

# Algorithmen und Datenstrukturen 2

**Eine Zusammenfassung von Bernhard Kabelka  
zur Vorlesung von Prof. Mutzel im WS 2001/02**

Version 1.02, 15. März 2004

Es sei ausdrücklich betont, dass

- (1) dieses Essay ohne das Wissen und die Mitarbeit von Prof. Mutzel entstanden ist,
- (2) trotz großer Anstrengungen seitens des Autors, eine möglichst fehlerfreie und vollständige Zusammenfassung zu liefern, sich Fehler eingeschlichen haben könnten (sollte jemand einen Fehler entdecken, so bittet der Autor um Benachrichtigung, vorzugsweise per eMail an [bernhard@kabelka.net](mailto:bernhard@kabelka.net)),
- (3) die Lektüre dieser Zusammenfassung keinesfalls den persönlichen Besuch der Vorlesung bzw. das Studium des Skriptums ersetzen, sondern bestenfalls ergänzen kann.

Die aktuelle Version dieser Datei ist erhältlich unter:

<http://fsmat.at/~bkabelka/math/compmath/download/algodat2.pdf>

<http://fsmat.at/~bkabelka/math/compmath/download/algodat2.ps.gz>

# Inhaltsverzeichnis

<b>1</b>	<b>Graphen</b>	<b>1</b>
<b>2</b>	<b>Optimierung</b>	<b>2</b>
2.1	Exakte Algorithmen für schwierige Optimierungsprobleme . . . .	2
2.2	Approximative Algorithmen und Gütegarantie . . . . .	3
2.3	Verbesserungsheuristiken . . . . .	4
<b>3</b>	<b>Geometrische Algorithmen</b>	<b>4</b>
<b>4</b>	<b>Suchen in Texten</b>	<b>5</b>
4.1	Das Verfahren von Knuth-Morris-Pratt . . . . .	5
4.2	Das Verfahren von Boyer-Moore . . . . .	6
4.3	Tries . . . . .	6
<b>5</b>	<b>Randomisierte Algorithmen</b>	<b>8</b>
<b>6</b>	<b>Parallele Algorithmen</b>	<b>8</b>

# 1 Graphen

Ein **Graph** ist ein Tupel  $(V, E)$ , wobei  $V$  die (endliche) Menge der **Knoten** und  $E \subseteq V \times V$  die Menge der **Kanten** ist. Man nennt zwei Knoten **benachbart** oder **adjazent**, wenn eine Kante die beiden Knoten verbindet. Jede Kante ist mit ihren beiden Endknoten **inzident**. Die Menge  $N^+(v)$  bezeichnet die Menge aller in  $v$  eingehenden,  $N^-(v)$  die Menge aller von  $v$  ausgehenden Kanten. Der **Knotengrad**  $d(v)$  ist definiert als die Anzahl der zu  $v$  inzidenten Kanten.

Ein Graph kann entweder über eine **Adjazenzmatrix** oder eine **Adjazenzliste** beschrieben werden. Ist der Graph „dünn“ besetzt, so eignet sich letzteres besser, ist er sehr „dicht“, so ist ersteres geeigneter.

Man kann einen Graphen nun auf **Zusammenhang** (d. h. dass jeder Knoten von jedem anderen Knoten aus erreicht werden kann) testen, indem man bei einem Knoten startet, diesen markiert, und zu einem (noch nicht markierten) adjazenten Knoten wechselt. So fährt man rekursiv fort, bis man keinen adjazenten, nicht markierten Knoten mehr findet. Dann „geht“ man seinen Pfad wieder zurück und versucht, von den früher besuchten Knoten zu noch nicht besuchten Knoten zu gelangen. Sind am Ende dieser Prozedur alle Knoten markiert, so ist der Graph zusammenhängend.

Auf ähnliche Weise kann man die **Zusammenhangskomponenten** bestimmen: Man muss nur nach dem ersten „Durchlauf“ mit einem noch nicht besuchten Knoten analog weitermachen – allerdings muss man die Knoten anders markieren. So fährt man fort, bis alle Knoten markiert sind.

