offensichtlich kann jede Reversion maximal 2 Breakpoints eliminieren. Daraus folgt die Behauptung. ■

Aus diesem Lemma wird klar, warum wir die erweiterten statt der gegebenen Permutationen betrachten. Ansonsten gäbe es neben der Identität auch noch die Permutation $(n, n-1, \ldots, 2, 1)$ ohne Breakpoints.

14.01.25

Definition 6.7 *Sei $\pi \in S_n$ eine erweiterte Permutation.*

- *Ein* Strip *von π ist eine maximal konsekutive Teilfolge $(\pi_i, \ldots, \pi_j)$ von π ohne einen Breakpoint.*
- *Ein Strip heißt* steigend *bzw.* fallend, *wenn $\pi_i < \cdots < \pi_j$ bzw. $\pi_i > \cdots > \pi_j$ mit $i < j$ gilt.*
- *Einelementige Strips sind* fallend, *außer sie sind (0) oder $(n+1)$. Die möglichen Strips (0) und $(n+1)$ sind* steigend.

In Abbildung 6.12 sind für die Permutation π die steigenden Strips grün und die fallenden rot angezeichnet.

π: 0 | 4 3 2 | 7 | 1 | 5 6 | 8 9

Abbildung 6.12: Beispiel: Strips in einer erweiterten Permutation

Lemma 6.8 *Sei $\pi \in S_n$ eine erweiterte Permutation.*

a) Gehört $k \in [1:n]$ zu einem fallenden Strip und $k-1$ zu einem steigenden Strip, dann existiert eine Reversion ρ, die mindestens einen Breakpoint entfernt, d.h. $b(\pi \circ \rho) < b(\pi)$.

b) Gehört $\ell \in [1:n]$ zu einem fallenden Strip und $\ell+1$ zu einem steigenden Strip, dann existiert eine Reversion ρ, die mindestens einen Breakpoint entfernt, d.h. $b(\pi \circ \rho) < b(\pi)$.

Beweis: zu a): Betrachten wir zuerst den Fall, dass k in der Permutation vor $k-1$ auftaucht, wie in Abbildung 6.13 illustriert. Da k zu einem fallenden Strip gehört, kann $k+1$ nicht unmittelbar hinter k stehen. Da $k-1$ zu einem steigenden Strip gehört, muss $k-2$ unmittelbar vor $k-1$ stehen (man überlegt sich leicht, dass in diesem Fall $k-1 \notin \{0, n+1\}$). Also kann $k-1$ auch nicht unmittelbar hinter k stehen und nach k bzw. $k-1$ ist ein Breakpoint. Wie man leicht sieht, sorgt die dort angezeigte Reversion ρ dafür, dass mindestens ein Breakpoint verschwindet .

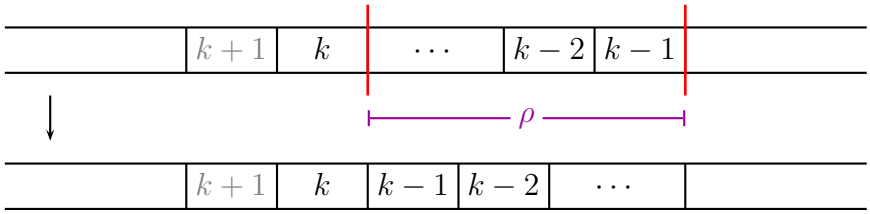


Abbildung 6.13: Skizze: Beweis zu Teil a), 1. Fall

Betrachten wir jetzt den Fall, dass k in der Permutation nach $k-1$ auftaucht, wie in Abbildung 6.14 illustriert. Da k zu einem fallenden und $k-1$ zu einem steigenden Strip gehört, kann $k-1$ nicht unmittelbar vor k in der Permutation stehen. Da $k-1$ zu einem steigenden Strip gehört, muss $k-2$ unmittelbar vor $k-1$ stehen (außer $k-1=0$). Weiterhin ist unmittelbar nach k bzw. $k-1$ ein Breakpoint. Wie man leicht sieht, sorgt die dort angezeigte Reversion ρ dafür, dass mindestens ein Breakpoint verschwindet.

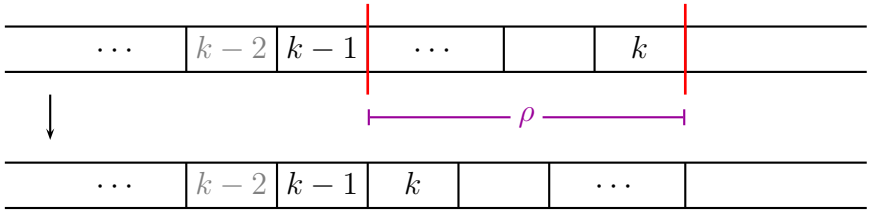


Abbildung 6.14: Skizze: Beweis zu Teil a), 2. Fall

zu b): Betrachten wir zuerst den Fall, dass ℓ in der Permutation vor $\ell+1$ auftaucht, wie in Abbildung 6.15 illustriert. Da ℓ zu einem fallenden und $\ell+1$ zu einem steigenden Strip gehört, kann ℓ nicht unmittelbar vor $\ell+1$ in der Permutation stehen. Da $\ell+1$ zu einem steigenden Strip gehört, muss $\ell+2$ unmittelbar hinter $\ell+1$ in der Permutation stehen (außer $\ell = n$). Weiterhin ist unmittelbar vor ℓ bzw. $\ell+1$ ein Breakpoint. Wie man leicht sieht, sorgt die dort angezeigte Reversion ρ dafür, dass mindestens ein Breakpoint verschwindet.

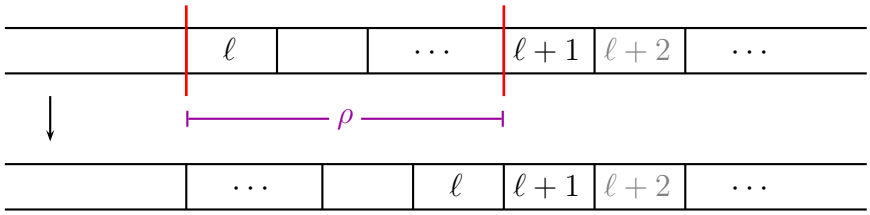


Abbildung 6.15: Skizze: Beweis zu Teil b), 1. Fall

Betrachten wir jetzt den Fall, dass ℓ in der Permutation nach $\ell+1$ auftaucht, wie in Abbildung 6.16 illustriert. Da $\ell+1$ zu einem steigenden Strip gehört, muss $\ell+2$ unmittelbar hinter $\ell+1$ in der Permutation stehen. Weiterhin ist unmittelbar vor ℓ bzw. $\ell+1$ ein Breakpoint. Wie man leicht sieht, sorgt die dort angezeigte Reversion ρ dafür, dass mindestens ein Breakpoint verschwindet. ■

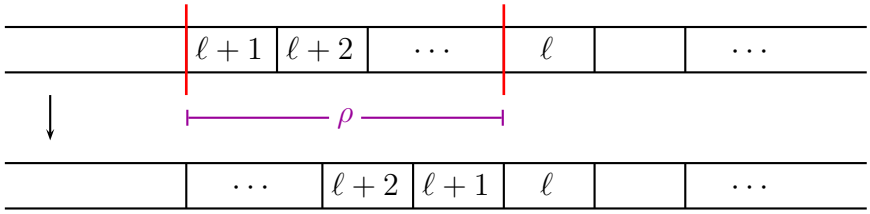


Abbildung 6.16: Skizze: Beweis zu Teil b), 2. Fall

Korollar 6.9 *Sei $\pi \in S_n$ eine erweiterte Permutation mit einem fallenden Strip. Dann existiert eine Reversion ρ mit $b(\pi \circ \rho) < b(\pi)$.*

Beweis: Sei k das kleinste Element, das in einem fallenden Strip liegt. Offensichtlich gilt $k \geq 1$. Dann muss also $k-1 \geq 0$ in einem steigenden Strip liegen. Mit dem vorhergehenden Lemma folgt die Behauptung.

Sei alternativ ℓ das größte Element, das in einem fallenden Strip liegt. Offensichtlich gilt $\ell \leq n$. Dann muss also $\ell+1 \leq n+1$ in einem steigenden Strip liegen. Mit dem vorhergehenden Lemma folgt die Behauptung. ■

Lemma 6.10 *Sei $\pi \in S_n$ eine erweiterte Permutation ohne fallenden Strip, dann ist entweder $\pi = id$ oder es existiert eine Reversion ρ, so dass $\pi \circ \rho$ einen fallenden Strip enthält und $b(\pi \circ \rho) \leq b(\pi)$.*

Beweis: Sei $\pi \neq \text{id}$. Sei $(0, \ldots, i)$ und $(j, \ldots, n+1)$ die beiden steigenden Strips, die die Endsymbole 0 und $n+1$ enthalten. Da $\pi \neq \text{id}$ gilt, muss auch $i+1 < j-1$ gelten. Führt man nun eine Reversion $\rho = (i+1, j-1)$ durch, so kann sich die Anzahl der Breakpoints nicht erhöhen. Nach der Reversion sind also Strips, die nicht $(0, \ldots, i)$ und $(j, \ldots, n+1)$ enthalten, fallende Strips, da sie vor der Reversion steigend waren. Dies ist in Abbildung 6.17 illustriert. ■

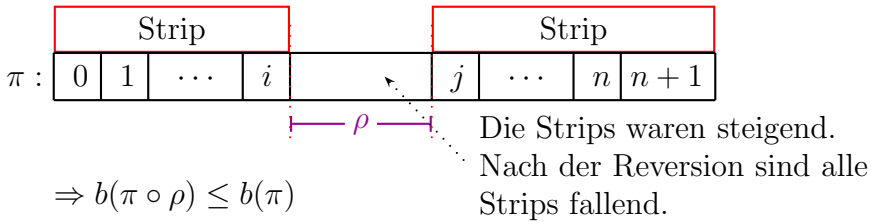


Abbildung 6.17: Skizze: Eine Permutation ohne fallende Strips

Damit erhalten wir eine 4-Approximation, da mit Lemma 6.10 immer ein fallender Strip erzeugt werden kann, ohne die Anzahl der Breakpoints zu erhöhen, und anschließend mit Korollar 6.9 ein Breakpoint entfernt werden kann. Somit kann mit maximal zwei Reversionen ein Breakpoint entfernt werden. Mit der unteren Schranke aus Lemma 6.6 folgt die Behauptung.

Lemma 6.11 *Sei $\pi \in S_n$ eine erweiterte Permutation mit mindestens einem fallendem Strip. Sei k bzw. ℓ das kleinste bzw. größte Element in einem fallenden Strip in π. Sei ρ bzw. σ die Reversion, die $k-1$ und k bzw. ℓ und $\ell+1$ zu benachbarten Elementen macht. Wenn weder $\pi \circ \rho$ noch $\pi \circ \sigma$ fallende Strips enthalten, dann ist $\rho = \sigma$ und $b(\pi \circ \rho) = b(\pi) - 2$.*

Beweis: Nehmen wir zunächst an, dass k vor $k-1$ in der Permutation π auftritt. Da k das kleinste Element in einem fallenden Strip ist, muss sich $k-1$ in einem aufsteigenden Strip befinden, d.h. $k-2$ steht unmittelbar vor $k-1$ in π. Die zugehörige Reversion ρ erzeugt dann aber einen fallenden Strip, siehe Abbildung 6.18.

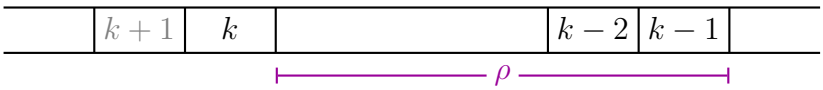


Abbildung 6.18: Skizze: k tritt vor $k-1$ auf

Es gilt also, dass $k-1$ vor k in der Permutation π auftritt, wie in Abbildung 6.19 illustriert.

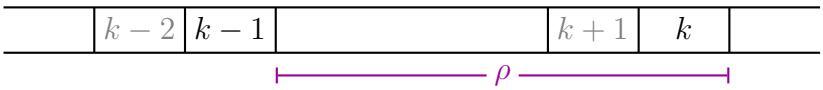


Abbildung 6.19: Skizze: $k-1$ tritt vor k auf

Analog gilt für die Betrachtung von σ, dass ℓ vor $\ell+1$ in der Permutation π auftreten muss. Siehe hierzu auch Abbildung 6.20.

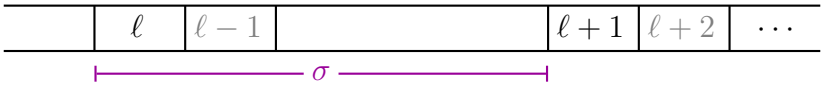


Abbildung 6.20: Skizze: ℓ tritt vor $\ell+1$ auf

Wir beobachten jetzt Folgendes: Die Reversion σ muss den Strip mit k umdrehen, sonst enthält $\pi \circ \sigma$ den fallenden Strip, der k enthält; Die Reversion ρ muss den Strip mit ℓ umdrehen, sonst enthält $\pi \circ \rho$ den fallenden Strip, der ℓ enthält.

Für einen Widerspruchsbeweis nehmen wir jetzt an, dass $\rho \neq \sigma$ gilt. Also existiert ein Strip, der nur von σ oder nur von ρ gedreht wird.

Fall 1: Es existiert ein Strip, der nur von ρ gedreht wird. Ist dieser Strip fallend, dann bleibt dieser Strip in $\pi \circ \sigma$ ebenfalls fallend. Dies ist aber ein Widerspruch zu unserer Annahme. Also muss dieser Strip steigend sein. Dann enthält aber $\pi \circ \rho$ einen fallenden Strip und wir erhalten auch in diesem Fall einen Widerspruch zu den Voraussetzungen.

Fall 2: Es existiert ein Strip, der nur von σ gedreht wird. Die Argumentation verläuft analog zum ersten Fall und wir erhalten auch hier einen Widerspruch.

Also muss $\rho = \sigma$ sein. Diese Situation ist in Abbildung 6.21 illustriert.

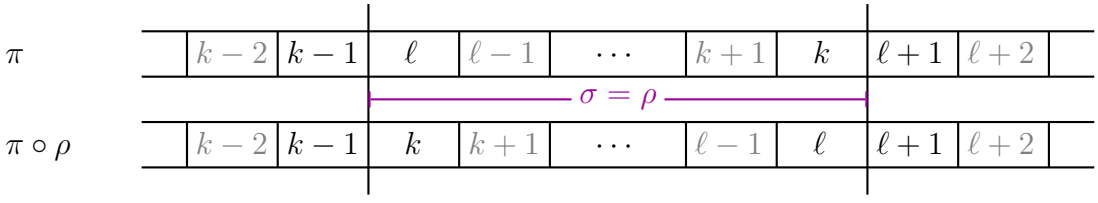


Abbildung 6.21: Skizze: Der Fall $\rho = \sigma$

Wäre $k = \ell$, dann wäre nach k bzw. vor ℓ kein Breakpoint. Also gilt $k \neq \ell$ und es werden offensichtlich genau zwei Breakpoints eliminiert. ■

6.2.3 Algorithmus und Laufzeitanalyse

In diesem Abschnitt entwerfen wir basierend auf den Lemmata des letzten Abschnitts einen Approximationsalgorithmus für das Min-SBR Problem. Der Algorithmus selbst ist in Abbildung 6.22 angegeben.

SBR_Approx (permutation π)

```
begin
    list Reversals = NULL;
    int numReversals = 0;
    while (π ≠ id) do
        if (π has a decreasing strip) then
            let k be the minimal element of any decreasing strip in π;
            let i be the position of k in π;
            let i' be the position of k − 1 in π;
            if (i' < i) then
                let ρ := (i' + 1, i);
            else
                let ρ := (i + 1, i');
            if (π ∘ ρ has no decreasing strip) then
                let ℓ be the maximal element of any decreasing strip in π;
                let j be the position of ℓ in π;
                let j' be the position of ℓ + 1 in π;
                if (j' > j) then
                    let ρ := (j, j' − 1);
                else
                    let ρ := (j', j − 1);
        else
            // π has no decreasing strips
            let (i, i + 1) be the leftmost breakpoint in π;
            let (j − 1, j) be the rightmost breakpoint in π;
            let ρ := (i + 1, j − 1);
        π := π ∘ ρ;
        Reversals = append(Reversals, ρ);
        numReversals++;
end
```

Abbildung 6.22: Algorithmus: 2-Approximation für Min-SBR

Dabei verwenden wir für Permutationen mit fallenden Strips im Wesentlichen die Aussage von Lemma 6.8 bzw. Korollar 6.9. Hierbei verwenden wir allerdings eine sol-

che Reversion, die entweder eine Permutation mit fallenden Strip generiert oder aber mindestens zwei Breakpoints entfernt, siehe Lemma 6.11. Besitzt die Permutation keinen fallenden Strip, dann verwenden wir die Aussage von Lemma 6.10.

Damit erhalten wir das folgende Theorem.

Theorem 6.12 *Der vorherige Algorithmus liefert eine 2-Approximation für Min-SBR mit Laufzeit $O(n^2)$.*

Beweis: Dass der Algorithmus eine Lösung des Min-SBR Problems liefert, ist klar. Wir müssen nur noch die Aussagen über die Approximationsgüte und die Laufzeit beweisen.

Nach Lemma 6.6 gilt: $d(\pi) \geq \lceil b(\pi)/2 \rceil$, wobei $d(\pi)$ die minimale Anzahl von Reversionen ist, die nötig sind, um π zu sortieren. Wir müssen für eine 2-Approximation also zeigen, dass mit jeder Reversion im Schnitt ein Breakpoint entfernt wird.

Für Permutationen mit fallendem Strip ist das klar, da wir dann jeweils eine Reversion anwenden, die mindestens einen Breakpoint entfernt (siehe Lemma 6.8 bzw. Korollar 6.9).

Wie kann jetzt überhaupt eine Permutation ohne fallende Strips entstehen? Entweder durch eine Reversion, die zuvor zwei Breakpoints eliminiert hat (siehe auch Lemma 6.11), oder in der ursprünglich gegebenen Permutation.

Im ersten Fall können wir die zwei eliminierten Breakpoints mit der folgenden Reversion verrechnen, die keinen Breakpoint entfernt (aber auch keine neuen erzeugt) und einen fallenden Strip generiert.

Außerdem überlegen wir uns leicht, dass die letzte Reversion ebenfalls zwei Breakpoints entfernen muss, da es keine erweiterte Permutation mit genau einem Breakpoint geben kann. Auch in diesem Fall können wir die zwei entfernten Breakpoint der allerletzten Reversion mit dem Nichtentfernen eines Breakpoints der allerersten Reversion errechnen, falls die Eingabepermutation keine fallenden Strips enthalten sollte.

Nun zur Laufzeit. Die Anzahl der Durchläufe der while-Schleife: ist $O(b(\pi)) = O(n)$. Weiterhin beträgt die Arbeit pro Schleifendurchlauf $O(n)$. Somit ist die Gesamtlaufzeit $O(b(\pi)n) = O(n^2)$. ■

Eine verbesserte Lösung mit einer Güte von 3/2 mit Laufzeit $O(n^2)$ ist von Christie entworfen worden. Der beste bislang in der Literatur bekannte Approximationsalgorithmus mit polynomieller Laufzeit hat eine Güte von 11/8. Für Details verweisen wir auf die Originalliteratur von Berman, Hannenhalli und Karpinski.

6.3 Eine bessere untere Schranke für Min-SBR

In diesem Abschnitt wollen wir eine bessere untere Schranke für die Anzahl benötigter Reversionen angeben.

6.3.1 Breakpoint-Graphen

Zuerst definieren wir den so genannten Breakpoint-Graphen.

Definition 6.13 *Sei $\pi \in S_n$ eine erweiterte Permutation. Der* Breakpoint-Graph *ist ein Graph $G(\pi) = (V, E)$, der wie folgt definiert ist:*

- $V := [0 : n+1]$;
- $E := R_\pi \cup D_\pi$, *wobei*

$$\begin{aligned} R_\pi &= \{\{\pi_i, \pi_{i+1}\} \,:\, i \in [0:n] \wedge |\pi_i - \pi_{i+1}| \neq 1\}, \\ D_\pi &= \{\{\pi_i, \pi_j\} \,:\, i,j \in [0:n+1] \wedge |\pi_i - \pi_j| = 1 \wedge |i-j| \neq 1\}. \end{aligned}$$

Die Kanten in R_π bzw. in D_π werden auch als Reality-Edges *bzw.* Desire-Edges *bezeichnet.*

Reality-Edges werden im Breakpoint-Graphen als rote Kanten und Desire-Edges als grüne Kanten gezeichnet. Manchmal ist es auch sinnvoll für benachbarte Positionen, die keinen Breakpoint darstellen, im Breakpoint-Graphen die entsprechenden Reality- und Desire-Edges einzuzeichnen. Formal handelt es sich hierbei um einen Multi-Graphen (da es dann natürlich zwischen zwei Knoten mehr als eine Kante geben kann, nämlich maximal zwei, die dann unterschiedlich gefärbt sind).

Definition 6.14 *Sei $\pi \in S_n$ eine erweiterte Permutation. Der* erweiterte Breakpoint-Graph *ist ein Multi-Graph $G'(\pi) = (V', E')$, der wie folgt definiert ist:*

- $V' := [0 : n+1]$;
- $E' := R'_\pi \cup D'_\pi$, *wobei*

$$\begin{aligned} R'_\pi &= \{\{\pi_i, \pi_{i+1}\} \,:\, i \in [0:n]\}, \\ D'_\pi &= \{\{\pi_i, \pi_j\} \,:\, i,j \in [0:n+1] \wedge |\pi_i - \pi_j| = 1\}. \end{aligned}$$

Die Kanten in R'_π bzw. in D'_π werden auch als Reality-Edges *bzw.* Desire-Edges *bezeichnet.*

Wir halten jetzt noch die folgende fundamentale Beziehung zwischen Breakpoint-Graphen und erweiterten Breakpoint-Graphen fest.

Lemma 6.15 *Sei $\pi \in S_n$ eine erweiterte Permutation, dann gilt $G(\pi) \subseteq G'(\pi)$.*

In Abbildung 6.23 ist für die Permutation $\pi = (4, 3, 2, 7, 1, 5, 6, 8)$ der zugehörige (erweiterte) Breakpoint-Graph illustriert.

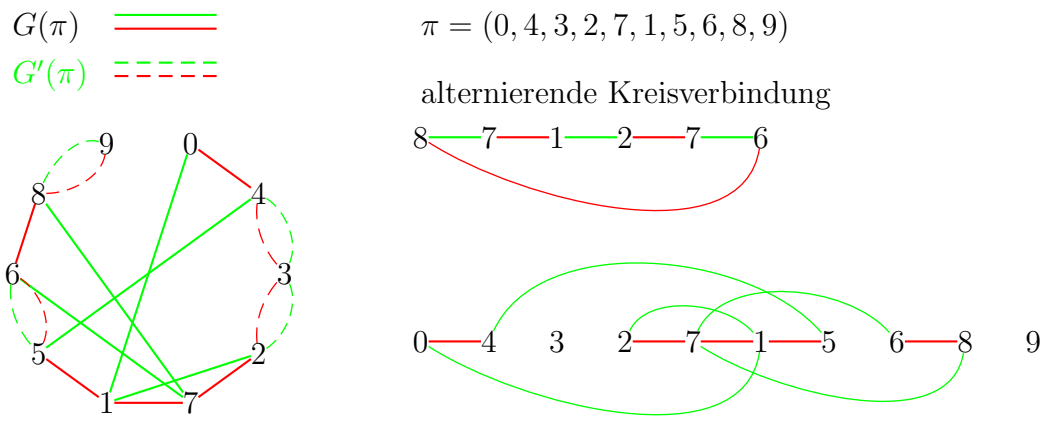


Abbildung 6.23: Beispiel: Ein (erweiterter) Breakpoint-Graph

Wir definieren jetzt noch das Gegenteil von Breakpoints, die so genannten Adjazenzen.

Definition 6.16 *Sei $\pi \in S_n$ eine erweiterte Permutation. Eine* Adjazenz *von π ist ein Paar $(i, i+1)$ mit $i \in [0 : n]$, so dass $|\pi_i - \pi_{i+1}| = 1$. Die Anzahl der Adjazenzen von π wird mit $a(\pi)$ bezeichnet.*

Die folgende Beobachtung entstammt der Tatsache, dass jeder Zwischenraum $(i, i+1)$ für $i \in [0 : n]$ entweder ein Breakpoint oder eine Adjazenz ist.

Beobachtung 6.17 *Es gilt $b(\pi) + a(\pi) = n + 1$ für jede erweiterte Permutation $\pi \in S_n$.*

Wir geben jetzt jetzt noch formal eine Notation für die optimale Anzahle an Reversionen

Notation 6.18 *Sei $\pi \in S_n$ eine erweiterte Permutation, dann bezeichnet $d(\pi)$ die minimale Anzahl an Reversionen, die nötig sind, um π zu sortieren.*

Für die zu entwickelnde untere Schranke werden Zyklenzerlegungen des Breakpoint-Graphen eine wichtige Rolle spielen.

16.01.25

Definition 6.19 *Sei $G = (V, E)$ mit $E = E_1 \cup E_2$ ein 2-kantengefärbter Multi-Graph, in dem jeder Knoten zu gleich vielen Kanten aus E_1 wie zu E_2 inzident ist, d.h. $|\{e \in E_1 : v \in e\}| = |\{e \in E_2 : v \in e\}|$ für jeden Knoten $v \in V$. Eine* alternierende Zyklenzerlegung *$C(G)$ von G ist eine Menge von Kreisen $C_1, .., C_k$ mit:*

- $\bigcup_{i=1}^{k} V(C_i) \subseteq V$,
- $\bigcup_{i=1}^{k} E(C_i) = E$,
- $E(C_i) \cap E(C_j) = \emptyset$ *für alle* $i \neq j \in [1 : k]$,
- *In jedem Kreis $C_i = (v_0^{(i)}, \ldots, v_{2k_i-1}^{(i)})$ der Länge $2k_i$ mit $i \in [1 : k]$ sind die Kanten alternierend gefärbt, d.h. für alle $j \in [0 : k_i - 1]$ gilt*

$$(v_{2j}^{(i)}, v_{2j+1}^{(i)}) \in E_1 \quad \textit{und} \quad (v_{2j+1}^{(i)}, v_{(2j+2) \bmod (2k_i)}^{(i)}) \in E_2.$$

In Abbildung 6.23 sind rechts zwei verschiedene alternierende Kreise dargestellt. Für den Breakpoint-Graphen $G(\pi)$ rechts in Abbildung 6.23 gibt es vier verschiedene alternierende Zyklenzerlegungen, nämlich

$$\{(0, 4, 5, 1), (7, 1, 2, 7, 6, 8)\}$$
$$\{(0, 4, 5, 1), (7, 2, 1, 7, 6, 8)\}$$
$$\{(0, 4, 5, 1, 2, 7, 8, 6, 7, 1)\}$$
$$\{(0, 4, 5, 1, 2, 7, 6, 8, 7, 1)\}$$

Basierend auf einer alternierenden Zyklenzerlegung definieren wir den wichtigen Parameter $c(\pi)$.

Notation 6.20 *Sei $\pi \in S_n$ eine erweiterte Permutation. Dann bezeichnet $c(\pi)$ bzw. $c'(\pi)$ die Anzahl von Kreisen einer maximalen alternierenden Zyklenzerlegung von $G(\pi)$ bzw. $G'(\pi)$.*

Wir wollen im Folgenden zeigen, dass dann $d(\pi) \geq b(\pi) - c(\pi)$ gilt. Da im Breakpoint-Graph jeder Kreis die Länge mindestens 4 hat und somit mindestens zwei Reality-Edges enthält, und die Anzahl Reality-Edges gerade $b(\pi)$ ist, gilt $c(\pi) \leq \frac{b(\pi)}{2}$. Daraus folgt, dass dies eine Verbesserung von Lemma 6.6 darstellt.

6.3.2 Elementare Beobachtungen

Bevor wir zum Beweis der unteren Schranke für das Sortieren mit Reversionen kommen, halten wir einige fundamentale Eigenschaften des Breakpoint-Graphen fest.

Beobachtung 6.21 *Sei $\pi \in S_n$ eine erweiterte Permutation und sei $G(\pi)$ (bzw. $G'(\pi)$) der zugehörige (erweiterte) Breakpoint-Graph. Dann gilt $c'(\pi) = c(\pi) + a(\pi)$.*

Den Beweis dieser Beobachtung überlassen wir dem Leser zur Übung.

Beobachtung 6.22 *Sei $\pi \in S_n$ eine erweiterte Permutation und sei $G'(\pi)$ der zugehörige erweiterte Breakpoint-Graph. Für jeden Knoten $v \in [1:n]$ gilt $\deg(v) = 4$ in $G'(\pi)$. Es gilt sogar, dass jeder Knoten zu genau zwei roten und zu genau zwei grünen Kanten inzident ist.*

Diese Beobachtung folgt unmittelbar aus der Definition des erweiterten Breakpoint-Graphen.

Lemma 6.23 *Sei $\pi \in S_n$ eine erweiterte Permutation und sei $G'(\pi)$ der zugehörige erweiterte Breakpoint-Graph. Eine Reversion ρ erzeugt eine neue alternierende Zyklenzerlegung von $G'(\pi \circ \rho)$, in der sich die Anzahl der Kreise gegenüber $c'(\pi)$ um höchstens eins ändert.*

Beweis: Eine Reversion kann nur die Reality-Edges des erweiterten Breakpoint-Graphen verändern. Durch die Reversion werden zwei Reality-Edges entfernt und zwei neue hinzugefügt. Genauer gesagt, werden zwei entfernte Reality-Edges $(i-1, i)$ und $(j, j+1)$ (mit $i < j$) durch $(i-1, j)$ und $(i, j+1)$ ersetzt. Dabei kann die Anzahl der Kreise sich nur um eins ändern. Dies kann man genauer der Abbildung 6.24 ent-

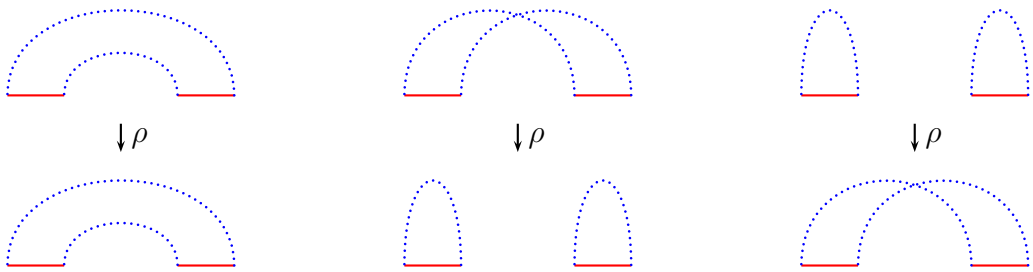


Abbildung 6.24: Skizze: Änderung der alternierenden Zyklenzerlegung durch eine Reversion

nehmen. Dort sind nur die Pfade der Zyklen (in blau) skizziert, die die (beiden roten) Breakpoints bzw. Adjazenzen verbinden, auf denen die Reversion operiert. Diese können nur entweder ein oder zwei Zyklen sein. Beachte dass die blauen alternierenden Pfade hier idealisiert dargestellt sind. Sie können jeweils beliebig lang und Teile der Pfade können sowohl innerhalb als auch außerhalb der beiden Reality-Edges liegen. ■

Nun beweisen wir die fundamentale Eigenschaft, dass jede Reversion die Anzahl von Kreisen in einer maximalen alternierenden Zyklenzerlegung um höchstens eins ändern kann.