Auch die **Kreissuche** verläuft ähnlich: Man überprüft nach dem „Weitergehen“ zu einem adjazenten, noch nicht markierten Knoten einfach, ob man zu einem markiertem Knoten (außer dem unmittelbaren Vorgänger, den man soeben verlassen hat) zurückkehren kann. Wenn ja, dann hat man einen Kreis gefunden.

Der **Algorithmus von Kruskal** wird zur Bestimmung eines minimalen Spannbaumes eines Graphen verwendet: Hat man einen (zusammenhängenden) Graphen  $(V, E)$  gegeben, wobei jeder Kante  $e_i$  ein Kantengewicht  $w_i$  zugeordnet ist, so ist der minimale Spannbaum ein zusammenhängender, kreisfreier Graph, der alle Knoten miteinander verbindet, und dessen Gesamtgewicht minimal ist. Nach Kruskal sortiert man die Kanten gemäß ihrem Gewicht aufsteigend, und nimmt iterativ die nächsthöhere Kante in den Spannbaum auf, sofern dadurch kein Kreis erzeugt wird. So erhält man den minimalen Spannbaum.

Dasselbe Problem löst auch der **Algorithmus von Prim**: Hier wird ein beliebiger Startknoten  $s$  gewählt, und  $C = \{s\}$  gesetzt. Anschließend sucht man rekursiv eine Kante  $e = (u, v)$  mit  $u \in C$  und  $v \in V \setminus C$  und minimalem Gewicht  $w_e$ , und fügt  $v$  zu  $C$  hinzu, bis  $|C| = |V|$ .

Diese beiden Algorithmen sind **Greedy-Algorithmen** – das sind „gierige“ Verfahren zur Lösung von Optimierungsaufgaben, wobei in jedem Schritt die lokal beste Lösung gewählt wird, und diese einmal getroffene Wahl nie wieder

geändert wird. Solche Algorithmen führen im Allgemeinen jedoch nur in seltenen Fällen zur optimalen Lösung.

## 2 Optimierung

Bei der **Optimierung** geht es darum aus mehreren Lösungen die (in einem gewissen Sinn) „optimale“ zu finden, z. B. minimale Spannbäume, das Handlungsreisendenproblem, das Rucksackproblem, etc.

### 2.1 Exakte Algorithmen für schwierige Optimierungsprobleme

#### 2.1.1 Enumerationsverfahren

Bei den **Enumerationsverfahren** werden alle Lösungen aufgezählt und aus diesen dann die optimale ausgewählt. Allerdings kann das Bestimmen aller Lösungen sehr lange dauern.

Man kann dieses Verfahren zum Beispiel auf das **Rucksackproblem** anwenden: Man hat eine Menge von  $N$  Gegenständen  $i$  mit Gewicht  $w_i$  und Wert  $c_i$  sowie einen Rucksack mit Fassungsvermögen  $K$  gegeben. Nun geht es darum, die Gegenstände mit möglichst hohem Gesamtwert einzupacken, ohne die Kapazität des Rucksacks zu überschreiten.

Auch das **Acht-Damen-Problem** kann mittels Enumeration gelöst werden. Es geht dabei darum, acht Damen – oder allgemeiner:  $n$  Damen – so auf einem  $n \times n$ -Schachbrett zu platzieren, dass sie einander nicht bedrohen. Bei der Enumeration sollte man allerdings beachten, dass in jeder Zeile und jeder Spalte genau eine Dame stehen muss, damit die Aufgabe überhaupt erfüllt werden kann. Beachtet man das nicht, so erreicht der Aufwand  $\Theta(n^{2n+2})$ .

#### 2.1.2 Dynamische Programmierung

Bei der **dynamischen Programmierung** zerlegt man das Problem in Teilprobleme, die jedoch voneinander abhängig sind.

Mit ihrer Hilfe kann die Lösung des Rucksackproblems deutlich vereinfacht werden: Man betrachtet zuerst nur die ersten  $l$  Gegenstände und merkt sich für alle möglichen Werte  $c$  diejenigen Lösungen, die das kleinste Gewicht ergeben. In der Praxis macht man das durch einen Entscheidungsbaum: Die Verzweigungen vor der  $i$ -ten Ebene entscheiden darüber, ob das  $i$ -te Element aufgenommen wird oder nicht. In den Knoten wird das jeweilige  $(c, w)$ -Paar gespeichert. Treten in einer Ebene Paare mit gleichem  $c$ -Wert auf, so wird nur jene Lösung mit kleinerem Gewicht weiterverfolgt, die andere wird verworfen. Gleiches geschieht mit Paaren  $(c, w)$  mit  $w > K$ . In der letzten Ebene sucht man dann unter jenen Paaren  $(c, w)$  mit  $w \leq K$  dasjenige heraus, das den größten Wert besitzt, und hat somit die Lösung gefunden.

### 2.1.3 Branch-and-Bound

Bei **Branch-and-Bound** geht man von einem Problem aus, von dem eine minimale Lösung gesucht werden soll (Maximierungsaufgaben können durch Vorzeichenumkehr auch so behandelt werden).

Zunächst sucht man (z.B. mit Hilfe einer Heuristik – siehe Abschnitt 2.2) eine zulässige, nicht notwendigerweise optimale Lösung mit Wert  $U$  und berechnet eine untere Schranke  $L$  für alle möglichen Lösungswerte. Ist  $U = L$ , so ist man fertig.

Andernfalls wird die Lösungsmenge partitioniert (*Branching*) und die Heuristiken werden auf die Teilmengen angewandt (*Bounding*). Man berechnet sich wieder eine untere Schranke  $L$  für alle möglichen Lösungswerte (der jeweiligen Teilmenge). Ist  $L \geq U$ , so muss man keine weiteren Lösungen aus dieser Teilmenge suchen und erspart sich somit weitere Enumeration.

Die Schwierigkeit bei dieser Art der Optimierung liegt meist in einer „guten“ Aufteilung der Lösungsmenge. Darüber hinaus sollten die berechneten unteren Schranken  $L$  möglichst gut sein, um möglichst viele Teilprobleme ausschließen zu können.

## 2.2 Approximative Algorithmen und Gütegarantie

Eine **Heuristik** ist ein Verfahren zur Bestimmung einer (nicht notwendigerweise optimalen) Lösung eines Minimierungs- bzw. Maximierungsproblems.

Ziel dabei ist es natürlich, möglichst nahe ans Optimum heranzukommen. Ist  $P$  eine Instanz des zu lösenden Problems,  $c_{OPT}(P)$  der Optimalwert für  $P$  und  $c_A(P)$  der Wert der Lösung, die mit Hilfe der Heuristik  $A$  gefunden wurde, so nennt man  $A$  einen  **$\varepsilon$ -approximativen Algorithmus**, wenn für alle Probleminstanzen  $P$  gilt:

$$\frac{c_A(P)}{c_{OPT}(P)} \leq \varepsilon \quad \text{bzw.} \quad \frac{c_A(P)}{c_{OPT}(P)} \geq \varepsilon$$

(für Minimierungsaufgaben)                      (für Maximierungsaufgaben)

$\varepsilon$  heißt dann **Gütegarantie** des Algorithmus  $A$ .

Ein Beispiele für einen  $\varepsilon$ -approximativen Algorithmus ist die First-Fit-Heuristik für das Bin-Packing-Problem ( $\varepsilon \sim \frac{17}{10}$ ).

Die **Spanning-Tree-Heuristik** ( $\varepsilon = 2$ ) wird zur Lösung des euklidische Travelling Salesman Problem (TSP) benutzt: Zuerst bestimme man einen minimalen Spannbaum und verdopple anschließend dessen Kanten. Im entstanden Graphen suche man nun eine **Eulertour** (d. h. eine Tour, die jede Kante des Graphen genau einmal enthält) und gebe dieser eine Orientierung. Dann starte man bei einem beliebigem Knoten  $p$ , markiere diesen und setze  $T = \emptyset$ . Dann folge man der Eulertour, bis man auf einen nichtmarkierten Knoten  $q$  trifft, markiere diesen, setze  $T = T \cup (p, q)$  sowie  $p = q$  und fahre solange fort, bis

alle Knoten markiert sind. Fügt man nun zu  $T$  noch jene Kante hinzu, die den letzten mit dem ersten besuchten Knoten verbindet, hat man eine Lösung des TSP gefunden.