Lemma 6.24 *Sei $\pi \in S_n$ eine erweiterte Permutation und sei ρ eine Reversion. Dann gilt $|c'(\pi \circ \rho) - c'(\pi)| \leq 1$.*

Beweis: Betrachten wir eine maximale alternierende Zyklenzerlegung von $G'(\pi)$ mit $c'(\pi)$ Zyklen. Nach dem vorherigen Lemma erzeugt eine Reversion ρ eine alternierende Zyklenzerlegung von $G'(\pi \circ \rho)$, deren Anzahl nur um eins niedriger ist, somit gilt für die maximale Anzahl in $G'(\pi \circ \rho)$ eben $c'(\pi \circ \rho) \geq c'(\pi) - 1$.

Betrachten wir eine maximale alternierende Zyklenzerlegung von $G'(\pi \circ \rho)$ mit $c'(\pi \circ \rho)$ Zyklen. Nach dem vorherigen Lemma erzeugt eine Reversion ρ eine alternierende Zyklenzerlegung von $G'(\pi \circ \rho \circ \rho)$, deren Anzahl der nur um eins niedriger ist, somit gilt für die maximale Anzahl in $G'(\pi \circ \rho)$ eben $c'(\pi \circ \rho \circ \rho) \geq c'(\pi \circ \rho) - 1$.

Da Reversionen zu sich selbst invers sind, d.h. $\rho^{-1} = \rho$, gilt dann auch

$$c'(\pi) = c'(\pi \circ \rho \circ \rho^{-1}) = c'(\pi \circ \rho \circ \rho) \geq c'(\pi \circ \rho) - 1.$$

Insgesamt gilt also $|c'(\pi) - c'(\pi \circ \rho)| \leq 1$ und somit die Behauptung. ■

Theorem 6.25 *Sei $\pi \in S_n$ eine erweiterte Permutation und sei ρ eine Reversion. Es gilt $|b(\pi \circ \rho) - c(\pi \circ \rho) - (b(\pi) - c(\pi))| \leq 1$.*

Beweis: Im erweiterten Breakpoint-Graphen $G'(\pi)$ gilt nach dem letzten Lemma

$$|c'(\pi \circ \rho) - c'(\pi)| \leq 1,$$

wobei $c'(\pi)$ die Anzahl von Zyklen einer maximalen Zyklenzerlegung des erweiterten Breakpoint-Graphen $G'(\pi)$ ist. Mithilfe der vorherigen Beobachtung 6.21, dass

$c'(\pi) = c(\pi) + a(\pi)$, wobei $c(\pi)$ die Anzahl der Zyklen einer maximalen Zyklenzerlegung des normalen Breakpoint-Graphen $G(\pi)$ ist, gilt dann

$$|c(\pi \circ \rho) + a(\pi \circ \rho) - c(\pi) - a(\pi)| \leq 1.$$

Mit der vorherigen Beobachtung 6.17, dass $a(\pi) = n + 1 - b(\pi)$, gilt weiter

$$|c(\pi \circ \rho) + (n+1) - b(\pi \circ \rho) - c(\pi) - (n+1) + b(\pi)| \leq 1.$$

Daher gilt auch

$$|b(\pi \circ \rho) - c(\pi \circ \rho) - (b(\pi) - c(\pi))| \leq 1$$

und somit die Behauptung. ■

6.3.3 Die untere Schranke

Jetzt können wir die angekündigte untere Schranke beweisen.

Theorem 6.26 *Für jede erweiterte Permutation $\pi \in S_n$ gilt $d(\pi) \geq b(\pi) - c(\pi)$.*

Beweis: Sei $\rho_k, \ldots, \rho_1$ eine kürzeste Folge von Reversionen, die π sortiert, und sei $\pi_k = \pi$, $\pi_{i-1} = \pi_i \circ \rho_i$ für $i \in [1:k]$ sowie $\pi_0 = \mathrm{id}$. Somit ist $d(\pi) = k$. Es gilt, dass dann auch $\rho_i, \ldots, \rho_1$ für alle $i \in [1:k]$ eine kürzeste Folge von Reversionen ist, die π_i sortiert (also $d(\pi_i) = i$). Nach Definition von d und π_i gilt zunächst $d(\pi_i) \leq d(\pi_{i-1}) + 1$. Wäre $d(\pi_i) < d(\pi_{i-1}) + 1$, dann gäbe es eine kürzere Folge als $\rho_i, \ldots, \rho_1$, die π sortiert, was nicht sein kann. Also gilt $d(\pi_i) = d(\pi_{i-1}) + 1$

Dann gilt aufgrund von Theorem 6.25 ($1 \geq c(\pi_i \circ \rho_i) - b(\pi_i \circ \rho_i) - (c(\pi_i) - b(\pi_i))$) für $i \in [1:k]$:

$$d(\pi_i) = d(\pi_{i-1}) + 1 \geq d(\pi_{i-1}) + c(\pi_i \circ \rho_i) - b(\pi_i \circ \rho_i) - (c(\pi_i) - b(\pi_i))$$

Daraus folgt sofort:

$$\begin{aligned} d(\pi_i) - (b(\pi_i) - c(\pi_i)) &\geq d(\pi_{i-1}) - (b(\pi_{i-1}) - c(\pi_{i-1})) \\ &\geq d(\pi_{i-2}) - (b(\pi_{i-2}) - c(\pi_{i-2})) \\ &\vdots \\ &\geq d(\pi_0) - (b(\pi_0) - c(\pi_0)) \\ &= 0. \end{aligned}$$

Mit $i = k$ folgt sofort

$$d(\pi) - (b(\pi) - c(\pi)) = d(\pi_k) - (b(\pi_k) - c(\pi_k)) \geq 0.$$

Also gilt $d(\pi) \geq b(\pi) - c(\pi)$. ■

Korollar 6.27 *Für jede erweiterte Permutation $\pi \in S_n$ gilt $d(\pi) \geq n + 1 - c'(\pi)$.*

Beweis: Nach dem vorherigen Satz gilt $d(\pi) \geq b(\pi) - c(\pi)$. Nach Beobachtung 6.17 gilt $b(\pi) = n + 1 - a(\pi)$ und nach Beobachtung 6.21 gilt $c'(\pi) = c(\pi) + a(\pi)$. Also folgt

$$\begin{aligned} d(\pi) &\geq b(\pi) - c(\pi) \\ &= (n + 1 - a(\pi)) - (c'(\pi) - a(\pi)) \\ &= n + 1 - c'(\pi) \end{aligned}$$

und somit die Behauptung. ■

Wir halten zum Abschluss dieses Abschnittes noch folgende Bemerkungen fest.

- Bereits die Bestimmung einer maximalen Zyklenzerlegung von $G(\pi)$ oder $G'(\pi)$ ist $\mathcal{NP}$-hart.
- Bei Verwendung von Präfix-Reversionen statt normaler Reversionen heißt das Problem auch *Sorting by Prefix Reversals* bzw. *Pancake Flipping* und wurde zuerst 1979 von Gates und Papadimitriou untersucht. Allerdings wurde dabei nicht der Abstand für ein Paar von Permutationen bestimmt, sondern der maximale mögliche Abstand für alle Paare von Permutationen (auch Durchmesser genannt). Hierfür gilt

  $$d(S_n) := \max\{d(\pi) \ : \ \pi \in S_n\} \leq \frac{5}{3}(n + 1).$$

  Interessanterweise wurde Bill Gates später für andere Entwicklungen bekannt.
- Eine erweiterte Fassung des Pancake Flipping ist da so genannte *Burnt Pancake Flipping* (auch *Sorting by Oriented Prefix Reversals*). Hierbei sind die Pfannkuchen von einer Seite angebrannt, die dann der Optik wegen besser auf der Unterseite zu liegen kommt (so dass man sie eben nicht sehen kann). Dieses Problem wurde Anfang der 90er von Cohen und Blum genauer untersucht. Interessanterweise ist David S. Cohen später als Mitentwickler von *Futurama* unter dem Namen David X. Cohen bekannt geworden.
- Der angegebene Approximationsalgorithmus verwendet Reversionen, die auf zwei Breakpoints operieren. Es gibt allerdings Permutationen, bei denen es besser ist, solche Reversionen zu verwenden, die nicht nur auf Breakpoints operieren. Man kann jedoch zeigen, dass nur das Aufbrechen kurzer Strips hilfreich sein kann.

- Auch allgemeinere Fragestellung wurden bereits untersucht. Gegeben sei eine Menge von Generatoren $\rho_1, \ldots, \rho_k$ von S_n und eine Permutation $\pi \in S_n$. Gesucht wird die kürzeste Darstellung für π mithilfe der Generatoren der symmetrischen Gruppe. Dieses Problem ist selbst für $k = 2$ bereits $\mathcal{PSPACE}$-vollständig.

6.4 Sorting by Oriented Reversals

In diesem Abschnitt wollen wir nun so genannte orientierte Permutationen betrachten. Bei einer Reversion wird ja nicht nur eine Gengruppe bzw. ein Gen umgedreht sondern auch die natürliche Ordnung innerhalb dieser Gengruppe bzw. des Gens, das durch die Leserichtung auf der DNA gegeben ist. Von daher erscheint es sinnvoll, sich nicht nur die Gengruppe zu merken, sondern auch deren Leserichtung bzw. auf welchem Strang der DNA sie kodiert sind. Bei einer Reversion werden sie ja dann auf den entsprechenden Komplementärstrang verlagert.

6.4.1 Orientierte Permutationen

Formalisieren wir zunächst solche orientierten Permutationen.

Definition 6.28 *Sei $\bar{\pi} = (\bar{\pi}_1, \ldots, \bar{\pi}_n) \in [-n : n]^n$ eine Folge ganzer Zahlen mit $|\bar{\pi}| := (|\bar{\pi}_1|, \ldots, |\bar{\pi}_n|) \in S_n$. Dann ist $\bar{\pi}$ eine* orientierte Permutation*. Die Menge aller orientierten Permutationen bezeichnen wir mit $\bar{S}_n$. $\bar{S}_n$ wird auch als* orientierte symmetrische Gruppe *bezeichnet.*

Man kann die Verknüpfung von zwei orientierten Permutationen wie folgt definieren $(\bar{\pi} \circ \bar{\sigma})(i) = \operatorname{sgn}(\bar{\pi}(i)) \cdot \bar{\sigma}(|\bar{\pi}|(i))$, wobei $|\bar{\pi}|(i) = |\bar{\pi}(i)|$ ist. Der Leser möge sich überlegen, dass mit dieser Verknüpfung von Abbildungen auf $\bar{S}_n$ eine Gruppe definiert wird.

Definition 6.29 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation, dann operiert eine* orientierte Reversion *(oder auch kurz eine* Reversion*) $\bar{\rho} = (i, j)$ wie folgt:*

$$\bar{\pi} \circ \bar{\rho} = (\bar{\pi}_1, \ldots, \bar{\pi}_{i-1}, -\bar{\pi}_j, \ldots, -\bar{\pi}_i, \bar{\pi}_{j+1}, \ldots, \bar{\pi}_n).$$

Auch hier definieren wir wieder erweiterte orientierte Permutationen.

Definition 6.30 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation, dann bezeichnet die Folge $(+0, \bar{\pi}_1, \ldots, \bar{\pi}_n, +(n+1))$ die zugehörige* erweiterte orientierte Permutation.

In Abbildung 6.25 ist ein Beispiel für eine erweiterte orientierte Permutation und eine orientierte Reversion $\bar{\rho} = (2, 4)$ angegeben.

$$\begin{aligned} \bar{\pi} &= (+0, +3, \underbrace{+1, +6, -2}_{\bar{\rho}}, +4, -5, +7) \\ \bar{\pi} \circ \bar{\rho} &= (+0, +3, +2, -6, -1, +4, -5, +7) \end{aligned}$$

Abbildung 6.25: Beispiel: Orientierte Reversion $(2, 4)$ in einer erweiterten orientierten Permutation

Damit erhalten wir die folgende Problemstellung.

SORTING BY ORIENTED REVERSALS (MIN-SOR)

Eingabe: Eine orientierte Permutation $\bar{\pi} \in \bar{S}_n$.
Gesucht: Eine kürzeste Folge von orientierten Reversionen $\bar{\rho}_1, \ldots, \bar{\rho}_k$, so dass gilt: $\bar{\pi} \circ \bar{\rho}_1 \circ \cdots \circ \bar{\rho}_k = \mathrm{id}$.

Auch für orientierte Permutationen können wir (orientierte) Breakpoints und (orientierte) Adjazenzen definieren.

Definition 6.31 *Sei $\bar{\pi} \in \bar{S}_n$ eine erweiterte orientierte Permutation. Dann ist $(i, i+1)$ eine* orientierte Adjazenz *oder kurz eine* Adjazenz, *wenn ein $j \in [0 : n]$ existiert mit:*

- $(\bar{\pi}_i, \bar{\pi}_{i+1}) = (+j, +(j+1))$ *oder*
- $(\bar{\pi}_i, \bar{\pi}_{i+1}) = (-(j+1), -j)$.

Andernfalls heißt $(i, i+1)$ ein orientierter Breakpoint *oder kurz ein* Breakpoint. *Die Anzahl der orientierten Breakpoints bzw. der Adjazenzen einer orientierten Permutation $\bar{\pi} \in \bar{S}_n$ bezeichnet man mit $\bar{b}(\bar{\pi})$ bzw. $\bar{a}(\bar{\pi})$.*

Nun können wir noch eine Bezeichnung für die minimale Anzahl orientierter Reversion zur Sortierung orientierter Permutationen festlegen.

Notation 6.32 *Sei $\bar{\pi} \in \bar{S}_n$ eine erweiterte orientierte Permutation. Dann bezeichnet $\bar{d}(\bar{\pi})$ die minimale Anzahl von orientierten Reversionen, die nötig sind, um $\bar{\pi}$ zu sortieren.*

Für eine einfache Weiterbearbeitung von orientierten Permutationen definieren wir jetzt die zugehörige unorientierte Permutation.

21.01.25

Definition 6.33 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation, dann ist $\pi \in S_{2n}$ die* zugehörige unorientierte Permutation*, wobei $\pi = (\pi_1, \ldots, \pi_{2n})$ mit:*

- $(\pi_{2i-1}, \pi_{2i}) = (2\bar{\pi}_i - 1, 2\bar{\pi}_i)$*, wenn* $\bar{\pi}_i > 0$*,*
- $(\pi_{2i-1}, \pi_{2i}) = (2|\bar{\pi}_i|, (2|\bar{\pi}_i| - 1))$*, wenn* $\bar{\pi}_i < 0$*.*

Im Allgemeinen werden wir die erweiterte zugehörige unorientierte Permutation betrachten. In Abbildung 6.26 ist ein Beispiel einer erweiterten orientierten Permutation mit ihrer zugehörigen erweiterten unorientierten Permutation angegeben.

$$\begin{aligned} \bar{\pi} &= (+0, +3, -2, -1, +4, -5, +6) \\ \pi &= (0, \overbrace{5, 6}^{+3}, \overbrace{4, 3}^{-2}, \overbrace{2, 1}^{-1}, \overbrace{7, 8}^{+4}, \overbrace{10, 9}^{-5}, 11) \end{aligned}$$

Abbildung 6.26: Beispiel: Zu einer orientierten Permutation zugehörige unorientierte Permutation

Im Folgenden werden wir immer die zugehörigen erweiterten unorientierten Permutationen und darauf operierende Reversionen betrachten. Dabei operieren Reversionen aber immer nur auf Breakpoints in der zugehörigen erweiterten unorientierten Permutation.

6.4.2 Reality-Desire-Diagram

Wir definieren nun das Analogon von Breakpoint-Graphen für orientierte Permutationen, so genannte Reality-Desire-Diagrams.

Definition 6.34 *Sei $\bar{\pi} \in \bar{S}_n$ eine erweiterte orientierte Permutation und sei π die zugehörige erweiterte unorientierte Permutation. Dann ist das* (erweiterte) Reality-Desire-Diagram *(kurz RDD) $G(\bar{\pi})$ bzw. $G'(\bar{\pi})$ der (erweiterte) Breakpoint-Graph $G(\pi)$ bzw. $G'(\pi)$ mit der Einschränkung, dass es für jedes $i \in [1:n]$ keine Reality- bzw. Desire-Edges zwischen $(2i-1)$ und $(2i)$ gibt.*

In Abbildung 6.27 ist für die Permutation $\bar{\pi} = (+3, -2, -1, +4, -5)$ das zugehörige erweiterte Reality-Desire-Diagram (RDD) angegeben.

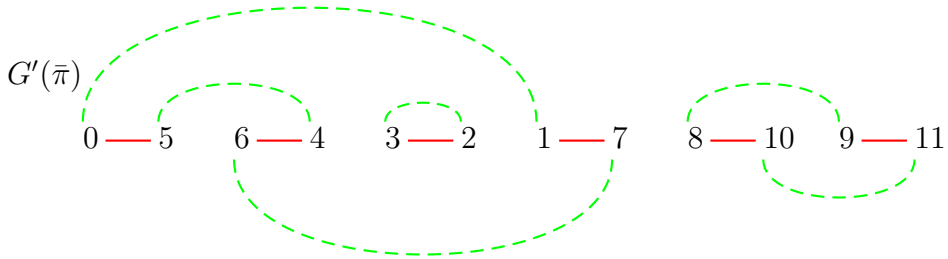


Abbildung 6.27: Beispiel: Erweitertes RDD einer orientierten Permutation

Zuerst halten wir fest, dass die maximalen alternierenden Zyklenzerlegungen im Falle von Reality-Desire-Diagram sehr einfach ermittelbar sind, da sie eindeutig ist.

Beobachtung 6.35 *Sei $\bar{\pi} \in \bar{S}_n$ eine erweiterte orientierte Permutation und sei $G'(\bar{\pi})$ das zugehörige erweiterte Reality-Desire-Diagram. Dann ist jeder Knoten in $G'(\bar{\pi})$ zu genau einer roten und genau einer grünen Kante inzident. Damit gibt es genau eine Zyklenzerlegung in alternierende Kreise und diese kann in Linearzeit ermittelt werden.*

Notation 6.36 *Sei $\bar{\pi} \in \bar{S}_n$ eine erweiterte orientierte Permutation. Die Anzahl der alternierenden Kreise im (erweiterten) Reality-Desire-Diagram $G(\bar{\pi})$ bzw. $G'(\bar{\pi})$ wird mit $\bar{c}(\bar{\pi})$ bzw. $\bar{c}'(\bar{\pi})$ bezeichnet.*

Damit folgt im Wesentlichen das folgende Lemma

Lemma 6.37 *Sei $\bar{\pi} \in \bar{S}_n$ eine beliebige erweiterte orientierte Permutation, dann gilt $\bar{d}(\bar{\pi}) \geq \bar{b}(\bar{\pi}) - \bar{c}(\bar{\pi})$ bzw. $\bar{d}(\bar{\pi}) \geq n + 1 - \bar{c}'(\bar{\pi})$.*

Beweis: Sei $\pi \in S_{2n}$ die zu $\bar{\pi} \in \bar{S}_n$ gehörige unorientierte Permutation. Dann gilt $\bar{b}(\bar{\pi}) = b(\pi)$ sowie $\bar{c}(\bar{\pi}) = c(\pi)$. Offensichtlich kann jede Folge von orientierten Reversionen von $\bar{\pi}$ in eine Folge von Reversionen in π verwandelt werden und umgekehrt, sofern für keine Reversion (i, j) (mit $i < j$ gilt) in π gilt, dass i gerade oder j ungerade ist. Denn genau dann werden die künstlich erzeugten Zweier-Strips der zugehörigen unorientierten Permutation nie aufgebrochen.

Insbesondere kann eine optimale Folge von orientierten Reversionen für die orientierte Permutation nicht kürzer sein als eine optimale Folge von Reversionen für die zugehörige unorientierte Permutation und die untere Schranke überträgt sich, d.h.

$$\bar{d}(\bar{\pi}) \geq d(\pi) \geq b(\pi) - c(\pi) = \bar{b}(\bar{\pi}) - \bar{c}(\bar{\pi}).$$

Für das erweiterte Reality-Desire-Diagram gilt $\bar{c}'(\bar{\pi}) = c'(\pi) - n$, da die Adjazenzen innerhalb der neu erzeugten Zweierstrips im erweiterten Reality-Desire-Diagram

nicht gezählt werden. Also gilt:

$$\bar{d}(\bar{\pi}) \geq d(\pi) \geq 2n + 1 - c'(\pi) = 2n + 1 - (\vec{c}'(\bar{\pi}) + n) = n + 1 - \vec{c}'(\bar{\pi}).$$

Damit sind die Behauptungen gezeigt. ■

6.4.3 Der Overlap-Graph

In diesem Abschnitt betrachten wir noch ein weiteres Hilfsmittel, den so genannten Overlap-Graphen. Zuerst definieren wir noch orientierte und unorientierte Desire-Edges im Reality-Desire-Diagram.

Definition 6.38 *Sei $\bar{\pi} \in \bar{S}_n$ eine erweiterte orientierte Permutation und sei $G(\bar{\pi})$ bzw. $G'(\bar{\pi})$ das zugehörige (erweiterte) Reality-Desire-Diagram. Eine Desire-Edge ist* orientiert*, wenn die beiden inzidenten Reality-Edges in $G(\bar{\pi})$ bzw. $G(\bar{\pi})$ in gegenläufiger Richtung durchlaufen werden. Ansonsten heißt die Desire-Edge* unorientiert*.*

In Abbildung 6.28 sind alle vier möglichen Fälle einer Desire-Edge mit ihrer Einordnung illustriert. Nun können wir die Orientierung von Desire-Edges auf Kreise in der eindeutigen alternierenden Zyklenzerlegung im Reality-Desire-Diagram übetragen.

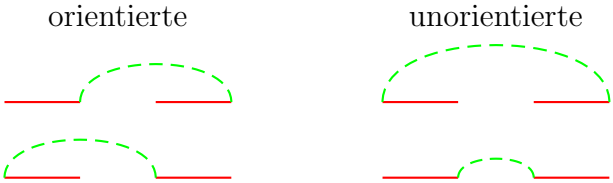


Abbildung 6.28: Skizze: Orientierte und unorientierte Desire-Edges

Definition 6.39 *Sei $\bar{\pi} \in \bar{S}_n$ eine erweiterte orientierte Permutation und sei $G(\bar{\pi})$ bzw. $G'(\bar{\pi})$ das zugehörige (erweiterte) Reality-Desire Diagram. Ein Kreis in $G(\bar{\pi})$ bzw. $G(\bar{\pi})$ heißt* orientiert*, wenn er eine orientierte Desire-Kante enthält. Die anderen Kreise bezeichnet man als* unorientiert*.*

In Abbildung 6.29 sind orientierte und unorientierte Kreise schematisch in der Kreis- und in der Intervall-Darstellung illustriert.

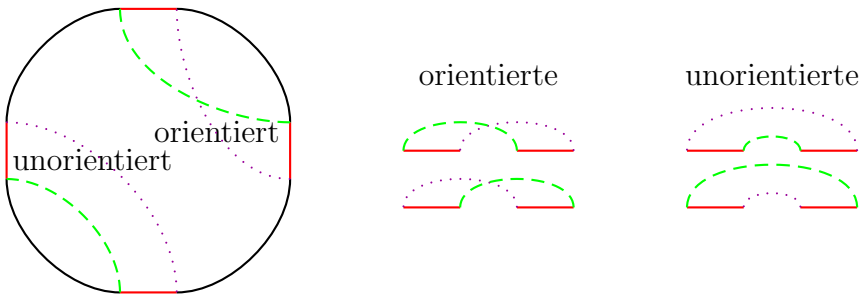


Abbildung 6.29: Skizze: Orientierte und unorientierte Kreise in der Kreis-Darstellung

Kommen wir nun zur Definition eines Overlap-Graphen.

Definition 6.40 *Sei $\bar{\pi} \in \bar{S}_n$ eine erweiterte orientierte Permutation und sei weiter $G'(\bar{\pi}) = (V', R_\pi \cup D_\pi)$ das zugehörige erweiterte Reality-Desire-Diagram. Der* Overlap-Graph $OV(\bar{\pi}) = (V, E)$ *ist dann wie folgt definiert:*

- $V := D_\pi$,
- $E := \{\{e, e'\} : e, e' \in D_\pi$ *und* e *schneidet* e' *in* $G'(\bar{\pi})\}$.

In der obigen Definition bedeutet, dass sich zwei Desire-Edges schneiden, wenn sich ihre Darstellung als Sehnen im Reality-Desire-Diagram schneiden. In der linearen Darstellung müssten bei der Bewertung des Schneidend alle Desire-Edges oberhalb (bzw. unterhalb) der Reality-Edges gezeichnet werden.

In Abbildung 6.30 ist für das Beispiel der bislang verwendeten orientierten Permutation $\bar{\pi} = (+3, -2, -1 + 4, -5)$ der zugehörige Overlap-Graph angegeben (die orientierten Desire-Edges sind blau dargestellt).

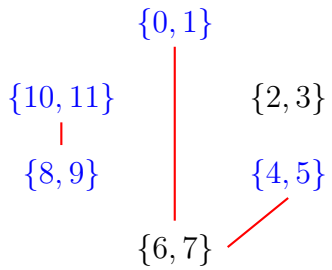


Abbildung 6.30: Beispiel: Ein Overlap-Graph für $\bar{\pi} = (+3, -2, -1, +4, -5)$

Der Begriff der Orientiertheit lässt sich nun auch auf Zusammenhangskomponenten des Overlap-Graphen ausdehnen (die, wir später sehen werden, Mengen von Kreisen entsprechen).

Definition 6.41 *Sei $\bar{\pi} \in \bar{S}_n$ eine erweiterte orientierte Permutation. Ein Knoten von $OV(\bar{\pi})$ heißt* orientiert*, wenn er zu einer orientierter Desire-Edge korrespondiert. Sonst heißt er* unorientiert*. Eine Zusammenhangskomponente von $OV(\bar{\pi})$ heißt* orientierte Komponente*, wenn sie einen orientierten Knoten enthält. Sonst heißt sie* unorientierte Komponente*.*

Nach diesen Definitionen können wir das folgende Lemma zeigen, das uns beim Beweis der Behauptung hilft, dass jede Zusammenhangskomponente im Overlap-Graphen ein Menge von Kreisen ist.

Lemma 6.42 *Sei $\bar{\pi} \in \bar{S}_n$ eine erweiterte orientierte Permutation. Sei weiter M eine Menge von Desire-Edges im erweiterten Reality-Desire-Diagram $G(\bar{\pi})$, die zu einer Zusammenhangskomponente von $OV(\bar{\pi})$ korrespondieren. Dann ist $\min(M)$ gerade und $\max(M)$ ungerade, wobei*

$$\begin{aligned} \min(M) &:= \min\{i, j \,:\, \{\pi_i, \pi_j\} \in M\}, \\ \max(M) &:= \max\{i, j \,:\, \{\pi_i, \pi_j\} \in M\}. \end{aligned}$$

Die Aussage des Lemmas ist in Abbildung 6.31 noch einmal illustriert. Kommen wir jetzt zum Beweis dieser Aussage.

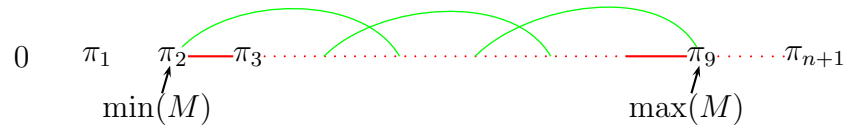


Abbildung 6.31: Skizze: Illustration der Aussage von Lemma 6.42

Beweis: Für einen Widerspruchsbeweis nehmen wir an, dass $\min(M)$ ungerade ist. Weiter sei $x = \min(M)$ und $y = \max(M)$ und $\pi \in S_{2n}$ die zu $\bar{\pi}$ gehörige unorientierte Permutation. Betrachten wir nun die zu π_x benachbarten Werte $\pi_x + 1$ bzw. $\pi_x - 1$. Einer der beiden muss an Position $x + 1$ stehen, da x ungerade ist. Der andere Wert ist von π_x über eine Desire-Edge erreichbar. Also gilt, dass sich die Elemente $\pi_x - 1$ und $\pi_x + 1$ im Intervall $[x + 1 : y] = [\min(M) + 1 : \max(M)]$ befinden. Beachte, dass $1 \leq \min(M) \leq \max(M) \leq 2n + 1$ gilt. Diese Situation für das Reality-Desire-Diagram ist in Abbildung 6.32 illustriert.

Wir betrachten jetzt den Knoten $\pi_x - 1$. Von dort aus folgen wir dem Pfad

$$(\pi_x - 1, \pi_x - 2, \pi_x - 3, \ldots, 1, 0),$$

der abwechselnd aus nebeneinander stehenden Knoten der Form $\{2i - 1, 2i\}$ und über Desire-Edges verbundene Knoten besteht. Dieser Pfad muss irgendwann aus

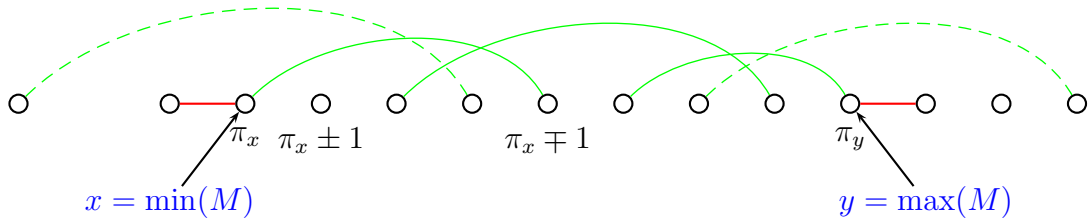


Abbildung 6.32: Skizze: Die Menge M von Desire-Edges

dem Intervall $[\min(M) : \max(M)]$ ausbrechen, da die Kanten aus M nach Definition einer Zusammenhangskomponente das gesamte Intervall $[\min(M) : \max(M)]$ überspannen. Bricht der Pfad mit einer Desire-Edge (siehe die gestrichelte Linie in Abbildung 6.32) aus, dann gehört diese Desire-Edge ebenfalls zu M und es muss $\min(M) < x$ oder $\max(M) > y$ gelten, was den gewünschten Widerspruch liefert. Bricht er über die ein Paar benachbarter Knoten der Form $\{2i-1, 2i\}$ über $(\pi_y, \pi_{y+1}{=}\pi_y - 1)$ aus, dann betrachten wir den Pfad

$$(\pi_x + 1, \pi_x + 2, \pi_x + 3, \ldots, 2n, 2n + 1),$$

der abwechselnd aus nebeneinander stehenden Knoten der Form $\{2i-1, 2i\}$ und über Desire-Edges verbundene Knoten besteht. Dieser Pfad muss irgendwann aus dem Intervall $[\min(M) : \max(M)]$ ausbrechen, da die Kanten aus M nach Definition einer Zusammenhangskomponente das gesamte Intervall $[\min(M) : \max(M)]$ überspannen. Bricht der Pfad mit einer Desire-Edge aus, dann gehört diese Desire-Edge ebenfalls zu M und es muss $\min(M) < x$ oder $\max(M) > y$ gelten, was den gewünschten Widerspruch liefert. Ein anderer Ausbruch ist nicht möglich, da $(\pi_y, \pi_{y+1}{=}\pi_y{+}1)$ bereits aufgrund des ersten Falles durch die Werte $(\pi_y, \pi_{y+1}{=}\pi_y{-}1)$ belegt ist.

Die Beweisführung für $\max(M)$ ist analog. ■

Lemma 6.43 *Sei $\bar{\pi} \in \bar{S}_n$ eine erweiterte orientierte Permutation. Jede Zusammenhangskomponente von $OV(\bar{\pi})$ korrespondiert zu einer Menge von Desire-Edges, die eine Vereinigung von alternierenden Zyklen in $G'(\bar{\pi})$ ist.*

Beweis: Für einen Widerspruchsbeweis nehmen wir an, dass es einen alternierenden Kreis C in $G'(\bar{\pi})$ gibt, dessen Desire-Edges zu mindestens 2 Zusammenhangskomponenten in $OV(\bar{\pi})$ gehören. Seien Z_1 und Z_2 zwei Zusammenhangskomponenten und $e_1 \in Z_1$ sowie $e_2 \in Z_2$ zwei Desire-Edges, die mit einer Reality-Edge verbunden sind. Man überlegt sich leicht, dass es solche Kanten e_1 und e_2 geben muss.

Fall 1: Wir betrachten zuerst den Fall, dass sich Z_1 und Z_2 überlappen. Dies ist in Abbildung 6.33 illustriert. Da Z_1 und Z_2 Zusammenhangskomponenten sind, kann

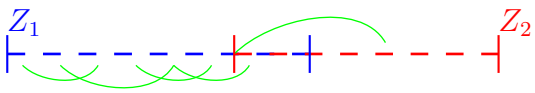


Abbildung 6.33: Skizze: Z_1 und Z_2 überlappen sich

dieser Fall nicht auftreten.

Fall 2: Wir nehmen jetzt an, dass eine Zusammenhangskomponente in der anderen enthalten ist. Betrachten wir zuerst den Unterfall, dass e_2 nicht in Z_1 eingebettet ist. Dies ist in Abbildung 6.34 illustriert. Damit muss aber $\min(Z_1)$ ungerade sein und wir erhalten einen Widerspruch zum vorherigen Lemma.

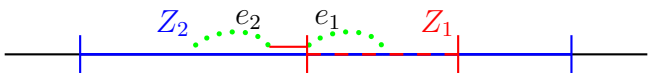


Abbildung 6.34: Skizze: Z_1 ist in Z_2 total in enthalten

Alternativ könnte ein Teil von Z_2 sich innerhalb von Z_1 befinden. Dies ist in Abbildung 6.35 illustriert. Da die Kantengruppe mit e_2 innerhalb von Z_1 zur Zusam-

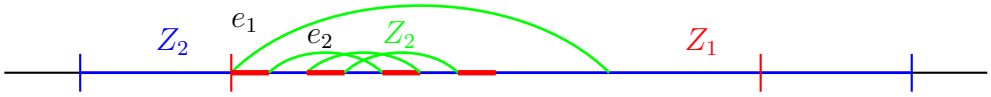


Abbildung 6.35: Skizze: Z_1 ist in Z_2 total in enthalten

menhangskomponente von Z_2 gehört, muss es eine Folge von sich schneidenden Desire-Edges aus Z_2 geben, die e_2 mit einer Desire-Edge im äußern Bereich von Z_2 verbindet. Offensichtlich muss eine davon e_1 schneiden und somit wären Z_1 und Z_2 keine verschiedenen Zusammenhangskomponenten.

Fall 3: Nun nehmen wir an, dass die beiden Zusammenhangskomponenten sich in disjunkten Intervallen befinden. Dies ist in Abbildung 6.36 illustriert.

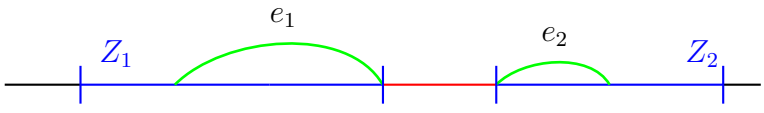


Abbildung 6.36: Skizze: Z_1 und Z_2 sind disjunkt

Auch in diesem Falle muss $\min(Z_2)$ ungerade sein und wir erhalten ebenfalls einen Widerspruch zum vorherigen Lemma. ■

Definition 6.44 *Eine Zusammenhangskomponente heißt* gut*, wenn sie orientiert ist. Ansonsten heißt sie* schlecht*.*

In Abbildung 6.37 sind für das Beispiel

$$\bar{\pi} = (+9, +8, +1, -3, +6, -4, +2, -5, +7, +10, -11)$$

im Reality-Desire-Diagram die Zusammenhangskomponenten in gute und schlechte Komponenten aufgeteilt, wobei mit A bis F die einzelnen Kreise bezeichnet werden. Nur der Kreis C enthält eine orientierte Desire-Edge (C besteht aus genau zwei orientierten Desire-Edges).

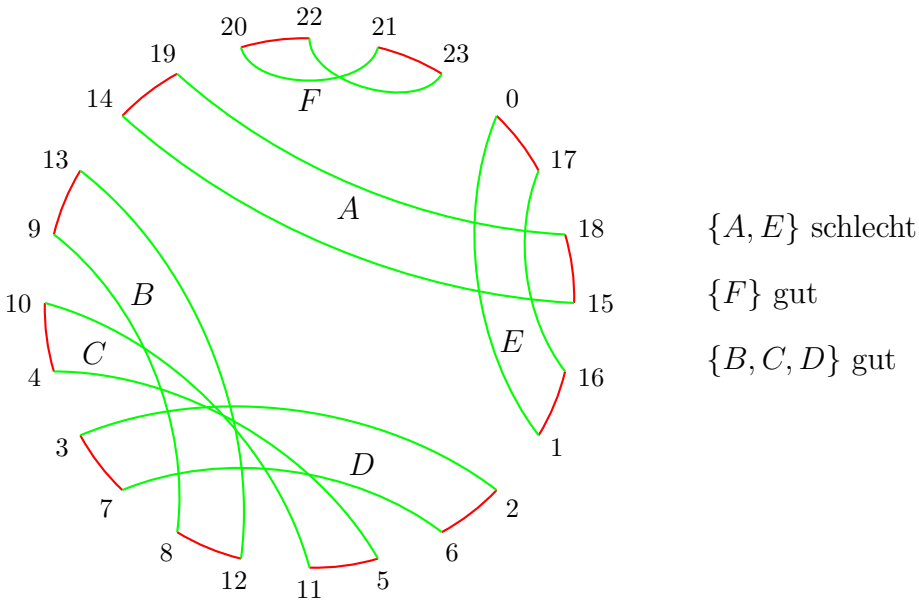


Abbildung 6.37: Skizze: Kreise und Komponenten in Reality-Desire-Diagram

6.4.4 Hurdles und Fortresses

Nach Definition können wir in guten Komponenten immer eine orientierte Reversion auf den beiden unterschiedlich durchlaufenen Reality-Edges, die adjazent zu einer orientierten Desire-Edge sind, anwenden und wir erhöhen die Anzahl der Kreise im erweiterten Reality-Desire-Diagram. Wie wir später noch sehen werden, lässt sich bei Vorhandensein orientierter Komponenten immer ein solches Paar von Reality-Edges auswählen, so dass man eine kürzesten Folge von orientierten Reversionen zur Sortierung der gegebenen orientierten Permutation erhält.

Problematisch sind also die schlechten Komponenten, die nur unorientierte Desire-Edges enthalten und somit jede orientierte Reversion auf zwei dazu adjazenten Reality-Edges nie die Anzahl der Kreise im erweiterten Reality-Desire-Diagram erhöhen kann. Weiterhin bleibt eine schlechte Komponente erhalten, bis eine orientierte Reversion innerhalb dieser Komponente ausgeführt wird oder auf zwei Komponenten ausgeführt wird.

Definition 6.45 *Sei $\bar{\pi} \in \bar{S}_n$ eine erweiterte orientierte Permutation und $OV(\bar{\pi})$ der zugehörige Overlap-Graph. Eine schlechte Komponente von $OV(\bar{\pi})$ heißt* Hurdle*, wenn sie keine anderen schlechten Komponenten trennt. $\bar{h}(\bar{\pi})$ bezeichne die Anzahl von Hurdles in einer gegebenen orientierten Permutation $\bar{\pi}$.*

Eine schlechte Komponente A trennt dabei genau dann zwei schlechte Komponenten B und C, wenn es je einen Reality-Edge in B und C gibt, so dass für die Reversion ρ, die auf diesen beiden Reality-Edges operiert, gilt, dass in $\pi \circ \rho$ dann A eine gute Komponente ist.

Im Beispiel in Abbildung 6.38 auf Seite 282 sind für die Permutation

$$\begin{aligned}\bar{\pi} \;=\; & (+11, +10, +1, +4, +3, +2, +5, +8, +7, +6, +9, +12,\\ & +19, +18, +13, +16, +15, +14, +17, +20, +23, +22, +21)\end{aligned}$$

die Komponenten A, C, D und F Hurdles, die anderen beiden B und E sind einfach nur schlechte Komponenten (im Beispiel sind der Einfachheit halber alle Komponenten schlecht). Hierbei sind die Komponenten statt im Overlap-Graphen der besseren Übersichtlichkeit wegen im Reality-Desire-Diagram angegeben.

Hurdles haben also die Eigenschaft, dass man auf ihnen orientierte Reversionen anwenden muss, um darin orientierte Kreise zu generieren. In schlechten Komponenten, die keine Hurdles sind, kann man eine orientierte Reversion auf zwei Desire-Edges in den zwei Komponente anwenden, die von dieser getrennt werden, und damit auch in der darin enthaltenen schlechten Komponente einen orientierten Kreis erzeugen. Hurdles sind somit diejenigen Komponenten die über die untere Schranke von $\bar{b}(\bar{\pi}) - \bar{c}(\bar{\pi})$ hinaus zusätzliche orientierte Reversionen erfordern, wie wir im Detail noch genauer sehen werden.

Definition 6.46 *Sei $\bar{\pi} \in \bar{S}_n$ eine erweiterte orientierte Permutation. Eine Hurdle, deren Löschung eine Nicht-Hurdle in eine Hurdle transformiert, heißt* Super-Hurdle*. Andernfalls nennen wir sie eine* einfache Hurdle*.*

Im Beispiel in Abbildung 6.38 ist die Hurdle F eine Super-Hurdle, die Hurdles A, C und D hingegen nicht. Bei Super-Hurdles muss man also aufpassen, da deren Eliminierung neue Hurdles erzeugt.

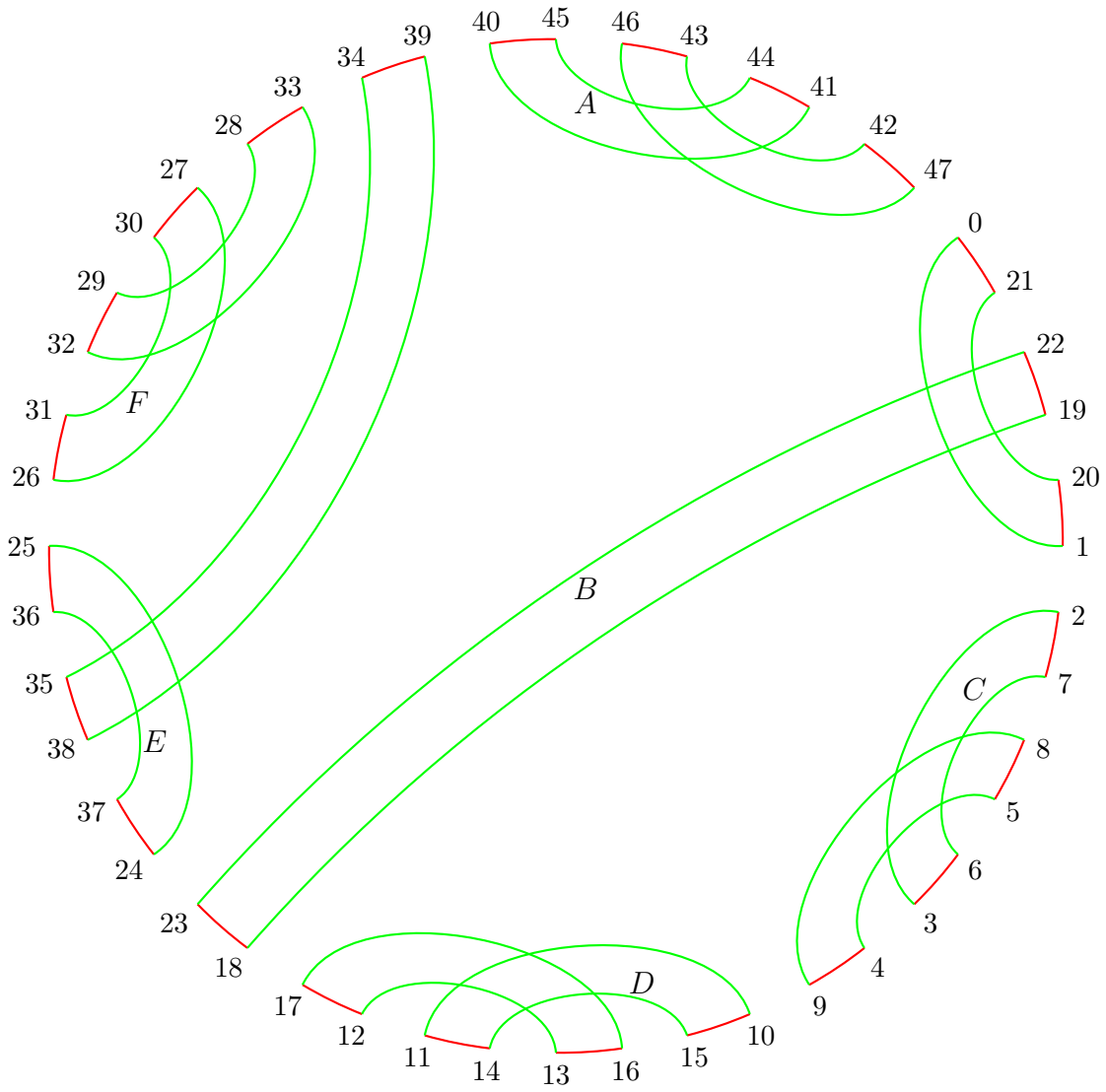


Abbildung 6.38: Skizze: Hurdles und schlechte Komponenten

Im Folgenden definieren wir noch eine besonders aufwendige Konstellation von orientierten Permutationen, den so genannte Fortresses, für die sich noch zeigen wird, dass sie besonders schwer zu sortieren sind.

Definition 6.47 *Sei $\bar{\pi} \in \bar{S}_n$ eine erweiterte orientierte Permutation. Die Permutation $\bar{\pi}$ heißt* Fortress*, wenn sie eine ungerade Anzahl von Hurdles besitzt, die alle Super-Hurdles sind. Dann ist $\bar{f}(\bar{\pi}) := 1$, wenn $\bar{\pi}$ ein Fortress ist, und $\bar{f}(\bar{\pi}) := 0$ sonst.*

Im Beispiel in Abbildung 6.38 ist die Permutation daher kein Fortress.

6.4.5 Eine untere Schranke für Min-SOR

Unser Ziel in diesem Abschnitt wird sein, die folgende Ungleichung zu zeigen:

$$\begin{aligned} \bar{d}(\bar{\pi}) &\geq \bar{b}(\bar{\pi}) - \bar{c}(\bar{\pi}) + \bar{h}(\bar{\pi}) + \bar{f}(\bar{\pi}) \\ &= (n+1) - \bar{c}'(\bar{\pi}) + \bar{h}(\bar{\pi}) + \bar{f}(\bar{\pi}). \end{aligned}$$

Da $\bar{h}(\bar{\pi}) > 0$ und $\bar{f}(\bar{\pi})$ gilt, ist dies eine stärkere Behauptung als die einfache Übertragung vom unorientierten auf den orientierten Fall mittels $\bar{d}(\bar{\pi}) \geq \bar{b}(\bar{\pi}) - \bar{c}(\bar{\pi})$. Hierfür benötigen wir insbesondere den folgenden Satz.

23.01.25

Theorem 6.48 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $\bar{\rho}$ eine orientierte Reversion, dann gilt:*

$$\bar{c}'(\bar{\pi} \circ \bar{\rho}) - \bar{h}(\bar{\pi} \circ \bar{\rho}) - (\bar{c}'(\bar{\pi}) - \bar{h}(\bar{\pi})) \leq 1.$$

Beweis: Wir halten zunächst einmal ein paar grundlegende Fakten fest:

- Wenn $\bar{\rho}$ nur auf einer Komponente operiert, so kann sie auch nur an dieser etwas ändern und somit maximal eine Hurdle eliminieren.
- Wenn $\bar{\rho}$ auf zwei Reality-Edges verschiedener Komponenten (egal, ob gute oder schlechte) operiert, dann sind folgende Fälle möglich:
  a) Wenn beide Komponenten Hurdles sind, so können diese nach Definition nicht durch eine weitere Hurdle getrennt sein und nur diese beiden Hurdles können schlimmstenfalls eliminiert werden.
  b) Wenn beide Komponenten keine Hurdles sind, so können maximal 2 Hurdles dazwischen liegen, die von $\bar{\rho}$ verändert werden können (eine orientierte Reversion einer gesamten Hurdle stellt dabei keine Veränderung dar). Weitere schlechte Komponenten, die dazwischen liegen, würden ansonsten diese beiden Hurdles trennen und wären dann keine Hurdle. Also können maximal zwei Hurdles eliminiert werden.
  c) Wenn genau eine Komponente eine Hurdle ist, so kann nach Definition einer Hurdle maximal eine weitere Hurdle zwischen den beiden Komponenten liegen. Also kann auch in diesem Fall $\bar{\rho}$ maximal zwei Hurdles verändern (eine orientierte Reversion einer gesamten Hurdle stellt dabei auch hier keine Veränderung dar). Also können ebenso nur maximal zwei Hurdles eliminiert werden.

Aus diesen Überlegungen folgt, dass $\bar{h}(\bar{\pi} \circ \bar{\rho}) - \bar{h}(\bar{\pi}) \geq -2$.

Weiter gilt die folgende Fallunterscheidung:

1) Gilt $\bar{c}'(\bar{\pi} \circ \bar{\rho}) - \bar{c}'(\bar{\pi}) = 1$, dann muss $\bar{\rho}$ auf einem orientierten Kreis operieren, also kann keine Hurdle eliminiert werden, d.h. $\bar{h}(\bar{\pi} \circ \bar{\rho}) - \bar{h}(\bar{\pi}) \geq 0$. Somit gilt:

$$\begin{aligned} \bar{c}'(\bar{\pi} \circ \bar{\rho}) - \bar{h}(\bar{\pi} \circ \bar{\rho}) - (\bar{c}'(\bar{\pi}) - \bar{h}(\bar{\pi})) &= \bar{c}'(\bar{\pi} \circ \bar{\rho}) - \bar{c}'(\bar{\pi}) - (\bar{h}(\bar{\pi} \circ \bar{\rho}) - \bar{h}(\bar{\pi})) \\ &\leq 1 - 0 = 1. \end{aligned}$$

2) Gilt $\bar{c}'(\bar{\pi} \circ \bar{\rho}) - \bar{c}'(\bar{\pi}) = 0$, dann muss $\bar{\rho}$ auf genau einem Kreis operieren, da sich nur dann die Anzahl der Kreise nicht verändern kann. Also kann $\bar{\rho}$ maximal eine Hurdle entfernen oder hinzufügen. Also gilt insbesondere $\bar{h}(\bar{\pi} \circ \bar{\rho}) - \bar{h}(\bar{\pi}) \geq -1$. Daraus folgt

$$\begin{aligned} \bar{c}'(\bar{\pi} \circ \bar{\rho}) - \bar{h}(\bar{\pi} \circ \bar{\rho}) - (\bar{c}'(\bar{\pi}) - \bar{h}(\bar{\pi})) &= \bar{c}'(\bar{\pi} \circ \bar{\rho}) - \bar{c}'(\bar{\pi}) - (\bar{h}(\bar{\pi} \circ \bar{\rho}) - \bar{h}(\bar{\pi})) \\ &\leq 0 - (-1) = 1. \end{aligned}$$

3) Gilt $\bar{c}'(\bar{\pi} \circ \bar{\rho}) - \bar{c}'(\bar{\pi}) = -1$, dann muss nach der obigen Überlegung in jedem Falle $\bar{h}(\bar{\pi} \circ \bar{\rho}) - \bar{h}(\bar{\pi}) \geq -2$ gelten, also ist

$$\begin{aligned} \bar{c}'(\bar{\pi} \circ \bar{\rho}) - \bar{h}(\bar{\pi} \circ \bar{\rho}) - (\bar{c}'(\bar{\pi}) - \bar{h}(\bar{\pi})) &= \bar{c}'(\bar{\pi} \circ \bar{\rho}) - \bar{c}'(\bar{\pi}) - (\bar{h}(\bar{\pi} \circ \bar{\rho}) - \bar{h}(\bar{\pi})) \\ &\leq -1 - (-2) = 1. \end{aligned}$$

Somit folgt die Behauptung. ■

Korollar 6.49 *Sei $\bar{\pi} \in \bar{S}_n$, dann gilt*

$$\bar{d}(\bar{\pi}) \geq (n+1) - \bar{c}'(\bar{\pi}) + \bar{h}(\bar{\pi}) = \bar{b}(\bar{\pi}) - \bar{c}(\bar{\pi}) + \bar{h}(\bar{\pi}).$$

Beweis: Da nach dem vorhergehenden Lemma jede orientierte Reversion den folgenden Parameter $\bar{h}(\bar{\pi}) - \bar{c}'(\bar{\pi})$ nur um maximal eins erniedrigen kann und da

$$(n+1) - \bar{c}'(\mathrm{id}) + \bar{h}(\mathrm{id}) = (n+1) - (n+1) + 0 = 0$$

gilt, folgt

$$\bar{d}(\bar{\pi}) \geq (n+1) - \bar{c}'(\bar{\pi}) + \bar{h}(\bar{\pi})$$

und somit die erste Behauptung.

Man überlegt sich wie schon früher, dass auch hier

$$(n+1) - \bar{c}'(\bar{\pi}) + \bar{h}(\bar{\pi}) = \bar{b}(\bar{\pi}) - \bar{c}(\bar{\pi}) + \bar{h}(\bar{\pi})$$

gilt und damit folgt die zweite Behauptung. ■

Das letzte Korollar lässt sich noch wie folgt verstärken.

Theorem 6.50 *Sei $\bar{\pi} \in \bar{S}_n$, dann gilt*

$$\bar{d}(\bar{\pi}) \geq (n+1) - \bar{c}'(\bar{\pi}) + \bar{h}(\bar{\pi}) + \bar{f}(\bar{\pi}) = \bar{b}(\bar{\pi}) - \bar{c}(\bar{\pi}) + \bar{h}(\bar{\pi}) + \bar{f}(\bar{\pi}).$$

Den Beweis, dass wir für eine Fortress eine weitere orientierte Reversion wirklich benötigen, lassen wir hier aus und verweisen den interessierten Leser auf die Originalliteratur.

6.4.6 Sortierung orientierter Komponenten

Im letzten Abschnitt haben wir eine untere Schranke für das Sortieren mit orientierten Permutationen kennen gelernt. Nun wollen wir zeigen, dass man diese untere Schranke auch in jedem Falle erreichen kann. Zuerst wollen wir uns mit dem Fall beschäftigen, dass die Permutation orientierte Komponenten enthält.

In Abbildung 6.39 ist das Reality-Desire-Diagram für die folgende orientierte Permutation $\bar{\pi} = (+3, -1, +6, +5, -2, +4)$ angegeben, wobei die orientierten Kanten blau hervorgehoben sind und die zugehörigen orientierten Reversionen $\bar{\rho}_1$ und $\bar{\rho}_2$ auf den jeweiligen inzidenten Reality-Edges.

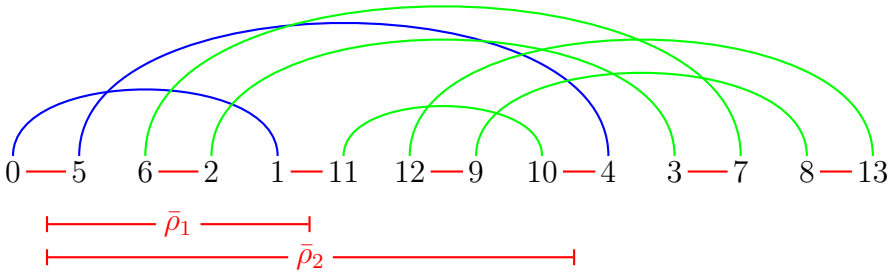


Abbildung 6.39: Beispiel: Orientierte Desire-Edges

Definition 6.51 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und $\bar{\rho}$ eine orientierte Reversion auf $\bar{\pi}$. Der* Score *von $\bar{\rho}$ ist definiert als die Anzahl orientierter Desire-Edges in $G(\bar{\pi} \circ \bar{\rho})$.*

In den Abbildungen 6.40 und 6.41 sind die Ergebnisse der orientierten Reversionen, die zu orientierten Desire-Edges im Beispiel in Abbildung 6.39 gehören, und ihre Scores angegeben.

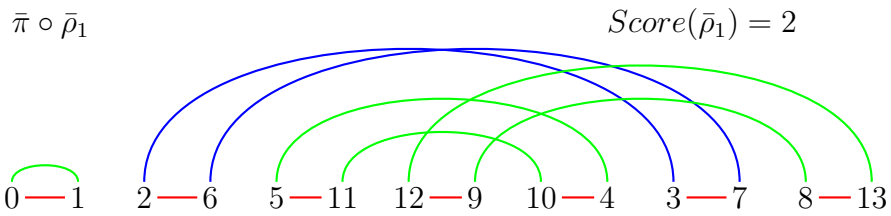


Abbildung 6.40: Beispiel: Score von $\bar{\pi} \circ \bar{\rho}_1$

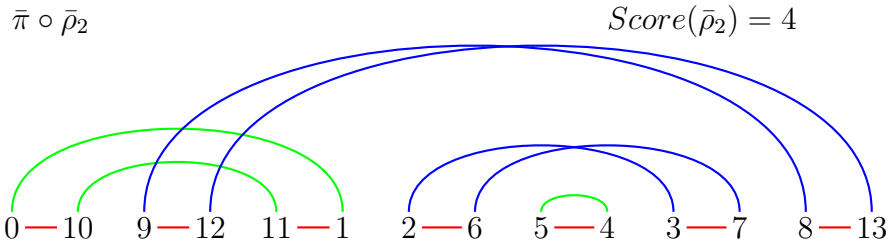


Abbildung 6.41: Beispiel: Score von $\bar{\pi} \circ \bar{\rho}_2$

Der zugehörige Greedy-Algorithmus geht nun wie folgt vor. Solange $G(\bar{\pi})$ eine orientierte Desire-Edge besitzt, wähle eine orientierte Reversion zu einer orientierten Desire-Edge mit maximalem Score. Ziel des Abschnittes wird der Beweis des folgenden Satzes sein.

Theorem 6.52 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation. Wendet man mit dem oben angegeben Algorithmus genau k orientierte Reversionen auf $\bar{\pi}$ an und erhält man somit $\bar{\pi}'$, dann gilt $\bar{d}(\bar{\pi}) = \bar{d}(\bar{\pi}') + k$.*

Bevor wir zum Beweis kommen, zeigen wir erst noch, wie man algorithmisch effizient die ausgeführten orientierten Reversionen und ihre Änderungen auf dem Overlap-Graphen darstellen kann.

Lemma 6.53 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation. Eine Desire-Edge, die zu $v \in V(OV(\bar{\pi}))$ korrespondiert, ist genau dann orientiert, wenn der Grad von v ungerade ist.*

Für die Permutation in Abbildung 6.39 ist in Abbildung 6.42 der zugehörige Overlap-Graph angegeben. Die orientierten Desire-Edges sind durch Knoten mit einer doppelten Kreislinie dargestellt.

Beweis: $\Rightarrow$**:** Sei $v \in V(OV(\bar{\pi}))$ eine orientierte Desire-Edge im Overlap-Graphen. Sei x die Anzahl der Desire-Edges, die vollständig innerhalb von der zu v gehörigen Desire-Edge verlaufen. Sei weiter y die Anzahl der Desire-Edges, die die zu v

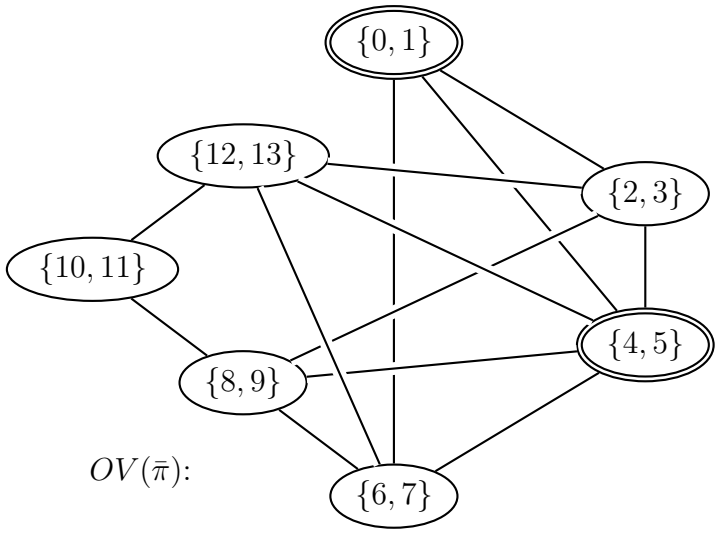


Abbildung 6.42: Beispiel: Overlap-Graph zur Permutation in Abbildung 6.39

gehörige Desire-Edge schneiden. Dann gilt, dass die Anzahl Endpunkte innerhalb der betrachteten Desire-Edge ist $2x + y$.

Die Anzahl der Endpunkte innerhalb der betrachteten Intervalls der Desire-Edge muss aber ungerade sein, da v ja orientiert ist. Also ist $2x + y$ ungerade und es muss dann auch y ungerade sein. Da y gerade der Grad des Knotens v im Overlap-Graphen ist, folgt die Behauptung. Dies ist in Abbildung 6.43 noch einmal illustriert.

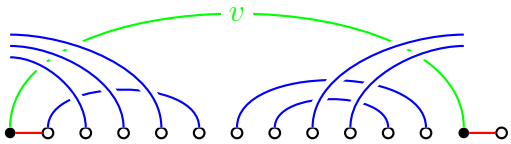


Abbildung 6.43: Skizze: Der Knoten v im Overlap-Graphen mit ungeradem Grad

⇐: Nach Voraussetzung gibt es eine ungerade Anzahl von Endpunkten von Kanten, die die zu v gehörige Desire-Edge schneiden. Die Anzahl Endpunkte von Kanten, die vollständig innerhalb der zu v gehörigen Desire-Edge verlaufen, ist offensichtlich gerade. Also muss die Anzahl der Endpunkte, die innerhalb der zu v gehörigen Desire-Edge liegen, ungerade sein. Diese Tatsache ist in Abbildung 6.43 illustriert.