Die **Christophides-Heuristik**, die ebenfalls für das euklidische TSP verwendet wird, erreicht durch ein geschickteres Vorgehen sogar  $\varepsilon = \frac{3}{2}$ .

### 2.3 Verbesserungsheuristiken

Verbesserungsheuristiken werden, wie der Name schon vermuten lässt, zur Verbesserung einer gegebenen Lösung eines Optimierungsproblems verwendet.

Bei **Austauschverfahren** werden aus einer gegebenen Lösung  $L$  einige Elemente entfernt, um eine Teillösung  $T$  zu erhalten. Anschließend versucht man, alle Lösungen zu bestimmen, die  $T$  enthalten, und wählt aus all diesen die beste. Dabei kann es allerdings vorkommen, dass es nicht mehr möglich ist, sogenannten „lokalen Optima“ zu entkommen.

Dieses Manko beseitigt **Simulated Annealing** zumindest zum Teil: Hierbei wird (ausgehend von einer gegebenen Lösung  $L$ ) in jedem Schritt eine „Nachbarlösung“  $L'$  bestimmt, die immer dann übernommen wird, wenn ihre Bewertung besser ist als jene der ursprünglichen Lösung. Schlechtere Lösungen werden allerdings auch mit einer gewissen Wahrscheinlichkeit akzeptiert (eben um lokalen Optima zu entkommen), wobei üblicherweise nur geringfügig schlechtere Lösungen eher übernommen werden als deutlich schlechtere. Außerdem nimmt diese Wahrscheinlichkeit mit fortschreitender Zeit noch zusätzlich ab.

**Evolutionäre Algorithmen** wiederum versuchen die Prinzipien der Evolution – Selektion, Rekombination und Mutation – zu simulieren. Ausgehend von einer Vielzahl von Lösungen (und nicht einer einzigen!) werden in jedem Schritt die besseren Lösungen zwar mit einer höheren Wahrscheinlichkeit selektiert als die schlechteren, aber auch letztere haben sozusagen „eine Chance“. Dann wird aus zwei „Elternlösungen“ (unter Umständen) eine neue Lösung kombiniert, bevor es schließlich (eventuell) durch zufällige kleine Änderungen zu einer Mutation einer oder mehrerer Lösungen kommt.

## 3 Geometrische Algorithmen

Oftmals tritt das Problem auf, dass mit gewissen „mehrdimensionalen“ Daten umgegangen werden muss. Zwei wichtige Grundprinzipien in diesem Bereich werden im Folgenden erläutert:

Das **Scan-Line-Prinzip** kann dann verwendet werden, wenn Datenobjekte in der Ebene gegeben sind. Die Grundstrategie dabei ist es, eine vertikale Gerade (beispielsweise) von links nach rechts über den Bereich zu führen, und jeweils nur die Situation entlang der Scan-Line zu betrachten. Hierbei müssen auch nicht alle möglichen  $x$ -Werte betrachtet werden, da nur an gewissen Punkten

„etwas passiert“ (Verwendet man dieses Prinzip beispielsweise zur Bestimmung der Schnittpunkte von Geradenstücken, so sind nur jene  $x$ -Werte interessant, an denen eine Strecke beginnt bzw. endet, sowie die  $x$ -Werte der Schnittpunkte selbst). Mit dieser Methode kann man das zweidimensionale Problem in eine dynamische Folge von eindimensionalen Problemen transformieren, die (unter Umständen) leichter zu behandeln sind.

**Mehrdimensionale Bereichssuche** kann am besten mit mehrdimensionalen Bäumen realisieren. Dies sei kurz am zweidimensionalen Beispiel erläutert: Die Daten werden ähnlich wie bei eindimensionalen Suchbäumen (siehe Zusammenfassung zu „Algorithmen und Datenstrukturen 1“ bzw. „Rekursive Prozeduren und flexible Datenstrukturen“, Kapitel 3.1) aufgeteilt, allerdings verwendet man im zweidimensionalen Fall immer abwechselnd die  $x$ - und die  $y$ -Koordinate als Schlüsselwert. Wählt man hierbei für jeden Teilbaum genau den Median – entweder bezüglich der  $x$ - oder bezüglich der  $y$ -Koordinate – als Wurzel, so entsteht in der Zeit  $\Theta(n \cdot \log_2 n)$  ein balancierter Baum. Hat man einmal die Ebene auf diese Weise in kleine Rechtecke geteilt (denn so kann diese Aufteilung interpretiert werden), lässt sich das Suchen in der Zeit  $\Theta(\sqrt{n} + R)$  bewerkstelligen (wobei  $R$  die Anzahl der Datenpunkte ist, die in dem durchsuchten Bereich liegen).