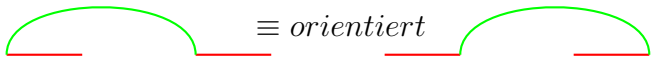


Abbildung 6.44: Skizze: Form der zugehörige Desire-Edge

Somit muss die zu v gehörige Desire-Edge eine der Formen besitzen, wie in Abbildung 6.44 abgebildet. Also ist diese Desire-Edge orientiert. ■

Lemma 6.54 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $\bar{\rho}$ eine orientierte Reversion, die zu einem orientierten Knoten $v \in V(OV(\bar{\pi}))$ korrespondiert. Dann entsteht $OV(\bar{\pi} \circ \bar{\rho})$ aus $OV(\bar{\pi})$ durch Komplementieren des von $\bar{N}(v)$ induzierten Teilgraphen, wobei*

$$\begin{aligned} N(v) &:= \{w : \{v, w\} \in E\}\,, \\ \bar{N}(v) &:= N(v) \cup \{v\}. \end{aligned}$$

In Abbildung 6.45 ist ein Beispiel für die Anwendung der orientierten Reversion $\bar{\rho}_2$ auf die Permutation in Abbildung 6.42 und der zugehörige Overlap-Graphen angegeben. Die Kanten, die verschwinden, sind rot gestrichelt gezeichnet, die neu entstandenen Kanten sind grün gezeichnet. Die neu entstandenen orientierten Knoten sind blau gezeichnet.

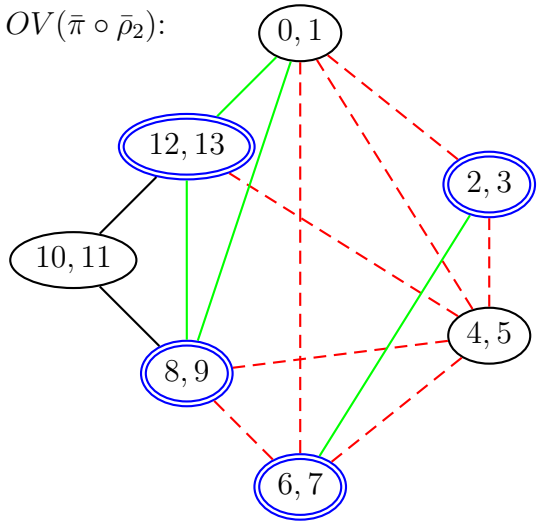


Abbildung 6.45: Beispiel: Der Overlap-Graph von $\bar{\pi} \circ \bar{\rho}_2$

Beweis: Zunächst einmal sei $v = (x, x+1)$ die betrachtete Desire-Edge, auf der $\bar{\rho}$ operiert. $\bar{\rho}$ sorgt dafür, dass x und $x+1$ in $\bar{\pi} \circ \bar{\rho}$ benachbart sind, d.h. im erweiterten Reality-Desire-Diagram befinden sich x und $x+1$ in einem Kreis, der aus genau einer Reality- und einer Desire-Edge besteht. Somit ist der zugehörige Knoten v anschließend auch im Overlap-Graphen ein isolierter Knoten, da die zugehörige Desire-Edge keine anderen Desire-Edges mehr schneiden kann.

Zuerst halten wir fest, dass nur Desire-Edges von $\bar{\rho}$ verändert werden können, die die zu v gehörige Desire-Edge schneiden oder vollständig innerhalb dieser Desire-Edge liegen.

Fall 1: Betrachten wir zuerst den Fall, dass zwei Knoten u und w beide v schneiden, sich aber selbst nicht schneiden. Die Auswirkung von $\bar{\rho}$ ist in Abbildung 6.46

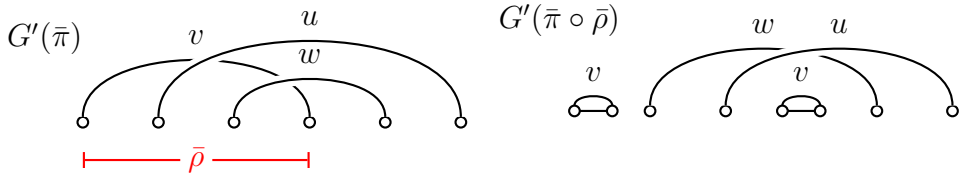


Abbildung 6.46: Skizze: Wirkung von $\bar{\rho}$, wenn sich u und w nicht schneiden

illustriert, wobei die Lage von v von der Richtung der inzidenten Reality-Edges abhängt. Wie man leicht sieht, schneiden sich u und w nicht mehr mit v, aber durch die Verdrehung von u mit w schneiden sich jetzt u und w.

Fall 2: Betrachten wir jetzt den Fall, dass sich u und w sowohl mit v als auch mit sich selbst schneiden. Die Auswirkung von $\bar{\rho}$ ist in Abbildung 6.47 illustriert, wobei die Lage von v von der Richtung der inzidenten Reality-Edges abhängt. Wie man

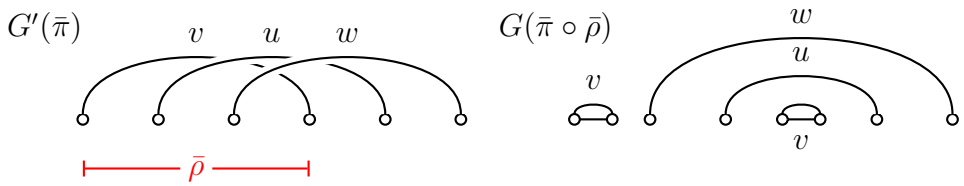


Abbildung 6.47: Skizze: Wirkung von $\bar{\rho}$, wenn sich u und w schneiden

leicht sieht, schneiden sich u und w nicht mehr mit v, aber durch die Entdrehung von u mit w schneiden sich jetzt auch u und w nicht mehr.

Fall 3: Jetzt betrachten wir den Fall, dass sich u mit v schneidet, aber v nicht mit w. Wenn sich u und w nicht schneiden, ändert sich bei der orientierten Reversion $\bar{\rho}$ daran nichts. Also nehmen wir an, dass sich u und w schneiden. Nach der Operation $\bar{\rho}$ schneiden sich u und w weiterhin. Die Auswirkung von $\bar{\rho}$ ist in Abbildung 6.48 illustriert, wobei die Lage von v von der Richtung der inzidenten Reality-Edges abhängt (hier nur der Fall, dass sich u und w schneiden).

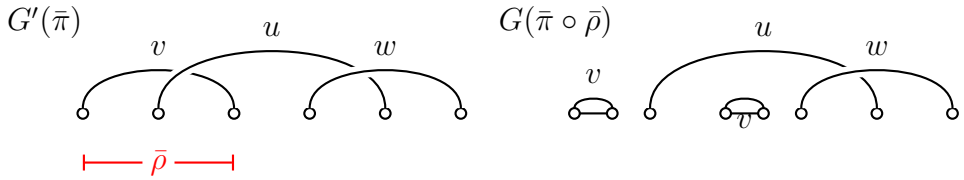


Abbildung 6.48: Skizze: Wirkung von $\bar{\rho}$, wenn sich u und w schneiden

Fall 4: Nun nehmen wir an, dass sich u und w nicht mit v schneiden. Wenn jetzt u und w außerhalb von v liegen, kann $\bar{\rho}$ an der Schnitt-Relation nichts ändern. Liegen u und w innerhalb von v, dann liegt die in Abbildung 6.49 skizzierte Situation vor.

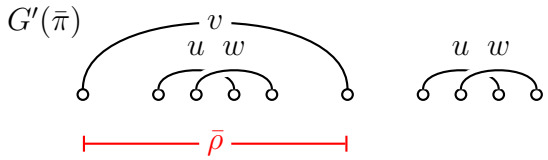


Abbildung 6.49: Skizze: v schneidet sich nicht mit u und w

Aus allen Fällen folgt, dass sich die Kanten im Teilgraphen $(\bar{N}(v), E \cap \bar{N}(v))$ komplementieren und die Behauptung ist gezeigt. ■

Lemma 6.55 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $\bar{\rho}$ eine orientierte Reversion mit maximalem Score, die zu einer orientierten Desire-Edge korrespondiert, die im Overlap-Graph durch den Knoten $v \in V(OV(\bar{\pi}))$ gegeben ist. Dann ändert jeder zu v adjazente Knoten in $OV(\bar{\pi})$ seine Orientierung in $OV(\bar{\pi} \circ \bar{\rho})$.*

Beweis: Nach Voraussetzung ist $v \in V(OV(\bar{\pi}))$ ein orientierter Knoten und somit ist $\deg(v)$ ungerade. Wir betrachten jetzt die Nachbarschaft $N(v)$ von v in $OV(\bar{\pi})$, wie in Abbildung 6.50 dargestellt.

Da v orientiert ist, muss $|N(v)| = 2k+1$ für ein $k \in \mathbb{N}$ gelten (also ungerade sein). Sei jetzt $w \in N(v)$ eine beliebiger zu v adjazenter Knoten. Weiter sei $j := |N(w) \cap N(v)|$

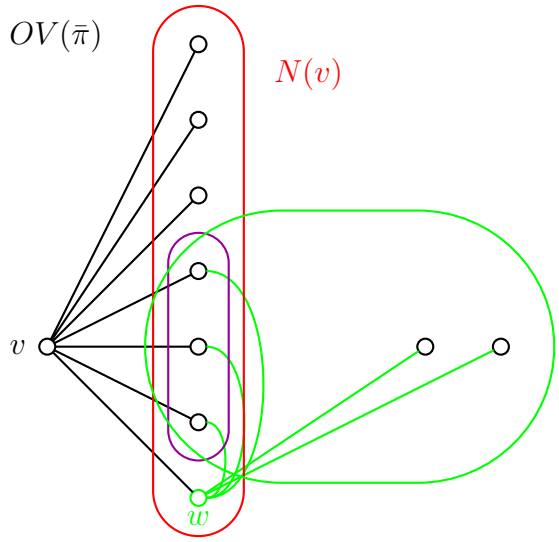


Abbildung 6.50: Skizze: Die Nachbarschaft eines Knotens v

(die Anzahl der Knoten im violetten Kreis in Abbildung 6.50). Im Folgenden sei $\deg(v)$ bzw. $\deg'(v)$ den Grad des Knotens v im Overlap-Graphen $OV(\bar{\pi})$ bzw. $OV(\bar{\pi} \circ \bar{\rho})$. Es gilt

$$\deg'(w) = \deg(w) - 1 - j + (2k - j) = \deg(w) + 2(k - j) - 1,$$

da die Kante $\{v, w\}$ und die Kanten von w zu $N(w) \cap N(v)$ wegfallen sowie die Kanten von w zu $N(v) \setminus (N(w) \cap N(v))$ hinzukommen.

Offensichtlich ist $2(k - j) - 1$ ungerade und somit muss sich die Parität des Grades von w ändern. ■

Notation 6.56 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation. Dann bezeichnet für ein festes $v \in V(OV(\bar{\pi}))$:*

- *$T(\bar{\pi})$ die Anzahl orientierter Knoten in $V(OV(\bar{\pi}))$;*
- *$O_v(\bar{\pi})$ die Anzahl orientierter Knoten in $V(OV(\bar{\pi})) \cap N(v)$;*
- *$U_v(\bar{\pi})$ die Anzahl unorientierter Knoten in $V(OV(\bar{\pi})) \cap N(v)$.*

Mit Hilfe dieser Notation können wir das Ergebnis aus dem letzten Beweis auch wie folgt formulieren.

Lemma 6.57 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $\bar{\rho}$ eine orientierte Reversion auf $\bar{\pi}$, die zu einer orientierten Desire-Edge korrespondiert, wobei v der zugehörige Knoten in $OV(\bar{\pi})$ ist. Für den Score von $\bar{\rho}$ gilt dann*

$$Score_{\bar{\pi}}(\bar{\rho}) = T(\bar{\pi}) + U_v(\bar{\pi}) - O_v(\bar{\pi}) - 1.$$

Beweis: Der Term $T(\bar{\pi})$ beschreibt die Anzahl der orientierten Knoten in $OV(\bar{\pi})$. Der Term $U_v(\bar{\pi}) - O_v(\bar{\pi}) - 1$ beschreibt die relative Änderung, die durch die orientierte Reversion $\bar{\rho}$ ausgelöst wird. ■

Definition 6.58 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation. Eine orientierte Reversion $\bar{\rho}$ heißt* sicher*, wenn*

$$\bar{c}'(\bar{\pi} \circ \bar{\rho}) - \bar{h}(\bar{\pi} \circ \bar{\rho}) > \bar{c}'(\bar{\pi}) - \bar{h}(\bar{\pi}).$$

Damit können wir den folgenden fundamentalen Satz beweisen.

Theorem 6.59 *Eine zu einer orientierten Desire-Edge korrespondierenden orientierten Reversion mit maximalem Score ist sicher.*

Beweis: Für den Beweis werden wir folgende stärkere Aussage zeigen: Es wird keine neue unorientierte Komponente erzeugt. Da keine neuen unorientierten Komponenten erzeugt werden, kann auch keine neue Hurdle erzeugt werden. Dies ist ausreichend, da eine orientierte Reversion immer die Anzahl der Kreise im erweiterten Reality-Desire $G'(\bar{\pi})$ erhöht. Außerdem kann dabei keine schlechte Komponente zu einer Hurdle werden.

Für einen Widerspruchsbeweis nehmen wir an, dass eine orientierte Reversion mit maximalen Score, die zur orientierten Desire-Edge $v \in V(OV(\bar{\pi}))$ gehört, eine neue unorientierte Komponente C erzeugt.

Dann muss einer der Knoten aus C in $OV(\bar{\pi})$ zu v adjazent gewesen sein, da sonst C schon in $OV(\bar{\pi})$ unorientiert gewesen wäre. Sei dieser Knoten $w \in V(OV(\bar{\pi}))$. Sei weiter $\text{Score}(v) = T + U_v - O_v - 1$ und $\text{Score}(w) = T + U_w - O_w - 1$.

Alle unorientierten Knoten aus $N(v)$ sind adjazent zu w in $OV(\bar{\pi})$. Sonst gäbe es einen orientierten Knoten in $OV(\bar{\pi} \circ \bar{\rho})$, der zu w in $OV(\bar{\pi})$ adjazent ist. Dann wäre C orientiert, was einen Widerspruch liefert. Somit gilt $U_w \geq U_v$.

Alle orientierten Knoten in $N(w)$ sind adjazent zu v in $OV(\bar{\pi})$. Sonst gäbe es einen orientierten Knoten $u \in N(w)$ in $OV(\bar{\pi})$, der zu v nicht adjazent ist. Da $u \notin \bar{N}(v)$ ist, wäre aber u auch in $OV(\bar{\pi} \circ \bar{\rho})$ orientiert und zu w adjazent. Dann wäre C orientiert, was einen Widerspruch liefert. Damit gilt $O_w \leq O_v$.

Somit gilt:

$$\text{Score}(v) = T + U_v - O_v - 1 \leq T + U_w - O_w - 1 = \text{Score}(w).$$

Wenn $\text{Score}(w) > \text{Score}(v)$, erhalten wir einen Widerspruch zur Voraussetzung. Also gilt $U_v = U_w$ und $O_v = O_w$. Dann gilt nach dem obigen auch $N(v) \cap U = N(w) \cap U$ und $N(w) \cap O = N(v) \cap O$, wobei O bzw. U die orientierten bzw. unorientierten Knoten in $OV(\bar{\pi})$ sind und somit $N(v) = N(w)$. Da v und w adjazent sind, gilt dann auch $\bar{N}(v) = \bar{N}(w)$.

Die orientierte Reversion $\bar{\rho}$ würde also aus w einen isolierten Knoten machen. Dies ist ein Widerspruch zur Tatsache, dass w in der unorientierten Komponente C enthalten ist. ■

Aus dem Beweis folgt unmittelbar auch das folgende Korollar.

Korollar 6.60 *Eine zu einer orientierten Desire-Edge korrespondierenden Reversion mit maximalem Score erzeugt keine neuen unorientierten Komponenten.*

Weiterhin gilt als Folge der Definition 6.58 zusammen mit Theorem 6.48

Korollar 6.61 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation. Eine orientierte Reversion $\bar{\rho}$ ist genau dann sicher, wenn*

$$\bar{c}'(\bar{\pi} \circ \bar{\rho}) - \bar{h}(\bar{\pi} \circ \bar{\rho}) = \bar{c}'(\bar{\pi}) - \bar{h}(\bar{\pi}) + 1.$$

Aus diesem Korollar folgt zusammen mit Theorem 6.59 sofort die Behauptung aus Theorem 6.52.

6.4.7 Eliminierung von Hurdles (+)

In diesem Abschnitt wollen wir nun beschreiben wie wir mit Hurdles umgehen.

Definition 6.62 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation. Eine orientierte Reversion, die auf zwei Reality-Edges in zwei verschiedenen, nicht benachbarten Hurdles operiert, wird als* Hurdle-Merging *bezeichnet.*

Ein solches Hurdle-Merging ist in Abbildung 6.51 schematisch dargestellt. Hierbei ist links die Kreis-Darstellung des Reality-Desire-Diagrams angegeben, wobei Kreise dort Hurdles darstellen. Im rechten Teil ist die Intervall-Darstellung angegeben, oben vor und unten nach der orientierten Reversion.

Wir wir uns schon überlegt haben, macht eine orientierte Reversion, die auf zwei Kreisen der beiden Hurdles operiert, aus diesen beiden Kreisen einen. Weiterhin sorgt eine solche orientierte Reversion dafür, dass eine orientierte Desire-Edge entsteht. Somit nimmt nach einem Hurdle-Merging die Zahl der Hurdles um 2 ab, während

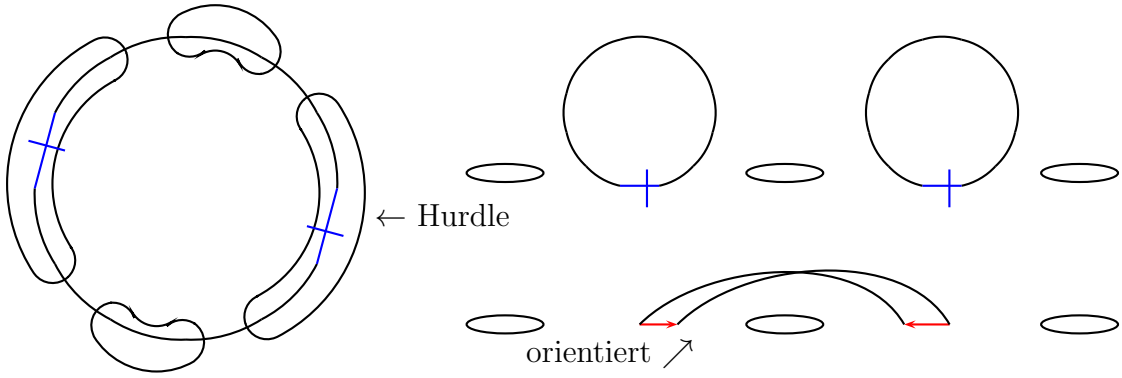


Abbildung 6.51: Skizze: Schematische Darstellung eines Hurdle-Merging

die Anzahl der Kreise im erweiterten Reality-Desire-Diagram um eins sinkt. Somit gilt aber insgesamt

$$(\bar{h}(\bar{\pi} \circ \bar{\rho}) - \bar{c}'(\bar{\pi} \circ \bar{\rho})) - (\bar{h}(\bar{\pi}) - \bar{c}'(\bar{\pi})) \leq -1.$$

Weiterhin kann man sich überlegen, dass die Eigenschaft ein Fortress zu sein unverändert bleibt, da keine neuen Super-Hurdles generiert werden und zwei Super-Hurdles eliminiert wurden (also die ungerade Anzahl an Hurdles, die alle Super-Hurdles sind, unverändert bleibt).

Hierbei werden nur nicht benachbarte Hurdles gemischt, ansonsten könnten beim Mischen von benachbarten Hurdles neue Hurdles entstehen. Siehe hierzu auch Abbildung 6.52. Würde man hier die Hurdles A und B mischen, würde aus der unorientierten Komponente C eine neue Hurdle. Durch das Mischen nicht benachbarter Hurdles wird dies vermieden.

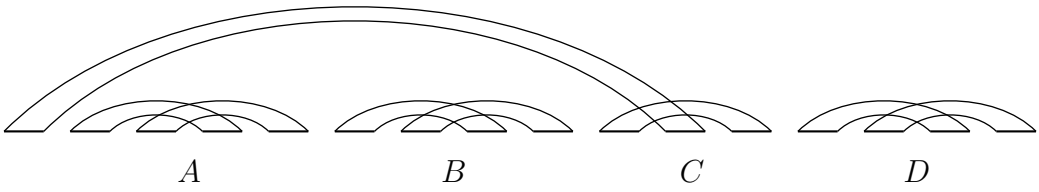


Abbildung 6.52: Skizze: Mögliche Probleme beim Hurdle-Merging

Auch über die Enden hinaus kann es benachbarte Hurdles geben, siehe Abbildung 6.53. Würde man hier die Hurdles A und D mischen, würde aus der unorientierten Komponente C eine neue Hurdle. Durch das Mischen nicht benachbarter Hurdles wird dies vermieden. Man kann also ein zielführendes Hurdle-Merging nur dann durchführen, wenn $\bar{h}(\bar{\pi}) \geq 4$ ist, da ansonsten jedes Paar von Hurdles benachbart wäre.

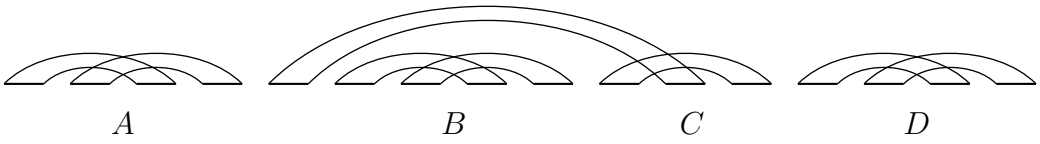


Abbildung 6.53: Skizze: Mögliche Probleme beim Hurdle-Merging

Technisch werden wir die orientiert Reversion zu den beiden Reality-Edges auswählen, die von den beiden nicht benachbarten Hurdles am nächsten beieinanderliegen. Nach dieser orientierten Reversion schneiden sich diese beiden inneren Desire-Edges und schneiden sich auch mit allen Desire-Edges, die nicht vollständig mit der

orientierten Reversion mitgedreht werden. Zusätzlich ändern sich die Eigenschaft das Schneiden dieser Desire-Edges, die von der orientierten Reversion nur teilweise gedreht werde auch untereinander. So kann man den Overlap-Graph relativ einfach aktualisieren.

Definition 6.63 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation. Eine orientierte Reversion, die auf zwei Reality-Edges einer Hurdle operiert, die durch eine Desire-Edge verbunden sind, wird als* Hurdle-Cutting *bezeichnet.*

Wir betrachten zunächst eine Hurdle, die ja eine unorientierte Komponente sein muss. Zuerst überlegt man sich leicht, dass diese eine Desire-Edge mit der Ausrichtung der inzidenten Reality-Edges, wie in Abbildung 6.54 angegeben, existieren muss.

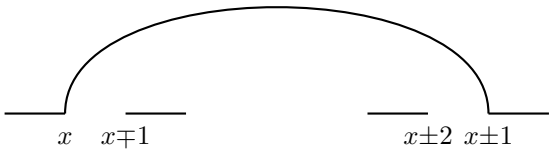


Abbildung 6.54: Skizze: Existenz einer schneidende Desire-Edge in einer Hurdle

Angenommen diese Desire Edge verbindet x und $x+1$ (bzw. $x-1$). Dann folgt man der Folge $(x, x-1, x-2, \ldots, 0)$, die alternierend aus Desire-Edges und benachbarten Knoten, die zu Strips von orientierten Werten gehören, beschreiben. Dieser Pfad verläuft zunächst innerhalb der betrachteten Desire-Edge und muss irgendwann einmal aus dem Inneren dieser Desire-Edge ausbrechen. Dies kann nur über eine andere Desire-Edge geschehen, die dann die betrachtete schneiden muss. Also enthält eine Hurdle mindestens zwei sich schneidende Desire-Edges, wie in Abbildung 6.55 illustriert.

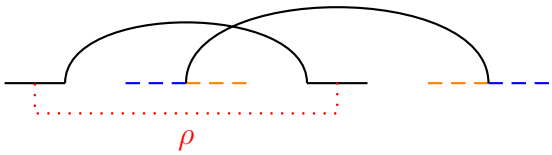


Abbildung 6.55: Skizze: Sich schneidende Desire-Edges in einer Hurdle

Wendet man jetzt die Reversion ρ (wie in Abbildung 6.55 angegeben) an, so sieht man leicht, dass die andere schneidende Desire-Edge anschließend orientiert sein muss, da die zur schneidenden Desire-Edge inzidenten blauen (bzw. orangefarbenen) Reality-Edges nun gegenläufig durchlaufen werden.

Also wird auch hier eine Hurdle eliminiert, da ein orientierter Kreis generiert wird. Die Anzahl der Zyklen steigt auch hier um 1. Ist beim Hurdle-Cutting die betroffene Hurdle einfach, dann entsteht durch Auflösung keine neue Hurdle (im Gegensatz zu Super-Hurdles). Damit bekommt man bei $\bar{h}(\bar{\pi}) = 3$ Probleme, wenn eben alle Hurdles Super-Hurdles sind. Dann ist die betroffene Permutation aber ein Fortress und man darf sich eine zusätzliche orientierte Reversion erlauben, die durch Generierung einer Hurdle auch benötigt wird.

In diesem Fall entspricht das Hurdle-Cutting einer orientierten Reversion, die an den beiden adjazenten Reality-Edges $v \in V(OV(\bar{\pi}))$ operiert. Im Overlap-Graphen entspricht dies einer Komplementierung des Teilgraphen auf den Knoten von $N(v)$ in $OV(\bar{\pi})$ (und nicht wie bei einer orientierten Desire-Edge dem Komplementieren des Teilgraphen auf $\bar{N}(v) = N(v) \cup [\{v\})$.

Somit haben wir für $\bar{h}(\bar{\pi}) \geq 3$ (sofern es sich nicht um drei Super-Hurdles handelt) eine Strategie kennen gelernt, die die untere Anzahl Schranke für die benötigte Anzahl an orientierten Reversionen respektiert. Bei genau zwei Hurdles wenden wir ebenfalls wieder ein Hurdle-Merging an, da wir hierbei keine neue Hurdle erzeugen können (der Leser möge sich überlegen warum). Bei einer Hurdle bleibt nichts anderes als ein Hurdle-Cutting übrig. Man überlegt sich hier leicht, dass diese dann eine einfache Hurdle sein muss und keine Super-Hurdle sein kann. Somit können wir auch hier keine neue Hurdle generieren.

6.4.8 Algorithmus für Min-SOR

Zusammenfassend aus den Ergebnisse der vorangegangenen Abschnitte erhalten wir also den in Abbildung 6.56 angegeben Algorithmus zum Sortieren mit orientierten Reversionen im Pseudo-Code. Wir eliminieren also immer zuerst alle orientierten Komponenten und wenden uns dann unorientierten Komponenten zu, die dann allerdings neue orientierte Komponenten generieren können.

Halten wir zum Abschluss das erzielte Ergebnis fest, wobei wir den Beweis der Laufzeit dem Leser überlassen.

Theorem 6.64 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation. Zum Sortieren von $\bar{\pi}$ sind genau $d(\bar{\pi}) = n + 1 - \bar{c}'(\bar{\pi}) - \bar{h}(\bar{\pi}) + \bar{f}(\bar{\pi})$ orientierte Reversionen nötig und ausreichend. Eine optimale sortierende Folge von orientierten Reversionen kann in polynomieller Zeit gefunden werden, d.h. Min-SOR $\in \mathcal{P}$.*

SOR (permutation $\bar{\pi}$)

```
begin
    while (π̄ ≠ id) do
        if (OV(π̄) contains an oriented vertex) then
            apply an oriented reversal (corresponding to an oriented desire-edge)
            with maximal score;
        else if (h̄(π̄) ≥ 4) then
            use hurdle-merging of two non-adjacent hurdles;
        else if (h̄(π̄) = 2) then
            use hurdle-merging of these two hurdles;
        else if ((h̄(π̄) ∈ {1, 3}) && (there is a simple hurdle)) then
            use hurdle-cutting on a simple hurdle;
        else /* π must be a fortress */
            use hurdle-merging on two (adjacent) hurdles;
end
```

Abbildung 6.56: Algorithmus: Pseudo-Code für Min-SOR

Zum Abschluss noch ein paar Bemerkungen zu Verbesserungen des angegebenen Algorithmus in der wissenschaftlichen Literatur, insbesondere zu den Laufzeiten:

- Die Laufzeit des ersten vorgeschlagenen Algorithmus von Hannenhalli und Pevzner beträgt $O(n^4)$.
- Eine Verbesserung von Berman und Hannenhalli benötigt $O(n^2\alpha(n))$, wobei $\alpha(n)$ die inverse der Ackermann-Funktion ist (für die Praxis quasi eine Konstante).
- Kaplan, Shamir und Tarjan haben eine weitere Laufzeitverbesserung entwickelt, die einen Zeitbedarf von $O((\bar{d}(\bar{\pi}) + \alpha(n)) \cdot n) = O(n^2)$ besitzt.
- Bader, Moret und Yan konnten die Laufzeit auf $O(\bar{d}(\bar{\pi}) \cdot n) = O(n^2)$ senken. Für die reine Berechnung der Reversal-Distanz beträgt die Lauftzeit sogar nur $O(n)$ entwickelt.
- Eine vereinfachte Darstellung zu Berechnung der Reversal-Distanz (ohne die Bestimmung der sortierenden Reversionen) wurde von Bergeron, Mixtacki und Stoye ebenfalls mit Laufzeit $O(n)$.

Man beachte, dass man in der Regel die Eingabe-Sequenz durch Zusammenfassen von Strips (Bereichen ohne Breakpoints) so modifiziert, dass $\bar{d}(\bar{\pi}) \geq \bar{b}(\bar{\pi})/2 = n/2$ gilt, also $\bar{d}(\bar{\pi}) = \Theta(n)$. Dies stellt keine Einschränkung dar.

28.01.25

6.5 Sorting by Transpositions (*)

In diesem Abschnitt wollen wir nun die Distanz zwischen zwei Genomen bestimmen, wenn nur Transpositionen als elementare Operationen zugelassen sind.

6.5.1 Elementare Definitionen

Zuerst halten wir in diesem Abschnitt noch ein paar grundlegende Definitionen und Beobachtungen fest. Wir bemerken als erstes, dass wir jetzt wieder unorientierte Permutationen betrachten, da wir mit einer Transposition nicht die Orientierung eines Teilstückes ändern können.

Sorting by Transpositions (Min-SBT)
Eingabe: Eine Permutation $\pi \in S_n$.
Gesucht: Eine kürzeste Folge von Transpositionen $\tau_1, \ldots, \tau_k$ mit $\pi \circ \tau_1 \circ \cdots \circ \tau_k = \mathrm{id}$.

Im Folgenden bezeichnen wir mit d_T die *Transpositions-Distanz*. Wie im Falle orientierter Permutationen definieren wir noch den Begriff einer zugehörigen unorientierten Permutation, auch wenn dies zunächst sehr seltsam klingen mag.

Definition 6.65 *Sei $\pi \in S_n$ eine Permutation, dann ist $\pi' \in S_{2n}$ die* zugehörige unorientierte Permutation*, wobei $\pi' = (\pi'_1, \ldots, \pi'_{2n})$ mit $(\pi'_{2i-1}, \pi'_{2i}) = (2\pi_i - 1, 2\pi_i)$.*

Auch hier werden wir im Folgenden die zugehörige unorientierte Permutation immer als erweiterte zugehörige unorientierte Permutation $\pi' = (0, \pi'_1, \ldots, \pi'_{2n}, 2n{+}1)$ interpretieren. Damit können wir analog zu den orientierten Permutation das Reality-Desire-Diagram definieren.

Definition 6.66 *Sei $\pi \in S_n$ eine erweiterte Permutation und sei π' die zugehörige unorientierte Permutation. Dann ist das* (erweiterte) Reality-Desire-Diagram *(kurz RDD) $G(\pi)$ bzw. $G'(\pi)$ der (erweiterte) Breakpoint-Graph $G(\pi')$ bzw. $G'(\pi')$ mit der Einschränkung, dass es für jedes $i \in [1:n]$ keine Reality- bzw. Desire-Edges zwischen $(2i-1)$ und $(2i)$ gibt.*

In Abbildung 6.57 ist ein Beispiel eines (erweiterten) Reality-Desire-Diagrams für die erweiterte Permutation $(0, 7, 4, 1, 3, 2, 6, 5, 8)$ angegeben.

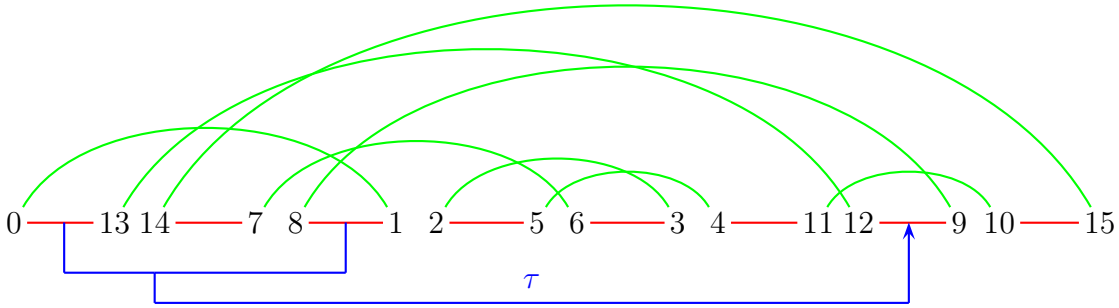


Abbildung 6.57: Beispiel: Reality-Desire-Diagram für $(0, 7, 4, 1, 3, 2, 6, 5, 8)$

In Abbildung 6.58 ist ein Beispiel angegeben, wie sich die Transposition $(1, 2, 6)$ auf die erweiterte Permutation $(0, 7, 4, 1, 3, 2, 6, 5, 8)$ im (erweiterten) Reality-Desire-Diagram für die zugehörige unorientierte Permutation auswirkt. Weiterhin ist dort in blau die Zyklenzerlegung in alternierende Kreise angegeben.

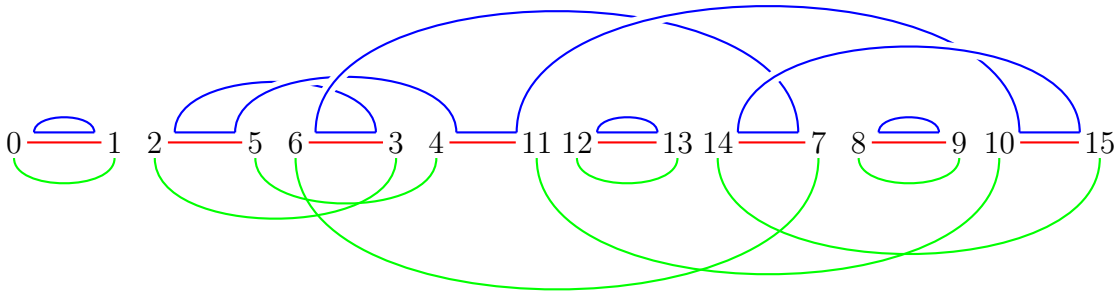


Abbildung 6.58: Beispiel: Erweitertes Reality-Desire-Diagram für die Permutation $(0, 7, 4, 1, 3, 2, 6, 5, 8)$ nach der Transposition $\tau = (1, 2, 6)$

6.5.2 Untere Schranken für die Transpositions-Distanz

Zunächst wollen wir aus dem Reality-Desire-Diagram eine untere Schranke für die mindestens benötige Anzahl von Transpositionen herleiten. Da jede Transposition auf maximal drei Breakpoints operieren kann und somit auch maximal drei Breakpoints eliminieren kann, gilt die folgende Beobachtung.

Beobachtung 6.67 *Für jede Permutation $\pi \in S_n$ gilt $d_T(\pi) \geq b(\pi)/3$.*

Mit Hilfe des folgenden Lemmas können wir eine bessere untere Schranke herleiten.

Lemma 6.68 *Sei $\pi \in S_n$ eine erweiterte Permutation und sei τ eine Transposition auf S_n. Dann gilt $c'(\pi \circ \tau) - c'(\pi) \in \{-2, 0, 2\}$*

Der Beweis erfolgt durch eine einfache Fallunterscheidung unter Berücksichtigung der Gestalt von Kreisen im erweiterten Reality-Desire-Diagram, wie sie im folgenden Abschnitt erläutert werden, und sei dem Leser als Übungsaufgabe überlassen.

Theorem 6.69 *Für jede Permutation $\pi \in S_n$ gilt $d_T(\pi) \geq \frac{(n+1)-c'(\pi)}{2} = \frac{b(\pi)-c(\pi)}{2}$.*

Diese Schranke lässt sich noch verbessern, wenn man nur Kreise ungerader Länge betrachtet (d.h. mit einer ungeraden Anzahl von Reality- bzw. Desire-Edges).

Notation 6.70 *Sei $\pi \in S_n$ eine erweiterte Permutation. Dann bezeichnet $\hat{c}(\pi)$ die Anzahl der Kreise in $G'(\pi)$, die eine ungerade Anzahl von Reality-Edges (oder Desire-Edges) besitzen.*

Korollar 6.71 *Sei $\pi \in S_n$ eine erweiterte Permutation und sei τ eine Transposition auf S_n. Dann gilt $\hat{c}(\pi \circ \tau) - \hat{c}(\pi) \in \{-2, 0, 2\}$*

Der folgende Satz folgt aus der Tatsache, dass die Identität im erweiterten Reality-Desire-Diagram aus genau $n+1$ Kreise der Länge 1 (wenn man nur Reality- oder Desire-Edges bei der Länge mitzählt) besteht.

Theorem 6.72 *Für jede Permutation $\pi \in S_n$ gilt $d_T(\pi) \geq \frac{(n+1)-\hat{c}(\pi)}{2} = \frac{b(\pi)-\hat{c}(\pi)}{2}$.*

6.5.3 Orientierte und Unorientierte Desire-Edges

Da das (erweiterte) Reality-Desire-Diagram nur aus unorientierten Permutationen erstellt wird (im Gegensatz zu Min-SOR), treten die Paare $(2i-1, 2i)$ für den Wert $+i$ in der zugehörigen unorientierten Permutation nur in dieser Reihenfolge auf. Somit sind im Reality-Desire-Diagram auch nur Desire-Edges bestimmter Gestalt möglich, wie in Abbildung 6.59 dargestellt (die rot durchgestrichenen können nicht auftreten).

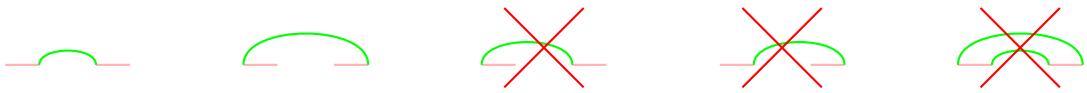


Abbildung 6.59: Skizze: Vorkommende Desire-Edges

Der rechteste Fall in Abbildung 6.59 illustriert, dass zwischen zwei Reality-Edges, die mit zwei Desire-Edges verbunden sind, mindestens eine weitere Reality-Edge

vorkommen muss. Andernfalls hätte die entsprechende Teilsequenz der zugehörigen unorientierte Permutation die Form: $(2i-1, 2i, 2j-1, 2j, 2i+1, 2i+2)$. Da wir aber im Reality-Desire-Diagram zwischen $2j-1$ und $2j$ keine Desire-Edge haben, kann dies nicht eintreten. Damit haben wir folgende Bobachtung gezeigt.

Beobachtung 6.73 *Sei $\pi \in S_n$ eine erweiterte Permutation und sei $G'(\pi)$ das zu π gehörige erweiterte Reality-Desire-Diagram. Seien (π_i, π_{i+1}) und (π_j, π_{j+1}) mit $i+1 < j \in [0 : 2n+1]$ zwei Reality-Edges, deren äußere oder innere Endpunkte mit einer Desire-Edge verbunden sind, dann existiert eine weitere Reality-Edge (π_k, π_{k+1}) mit $k \in [i+1 : j-2]$.*

Beim Sortieren mit Transpositionen ist die Definition von orientierten Kreisen etwas anders als beim Sortieren mit Reversionen, da alle Reality-Edges automatisch immer in derselben Richtung durchlaufen werden.

Definition 6.74 *Sei $\pi \in S_n$ eine erweiterte Permutation und $G'(\pi)$ das zugehörige erweiterte Reality-Desire-Diagram. Ein Kreis C in $G'(\pi)$ heißt* orientiert*, wenn alle bis auf eine Desire-Edge (die zu $\pi_{\max(C)}$ inzident ist) nicht in derselben Richtung durchlaufen werden. Andernfalls heißt der Kreis* orientiert*.*

Wir merken hier noch an, dass ein Kreis aus einer Reality- und einer Desire-Edge nach dieser Definition unorientiert ist. In Abbildung 6.60 ist links ein unorientierter Kreis und in der Mitte sowie rechts jeweils ein orientierter Kreis schematisch dargestellt. Man beachte, dass alle unorientierten Kreise eine Gestalt wie der lin-

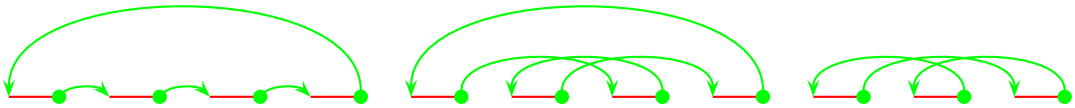


Abbildung 6.60: Skizze: Orientierte und unorientierte Kreise im RDD

keste Kreis in Abbildung 6.60 haben müssen. Wenn nämlich die Desire-Edge mit dem rechtesten Ende von rechts nach links durchlaufen wird, müssen aufgrund der Unorientiertheit alle andere Desire-Edges von links nach rechts durchlaufen werden.

6.5.4 Eine 2-Approximation

In diesem Abschnitt wollen wir einen Approximationsalgorithmus zur Lösung für Min-SBT entwickeln. Zunächst geben wir die Definition von *x-Moves* an.

Definition 6.75 *Sei $\pi \in S_n$ eine erweiterte Permutation. Eine Transposition τ auf π heißt* 2-Move*,* 0-Move *bzw.* -2-Move*, wenn $c(\pi \circ \tau) - c(\pi)$ gleich 2, 0 bzw. -2 ist.*

Vermutlich sollte man beim Sortieren mit Transpositionen nach Möglichkeit nur 2-Moves verwenden. Wir merken aber bereits jetzt an, dass es noch ungeklärt ist, ob eine optimale Lösung ohne -2-Moves auskommen kann.

Das folgende Lemma wird zentral für unseren Approximationsalgorithmus sein.

Lemma 6.76 *Sei $\pi \in S_n$ eine erweiterte Permutation. Besitzt $G'(\pi)$ einen orientierten Kreis, dann existiert ein 2-Move τ.*

Beweis: Nach Voraussetzung gibt es also einen orientierten Kreis. Neben der am weitesten rechts beginnenden Desire-Edge muss eine weitere von rechts nach links durchlaufen werden. Dies ist in Abbildung 6.61 oben dargestellt, wobei wir dort unterscheiden, ob die zweite Desire-Edge vollständig innerhalb der ersten Desire-Edge (links) verläuft oder nicht (rechts). Da alle Reality-Edges von links nach rechts durchlaufen werden, ist die Form dieser zweiten Desire-Edge auch klar.

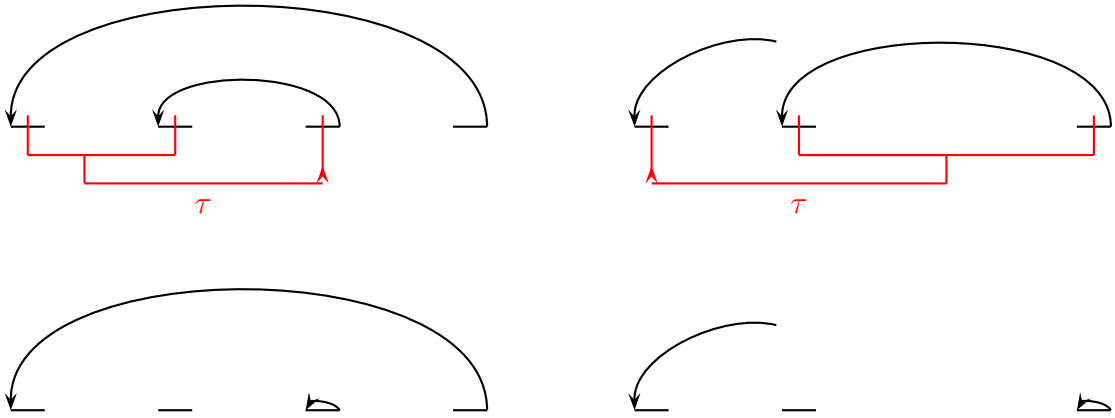


Abbildung 6.61: Skizze: Orientierter Kreis in $G'(\pi)$)

Verläuft die zweite Desire-Edge vollständig innerhalb der ersten, wie links in Abbildung 6.61 dargestellt, dann wenden wir die dort angegebene Transposition an. Andernfalls liegt die Situation vor, wie im rechten Teil der Abbildung 6.61 skizziert. Dann wenden wir die dort angegebene Transposition an. Man beachte, dass beide Transpositionen jeweils auf einem Kreis operieren, d.h. die Anzahl der Kreise in keinem Fall sinken kann. In jedem Fall wird aber nach Ausführung der jeweiligen Transposition die Anzahl der Kreise in $G'(\pi \circ \tau)$ um mindestens eins erhöht, da wir ja eine Adjazenz erzeugen. Nach Lemma 6.68 muss dann aber $G'(\pi \circ \tau)$ genau zwei Kreise mehr besitzen. Also ist τ ein 2-Move. ■

Lemma 6.77 *Sei $\pi \in S_n$ eine erweiterte Permutation. Besitzt $G'(\pi)$ keinen orientierten Kreis, dann existiert kein 2-Move τ auf π.*

Den Beweis dieses Lemmas überlassen wir dem Leser als Übungsaufgabe.

Lemma 6.78 *Sei $id \neq \pi \in S_n$ eine erweiterte Permutation. Dann besitzt π einen 2-Move oder es existieren zwei Transposition τ und τ', wobei τ ein 0-Move für π und τ' ein 2-Move für $\pi \circ \tau$ ist.*

Beweis: Enthält $G'(\pi)$ einen orientierten Kreis, dann folgt die Behauptung aus dem Lemma 6.76. Also seien alle Kreise in $G'(\pi)$ unorientiert. Wir erinnern noch einmal an die Beobachtung, dass unorientierte Kreise die Gestalt der linkesten Illustration in Abbildung 6.60 besitzen müssen.

Wir betrachten einen unorientierten Kreis C in $G'(\pi)$, der mindestens zwei Desire-Edges enthält (sonst wäre die Permutation bereits sortiert). Weiterhin muss es eine Desire-Edge zwischen $\max(C)$ und $\min(C)$ geben, da sonst C orientiert wäre. Weiter gilt $(\min(C), \max(C)) \neq (0, n+1)$.

Sei $k = \max(C)$ und sei also $(\pi_k - 1, \pi_k)$ die am weitesten rechts endende Desire-Edge in C (die dann auch in $\min(C)$ beginnen muss, da C unorientiert ist). Wir setzen zunächst $i = k$. Dann betrachten wir die von rechts her nächste darin beginnende bzw. endende Desire-Edge. Ist (π_{i-1}, π_i) eine Reality-Edge, dann hat diese die Form $(\pi_j - 1, \pi_j) = (\pi_{i-1} - 1, \pi_{i-1})$ und sonst $(\pi_j + 1, \pi_j) = (\pi_i - 1, \pi_i - 2)$. Diese beiden Fälle sind in Abbildung 6.62 illustriert.

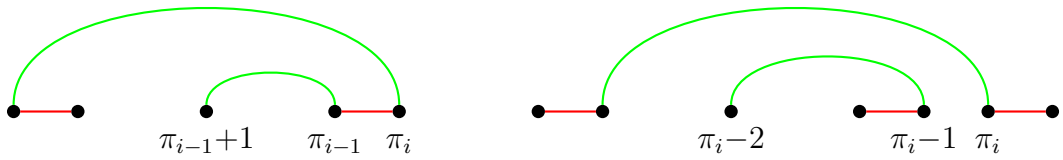


Abbildung 6.62: Skizze: Mögliche nächstinnere Desire-Edges

Schneidet $(\pi_j - 1, \pi_j)$ bzw. $(\pi_j + 1, \pi_j)$ keine der bislang betrachteten Desire-Edges, so setzen wir $i = j$ und führen dieses Verfahren fort. Andernfalls untersuchen wir die zu π_j inzidente Desire-Edge genauer. Irgendwann müssen wir eine Desire-Edge finden, die aus dem betrachteten Intervall ausbricht (sonst würde die betrachtete unorientierte Permutation nicht aus einer Permutation mit ausschließlich positiven Vorzeichen stammen).

Für die so gefundenen ausbrechende Desire-Edge unterscheiden wir zwei Fälle, je nachdem, wie die zu π_j inzidente Desire-Edge aussieht.

Fall 1: Wir nehmen zuerst an, dass die zu π_j inzidente Kante zu einer Position größer als k führt. Man beachte, dass die betrachte Desire-Edge nicht an den Positionen

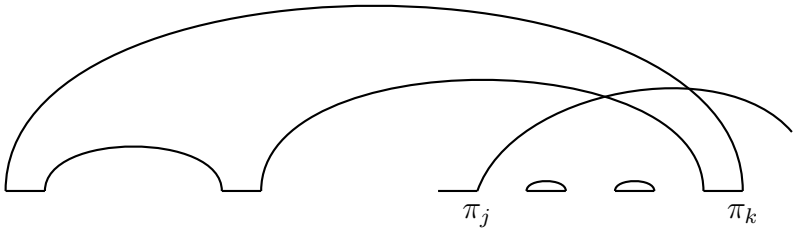


Abbildung 6.63: Skizze: Fall 1: Eine Kante bricht nach rechts aus

$i+1$ mit k enden kann und dass diese Desire-Edge zu einem anderen Kreis gehören muss. Dies ist in Abbildung 6.63 illustriert.

Diese Situation ist in Abbildung 6.64 etwas genauer dargestellt. Dort ist (f, i) die ausbrechende Kante und die rot-gepunkteten Linien entsprechend alternierenden Pfaden im Reality-Desire-Diagram.

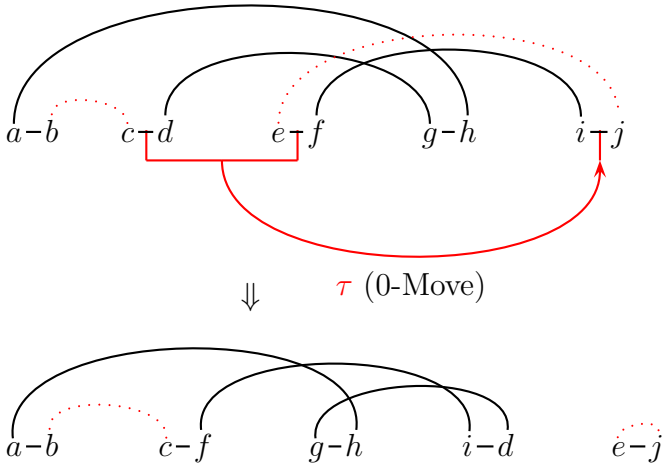


Abbildung 6.64: Skizze: Fall 1: 0-Move der einen orientierten Kreis generiert

Wie man der Abbildung 6.64 entnimmt, gibt es eine Transposition τ, die ein 0-Move ist: Da diese Transposition auf zwei Kreisen operiert und mindestens zwei Kreise übrig lässt (unter anderem erzeugen wir ja die Adjazenz (e, j)), muss es sich um einen 0-Move oder 2-Move handeln (nach Lemma 6.77 ist letzteres aber nicht möglich). Da nach der Ausführung von τ ein orientierter Kreis generiert wurde, gibt es nach Lemma 6.76 jetzt einen 2-Move.

Fall 2: Jetzt nehmen wir an, dass diese Kante nach links ausbricht und eine der vorher betrachteten Desire-Edges schneidet. Dies ist in Abbildung 6.65 illustriert. Hierbei ist jetzt jedoch nicht klar, ob der andere Endpunkt noch im aufspannenden Intervall des zugehörigen Kreises liegt oder nicht. Auch hier muss die schneidende Kante wieder zu einem andere Kreis gehören

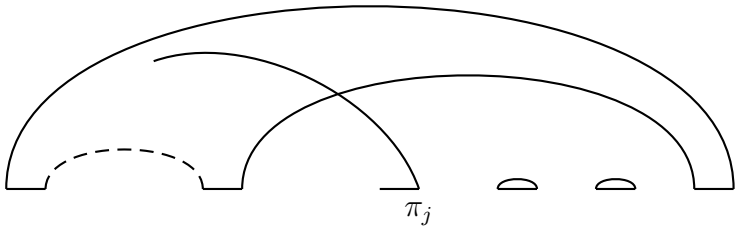


Abbildung 6.65: Skizze: Fall 2: Desire-Edge bricht nach links aus

Wir betrachten diese Situation jetzt in Abbildung 6.66 genauer. Hier ist die Desire-Edge (e, h) eine der zuletzt betrachteten Kanten, die von der aktuellen Kante (c, g) geschnitten wird. Man beachte, dass c auch links von a liegen kann.

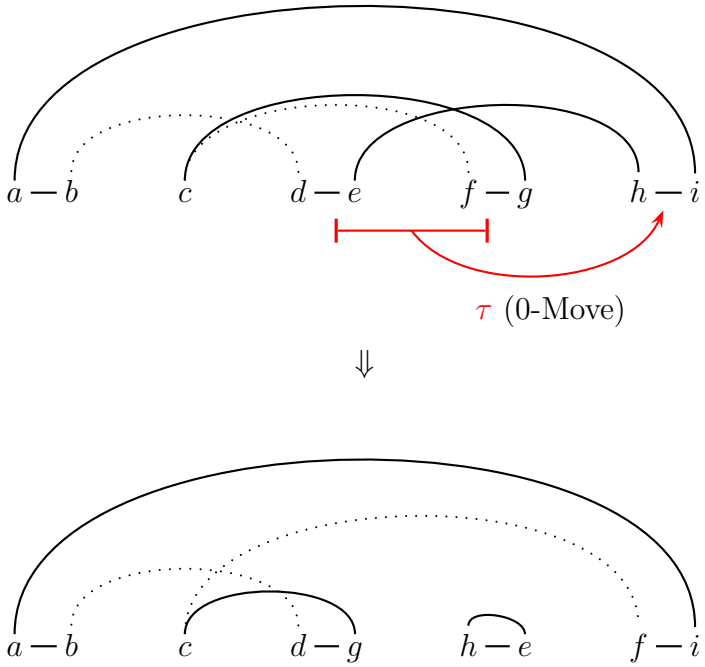


Abbildung 6.66: Skizze: Fall 2: 0-Move der einen orientierten Kreis generiert

Wie man der Abbildung 6.66 entnimmt, gibt es eine Transposition τ, die ein 0-Move ist: Da diese Transposition auf zwei Kreisen operiert und mindestens zwei Kreise übrig lässt (unter anderem erzeugen wir ja die Adjazenz (h, e)), muss es sich um einen 0-Move oder 2-Move handeln (nach Lemma 6.77 ist letzteres aber nicht möglich). Da nach der Ausführung von τ ein orientierter Kreis generiert wurde, gibt es nach Lemma 6.76 jetzt einen 2-Move. ■

Aus diesem Lemma folgt sofort, dass nach maximal 2 Transpositionen ein 2-Move erfolgt und dass ansonsten nur 0-Moves erfolgen können. Es sind also maximal

$(n+1) - c'(\pi)$ Transpositionen nötig, um π zu sortieren. Damit haben wir den folgenden Satz bewiesen.

Theorem 6.79 *Es existiert eine polynomielle 2-Approximation für Min-SBT.*

Zum Abschluss dieses Abschnitts wollen wir noch auf die folgenden Verbesserungen bei der Approximationsgüte hinweisen.

- Man kann zeigen, dass nach einem 0-Move mindestens zwei 2-Moves möglich sind, sofern $G'(\pi)$ lange orientierte Kreise mit mindestens 3 Reality-Edges enthält. Damit ist eine Approximation mit Güte 1,75 erreichbar.
- Eine Approximation mit Güte von 1,5 kann in Zeit $O(n^2)$ erstellt werden. Für Details verweisen wir auf die Originalarbeit von Hartman und Shamir.
- Eine Approximation mit der bislang besten bekannten Güte von 1,375 kann in Zeit $O(n^2)$ erstellt werden. Für Details verweisen wir auf die Originalarbeit von Elias und Hartman. Für eine schnellere Variante in Zeit $O(n \log(n))$ mit derselben Approximationsgüte verweisen wir auf die Arbeiten von Cunha, Kowada, Hausen und de Figueiredo.
- Es ist bislang noch ungeklärt, ob eine optimale Lösung -2-Moves enthalten kann oder nicht.
- Bislang ist auch noch unbekannt, ob Min-SBT in $\mathcal{P}$ ist oder ob das Problem bereits $\mathcal{NP}$-hart ist.

6.6 Sorting by Transpositions und Reversals (*)

In diesem Abschnitt wollen wir jetzt die Distanz von zwei Permutationen ermitteln, wenn sowohl Reversionen als auch Transpositionen erlaubt sind.

6.6.1 Problemstellung

In diesem Abschnitt erlauben wir neben den normalen Operationen Transposition und Reversionen teilweise auch noch so genannte *Transversals*. Hierbei handelt es sich um eine Transposition, wobei anschließend einer der beiden vertauschten Teile noch umgedreht wird. Weiter erlauben wir teilweise noch *doppelte Reversionen*, d.h.

es werden gleichzeitig zwei benachbarte Teilstücke umgedreht ohne sie zu vertauschen. Auch diese Transversals bzw. doppelten Reversionen können wir in Form von Permutation angegeben:

$$\begin{aligned}
\text{Transversal}_1(i,j,k) &= \begin{pmatrix} 1\cdots i-1 & i\cdots j & j+1\cdots k & k+1\cdots n \\ 1\cdots i-1 & j+1\cdots k & j\cdots i & k+1\cdots n \end{pmatrix}, \\
\text{Transversal}_2(i,j,k) &= \begin{pmatrix} 1\cdots i-1 & i\cdots j & j+1\cdots k & k+1\cdots n \\ 1\cdots i-1 & k\cdots j+1 & i\cdots j & k+1\cdots n \end{pmatrix}, \\
\text{DReversal}(i,j,k) &= \begin{pmatrix} 1\cdots i-1 & i\cdots j & j+1\cdots k & k+1\cdots n \\ 1\cdots i-1 & j\cdots i & k\cdots j+1 & k+1\cdots n \end{pmatrix}.
\end{aligned}$$

Dabei werden wir im Folgenden bei einer Reversion innerhalb eines Transversals oder einer doppelten Reversion auch immer die Vorzeichen ändern, da wir hier nur orientierte Permutationen untersuchen werden. In Abbildung 6.67 ist ein Beispiel für eine sortierende Folge von Operationen angegeben, die auch ein Transversal und eine doppelte Reversion verwendet.

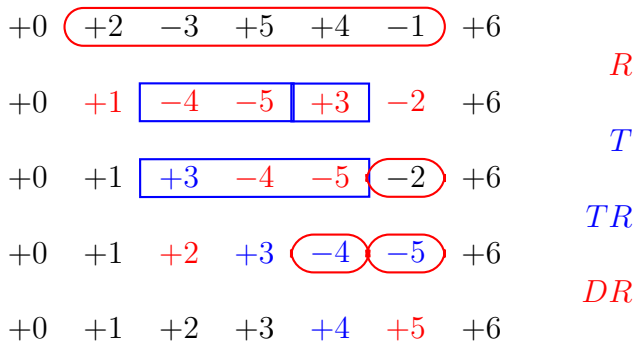


Abbildung 6.67: Beispiel: Sortierung mit Transversals und doppelter Rotation

Somit können wir nun die zu untersuchenden Probleme genau definieren.

Sorting by Reversals and Transpositions (Min-SBRT)

Eingabe: Eine Permutation $\pi \in S_n$.
Gesucht: Eine kürzeste Folge von Reversionen und Transpositionen $\rho_1, \ldots, \rho_k$ mit $\pi \circ \rho_1 \circ \cdots \circ \rho_k = \mathrm{id}$.

Sorting by Transversals (Min-SBTR)

Eingabe: Eine Permutation $\pi \in S_n$.
Gesucht: Eine kürzeste Folge von Reversionen, Transpositionen, Transversals und doppelten Reversionen $\rho_1, \ldots, \rho_k$ mit $\pi \circ \rho_1 \circ \cdots \circ \rho_k = \mathrm{id}$.

Notation 6.80 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation. d_{TR} bezeichnet die minimale Anzahl von Reversionen und Transpositionen, die nötig sind um $\bar{\pi}$ zu sortieren. d'_{TR} bezeichnet die* Transversal-Distanz*, d.h. die minimale Anzahl von Reversionen, Transpositionen, Transversals und doppelten Reversionen, die nötig sind um $\bar{\pi}$ zu sortieren*

Weiterhin kürzen wir die Operationen Reversionen, Transpositionen, Transversals bzw. doppelte Reversionen oft mit T, R, TR bzw. DR ab. Man überlege sich ferner, warum man bei Transversals nicht die Reversion beider Teile nach einer Transposition zulässt. Wir wiederholen auch hier noch einmal kurz zur Erinnerung die Definition eines Reality-Desire-Diagrams.

Definition 6.81 *Sei $\bar{\pi} \in \bar{S}_n$ eine erweiterte orientierte Permutation. Dann ist das* (erweiterte) Reality-Desire-Diagram *(kurz RDD) $G(\bar{\pi})$ bzw. $G'(\bar{\pi})$ der (erweiterte) Breakpoint-Graph $G(\pi)$ bzw. $G'(\pi)$, wobei π die zu $\bar{\pi}$ gehörige erweiterte unorientierte Permutation ist, mit der Einschränkung, dass es für jedes $i \in [1:n]$ keine Reality- bzw. Desire-Edges zwischen $(2i-1)$ und $(2i)$ gibt.*

6.6.2 Untere Schranken für die Transversal-Distanz

Nun wollen wir eine untere Schranke für das orientierte Sortieren mit Transversals herleiten.