## 4 Suchen in Texten

Hierbei geht es darum, in einer gegebenen Zeichenfolge  $T$ , die aus  $N$  Buchstaben des (endlichen) Alphabets  $\Sigma$  aufgebaut ist, ein gewisses Muster  $P$  der Länge  $M$  zu finden.

Ein naives Verfahren, das das leistet, beginnt an der Stelle 1 das Muster anzulegen, und vergleicht so lange die Buchstaben des Musters mit jenen der Zeichenfolge  $T$ , bis keine Übereinstimmung mehr vorliegt bzw. ein Vorkommen des Musters gefunden wurde. Anschließend wird das Muster an der Stelle 2 angelegt, usw. Dieser Algorithmus hat allerdings den Nachteil, dass er bereits eingelesene Information nicht wiederverwertet, und hat daher die Worst-Case-Laufzeit von  $O(M \cdot N)$ .

### 4.1 Das Verfahren von Knuth-Morris-Pratt

Die Idee der Wiederverwertung bereits gewonnener Information greift der Verfahren von **Knuth-Morris-Pratt** auf, indem er im Falle eines Mismatches das Muster eventuell um mehr als eine Stelle nach hinten verschiebt: Es mögen die ersten  $q$  Stellen des Musters mit den letzten  $q$  gelesenen Zeichen im Text übereinstimmen, das nächste Zeichen im Text sei aber verschieden vom  $(q + 1)$ -ten Zeichen des Musters. Weiters bezeichne  $P_q$  das Teilmuster  $P[1] \dots P[q]$ . Man sucht nun das längste Endstück (**Suffix**) des Teilmusters  $P_q$  mit Länge  $l < q$ , das gleichzeitig auch Anfangsstück (**Präfix**) des Musters ist. Das Muster kann dann um  $q - l$  Stellen verschoben werden. Man erreicht so eine Laufzeit von  $\Theta(N + M)$ .

## 4.2 Das Verfahren von Boyer-Moore

Beim Verfahren von **Boyer-Moore** werden zwei verschiedene Verschiebungskriterien berechnet, und dann das größere der beiden verwendet. Ein weiterer Unterschied zu Knuth-Morris-Pratt liegt in der Abarbeitung der Daten: Die Muster wird weiterhin von links nach rechts angelegt, die Zeichen werden allerdings von rechts nach links verglichen. Das hat den Vorteil, dass das Muster oft um seine ganze Länge  $M$  verschoben werden kann.

Es gelte nun, dass der erste Mismatch an der Stelle  $j$  des Musters aufgetreten ist. Das Zeichen an dieser Stelle im Text sei  $c$ . In diesem Fall kann man das Muster zumindest so weit nach rechts schieben, bis das Zeichen  $c$  im Text über dem rechtesten Zeichen gleich  $c$  im Muster ist. Kommt  $c$  im Muster nicht vor, so kann das Muster bis hinter die Position von  $c$  im Text verschoben werden. Implementiert man allerdings nur diese sogenannte **Last-Verschiebung**, so ist die Laufzeit im Worst Case wieder  $O(M \cdot N)$ .

Deswegen verwendet man noch die sogenannte **Suffix-Verschiebung**, das ist die kleinste Verschiebung  $s$ , sodass gilt:

$$P[k - s] = P[k] \quad \forall k \in \{j + 1, \dots, M\}$$

Mit anderen Worten verschieben wird das Muster so weit nach rechts, bis die ersten Zeichen im Muster mit den bisher untersuchten gemachten Zeichen im Text übereinstimmen.

Kombiniert man nun Last- und Suffix-Verschiebung, so erreicht man eine Laufzeit von  $\Theta(N + M)$ .