Lemma 6.82 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $\bar{\rho}$ eine Operation vom Typ R, T, TR_1, TR_2 oder DR, dann gilt:*

$$|\bar{b}(\bar{\pi} \circ \bar{\rho}) - \bar{c}(\bar{\pi} \circ \bar{\rho}) - (\bar{b}(\bar{\pi}) - \bar{c}(\bar{\pi}))| \leq 2 \qquad \textit{bzw.} \qquad |\bar{c}'(\bar{\pi} \circ \bar{\rho}) - \bar{c}'(\bar{\pi} \circ \bar{\rho})| \leq 2.$$

Beweis: Auch hier gilt immer noch $\bar{c}'(\bar{\pi}) = \bar{c}(\pi)+(n+1)-\bar{b}(\bar{\pi})$. Wir müssen also nur $|\bar{c}'(\bar{\pi} \circ \bar{\rho}) - \bar{c}'(\bar{\pi})| \leq 2$ zeigen. Es sind in jeder Operation jedoch maximal drei Kreise involviert, d.h. im schlimmsten Fall werden aus einem Kreis drei Kreise bzw. aus drei Kreisen ein Kreis. Damit folgt im Wesentlichen auch schon die Behauptung. ■

Theorem 6.83 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation. Dann gilt:*

$$d'_{TR}(\bar{\pi}) \geq \frac{\bar{b}(\bar{\pi}) - \bar{c}(\bar{\pi})}{2} = \frac{n+1-\bar{c}'(\bar{\pi})}{2}.$$

Der Beweis erfolgt analog wie beim Sortieren mit Reversionen und sei dem Leser zur Übung überlassen.

Korollar 6.84 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation. Dann gilt:*

$$d_{TR}(\bar{\pi}) \geq \frac{\bar{b}(\bar{\pi}) - \bar{c}(\bar{\pi})}{2} = \frac{n + 1 - \bar{c}'(\bar{\pi})}{2}.$$

Notation 6.85 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation. Dann bezeichnet $\hat{c}(\bar{\pi})$ die Anzahl der Kreise in $G'(\bar{\pi})$, die eine ungerade Anzahl von Reality-Edges (oder Desire-Edges) besitzen.*

Im Folgenden sagen wir, dass ein Kreis gerade bzw. ungerade Länge hat, wenn er eine gerade bzw. ungerade Anzahl an Reality-Edges enthält.

Lemma 6.86 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $\bar{\rho}$ eine Operation vom Typ R, T, TR_1, TR_2, DR, dann gilt $|\hat{c}(\bar{\pi} \circ \bar{\rho}) - \hat{c}(\bar{\pi} \circ \bar{\rho})| \leq 2$.*

Beweis: Die Behauptung folgt im Wesentlichen aus folgender Beobachtung in Kombination mit dem Beweis von Lemma 6.82. Aus einem Kreis gerader Länge können maximal zwei Kreise ungerader Länge entstehen, da alle Kanten, die vorher in einem Kreis waren, auch nachher wieder in einem Kreis enthalten sein müssen. Andererseits kann aus drei Kreisen ungerader Länge kein Kreis gerader Länge entstehen. Da ein Transversal auf maximal drei Kreisen operiert, kann sich die Anzahl ungerader Kreise nur um höchstens zwei ändern. ■

Theorem 6.87 *Für jede orientierte Permutation $\bar{\pi} \in \bar{S}_n$ gilt: $d'_{TR}(\bar{\pi}) \geq \frac{n+1-\hat{c}(\bar{\pi})}{2}$.*

Korollar 6.88 *Für jede orientierte Permutation $\bar{\pi} \in \bar{S}_n$ gilt: $d_{TR}(\bar{\pi}) \geq \frac{n+1-\hat{c}(\bar{\pi})}{2}$.*

6.6.3 Eine 2-Approximation

In diesem Abschnitt wollen wir jetzt einen polynomiellen Approximationsalgorithmus mit einer Güte von 2 für Min-SBRT entwickeln.

Lemma 6.89 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $G'(\bar{\pi})$ das zugehörige erweiterte Reality-Desire-Diagram. Sei weiter C ein Kreis in $G'(\bar{\pi})$ mit einem Paar von sich schneidender Desire-Edges. Dann existiert eine Operation τ vom Typ T oder R, die die Länge von C um eins bzw. zwei verkürzt, eine bzw. zwei neue Adjazenzen erzeugt und die Länge der übrigen Kreise in $G'(\bar{\pi})$ unverändert lässt.*

Beweis: Wir werden dazu zeigen, dass wir jeweils aus einem Kreis mindestens eine neue Adjazenz erzeugen können. Die restlichen Kanten des Kreises müssen dann weiterhin einen Kreis bilden. Wir betrachten hierzu die folgende Fallunterscheidung. Für die linke der beiden sich schneidenden Desire-Edges betrachten wir die vier möglichen Kombinationen, wie die hierzu inzidenten Reality-Edges liegen können.

Fall 1: Wir nehmen zuerst an, dass die beiden inzidenten Reality-Edges nach außen führen. Wie man in Abbildung 6.68 sieht, gibt es eine Transposition, die mindestens eine neue Adjazenz erzeugt. Hierbei ist die Ausrichtung der Reality-Edge (c, d) unerheblich (d.h., ob sie von d nach links oder rechts führt), die Transposition muss nur zwischen c und d operieren.

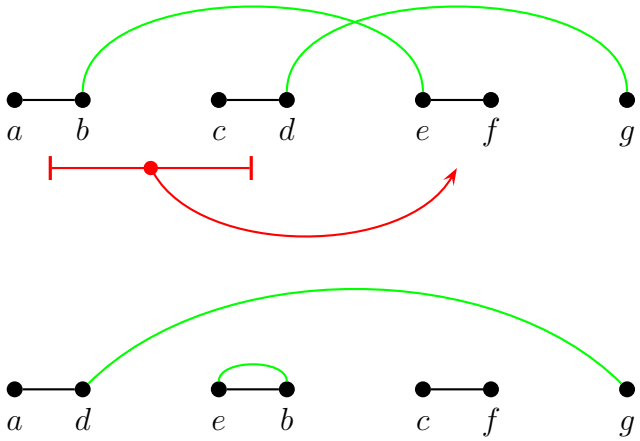


Abbildung 6.68: Skizze: Fall 1: Reality-Edges führen nach außen

Fall 2: Wir nehmen jetzt an, dass die beiden inzidenten Reality-Edges nach links führen. Siehe hierzu auch Abbildung 6.69.

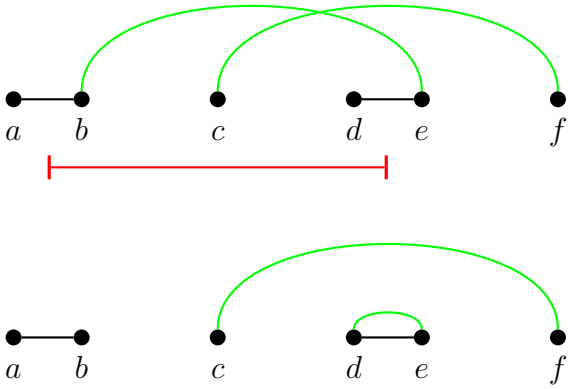


Abbildung 6.69: Skizze: Fall 2: Reality-Edges führen nach links

Wie man in Abbildung 6.69 sieht, gibt es eine Reversion, die mindestens eine neue Adjazenz erzeugt.

Fall 3: Wir nehmen jetzt an, dass die beiden inzidenten Reality-Edges nach rechts führen. Siehe hierzu auch Abbildung 6.70.

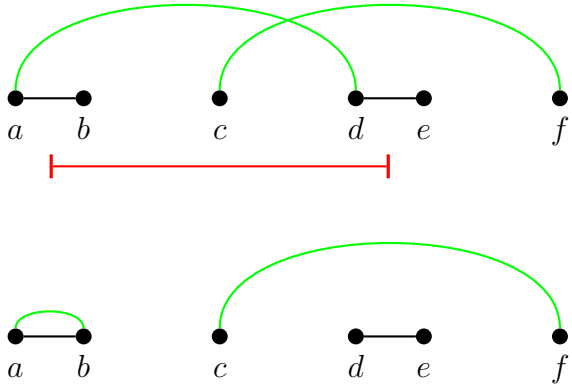


Abbildung 6.70: Skizze: Fall 3: Reality-Edges führen nach rechts

Wie man in Abbildung 6.70 sieht, gibt es eine Reversion, die mindestens eine neue Adjazenz erzeugt.

Fall 4: Wir nehmen zum Schluss an, dass die beiden inzidenten Reality-Edges nach innen führen. Siehe hierzu auch Abbildung 6.71.

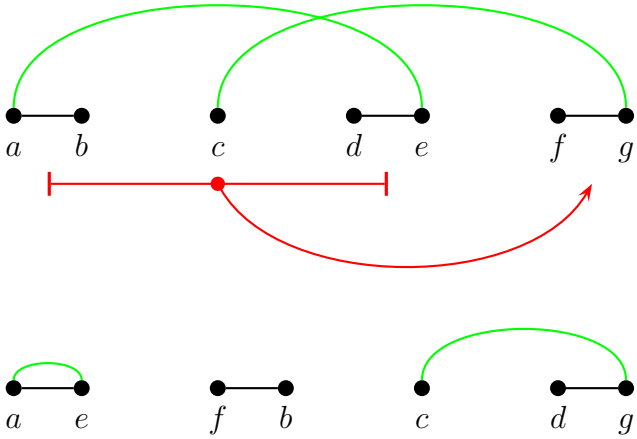


Abbildung 6.71: Skizze: Fall 4: Reality-Edges führen nach innen

Wie man in Abbildung 6.71 sieht, gibt es eine Transposition, die eine Adjazenz erzeugt. Hierbei ist die Ausrichtung der Reality-Edge (f, g) unerheblich (d.h., ob sie von g nach links oder rechts führt), die Transposition muss nur zwischen f und g operieren. ■

Lemma 6.90 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $G'(\bar{\pi})$ das zugehörige erweiterte Reality-Desire-Diagram. Sei weiter C ein Kreis in $G'(\bar{\pi})$ ohne sich schneidende Desire-Edges und sei $(x, x+1)$ eine Desire-Edge in C. Dann existiert eine Reality-Edge (y, y') eines anderen Kreises C' in $G'(\bar{\pi})$ zwischen $(x, x+1)$, die nicht in einer Adjazenz enthalten ist.*

Beweis: Ein Kreis in $G'(\bar{\pi})$ mit sich nicht schneidenden Kanten muss eine wie in Abbildung 6.72 angegebene Gestalt besitzen.

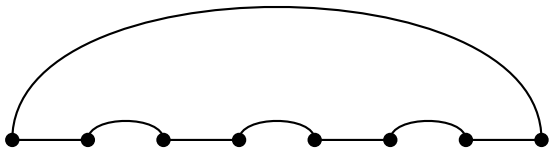


Abbildung 6.72: Skizze: Kreis ohne sich schneidende Desire-Edges

Für einen Widerspruchsbeweis nehmen an, dass in einer Lücke $(x, x+1)$ nur Reality-Edges liegen, die zu Adjazenzen gehören. Da die betrachtete Desire-Edge x mit $x+1$ verbindet, muss nach Konstruktion des erweiterte Reality-Desire-Graphen neben x der Wert $x-1$ stehen (und x muss gerade sein). Da $x-1$ zu einer Adjazenz gehört, ist der folgenden Wert $x-2$. Darauf folgt aufgrund der Konstruktion des erweiterten Reality-Desire-Diagrams der Wert $x-3$. Da nun auch $x-3$ zu einer Adjazenz gehört, ist der folgenden Wert $x-4$ usw. Dies ist in Abbildung 6.73 illustriert.

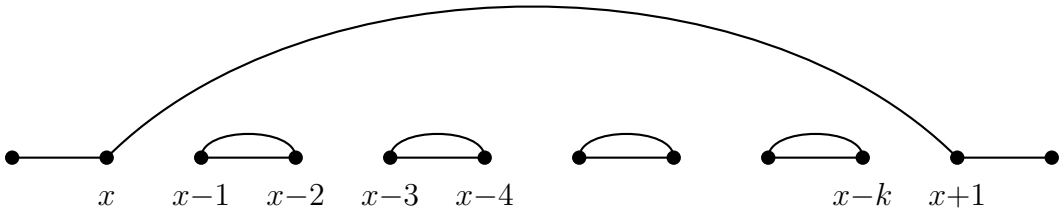


Abbildung 6.73: Skizze: Nur Adjazenzen in einer Desire-Lücke

Somit muss am Ende dieser Desire-Edge $x-k$ neben $x+1$ stehen und diese müssen aus einem vorzeichenbehafteten Element einer orientierten Permutation vermöge der Vorschrift $\{2i-1, 2i\}$ entstanden sein. Dies ist offensichtlich der gewünschte Widerspruch. ■

Lemma 6.91 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $G'(\bar{\pi})$ das zugehörige erweiterte Reality-Desire-Diagram. Gibt es in $G'(\bar{\pi})$ keine Kreise mit sich schneidenden Desire-Edges, dann existieren überlappende Kreise in $G'(\bar{\pi})$.*

Beweis: Sei π die zu $\bar{\pi}$ gehörige unorientierte Permutation und sei e der linkeste Breakpoint in π im Kreis C. Betrachte die darauf folgende Reality-Edge e', die nicht zu einer Adjazenz gehört. Schneidet eine der zugehörigen inzidenten Desire-Edges den Kreis C, ist die Behauptung gezeigt, da diese zu einem anderen Kreis gehören müssen. Dies ist im oberen Teil der Abbildung 6.74 illustriert.

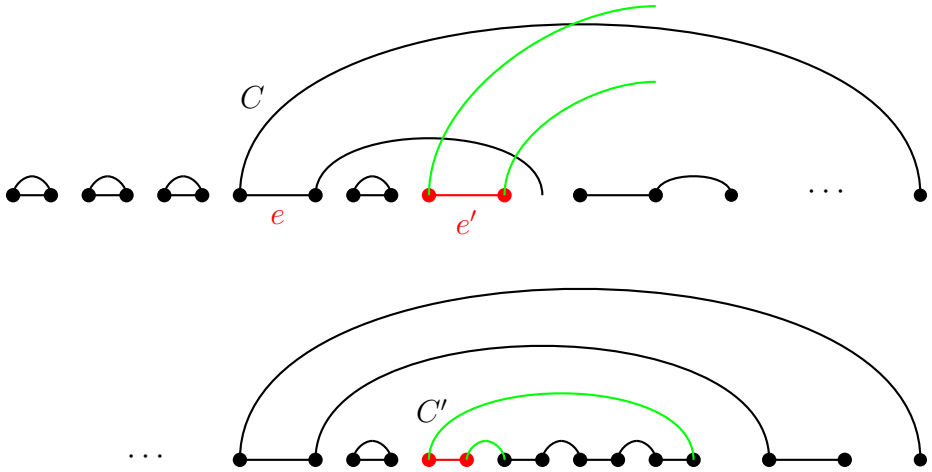


Abbildung 6.74: Skizze: Zum Beweis von Lemma 6.91

Schneiden die Desire-Edges des Kreises C' nicht den Kreis C, so betrachten wir jetzt den Kreis C' statt C. Dies ist im unteren Teil von Abbildung 6.74 dargestellt. Wir führen dieselbe Argumentation mit e' und C' fort. Entweder wir finden sich überlappende Kreise oder wir würden nach Lemma 6.90 unendlich viele Kreise inspizieren müssen. Dies ist offensichtlich nicht möglich, also finden wir ein Paar überlappender Kreise. ■

Lemma 6.92 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $G'(\bar{\pi})$ das zugehörige erweiterte Reality-Desire-Diagram. Seien C und C' zwei sich überlappende Kreise in $G'(\bar{\pi})$, die keine sich selbst schneidenden Desire-Edges enthalten. Dann existieren zwei Transpositionen τ und τ', die die Länge von C und C' jeweils um eins reduzieren und zwei neue Adjazenzen einführen und darüber hinaus die Längen der übrigen Kreise unverändert lassen.*

Beweis: Zuerst einmal überlegt man sich, dass es bei zwei sich überlappenden Kreisen, zwei sich schneidende Desire-Edges geben muss, deren inzidente Reality-Edges nach außen weisen. Dazu benötigen wir wiederum die Voraussetzung, dass sich die Kreise nicht selbst schneiden und damit wieder eine feste Gestalt haben müssen, wie bereits in Abbildung 6.72 illustriert. Dies ist in Abbildung 6.75 im oberen Teil dargestellt, wobei die beiden sich schneidenden Desire-Edges grün gezeichnet sind.

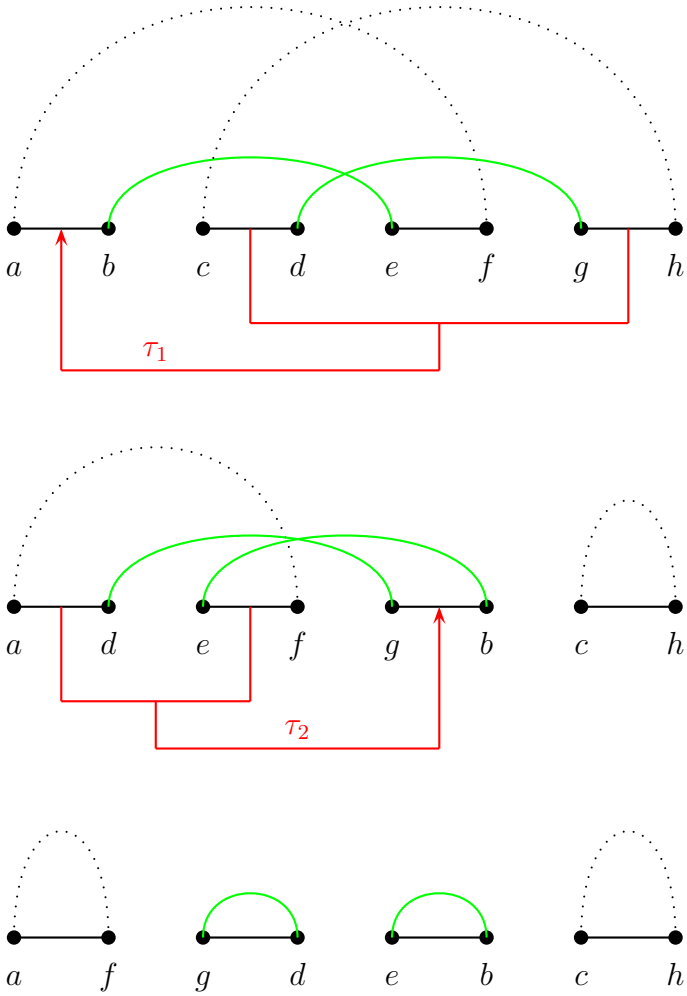


Abbildung 6.75: Skizze: Zwei Transpositionen, die die Längen von zwei Kreisen in $G'(\bar{\pi})$ um jeweils eins verkürzen

Wendet man jetzt die beiden Transpositionen an, wie es in Abbildung 6.75 illustriert ist, so sieht man leicht, dass beide Kreise am Ende um jeweils eine Reality-Edge kürzer sind und dass zwei neue Adjazenzen erzeugt wurden. Die Längen der anderen Kreise blieben offensichtlich unverändert. ■

Theorem 6.93 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation. Es existiert eine Folge von k Reversionen und Transpositionen, die $\bar{\pi}$ in Zeit $O(n^2)$ sortieren, wobei $k \leq 2 \cdot d_{TR}(\bar{\pi})$.*

Beweis: Wie aus den vorhergehenden Lemmata folgt, finden wir immer ein Folge Operationen, die im Schnitt die Länge eines Kreises von $G'(\bar{\pi})$ um eins verkürzt.

Seien $\ell_1, \ldots, \ell_{\bar{c}'(\bar{\pi})}$ die Längen der $\bar{c}'(\bar{\pi})$ Kreise in $G'(\bar{\pi})$ (wobei hier mit Länge wieder die Anzahl von Reality-Edges gemeint ist). Da es im erweiterten Reality-Desire-Diagram zu $\bar{\pi}$ genau $n+1$ Reality-Edges gibt, ist die Anzahl ausgeführter Operationen durch

$$\sum_{i=1}^{\bar{c}'(\bar{\pi})} (\ell_i - 1) \leq (n+1) - \bar{c}'(\bar{\pi})$$

nach oben beschränkt. Mit dem Korollar 6.84 folgt die Behauptung über die Approximationsgüte.

Dass dieser Algorithmus in quadratischer Zeit implementiert werden kann, bleibt dem Leser als Übungsaufgabe überlassen. ■

Eine Verbesserung des vorhergehenden Ergebnisses halten wir im folgenden Satz ohne Beweis fest.

Theorem 6.94 *Jede Permutation $\bar{\pi} \in \bar{S}_n$ lässt sich in Zeit $O(n^2)$ mit Operationen vom Typ R, T, TR und DR mit einer Approximationsgüte von* 1,75 *sortieren.*

Die entsprechenden Approximationsgüten gelten trivialerweise auch noch, wenn man Transversals oder doppelte Reversionen zulässt. Den Leser verweisen wir auf die Originalliteratur von Lin und Xue.

Ein weitere Verbesserung der Approximationsgüte halten wir noch im folgenden Satz fest.

Theorem 6.95 *Jede Permutation $\bar{\pi} \in \bar{S}_n$ lässt sich in Zeit $O(n^{3/2}\sqrt{\log(n)})$ mit Operationen vom Typ R, T, TR und DR mit einer Approximationsgüte von* 1,5 *sortieren.*

Den Leser verweisen wir auf die Originalliteratur von Hartman und Sharan.

Selbst im orientierten Fall ist bislang nicht bekannt, ob die Fragestellung $\mathcal{NP}$-hart ist. Für den unorientierten Fall sind bislang noch keine (außer den trivialen) polynomiellen Approximationen bekannt.

6.7 Sorting by Weighted Transversals (*)

In diesem Abschnitt wollen wir nun untersuchen, ob man eine optimale Lösung für das Sortieren mit Reversionen und Transpositionen effizient konstruieren kann,

wenn man beide Operationen unterschiedlich gewichtet. Bei einer Reversion muss die DNA nur an zwei Stellen geschnitten werden, bei einer Transposition an drei Stellen. Wir werden nun versuchen die wahrscheinlicheren Reversionen billiger als die aufwendigeren Transpositionen zu machen.

6.7.1 Gewichtete Transversal-Distanz

Um die neue gewichtete Transversal-Distanz zu definieren benötigen wir erst noch eine Notation.

Notation 6.96 *Sei s eine Folge von Reversionen und Transpositionen. Mit $rev(s)$ bezeichnet man die Anzahl von Reversionen in s und mit $trp(s)$ die Anzahl von Transpositionen in s.*

Nun zur Definition der gewichteten Transversal-Distanz.

Definition 6.97 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation. $\mathcal{R}(\bar{\pi})$ bezeichnet alle Folgen von Reversionen, die $\bar{\pi}$ sortieren und $\mathcal{TR}(\bar{\pi})$ alle Folgen von Reversionen und Transpositionen, die $\bar{\pi}$ sortieren.*

$$\begin{aligned} \bar{d}(\bar{\pi}) = \bar{d}_R(\bar{\pi}) &:= \min\{rev(s) \,:\, s \in \mathcal{R}(\bar{\pi})\} \\ \bar{d}_{TR}(\bar{\pi}), &:= \min\{rev(s) + trp(s) \,:\, s \in \mathcal{TR}(\bar{\pi})\}, \\ \tilde{d}(\bar{\pi}) = \tilde{d}_{TR}(\bar{\pi}) &:= \min\{rev(s) + 2trp(s) \,:\, s \in \mathcal{TR}(\bar{\pi})\} \end{aligned}$$

$\bar{d} = \bar{d}_R$ heißt Reversions-Distanz*, $\bar{d}_{TR}$ heißt* einfache Transversal-Distanz *und $\tilde{d}$ heißt* gewichtete Transversal-Distanz *.*

Wir erinnern uns, dass $\bar{d}_R(\bar{\pi}) = \bar{b}(\bar{\pi}) - \bar{c}(\bar{\pi}) + \bar{h}(\bar{\pi}) + \bar{f}(\bar{\pi})$ gilt. Zunächst einmal halten wir fest, dass es für die gewichtete Transversal-Distanz unerheblich ist, ob wir Transversals bzw. doppelte Reversionen zulassen oder nicht, da wir diese jeweils

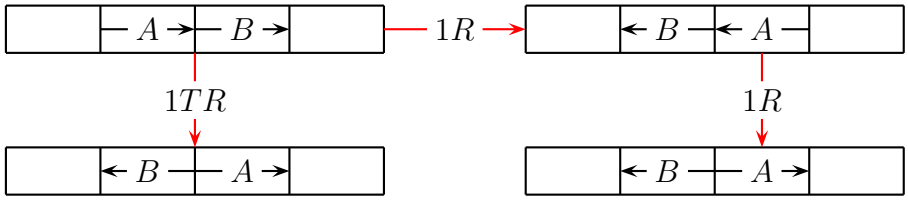


Abbildung 6.76: Skizze: Ersetzung von Transversals

durch zwei Reversionen mit gleichen Kosten ersetzen können (siehe Abbildung 6.76 für die Ersetzung eines Transversals).

Damit können wir nun auch die Problemstellung angeben.

Sorting by Weighted Transversals (Min-SWT)

Eingabe: Eine orientierte Permutation $\bar{\pi} \in \bar{S}_n$.
Gesucht: Eine Folge $s = (\bar{\rho}_1, \ldots, \bar{\rho}_k)$ bestehend aus Reversionen und Transpositionen mit $\bar{\pi} \circ \bar{\rho}_1 \circ \cdots \circ \bar{\rho}_k = \mathrm{id}$, so dass $\tilde{d}(\bar{\pi}) = \mathrm{rev}(s) + 2\mathrm{trp}(s)$.

6.7.2 Starke Orientierung und starke Hurdles

Für das Sortieren mit gewichteten Transversals benötigen wir noch die Begriffe der starken Orientierung und starken Hurdles.

Definition 6.98 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $G'(\bar{\pi})$ das zugehörige erweiterte Reality-Desire-Diagram. Für eine Zusammenhangskomponente C bezeichnet $\tilde{d}(C)$ die Anzahl der Operationen, die benötigt werden, um alle Elemente aus C zu sortieren (interpretiert als eine neue erweiterte orientierte Permutation).*

- *C heißt* stark unorientiert, *wenn $\tilde{d}(C) > \bar{b}(C) - \bar{c}(C)$.*
- *Trennt eine stark unorientierte Komponente C keine anderen stark unorientierten Komponenten, dann heißt C eine* starke Hurdle.
- *Eine starke Hurdle heißt* starke Super-Hurdle, *wenn deren Löschung eine andere starke unorientierte Komponente in eine starke Hurdle umwandelt.*
- *Besteht eine Permutation aus einer ungeraden Anzahl von starken Hurdles, die alle starke Super-Hurdles sind, dann ist $\bar{\pi}$ ein* starkes Fortress.

Darauf aufbauend ergänzen wir nun noch die entsprechenden Notationen für starke Hurdles und starke Fortresses.

Notation 6.99 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $G'(\bar{\pi})$ das zugehörige erweiterte Reality-Desire-Diagram. $\tilde{h}(\bar{\pi})$ bezeichnet die Anzahl starker Hurdles in $\bar{\pi}$. Es gilt $\tilde{f}(\bar{\pi}) = 1$, wenn $\bar{\pi}$ ein starkes Fortress ist, und $\tilde{f}(\bar{\pi}) = 0$ sonst.*

Die Begriffe orientiert, unorientiert, Hurdle und Fortress beziehen sich im Folgenden immer auf das Sortieren mit Reversionen. Im Zusammenhang mit gewichteten Trans-

versals reden wir immer von stark orientiert, stark unorientiert, starken Hurdles und starken Fortresses.

Lemma 6.100 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $G'(\bar{\pi})$ das zugehörige erweiterte Reality-Desire-Diagram. Jede stark unorientierte Komponente in $G'(\bar{\pi})$ ist auch unorientiert.*

Beweis: Für eine orientierte Komponente C in $G'(\bar{\pi})$ gilt: $\bar{d}_R(C) = \bar{b}(C) - \bar{c}(C)$. Angenommen, eine stark unorientierte Komponente C sei orientiert. Dann gilt

$$\bar{d}_R(C) \geq \tilde{d}_{TR}(C) > \bar{b}(C) - \bar{c}(C).$$

Damit erhalten wir den gewünschten Widerspruch. ■

Es kann jedoch unorientierte Komponenten geben, die stark orientierte Komponenten sind. Man bedenke insbesondere an unorientierte Komponenten, bei denen man mit einer Transposition im erweiterten Reality-Desire-Diagram die Anzahl der Kreise um zwei erhöhen kann.

6.7.3 Eine untere Schranke für die Transversal-Distanz

In diesem Abschnitt wollen wir eine zur Reversal-Distanz analoge Formel für die gewichtete Transversal-Distanz angeben.

Theorem 6.101 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $G'(\bar{\pi})$ das zugehörige erweiterte Reality-Desire-Diagram. Es gilt:*

$$\begin{aligned} \tilde{d}(\bar{\pi}) &= \bar{b}(\bar{\pi}) - \bar{c}(\bar{\pi}) + \tilde{h}(\bar{\pi}) + \tilde{f}(\bar{\pi}), \\ \tilde{d}(\bar{\pi}) &= (n+1) - \bar{c}'(\bar{\pi}) + \tilde{h}(\bar{\pi}) + \tilde{f}(\bar{\pi}). \end{aligned}$$

Wir geben hier nur die Idee des Beweises an und beginnen mit einer groben Interpretation der Formel. Wie im Fall orientierter Reversionen soll also gelten

$$\tilde{d}(\bar{\pi}) = (n+1) - \bar{c}'(\bar{\pi}) + \tilde{h}(\bar{\pi}) + \tilde{f}(\bar{\pi}) = \bar{b}(\bar{\pi}) - \bar{c}'(\bar{\pi}) + \tilde{h}(\bar{\pi}) + \tilde{f}(\bar{\pi}).$$

Beachte, dass eine Reversion den Wert $(n+1)-\bar{c}'(\bar{\pi})$ im besten Falle um eins und eine Transposition um zwei erniedrigen kann. Somit ist nach Definition der gewichteten Transversal-Distanz $(n+1) - \bar{c}'(\bar{\pi})$ schon eine triviale untere Schranke. Der Term $\tilde{h}(\bar{\pi})+\tilde{f}(\bar{\pi})$ gibt hierbei auch die Anzahl zusätzlicher Operationen an, um aus starken Hurdles stark orientierte Komponenten zu machen.

Beweis: Zunächst einmal halten wir fest, dass wir jede orientierte Komponente C wie beim Sortieren mit Reversionen mit nur $\bar{b}(C) - \bar{c}(C)$ Reversionen sortieren können. Weiter können unorientierte Komponenten, die nicht stark unorientiert sind, nach Definition mit Hilfe von Transpositionen mit $(\bar{b}(C) - \bar{c}(C))$ Operationen sortiert werden. Wir müssen uns also nur noch um stark unorientierte Komponenten kümmern, die nach Lemma 6.100 auch unorientiert sind.

Wir bemerken, dass eine Reversion angewendet auf $\bar{\pi}$ den Wert $\bar{c}'(\bar{\pi})$ um maximal eins verändern kann. Eine Transposition angewendet auf $\bar{\pi}$ kann $\bar{c}'(\bar{\pi})$ um die Werte -2, 0 oder $+2$ verändern. Man beachte, dass wir nur noch unorientierte Komponenten (und somit auch Permutationen) betrachten, d.h. wir können annehmen, dass alle Vorzeichen positiv sind. Somit folgt diese Tatsache aus Lemma 6.68.

Es bleibt also die Frage, inwieweit Transpositionen beim Aufbrechen starker Hurdles helfen können. Wir unterscheiden jetzt drei Fälle, je nachdem, wie eine Transposition auf Kreisen einer Komponente operiert.

Fall 1: Eine Transposition bricht einen Kreis in 3 Kreise auf. War der Kreis Teil einer starke Hurdle C, dann muss sich mindestens einer der neuen neuen Kreise in einer starker Hurdle befinden, da sonst nach Definition die ganze starke Hurdle mit $\bar{b}(C) - \bar{c}(C)$ Operationen aufgelöst werden kann. Somit kann sich der Parameter $\tilde{h}(\bar{\pi})$ nicht verringern.

Fall 2: Eine Transposition auf 2 Kreisen liefert 2 neue Kreise. Um die gewichtete Transversal-Distanz um 3 erniedrigen zu können, müssten 3 Hurdles eliminiert werden. Dies ist unmöglich, da nur 2 Kreise verändert werden.

Fall 3: Wir betrachten jetzt eine Transpositionen, die auf 3 Kreisen operiert. Dabei entsteht ein einziger Kreis ergibt. Da $\bar{c}'(\bar{\pi})$ um 2 wächst, müssten 5 Hurdles eliminiert werden, was offensichtlich nicht möglich ist.

Eine Transposition kann also den Parameter $\tilde{h}(\bar{\pi})$ nicht noch zusätzlich zum Parameter $(n+1)-\bar{c}'(\bar{\pi})$ erniedrigen. Wir müssen uns nur noch überlegen, ob eine Transposition beim Eliminieren eines Fortress auch noch den Parameter $\tilde{h}(\bar{\pi})$ erniedrigen kann. Nach unserer Analyse muss aber eine Transposition, die $(n+1)-\bar{c}'(\bar{\pi})+\tilde{h}(\bar{\pi})$ um zwei erniedrigt, auch $\tilde{h}(\bar{\pi})$ um eine gerade Zahl verändern und dabei werden keine außer den entfernten starken Hurdles verändert. Also kann keine Transposition den Parameter $(n+1)-\bar{c}'(\bar{\pi})+\tilde{h}(\bar{\pi})+\tilde{f}(\bar{\pi})$ um drei erniedrigen. Somit gilt die untere Schranke

$$\tilde{d}(\bar{\pi}) \geq (n+1) - \bar{c}'(\bar{\pi}) + \tilde{h}(\bar{\pi}) + \tilde{f}(\bar{\pi}),$$

wie in der Behauptung gefordert. ■

Man beachte, dass hier im Gegensatz zum Sortieren mit Reversionen die Begriffe starke Hurdle und starkes Fortress über die Eigenschaft definiert wurde, ob man

zusätzliche Operationen über den Parameter $\bar{b}(\bar{\pi}) - \bar{c}(\bar{\pi})$ hinaus zur Eliminierung der entsprechenden Komponenten benötigt. Es ist also momentan algorithmisch überhaupt nicht klar, wie man entscheidet, ob eine Komponente stark orientiert bzw. stark unorientiert ist oder ob es eine starke Hurdle ist.

6.7.4 Eine Approximation für Min-SWT

Nun versuchen wir einen Algorithmus zu entwickeln, der mit möglichst wenig Operationen auskommt, d.h. der unteren Schranke aus dem vorherigen Abschnitt sehr nahe kommt. Zuerst geben wir eine algorithmisch verwendbare Bedingung an, wann eine Komponente stark unorientiert ist.

Lemma 6.102 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $G'(\bar{\pi})$ das zugehörige erweiterte Reality-Desire-Diagram. Enthält eine unorientierte Komponente von $G'(\bar{\pi})$ einen Kreis gerader Länge, dann ist sie stark unorientiert.*

Auch hier ist die Länge eines Kreises wieder die Anzahl seiner Reality-Edges.

Beweis: Da die Komponente unorientiert ist, kann eine Reversion die Anzahl der Kreise im erweiterten Reality-Desire-Diagram nicht erhöhen. Weiter wissen wir, dass eine Transposition die unorientierte Kreise unorientiert lässt und die Anzahl der unorientierten Kreise nur um -2, 0, oder $+2$ ändern kann. Somit ist eine Anwendung einer Transposition, die die Anzahl der Kreise um zwei erhöht, die einzige sinnvolle Wahl. Damit bleibt die Parität der Anzahl der Kreise unverändert, da jede Anwendung einer Transposition auf einen unorientierten Kreis gerader Länge einen Kreis gerader Länge sowie zwei Kreisen mit entweder gerader oder ungerader Länge erzeugt. Ein Kreis gerader Länge kann also nie mit Transpositionen endgültig eliminiert werden.

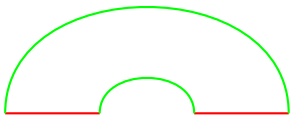


Abbildung 6.77: Skizze: ein unorientierter Kreis der Länge 2

Es bleibt also ein Kreis der Länge 2 übrig. Dieser muss eine Form, wie in Abbildung 6.77 illustriert, besitzen. Es existiert auch hier keine Transposition mehr, die $\bar{c}'(C)$ erhöhen kann. Also ist C eine stark unorientierte Komponente. ■

Es bleibt die Frage, wie man mit unorientierten Komponenten umgeht, die nur aus Kreisen ungerader Länge bestehen. Hier ist a priori nicht klar, ob sie stark

orientiert oder stark unorientiert sind. Man kann leicht überprüfen, dass es nur eine unorientierte Komponente mit genau drei Reality-Edges gibt. Da sich diese jedoch mit einer Transposition auflösen lässt, ist sie stark orientiert. Diese Komponente ist in Abbildung 6.78 ganz links dargestellt.

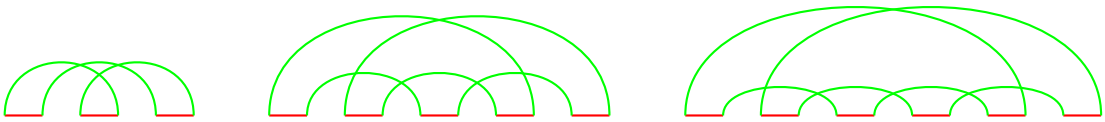


Abbildung 6.78: Skizze: Kleine unorientierte Komponenten

In Abbildung 6.78 sind in der Mitte und rechts die einzigen stark unorientierten Komponenten mit höchstens 6 Reality-Edges dargestellt, die nur aus Kreisen ungerader Länge bestehen. Mit dieser vollständigen Aufzählung lassen sich also kleine Komponenten leicht daraufhin testen, ob sie stark unorientiert sind oder nicht.

Definition 6.103 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $G'(\bar{\pi})$ das zugehörige erweiterte Reality-Desire-Diagram. Das* Gewicht *einer Komponente C von $G'(\bar{\pi})$ ist durch $\gamma(C) := \bar{b}(C) - \bar{c}(C)$ definiert.*

In Abbildung 6.78 sind in der Mitte und rechts alle stark unorientierten Komponenten mit einem Gewicht von höchstens 4 dargestellt.

Lemma 6.104 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $G'(\bar{\pi})$ das zugehörige erweiterte Reality-Desire-Diagram. Das Gewicht aller stark unorientierten Zusammenhangskomponenten in $G'(\bar{\pi})$, die nur aus Kreisen ungerader Länge bestehen, ist gerade.*

Den Beweis diese Lemmas überlassen wir dem Leser zur Übung. Für das Folgende ist nur wichtig, dass es keine stark orientierten Komponenten mit Gewicht 5 gibt, die nur aus Kreisen ungerader Länge bestehen.

Der Algorithmus geht wie folgt vor. Orientierte Komponenten werden mit Hilfe von Reversionen wie im Falle des Sortierens mit Reversionen behandelt. Unorientierte Komponenten, die einen Kreis gerader Länge enthalten, werden als stark unorientierte Komponente klassifiziert. Unorientierte Komponenten, die nur aus ungeraden Kreisen bestehen, werden mit den bekannten stark unorientierten Komponenten kleinen Gewichtes (im einfachsten Falle kleiner als 6) verglichen und als stark orientiert bzw. stark unorientiert klassifiziert. Die stark orientierten werden dann mit Hilfe von Transpositionen und Reversionen aufgelöst (deren prinzipielle Lösung vorab

gespeichert wurde). Alle übrigen unorientierten Komponenten, die nur aus ungeraden Kreisen bestehen, werden aus Unwissenheit der Einfachheit halber als stark unorientiert klassifiziert. Stark unorientierte Komponenten werden mit Hilfe von Reversionen wie im Falle von Sortieren mit Reversionen aufgelöst. Durch die möglichen Klassifizierungsfehler werden eventuell überflüssige Reversionen ausgeführt.

Man beachte, dass der Algorithmus hauptsächlich Reversionen verwendet und nur bei stark orientierten Komponenten, die unorientiert sind und ein kleines Gewicht besitzen, auf Transpositionen zurückgreift.

6.7.5 Approximationsgüte

Im letzten Teil kommen wir zur Bestimmung der Approximationsgüte des im eben vorgestellen polynomiellen Approximationsalgorithmus. Hierfür benötigen wir erst noch ein paar Notationen.

Notation 6.105 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $G'(\bar{\pi})$ das zugehörige erweiterte Reality-Desire-Diagram. $\tilde{h}_u(\bar{\pi})$, $\bar{c}_u(\bar{\pi})$ bzw. $\bar{b}_u(\bar{\pi})$ bezeichnet die Anzahl der unorientierten Komponenten, Kreise bzw. Breakpoints in unorientierten Komponenten, die nur aus Kreisen ungerader Länge bestehen und mindestens ein Gewicht von 6 besitzen. $\bar{c}_o(\bar{\pi})$ bzw. $\bar{b}_o(\bar{\pi})$ bezeichnet die Anzahl der Kreise bzw. Breakpoints in den übrigen Komponenten.*

Notation 6.106 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $G'(\bar{\pi})$ das zugehörige erweiterte Reality-Desire-Diagram. $\tilde{h}_{\min}(\bar{\pi})$ bzw. $\tilde{h}_{\max}(\bar{\pi})$ bezeichnet die Anzahl der starken Hurdles mit Gewicht mindestens 6, wenn keine bzw. alle unorientierten Komponenten, die nur aus Kreisen ungerader Länge bestehen, starke Hurdle sind.*

Beobachtung 6.107 *Sei $\bar{\pi} \in \bar{S}_n$ eine orientierte Permutation und sei $G'(\bar{\pi})$ das zugehörige erweitert Reality-Desire-Diagram, dann gilt*

$$\tilde{h}_{\min}(\bar{\pi}) \leq \tilde{h}(\bar{\pi}) \leq \tilde{h}_{\max}(\bar{\pi})$$

sowie

$$\begin{aligned} \tilde{h}_{\max}(\bar{\pi}) &= \tilde{h}_{\min}(\bar{\pi}) + \tilde{h}_u(\bar{\pi}), \\ \tilde{h}_u(\bar{\pi}) &\leq \bar{c}_u(\bar{\pi}), \\ \tilde{h}_u(\bar{\pi}) &\leq \frac{\bar{b}_u(\bar{\pi}) - \bar{c}_u(\bar{\pi})}{6}. \end{aligned}$$

Die Abschätzung für $\tilde{h}_u(\bar{\pi})$ folgt aus der Tatsache, dass hier nur unorientierte Komponenten mit Gewicht mindestens 6 gezählt werden. Im Regelfall ist diese Abschätzung nicht sehr genau, aber für unsere Zwecke ausreichend.

Die konkrete Zahl 6 in den vorhergehenden Notationen und Beobachtung stammt daher, dass wir für unorientierte Komponenten mit einem Gewicht kleiner 6, die nur aus ungeraden Kreisen bestehen, wissen, welche davon stark orientiert sind und welche nicht. Wenn wir das auch für größere Gewichte wissen, kann diese Konstante entsprechend erhöht werden.

Sei für die folgenden Betrachtungen $\bar{\pi} \in \bar{S}_n$ kein starkes Fortress. Dann gilt für die gewichtete Anzahl ausgeführter Operationen $A(\bar{\pi})$ nach dem im vorigen Abschnitt angegebenen Algorithmus:

$$\begin{aligned} A(\bar{\pi}) &= \bar{b}(\bar{\pi}) - \bar{c}(\bar{\pi}) + \tilde{h}(\bar{\pi}) \\ &\leq \bar{b}_o(\bar{\pi}) + \bar{b}_u(\bar{\pi}) - \bar{c}_o(\bar{\pi}) - \bar{c}_u(\bar{\pi}) + \tilde{h}_{\max}(\bar{\pi}) \\ &= \bar{b}_o(\bar{\pi}) + \bar{b}_u(\bar{\pi}) - \bar{c}_o(\bar{\pi}) - \bar{c}_u(\bar{\pi}) + \tilde{h}_{\min}(\bar{\pi}) + \tilde{h}_u(\bar{\pi}) \\ &\leq \bar{b}_o(\bar{\pi}) - \bar{c}_o(\bar{\pi}) + \bar{b}_u(\bar{\pi}) - \bar{c}_u(\bar{\pi}) + \tilde{h}_{\min}(\bar{\pi}) + \frac{\bar{b}_u(\bar{\pi}) - \bar{c}_u(\bar{\pi})}{6}. \end{aligned}$$

Weiter gilt offensichtlich mit Theorem 6.101:

$$\begin{aligned} \tilde{d}(\bar{\pi}) &\geq \bar{b}(\bar{\pi}) - \bar{c}(\bar{\pi}) + \tilde{h}(\bar{\pi}) \\ &\geq \bar{b}_o(\bar{\pi}) - \bar{c}_o(\bar{\pi}) + \bar{b}_u(\bar{\pi}) - \bar{c}_u(\bar{\pi}) + \tilde{h}_{\min}(\bar{\pi}). \end{aligned}$$

Somit gilt für die Approximationsgüte:

$$\frac{A(\bar{\pi})}{\tilde{d}(\bar{\pi})} \leq \frac{\bar{b}_o(\bar{\pi}) - \bar{c}_o(\bar{\pi}) + \frac{7}{6}(\bar{b}_u(\bar{\pi}) - \bar{c}_u(\pi)) + \tilde{h}_{\min}(\bar{\pi})}{\bar{b}_o(\bar{\pi}) - \bar{c}_o(\bar{\pi}) + \bar{b}_u(\bar{\pi}) - \bar{c}_u(\bar{\pi}) + \tilde{h}_{\min}(\bar{\pi})} \leq \frac{7}{6}.$$

Ein analoger Beweis für diese Approximationsrate ist auch für Permutationen möglich, die starke Fortresses sind. Halten wir das Ergebnis im folgenden Satz fest.

Theorem 6.108 *Es gibt es eine 7/6-Approximation, die in polynomieller Zeit eine orientierte Permutation mittels gewichteter Reversionen und Transpositionen sortiert.*

Man beachte, dass die Approximationsgüte im Wesentlichen aus der nicht sehr genauen Abschätzung für $\tilde{h}_u(\bar{\pi})$ aus Beobachtung 6.107 stammt. Meist ist die gefundene Lösung also viel genauer als die Approximationsgüte vorhersagt.

Wenn für alle unorientierten Komponenten mit einem Gewicht kleiner als k, die nur aus Kreisen ungerader Länge bestehen, vorab nach der starken Orientiertheit (samt

einer Lösung) klassifiziert wird, dann kann man auch den folgenden Satz zeigen, wobei dort $\varepsilon = 1/k$ ist.

Theorem 6.109 *Für jedes $\varepsilon > 0$ gibt es eine $(1+\varepsilon)$-Approximation, die in polynomieller Zeit eine orientierte Permutation mittels gewichteter Reversionen und Transpositionen sortiert.*

Wie schon bei Angabe des Algorithmus angegeben, verwendet dieser hauptsächlich Reversionen und nur in einigen Ausnahmefällen werden Transpositionen eingesetzt. Das legt die Vermutung nahe, dass die Gewichtung von Reversion zu Transposition für die Reversion zu günstig ausfällt. Insbesondere basiert das ganze Verfahren auch auf dem Trick, dass die Kosten einer Operation gerade genau groß wie die Anzahl der hinzugefügten Kreise im günstigsten Falle ist. Vermutlich wäre eine andere Gewichtung, wie etwa 2 pro Reversion und 3 pro Transposition günstiger, da sie den echten Aufwand, die Anzahl Stellen, an denen die DNA aufgeschnitten wird, besser repräsentiert.

Bader und Ohlebusch haben eine 1,5-Approximation entwickelt, auch wenn das Gewicht der Reversion mit 1 und der Transposition bzw. Transversal mit $\gamma \in [1:2]$ festgelegt ist. Für die Details verweisen wir auf die Originalliteratur.

6.8 Weitere Modelle (*)

Zum Abschluss dieses Kapitels wollen wir noch kurz auf einige Erweiterungen der Modelle und auf bereits dort erzielte Ergebnisse eingehen.

6.8.1 Gewichtung der Operationen durch ihre Länge

Unter der Annahme, dass kurze Operationen (wie Reversionen oder Transpositionen) wahrscheinlicher sind als lange, kann man das Modell dahingehend erweitern, die Kosten pro Operation von der Länge abhängig zu machen.

Eine erste, mehr mathematische Methode, die Länge ins Spiel zu bringen, ist, nur Reversionen fester Länge k zuzulassen. Für $k = 3$ gibt es dann bereits Permutation, die mit solchen Reversionen überhaupt nicht mehr sortiert werden können. Erste Ergebnisse hierzu findet man in der Arbeit von Chen und Skiena.

Heath und Vergara haben eine ähnliche Variante für Transpositionen untersucht, wobei die Anzahl der transponierten Elemente beschränkt ist. Ist diese Beschränkung nicht konstant, sondern von der Länge der Permutation abhängig, hat dieses

Problem bereits dieselbe (unbekannte) Komplexität wie das normale Sortieren mit Transpositionen. Für Transposition, die maximal drei Elemente transponieren, wird hingegen die Existenz eines polynomiellen Algorithmus vermutet.

Ein erster Ansatz für allgemeinere Gewichtsfunktionen, die wirklich von der Länge der Operationen abhängen, stammt von Pinter und Skiena. Für das Sortieren unorientierter Permutationen mittels Reversionen konnten für sehr spezielle Gewichtsfunktionen erste Approximationsalgorithmen für die Reversions-Distanz ermittelt werden. Bender, Ge, He, Hu, Pinter, Skiena und Swidan haben eine Klasse von Gewichtsfunktion der Gestalt $f_\alpha(\ell) = \ell^\alpha$ mit $\alpha > 0$ für unorientierte Reversionen genauer untersucht und untere bzw. obere Schranken sowie auch einige Approximationen sowohl für den Durchmesser (Maximum der minimalen Distanz über alle Paare von Permutationen) als auch für den minimalen Abstand zweier gegebener Permutationen ermittelt.

Auch die Erweiterung auf so genannte Block-Interchange Operationen wurden inzwischen genauer untersucht. Hierbei werden im Gegensatz zu Transpositionen zwei nicht benachbarte Blöcke vertauscht. Für den Fall unorientierter Permutationen konnte Christie einen quadratischen Algorithmus angeben.

6.8.2 Duplikationen

In Eukaryonten tauchen häufig Gene nicht nur einmal, sondern mehrfach im Genom auf. Von daher sind auch Varianten von Interesse, die solche Duplikationen berücksichtigen. Leider lassen sich dann die Genome nicht mehr als Permutationen modellieren. Da für unorientierte Permutationen Min-SBR bereits ein Spezialfall ist, bleibt dieses Problem $\mathcal{NP}$-hart. Für eine feste Anzahl von Genen (unabhängig von der Länge des Genoms) konnten Christie und Irving zeigen, dass das Problem auch hier $\mathcal{NP}$-hart bleibt. Sogar für nur zwei Gene (mit entsprechender Vielfachheit) bleibt die Bestimmung der Reversions-Distanz (im orientierten Fall) $\mathcal{NP}$-hart. Für die Transpositions-Distanz ist dies noch offen. Für orientierte Permutationen gibt es bislang noch keine Ergebnisse.

6.8.3 Multi-chromosomale Genome

Genome höherer Lebewesen bestehen ja nicht nur aus einem, sondern in der Regel aus mehreren Chromosomen. Im Falle orientierter Permutationen (aufgeteilt auf mehrere Chromosomen) konnte Hannenhalli einen ersten polynomiellen Algorithmus zur Bestimmung der so genannten Translokations-Distanz angeben. Diesen Algorithmus konnten Hannenhalli und Pevzner noch um die Berücksichtigung von Reversio-

nen erweitern. Dieser kann die exakte Distanz bis auf 1 ermitteln, was für praktische Zwecke jedoch völlig ausreichend ist.

Berücksichtigt man bei mehreren Chromosomen nur die Operationen Fusion, Fission sowie Transposition und belastet die Transpositionen mit doppelten Kosten im Vergleich zu Fusions und Fissions, so lässt sich die Distanz leicht berechnen, wie Meidanis und Dias zeigen konnten.

Unter der Prämisse, dass die Duplikation nur auf eine Duplikation aller Chromosomen eines urzeitlichen Genoms zurückzuführen sind, konnten El-Mabrouk und Sankoff eine effiziente Lösung vorstellen.

Betrachtet man bei multi-chromosomalen Genomen nur die Zuordnung der Gene auf die Chromosomen und vergisst deren Ordnung innerhalb der Chromosome, so kommt man zu den so genannten *Synteny-Problemen*. Dabei werden in der Regel nur die Operationen Translokation, Fission und Fusion betrachtet. DasGupta, Jiang, Kannan, Li und Sweedyk konnten zeigen, dass auch dieses Problem bereits $\mathcal{NP}$-hart ist und haben eine 2-Approximation entwickelt. Kleinberg und Liben-Nowell konnten zeigen, dass die syntenische Distanz von zwei Genomen mit n Chromosomen höchstens $2n - 4$ ist.

6.8.4 Multiple Genome Rearrangements

Man kann auch versuchen, aus den Rearrangements der Genome mehrerer Spezies direkt einen evolutionären Baum aufzubauen, anstatt nur die daraus ermittelten Distanzen für Paare zur Rekonstruktion von evolutionären Bäumen zu verwenden. Dies führt zu dem so genannten Median Problem, das wir hier nur in seiner einfachsten Form darstellen wollen.

Generic Median Problem

Eingabe: Drei Permutationen $\pi_1, \pi_2, \pi_3 \in S_n$ und eine Distanz $d : S_n \times S_n \to \mathbb{N}_0$.
Gesucht: Eine Permutation $\sigma \in S_n$, so dass $\sum_{i=1}^{3} d(\pi_i, \sigma)$ minimal ist.

Man kann sich leicht überlegen, wie dieses Problem zu erweitern ist, wenn man für k Permutationen einen Baum mit k Blättern (die die Genome respektive ihre Spezies darstellen) und minimaler Distanz sucht. Hierbei besteht als erste, einfachste Möglichkeit den Baum als einen Stern auszuwählen oder den Grad eines jedes inneren Knoten auf drei festzusetzen.

Als Abstandsfunktion kann man beispielsweise die Reversions-Distanz verwenden, dann erhält man das so genannte Reversal Median Problem.

REVERSAL MEDIAN PROBLEM

Eingabe: Drei Permutationen $\bar{\pi}_1, \bar{\pi}_2, \bar{\pi}_3 \in \bar{S}^n$.
Gesucht: Eine Permutation $\bar{\sigma} \in \bar{S}^n$, so dass $\sum_{i=1}^{3} \bar{d}_R(\bar{\pi}_i, \bar{\sigma})$ minimal ist.

Dieses Problem ist bereits dann $\mathcal{NP}$-hart, wenn man orientierte Permutationen betrachtet. Für k statt drei Sequenzen ist das Problem sogar $\mathcal{APX}$-vollständig, wie Caprara zeigen konnte.

Eine andere Wahl für die Distanzfunktion ist die so genannte *Breakpoint-Distanz* d_b. Diese liefert die Anzahl derjenigen Gene, die in beiden Genomen auftreten und in den beiden Genomen unterschiedliche Nachfolger besitzen. Manchmal wird diese Distanz auch noch durch die Anzahl der in beiden Genomen gemeinsamen Gene normalisiert.

BREAKPOINT MEDIAN PROBLEM

Eingabe: Drei Permutationen $\pi_1, \pi_2, \pi_3 \in S_n$.
Gesucht: Eine Permutation $\sigma \in S_n$, so dass $\sum_{i=1}^{3} d_b(\pi_i, \sigma)$ minimal ist.

Auch hier konnten Bryant bzw. Pe´er und Shamir zeigen, dass dieses Problem bereits $\mathcal{NP}$-hart ist.

Eine weitere Variante des Median Problems ist die Steiner-Baum Variante. Hier betrachtet man die Menge der Permutationen als Graphen (den so genannten Reversionsgraphen) und verbindet zwei Permutationen, wenn es eine Reversion gibt, die die beiden ineinander überführt. Gesucht ist dann ein minimaler Teilbaum dieses Graphen, der alle gegebenen Permutationen aufspannt.

STEINER REVERSAL MEDIAN PROBLEM

Eingabe: Eine Menge von Permutationen $\pi_1, \ldots, \pi_k \in S_n$.
Gesucht: Ein für $\pi_1, \ldots, \pi_k \in S_n$ minimal aufspannender Teilbaum des Reversionsgraphen von S_n.

Auch dieses Problem konnte Caprara als $\mathcal{APX}$-vollständig nachweisen. Man kann sich auch hier leicht Varianten des Modells vorstellen, beispielsweise für andere Permutation, mit Gewichten auf den Operationen oder man kann den vollständigen Graphen auf S_n betrachten, wobei dann die Kantengewichte durch die entsprechenden Distanzen gegeben sind.

Literaturhinweise A

A.1 Lehrbücher zur Vorlesung

D. Adjeroh, T. Bell, A. Mukherjee: *The Burrows-Wheeler Transform*, Springer, 2008

S. Aluru (Ed.): *Handbook of Computational Molecular Biology*; Chapman and Hall/CRC, 2006.

H.-J. Böckenhauer, D. Bongartz: *Algorithmische Grundlagen der Bioinformatik: Modelle, Methoden und Komplexität*; Teubner, 2003.

G. Fertin, A. Labarre, I. Rusu, E. Tannier, S. Vialette: *Combinatorics of Genome Rearrangements*; MIT Press, 2009.

D. Gusfield: *Algorithms on Strings, Trees, and Sequences — Computer Science and Computational Biology*; Cambridge University Press, 1997.

M. Lothaire: *Applied Combinatorics on Words*, Encyclopedia of Mathematics and Its Applications, Cambridge University Press, 2005.

V. Mäkinen, D. Belazzougui, F. Cunial, A.I. Tomescu: *Genome-Scale Algorithm Design: Biological Sequence Analysis in the Era of High-Throughput Sequencing*, Cambridge University Press, 2015.

E. Ohlebusch: *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction*, Oldenbusch Verlag, 2013.

P.A. Pevzner: *Computational Molecular Biology — An Algorithmic Approach*; MIT Press, 2000.

J.C. Setubal, J. Meidanis: *Introduction to Computational Molecular Biology*; PWS Publishing Company, 1997.

A.2 Skripten anderer Universitäten

J. Fischer: *Text-Indexierung und Information Retrieval*, Technische Universität Dortmund, 2015.
ls11-www.cs.tu-dortmund.de/_media/fischer/teaching/tir-ws2014/script-tir-ws14.pdf

S. Kurtz: *Lecture Notes for Foundations of Sequence Analysis*, Universität Bielefeld, 2001.
citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=39590fbdafdb7597e4f5f59c754f4629cfcbd8c9

A.3 Originalarbeiten

A.3.1 Optimal Scoring Subsequences

K.-M. Chung, H.-I. Lu: An Optimal Algorithm for the Maximum-Density Segment Problem, *SIAM Journal on Computing*, Vol. 34, No. 2, 373–387, 2004.
DOI: 10.1137/S0097539704440430

M. Csűrös: Maximum-Scoring Segment Sets, *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, Vol. 1, No. 4, 139–150, 2004.
DOI: 10.1109/TCBB.2004.43

P. Fariselli, M. Finelli, D. Marchignoli, P.L. Martelli, I. Rossi, R. Casadio: MaxSubSeq: An Algorithm for Segment-Length Optimization. The Case Study of the Transmembrane Spanning Segments, *Bioinformatics*, Vol. 19, 500–505, 2003.
DOI: 10.1093/bioinformatics/btg023

M.H. Goldwasser, M.-Y. Kao, H.-I. Lu: Linear-Time Algorithms for Computing Maximum-Density Sequence Segments with Bioinformatics Applications, *Journal of Computer and System Sciences*, Vol.70, No. 2, 128–144, 2005.
DOI: 10.1016/j.jcss.2004.08.001

S.K. Kim: Linear-Time Algorithm for Finding a Maximum-Density Segment of a Sequence, *Information Processing Leter*, Vol. 86, 339–342, 2003.
DOI: 10.1016/S0020-0190(03)00225-4

Y.-L. Lin, T. Jiang, K.-M. Chao: Efficient Algorithms for Locating the Length-Constrained Heaviest Segments with Applications to Biomolecular Sequence Analysis, *Journal of Computer and System Sciences*, Vol. 65, 570–586, 2002.
DOI: 10.1016/S0022-0000(02)00010-7

W.L. Ruzzo, M. Tompa: A Linear Time Algorithm for Finding All Maximal Scoring Subsequences, *Proceedings of the 7th International Conference on Intelligent Systems for Molecular Biology (ISMB'99)*, 234–241, 1999.