## 4.3 Tries

Der **Trie** (vom englischen „retrieve“) ist eine wichtige Datenstruktur in der Textverarbeitung.

### 4.3.1 Radix Trie

Ein **Radix Trie** ist ein binärer Baum, bei dem die Daten nur in den Blättern gespeichert sind. Die inneren Knoten enthalten nur Verweise  $next[0]$  und  $next[1]$  auf die Nachfolgeknoten. Jeder Schlüsselwert wird durch eine Bitfolge  $S_1 \dots S_l$  von fixer Länge  $l$  repräsentiert.

In einem Radix Trie gilt:

- (1) Ein inneren Knoten hat mindestens einen Nachfolger.
- (2) Ein innerer Knoten mit nur einem Nachfolger verweist immer auf einen anderen inneren Knoten.
- (3) Ein Radix Trie hat immer maximal  $l + 1$  Ebenen und es können  $2^l$  Datensätze gespeichert werden.

- (4) Für jeden inneren Knoten der Ebene  $i$  ( $i = 1, 2, \dots, l$ ) gilt, dass in seinem linken Teilbaum nur Datensätze mit  $S_i = 0$  und in seinem rechten Teilbaum nur Datensätze mit  $S_i = 1$  gespeichert sind.

Nachteil des Radix Trie ist jedoch, dass bei wachsender Schlüssellänge die Baumhöhe sehr rasch wächst.

### 4.3.2 Indexed Trie

Ein **Indexed Trie** wird zur Speicherung von Worten (z. B. für ein Wörterbuch zur Rechtschreibüberprüfung) verwendet. Nun besteht jeder innere Knoten aus einem Feld, das für jeden Buchstaben  $c$  aus dem Alphabet  $\Sigma$  einen Verweis *next* auf den Nachfolgeknoten sowie eine Boole'sche Variable *end*, die angibt, ob mit dem entsprechenden Buchstaben ein gültiges Wort endet, enthält. Die maximale Tiefe eines Radix Tries ist daher die Länge des längsten darin gespeicherten Wortes.

### 4.3.3 Linked Trie

Ein **Linked Trie** ähnelt sehr einem Indexed Trie mit dem einzigen Unterschied, dass für jeden Knoten nur jene  $c \in \Sigma$  gespeichert werden, für die *end*  $\neq$  False oder *next*  $\neq$  NULL ist.

### 4.3.4 Suffix Compression

Für einen Indexed oder Linked Trie kann man eine **Suffix Compression** durchführen, d. h. jene Knoten, die für alle  $c \in \Sigma$  dieselben *end*- und *next*-Einträge haben, werden zu einem Knoten zusammengefasst, und von mehreren Elternknoten wird auf ein und denselben Unterknoten verwiesen. Dadurch geht die Baumstruktur verloren, und es können keine weiteren Worte mehr eingefügt werden. Allerdings wird (unter Umständen) viel Speicherplatz eingespart.

### 4.3.5 Packed Trie

In einem **Packed Trie** werden die Informationen nicht mehr in einzelnen Feldern (für jeden Knoten eines), sondern in einem einzigen Feld „ineinander verzahnt“ gespeichert. Jeder Eintrag besteht nun aus einem Buchstaben  $c$ , einer Boole'schen Variable *end* und einem Verweis *next*, der jetzt allerdings nicht mehr ein Zeiger auf einen Nachfolgeknoten, sondern der Feldindex des ersten Buchstaben des Nachfolgeknoten ist. So wird im Allgemeinen eine große Menge an Speicherplatz (im Gegensatz zum Indexed Trie) eingespart.

## 5 Randomisierte Algorithmen

Ein **randomisierter Algorithmus** ist ein deterministischer Algorithmus, der zusätzlich Zufallsexperimente durchführen kann. Man unterscheidet hierbei zwei verschiedene Arten von randomisierten Algorithmen: Während ein **Las-Vegas-Verfahren** immer ein korrektes Ergebnis liefert, ist das Ergebnis eines **Monte-Carlo-Verfahrens** mit einer gewissen Fehlerwahrscheinlichkeit behaftet.