A.3.2 Suffix-Trees

R. Giegerich, S. Kurtz, J. Stoye: Efficient Implementation of Lazy Suffix Trees, *Software — Practice and Experience*, Vol. 33, 1035–1049, 2003.
DOI: 10.1002/spe.535

M. Maaß: *Suffix Trees and Their Applications*, Ausarbeitung von der Ferienakademie, Kurs 2, Bäume: Algorithmik und Kombinatorik, 1999.
www14.in.tum.de/konferenzen/Ferienakademie99/

E.M. McCreight: A Space-Economical Suffix Tree Construction Algorithm; *Journal of the ACM*, Vol. 23, 262–272, 1976.
DOI: 10.1145/321941.321946

E. Ukkonen: On-Line Construction of Suffix Trees, *Algorithmica*, Vol. 14, 149–260, 1995.
DOI: 10.1007/BF01206331

A.3.3 Repeats

A.S. Fraenkel, J. Simpson: How Many Squares Can a String Contain?, *Journal of Combinatorial Theory, Series A*, Vol. 82, 112–120, 1998.
DOI: 10.1006/jcta.1997.2843

D. Gusfield, J. Stoye: Linear Time Algorithm for Finding and Representing All the Tandem Repeats in a String, *Journal of Computer and System Sciences*, Vol. 69, 525–546, 2004; see also *Technical Report* CSE-98-4, Computer Science Department, UC Davis, 1998.
DOI: 10.1016/j.jcss.2004.03.004

R. Kolpakov, G. Kucherov: Finding Maximal Repetitions in a Word in Linear Time, *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science (FOCS'99)*, 596–604, 1999.
DOI: 10.1109/SFFCS.1999.814634

R. Kolpakov, G. Kucherov: On Maximal Repetitions in Words, *Proceedings of the 12th International Symposium on Fundamentals of Computation Theory (FCT'99)*, Lecture Notes in Computer Science, Vol. 1684, 374–385, 1999.
DOI: 10.1007/3-540-48321-7_31

R. Kolpakov, G. Kucherov: Finding Approximate Repetitions Under Hamming Distance, *Theoretical Computer Science*, Vol. 303, 135–156, 2003.
DOI: 10.1016/S0304-3975(02)00448-6

G.M. Landau, J.P. Schmidt: An Algorithm for Approximate Tandem Repeats, *Proceedings of the 4th Symposium on Combinatorial Pattern Matching (CPM'93)*, Lecture Notes in Computer Science, Vol. 684, 120–133, 1993.
DOI: 10.1007/BFb0029801

G.M. Landau, J.P. Schmidt, D. Sokol: An Algorithm for Approximate Tandem Repeats, *Journal of Computational Biology*, Vol. 8, No. 1, 1–18, 2001.
DOI: 10.1089/106652701300099038

M.G. Main, R.J. Lorentz: An $O(n \log n)$ Algorithm for Finding All Repetitions in a String, *Journal of Algorithms*, Vol. 5, No. 3, 422–432, 1984.
DOI: 10.1016/0196-6774(84)90021-X

J. Stoye, D. Gusfield: Simple and Flexible Detection of Contiguous Repeats Using a Suffix Tree, *Theoretical Computer Science*, Vol. 270, 843–856, January 2002.
DOI: 10.1016/S0304-3975(01)00121-9

A.3.4 Lowest Common Ancestors and Range Minimum Queries

S. Alstrup, C. Gavoille, H. Kaplan, T. Rauhe: Nearest Common Ancestors: A Survey and a New Distributed Algorithm, *Theory of Computing Systems*, Vol. 37, No. 3, 441–456, 2004.
DOI: 10.1007/s00224-004-1155-5

M.A. Bender, M. Farach-Colton: The LCA Problem Revisited, *Proceedings of the 4th Latin American Symposium on Theoretical Informatics (LATIN'00)*, Lecture Notes in Computer Science, Vol. 1776, 88–94, 2000.
DOI: 10.1007/10719839_9

O. Berkman, U. Vishkin: Recursive Star-Tree Parallel Data Structure, *SIAM Journal on Computing*, Vol. 22, 221-242, 1993.
DOI: 10.1137/0222017

J. Fischer, V. Heun. Space-Efficient Preprocessing Schemes for Range Minimum Queries on Static Arrays, *SIAM Journal on Computing*, Vol. 40, No. 2, 465–492, 2011.
DOI: 10.1137/090779759

B. Schieber, U. Vishkin: On Finding Lowest Common Ancestors: Simplification and Parallelization, *SIAM Journal on Computing*, Vol. 17, 1253–1262, 1988.
DOI: 10.1137/0217079

A.3.5 Construction of Suffix-Arrays

S. Burkhardt, J. Kärkkäinen: Fast Lightweight Suffix Array Construction and Checking, *Proceedings of the 14th Symposium on Combinatorial Pattern Matching (CPM'03)*, Lecture Notes in Computer Science, Vol. 2676, 55–69, 2003.
DOI: 10.1007/3-540-44888-8_5

S. Gog, T. Beller, A. Moffat. M. Petri: From Theory to Practice: Plug and Play with Succinct Data Structures, *Proceedinsg of the 13th International Symposium on Experimental Algorithms (SEA 2014)*, Lecture Notes in Computer Science, Vol. 8504, 326–337, 2014.
DOI: 10.1007/978-3-319-07959-2_28
SDSL Homepage: algo2.iti.kit.edu/gog/docs/html/index.html

D.K. Kim, J.S. Sim, H. Park, K. Park: Linear-Time Construction of Suffix Arrays, *Proceedings of the 14th Symposium on Combinatorial Pattern Matching (CPM'03)*, Lecture Notes in Computer Science, Vol. 2676, 186–199, 2003.
DOI: 10.1007/3-540-44888-8_14

P. Ko, A. Aluru: Space Efficient Linear Time Construction of Suffix Arrays, *Proceedings of the 14th Symposium on Combinatorial Pattern Matching (CPM'03)*, Lecture Notes in Computer Science, Vol. 2676, 200–210, 2003.
DOI: 10.1007/3-540-44888-8_15

J. Kärkkäinen, P. Sanders: Simple Linear Work Suffix Array Construction, *Proceedings of the 30th International Colloquium on Automata, Languages and Programming (ICALP'03)*, Lecture Notes in Computer Science, Vol. 2719, 943–955, 2003.
DOI: 10.1007/3-540-45061-0_73

G. Nong, S. Zhang, W.H. Chan: Two Efficient Algorithms for Linear Time Suffix Array Construction, *IEEE Transaction on Computers*, Vol 60, No. 10, 1471–1484, 2011.
DOI: 10.1109/TC.2010.188

U. Manber, G. Myers: Suffix Arrays: A New Method for On-Line String Searches, *SIAM Journal on Computing*, Vol. 22, 935–948, 1993.
DOI: 10.1137/0222058

Y. Mori: Implementation of SAIS in Different Programming Languages (C/C++, C#, Java). Web: sites.google.com/site/yuta256/sais vom 15.12.2017, zuletzt zugegriffen am 08.01.2018 (momentan nicht verfügbar).

A.3.6 Applications of Suffix-Arrays

M.I. Abouelhoda, S. Kurtz, E. Ohlebusch: The Enhanced Suffix Array and Its Applications to Genome Analysis, *Proceedings of the 2nd Workshop on Algorithms in Bioinformatics (WABI'02)*, Lecture Notes in Computer Science, Vol. 2452, 449–463, 2002.
DOI: 10.1007/3-540-45784-4_35

M.I. Abouelhoda, E. Ohlebusch, S. Kurtz: Optimal Exact String Matching Based on Suffix Arrays, *Proceedings of the 9th International Symposium on String Processing and Information Retrieval (SPIRE'02)*, Lecture Notes in Computer Science, Vol. 2476, 31–43, 2002.
DOI: 10.1007/3-540-45735-6_4

M.I.Abouelhoda, S. Kurtz, E. Ohlebusch: Replacing Suffix Trees with Enhanced Suffix Arrays, *Journal of Discrete Algorithms*, Vol. 2, 53–86, 2004.
DOI: 10.1016/S1570-8667(03)00065-0

M. Burrows, D.J. Wheeler: A Block-Sorting Lossless data Compression Algorithm; *Research Report*, Digital Research Center, SRC-Report 124, 1994.
https://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.html

P. Ferragina, G. Manzini: Indexing Compressed Text; *Journal of the ACM*, Vol. 52, Issue 4, 552–581, 2005
DOI: 10.1145/1082036.1082039

J. Fischer: Combined Data Structure for Previous- and Next-Smaller-Values; *Theoretical Computer Science*, Vol. 412, Issue 22, 2451–2456, 2011.
DOI: 10.1016/j.tcs.2011.01.036

J. Fischer, V. Heun: A New Succinct Representation of RMQ-Information and Improvements in the Enhanced Suffix Array, *Proceedings of the International Symposium on Combinatorics, Algorithms, Probabilistic and Experimental Methodologies (ESCAPE'07)*, Lecture Notes in Computer Science, Vol. 4614, 459–470, Springer, 2007.
DOI: 10.1007/978-3-540-74450-4_41

T. Kasai, G. Lee, H. Arimura, S. Arikawa, K. Park: Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications, *Proceedings of the 12th Symposium on Combinatorial Pattern Matching (CPM'01)*, Lecture Notes in Computer Science, Vol. 2089, 181–192, 2001.
DOI: 10.1007/3-540-48194-X_17

G. Manzini: Two Space Saving Tricks for Linear Time LCP Array Computation, *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory (SWAT'04)*, Lecture Notes in Computer Science, Vol. 3111, 372–383, 2004.
DOI: 10.1007/b98413

G. Navarro, V. Mäkinen: Compressed Full-Text Indexes, *ACM Computing Surveys*, Vol. 39, No. 1, 2007.
DOI: 10.1145/1216370.1216372

K. Sadakane: Succinct Representations of LCP Information and Improvements in the Compressed Suffix Arrays, *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'02)*, 225-232, 2002.

K. Sadakane: Compressed Suffix Trees with Full Functionality, *Theory of Computing Systems*, Vol. 41, No. 4, 589–607, 2007.
DOI: 10.1007/s00224-006-1198-x

A.3.7 Sorting by Reversals

V. Bafna, P.A. Pevzner: Genome Rearrangements and Sorting by Reversals, *SIAM Journal on Computing*, Vol. 25, 272–289, 1996.
DOI: 10.1137/S0097539793250627

P. Berman, S. Hannenhalli, M. Karpinski: A 1.375-Approximation Algorithm for Sorting by Reversals, *Proceedings of the 10th Annual European Symposium on Algorithms (ESA'02)*, Lecture Notes in Computer Science, Vol. 2461, 200–210, 2002.
DOI: 10.1007/3-540-45749-6_21

P. Berman, M. Karpinski: On Some Tighter Inapproximability Results, *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP'99)*, Lecture Notes in Computer Science, Vol. 1644, 200–209, 1999.
DOI: 10.1007/3-540-48523-6_17

D. Christie: A 3/2-Approximation Algorithms for Sorting by Reversals, *Proceedings of the 9th ACM Symposium on Discrete Algorithms (SODA'98)*, 244–252, 1998.

A.3.8 Sorting by Oriented Reversals

D.A. Bader, N.M.E. Moret, M. Yan: A Linear-Time Algorithm for Computing Inversion Distance Between Signed Permutations With an Experimental Study, *Journal of Computational Biology*, Vol. 8, No. 5, 483–491, 2001.
DOI: 10.1089/106652701753216503

A. Bergeron: A Very Elementary Presentation of the Hannenhalli-Pevzner Theory, *Discrete Applied Mathematics*, Vol. 146, No. 2, 134–145, 2005.
DOI: 10.1016/j.dam.2004.04.010

A. Bergeron, J. Mixtacki, J. Stoye: Reversal Distance without Hurdles and Fortresses, *Proceedings of the 15th Symposium on Combinatorial Pattern Matching (CPM'04)*, Lecture Notes in Computer Science, Vol. 3109, 388–399, 2004.
DOI: 10.1007/b98377

P. Berman, S. Hannehalli: Faster Sorting by Reversals, *Proceedings of the 7th Symposium on Combinatorial Pattern Matching (CPM'96)*, Lecture Notes in Computer Science, Vol. 1075, 168–185, 1996.
DOI: 10.1007/3-540-61258-0_14

S. Hannehalli, P. Pevzner: Transforming Cabbage into Turnip: Polynomial Algorithm for Sorting Signed Permutations by Reversals, *Journal of the ACM*, Vol. 46, 1–27, 1999; also in *Proceedings of the 27th Annual ACM Symposium on Computing (STOC'95)*, 178–189, 1995.
DOI: 10.1145/300515.300516

S. Hannehalli, P. Pevzner: To Cut ... or Not to Cut (Applications of Comparative Physical Maps in Molecular Evolution), *Proceedings of the 7th ACM Symposium on Discrete Algorithms (SODA'96)*, 304–313, 1996.

H. Kaplan, R. Shamir, R. Tarjan: Faster and Simpler Algorithm for Sorting Signed Permutations by Reversals, *SIAM Journal on Computing*, Vol. 29, 880–892, 1999; also in *Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms (SODA'97)*, 344–351, 1997.
DOI: 10.1137/S0097539798334207

A.C. Siepel: An Algorithm to Enumerate all Sorting Reversals, *Journal Computational Biology*, Vol. 10, 575–597, 2003; also in *Proceedings of the 6th Annual International Conference on Computational Biology (RECOMB'02)*, 281–290, 2002.
DOI: 10.1145/565196.565233

A.3.9 Sorting by Transpositions

V. Bafna, P.A. Pevzner: Sorting by Transpositions, *SIAM Journal on Discrete Mathematics*, Vol. 11, No. 2, 224–240, 1998.
DOI: 10.1137/S089548019528280X

L.F.I. Cunha, L.A.B. Kowada, R. de A. Hausen, C.M.H. de Figueiredo: A faster 1.375-approximation algorithm for sorting by transpositions, *Journal on Computational Biology*, Vol. 22, No. 11, 1044-56, 2015.
DOI: 10.1089/cmb.2014.0298

I. Elias, T. Hartman: A 1.375-Approximation Algorithm for Sorting by Transpositions, *Proceedings of the Fifth Workshop on Algorithms in Bioinformatics (WABI'05)*, Lecture Notes in Computer Science, Vol. 3692, 204–215, 2005. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, Vol. 3,No. 4, 369–379, 2006.
DOI: 10.1109/TCBB.2006.44

T. Hartman: A Simpler 1.5-Approximation Algorithms for Sorting by Transpositions, *Proceedings of the 14th Symposium on Combinatorial Pattern Matching (CPM'03)*, Lecture Notes in Computer Science, Vol. 2676, 156–169, 2003.
DOI: 10.1007/3-540-44888-8_12

A.3.10 Sorting by Transversals

M. Bader, E. Ohlebusch: Sorting by Weighted Reversals, Transpositions, and Inverted Transpositions, *Proceedings of the 10th Annual International Conference on Research in Computational Molecular Biology (RECOMB'06)*, Lecture Notes in Computer Science, Vol. 3909, 563–577, Springer, 2006.
DOI: 10.1007/11732990_46

N. Eriksen: $(1+\varepsilon)$-Approximation of Sorting by Reversals and Transpositions, *Theoretical Computer Science*, Vol. 289, 517–529, 2002.
DOI: 10.1016/S0304-3975(01)00338-3

Q.-P. Gu, S. Peng, I.H. Sudborough: A 2-Approximation Algorithm for Genome Rearrangements by Reversals and Transpositions, *Theoretical Computer Science*, Vol. 210, 327–339, 1999.
DOI: 10.1016/S0304-3975(98)00092-9

T. Hartman, R. Sharan: A 1.5-Approximation Algorithm for Sorting by Transpositions and Transreversals, *Proceedings of the 4th Workshop on Algorithms in Bioinformatics (WABI'04)*, Lecture Notes in Computer Science, Vol. 3240, 50–61, Springer, 2004.
DOI: 10.1007/978-3-540-30219-3_5

T. Hartman, R.Sharan: A 1.5-Approximation Algorithm for Sorting by Transpositions and Transreversals. *Journal of Computer and System Sciences*, Vol. 70, No. 3, 300–320, 2005.
DOI: 10.1016/j.jcss.2004.12.006

G-H. Lin, G. Xue: Signed Genome Rearrangements by Reversals and Transpositions: Models and Approximations, *Theoretical Computer Science*, Vol. 259, 513–531, 2001
DOI: 10.1016/S0304-3975(00)00038-4

A. Rahmana, S. Shatabdaa, M. Hasan: An approximation algorithm for sorting by reversals and transpositions; *Journal of Discrete Algorithms*, Vol. 6, No. 3, 449–45, 2008.
DOI: 10.1016/j.jda.2007.09.002

A.3.11 Erweiterungen zu Genome Rearrangements

M.A. Bender, D. Ge, S. He, H. Hu, R.Y. Pinter, S. Skiena, F. Swidan: Improved Bounds on Sorting with Length-Weighted Reversals, *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA'04*, 919–928, 2004.

A. Bergeron, J. Mixtacki, J. Stoye: On Sorting by Translocations, *Proceedings of the 9th Annual International Conference on Computational Biology (RECOMB'05)*, Lecture Notes in Computer Science, Vol. 3500, 615–629, Springer, 2005.
DOI: 10.1007/11415770_47

A. Bergeron, J. Mixtacki, J. Stoye: A Unifying View of Genome Rearrangements, *Proceedings of the 6th International Workshop on Algorithms in Bioinformatics, (WABI'06)*, Lecture Notes in Computer Science, Vol. 4175, 163–173, Springer, 2006.
DOI: 10.1007/11851561_16

D. Bryant: A Lower Bound for the Breakpoint Phylogeny Problem, *Proceedings of the 11th Workshop on Combinatorial Pattern Matching, CPM'00*, 235–247, 2000.
DOI: 10.1007/3-540-45123-4_21

A. Caprara: Formulations and Hardness of Multiple Sorting by Reversals, *Proceedings of the 3rd Annual International Conference on Computational Biology (RECOMB'99)*, 84–93, 1999.
DOI: 10.1145/299432.299461

T. Chen, S.S. Skiena: Sorting with Fixed-Length Reversals, *Discrete Applied Mathematics*, Vol .71, 269–295, 1996.
DOI: 10.1016/S0166-218X(96)00069-8

D.A. Christie, R.W. Irving: Sorting Strings by Reversals and by Transpositions, *SIAM Journal on Discrete Mathematics*, Vol. 14, 193–206, 2001.
DOI: 10.1137/S0895480197331995

D.A. Christie: Sorting Permutations by Block-Interchanges, *Information Processing Letters*, Vol. 60, 165–169, 1996.
DOI: 10.1016/S0020-0190(96)00155-X

D.S. Cohen, M. Blum: On the Problem of Sorting Burnt Pancakes, *Discrete Applied Mathematics*, Vol. 61, No. 2, 105–120, 1995.
DOI: 10.1016/0166-218X(94)00009-3

B. DasGupta, T. Jiang, S. Kannan, M. Li, E. Sweedyk: On the Complexity and Approximation of Syntenic Distance, *Discrete Applied Mathematics*, Vol. 88, 59–82, 1998.
DOI: 10.1016/S0166-218X(98)00066-3

N. El-Mabrouk, D. Sankoff: The Reconstruction of Doubled Genomes, *SIAM Journal on Computing*, Vol. 32, 754–792, 2003.
DOI: 10.1137/S0097539700377177

W. Gates, C. Papadimitriou: Bounds for Sorting by Prefix Reversal, *Discrete Mathematics*, Vol. 27, 47–57, 1979.
DOI: 10.1016/0012-365X(79)90068-2

S. Hannehalli: Polynomial-Time Algorithm for Computing Translocation Distance Between Genomes, *Discrete Applied Mathematics*, Vol .71, 137–151, 1996.
DOI: 10.1016/S0166-218X(96)00061-3

S. Hannenhalli, P.A. Pevzner: To cut ... or not to cut (applications of comparative physical maps in molecular evolution) , *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 96)*, 304–313, 1996.

S. Hannenhalli, P.A. Pevzner: Transforming Men into Mice (Polynomial Algorithm for Genomic Distance), *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science (FOCS'95)*, 581–592, 1995.
DOI: 10.1145/640075.640108

L.S. Heath, J.P. Vergara: Sorting by Bounded Block-Moves, *Discrete Applied Mathematics*, Vol. 88, 181–206, 1998.
DOI: 10.1016/S0166-218X(98)00072-9

J. Kleinberg, D. Liben-Nowell: The Syntenic Diameter of the Space of n-Chromosome Genomes, in *Comparative Genomics*, David Sankoff and Joseph H. Nadeau (Eds.), Kluwer Academic Press, 2000.

J. Meidanis, Z. Dias: Genome Rearrangements Distance by Fusion, Fission, and Transposition is Easy, *Proceedings of the 8th Symposium on String Processing and Information Retrieval, SPIRE'01*, 250–253, 2001
DOI: 10.1109/SPIRE.2001.989776

R.Y. Pinter, S. Skiena: Genomic Sorting with Length-Weighted Reversals, *Genome Informatics*, Vol. 13, 103–111, 2002.

Index B

Symbole

−2-Move, 301
0-Move, 301
2-Move, 301

A

a-MSS, 17
Adjazenz, 265, 272
    orientierte, 272
aktives Suffix, 74
Alignment-Distanz, 85
All Maximal Scoring Subsequences, 10, 11
alternierende Zyklenzerlegung, 266
AMSS, 10, 11
$\mathcal{APX}$, 256
atomic suffix tree, 61
ausreichend, 133

B

BAMSS, 27
Baum
    Tiefe, 141
    vollständiger, 155
binärer Baum, 155
Bit-Feld, 244
Bit-Vektor, 244
Blattliste, 98
BMSS, 29
Bounded All Maximum Scoring Subsequences, 27
Bounded Maximal Scoring Subsequence, 29
Breakpoint, 257, 272
    orientierter, 272
Breakpoint Median Problem, 327
Breakpoint-Distanz, 327
Breakpoint-Graph, 264
    erweiterter, 264
Burnt Pancake Flipping, 270
Burrows-Wheeler-Transformation, 231
bzip2, 235

C

Child-Tabelle, 209
compact suffix tree, 61

D

Darstellung eines Wortes, 58
Deletion, 251
Desire-Edges, 264
Divide-and-Conquer, 5
doppelte Reversion, 306
Duplikation, 252
dynamische Programmierung, 4, 143

E

EDIT-Distanz, 85, 116
einfache Hurdle, 281
einfache Transversal-Distanz, 316
Elter-Intervall, 206, 220
Enhanced-Suffix-Array, 209, 215
enthalten von ℓ-Intervallen, 206, 220
erfolgloser Suffix-Link-Walk, 133
erfolgreicher Suffix-Link-Walk, 133
erweiterte orientierte Permutation, 271
erweiterte Permutation, 256
erweiterter Breakpoint-Graph, 264
erweitertes Reality-Desire-Diagram, 273, 298, 308
Euler-Kontur, 140
exaktes Paar, 85
exaktes Repeat, 85

F

fallend rechtsschiefe Partition, 34

fallender Strip, 257
Feld
    inkrementelles, 145
    normalisiertes, 145
Fission, 252
FM-Index, 242
Folge
    linksnegative, 30
    linksschiefe, 51
    rechtsschiefe, 34
Fortress, 282
    starkes, 317
Fusion, 252
Futurama, 270

G

GC-reiche Regionen, 3
Geburtstagsparadoxon, 65
Generisches Median Problem, 326
Gewicht, 321
gewichtete Transversal-Distanz, 316
Gruppe
    orientierte symmetrische, 271
    symmetrische, 254
gute Komponente, 280

H

Hamming-Distanz, 85
Hashfunktion, 65
Hashing, 65
Höhe
    eines Knotens, 159
Huffman-Codierung, 235
Hurdle, 281
    einfache, 281
    starke, 317
Hurdle-Cutting, 295
Hurdle-Merging, 293

I

Incremental Range Minimum Query, 145
inkrementelles Feld, 145
Inorder-Liste, 154
Inorder-Nummerierung, 155, 156
Insertion, 251
Inversion, 252
iterierter rechtsschiefer Zeiger, 43

J

j-ter Block, 120

K

k-difference Repeat, 85
k-mismatch Repeat, 85
k-mismatch Tandem-Repeat, 112
kanonische Lokation, 79
Kante
    orientierte, 275
    unorientierte, 275
Kartesischer Baum, 148
Kette von Suffix-Link-Walks, 133
Kind-Intervall, 206, 220
Knoten
    Höhe, 159
    orientierter, 277
    Tiefe, 141
    unorientierter, 277
Kollision, 65
kompakter Σ^+-Baum, 58
kompaktifizierter Trie, 58
komplementäres Zeichen, 94
Komponente
    gute, 280
    orientierte, 277
    schlechte, 280
    unorientierte, 277
konservierte Regionen, 3
Kopf des Runs, 161
Kreis
    orientierter, 275, 301
    unorientierter, 275, 301

L

ℓ-Index, 206, 222
ℓ-Intervall, 205, 219

Länge eines Tandem-Repeat-Paares, 96
Länge eines Tandem-Repeats, 96
längster gemeinsamer Präfix, 198
lca, 110
lce, 109
lcp, 198
LCP-Intervall vom Typ ℓ, 205, 219
LCP-Intervall-Baum, 206, 220
LCP-Tabelle, 202
least common ancestor, 139
Lempel-Ziv-Zerlegung, 120
linkeste Überdeckung, 117
linksdivers, 90
linksnegative Folge, 30
linksnegativer Zeiger, 31
Linksrotation, 97
  zyklische, 233
linksschiefe Folge, 51
linksverzweigend, 96, 115
linksverzweigendes Tandem-Repeat-Paar, 96, 115
Linkszeichen
  einer Position, 90
  eines Blattes, 90
Lokation, 78
  kanonische, 79
  offene, 79
longest common backward extension, 109
longest common extension, 109, 110
longest common forward extension, 109
longest common prefix, 198
lowest common ancestor, 110, 139
Lowest Common Ancestor Query, 139

M

MASS, 34
Maximal Average Scoring Subsequence, 34, 45
maximal bewertete Teilfolge, 10, 11
Maximal Scoring Subsequence, 1
maximaler Repeat, 88
maximales Paar, 88
Median Problem
  Breakpoint, 327
  generisches, 326
  Reversal, 327
  Steiner, 327
Menge aller maximalen Paare, 88
Menge aller maximaler Repeats, 88
Mensch-Maus-Genom, 3
Min-SBR, 255
Min-SBRT, 307
Min-SBT, 298
Min-SBTR, 307
Min-SOR, 272
Min-SWT, 317
minimal linksnegative Partition, 30
Move-to-Front-Codierung, 235
MSS, 1
MSS(a), 11

N

nearest common ancestor, 139
nested suffix, 72
Next Smaller Values, 226
Next-Tabelle, 226
niedrigster gemeinsamer Vorfahre, 110, 139
normalisiertes Feld, 145
NSV, 226

O

offene Lokation, 79
offene Referenz, 79
orientierte Adjazenz, 272
Orientierte Kante, 275
orientierte Komponente, 277
orientierte Permutation, 271
  erweiterte, 271
orientierte Reversion, 271
orientierte symmetrische Gruppe, 271
orientierter Breakpoint, 272

orientierter Knoten, 277
orientierter Kreis, 275, 301
Overlap-Graph, 276

P

Paar
    exaktes, 85
    maximales, 88
    Tandem-Repeat-, 96
Pancake Flipping, 270
Partition
    fallend rechtsschiefe, 34
    minimal linksnegative, 30
    steigend linksschiefe, 51
PCP-Theorem, 256
Permutation, 254
    als Abbildung, 255
    als Umordnung, 255
    erweiterte, 256
    erweiterte orientierte, 271
    orientierte, 271
    zugehörige unorientierte, 273, 298
Position vom Typ L, 186
Position vom Typ S, 186
Präfix
    längster gemeinsamer, 198
Previous Smaller Values, 226
Previous-Tabelle, 226
primitiv, 97
primitives Tandem-Repeat, 97
probabilistisch verifizierbare Beweise, 256
PSV, 226

Q

Query
    Lowest Common Ancestor, 139
    Range Minimum, 140

R

Range Minimum Query, 140, 218, 228
Rank-Funktion, 244
Reality-Desire-Diagram, 273, 298, 308
    erweitertes, 273, 298, 308
Reality-Edges, 264
Rearrangement Distance, 254
Rechtsrotation, 97
    zyklische, 233
rechtsschiefe Folge, 34
rechtsschiefer Zeiger, 39
    iterierter, 43
rechtsverzweigend, 96, 115
rechtsverzweigendes Tandem-Repeat-Paar, 96, 115
rechtsverzweigendes Teilwort, 72
Referenz, 61
    offene, 79
Repeat, 85
    exaktes, 85
    k-difference, 85
    k-mismatch, 85
    maximaler, 88
    revers-komplementäres, 95
    Tandem, 96
revers-komplementäres Repeat, 95
revers-komplementäres Wort, 94
Reversal Median Problem, 327
Reversion, 252, 271
    doppelte, 306
    orientierte, 271
    sichere, 291
Reversions-Distanz, 316
Reversionsgraph, 327
rightbranching, 72
Run, 161
    Kopf, 161
Run von Tandem-Repeats, 116

S

S-Distanz, 187
SAIS, 197
schlechte Komponente, 280
Score, 285
Select-Funktion, 244
sichere Reversion, 291

Σ-Baum, 57
Σ+-Baum, 57
    kompakter, 58
Singletons, 206, 220
Sorting by Oriented Prefix Reversals, 270
Sorting by Oriented Reversals, 272
Sorting by Prefix Reversals, 270
Sorting by Reversals, 255
Sorting by Reversals and Transpositions, 307
Sorting by Transpositions, 298
Sorting by Transversals, 307
Sorting by Weighted Transversals, 317
stark unorientiert, 317
starke Hurdle, 317
starke Super-Hurdle, 317
starkes Fortress, 317
steigend linksschiefe Partition, 51
steigender Strip, 257
Steiner Reversal Median Problem, 327
Strip, 257
    fallend, 257
    steigend, 257
Suffix
    aktives, 74
    nested, 72
    verschachteltes, 72
Suffix Tree, 60
Suffix Trie, 59
Suffix-Array, 173
    Enhanced-, 215
Suffix-Array-Induced-Sorting, 197
Suffix-Baum, 60
Suffix-Link, 70
Suffix-Link an Position i, 217, 227
Suffix-Link-Intervall, 218, 228
Suffix-Link-Walk, 133
    erfolgloser, 133
    erfolgreicher, 133
    Kette, 133
Super-Hurdle, 281
    starke, 317
symmetrische Gruppe, 254
    orientierte, 271
Synteny, 326

T

Tandem-Arrays, 97
Tandem-Repeat, 96
    k-mismatch, 112
    Länge, 96
    primitives, 97
Tandem-Repeat-Paar, 96
    gleicher Typ, 116
    Länge, 96
    linksverzweigend, 96, 115
    rechtsverzweigend, 96, 115
    Zentrum, 122
Teilwort, 59
    rechtsverzweigendes, 72
Teilwort vom Typ L, 186
Teilwort vom Typ S, 186
Tiefe
    eines Baums, 141
    eines Knotens, 141
Translokation, 252
Translokations-Distanz, 325
Transmembranproteine, 2
Transposition, 252
Transpositions-Distanz, 298
Transversal-Distanz, 308
    einfache, 316
    gewichtete, 316
Transversals, 306
Trie, 57
    kompaktifizierter, 58
Tripel, 193
Typ
    gleicher, 116
Typ L, 180
Typ S, 180

U

überdecken, 117

Überdeckung, 117
    linkeste, 117
ungapped local alignment, 3
unorientiert
    stark, 317
unorientierte Kante, 275
unorientierte Komponente, 277
unorientierter Knoten, 277
unorientierter Kreis, 275, 301

V

verschachteltes Suffix, 72
Vokabular, 116
vollständiger Baum, 155
Vorfahre
    niedrigster gemeinsamer, 139

W

Wavelet-Trees, 248
Weighted Maximal Average Scoring Subsequence, 45
WMASS, 45
Wort
    revers-komplementäres, 94
Worttiefe, 58, 98
WOTD-Algorithmus, 67

X

x-Move, 301

Z

Zeichen
    komplementäres, 94
    Links-, 90
Zeiger
    iterierter rechtsschiefer, 43
    linksnegativer, 31
    rechtsschiefer, 39
Zentrum eines Tandem-Repeats-Paares, 122
zugehörige unorientierte Permutation, 273, 298
Zyklenzerlegung
    alternierende, 266
zyklische Linksrotation, 233
zyklische Rechtsrotation, 233