Ein Beispiel für ein Monte-Carlo-Verfahren ist der Algorithmus von **Miller-Rabin** zum Primzahltest, der  $s$  Mal den (kleinen) Satz von Fermat

$$a^{n-1} \equiv \text{mod } n \quad \forall a \in 1, \dots, n-1, \text{ falls } n \text{ Primzahl}$$

für zufällig gewählte Zahlen  $a$  anwendet.

Die Laufzeit des Algorithmus beträgt  $O(s \cdot k^3)$ , wobei  $k$  die Anzahl der Bits in der Binärdarstellung von  $n-1$  ist. Das ist eine deutliche Verbesserung gegenüber der Laufzeit des naiven Algorithmus (Division durch alle ganzen Zahlen kleiner oder gleich  $\sqrt{n}$ ).

Schließlich beträgt die Fehlerwahrscheinlichkeit dieses Algorithmus höchstens  $2^{-s}$ , in der Praxis ist sie sogar noch deutlich kleiner.

## 6 Parallele Algorithmen

In diesem Kapitel geht es um den Einsatz nicht nur eines, sondern vieler Prozessoren zur Lösung eines Problems. Es wird dabei von dem Modell einer PRAM (**Parallel Random Access Machine**) ausgegangen, bei der jeder Prozessor einen lokalen Speicherplatz besitzt, und außerdem noch ein gemeinsamer Speicher existiert. Ein Rechenschritt einer PRAM geht dann wie folgt vor sich:

- (1) Einlesen eines Datums vom gemeinsamen Speicher in den lokalen Speicher (**Lese**phase)
- (2) Durchführung der (konstant vielen) Rechenoperationen (**Rechen**phase)
- (3) Schreiben eines Datums vom lokalen Speicher in den gemeinsamen Speicher (**Schreib**phase)

In Bezug auf die Lese- und Schreibzugriffe kann man eine PRAM in unterschiedliche Klassen einteilen: Eine CR-PRAM (**Concurrent Read**) erlaubt gleichzeitige Lesezugriffe, eine CW-PRAM (**Concurrent Write**) erlaubt gleichzeitige Schreibzugriffe. Im Gegensatz dazu spricht man von einer ER- (**Exclusive Read**) bzw. EW-PRAM (**Exclusive Write**), wenn keine gleichzeitigen Lese- bzw. Schreibzugriffe erlaubt sind.

Es stellt sich nun die Frage, ob die entsprechend größere Anzahl an Prozessoren in einem angemessenen Verhältnis zum Zeitgewinn steht. Dazu bezeichne im Folgenden  $t_s(n)$  die sequentielle und  $t_p(n)$  die parallele Laufzeit, sowie  $p(n)$  die Anzahl der Prozessoren bei der parallelen Berechnung. Dann definiert man

- die **Arbeit** als  $w_p(n) := t_p(n) \cdot p(n)$ ,
- die **Effizienz** als  $\frac{\text{Arbeit sequentiell}}{\text{Arbeit parallel}}$  und
- den **Speedup** als  $\frac{\text{Laufzeit sequentiell}}{\text{Laufzeit parallel}}$ .

Man nennt einen parallelen Algorithmus

- **optimal**, wenn gilt:  $t_p(n) = O(\log^k n) \wedge w_p(n) = O(t_s(n))$
- **effizient**, wenn gilt:  $t_p(n) = O(\log^k n) \wedge w_p(n) = O(t_s(n) \cdot \log^k n)$

In der Vorlesung wurde die parallele Minimum-Berechnung und die parallele Präfixsummen-Berechnung genauer besprochen, worauf hier nicht näher eingegangen wird (bei Interesse möge das Skriptum studiert werden).

## Index

- Acht-Damen-Problem, 2
- Adjazenzliste, 1
- Adjazenzmatrix, 1
- Algorithmus
  - $\varepsilon$ -approximativer, 3
  - evolutionärer, 4
  - paralleler, 8
    - effizienter, 9
    - optimaler, 9
  - randomisierter, 8
- Arbeit, 9
- Austauschverfahren, 4
  
- Bereichssuche, 5
- Boyer-Moore-Verfahren, 6
- Branch-and-Bound, 3
  
- Christophides-Heuristik, 4
  
- dynamische Programmierung, 2
  
- Effizienz, 9
- Enumerationsverfahren, 2
- Eulertour, 3
  
- Gütegarantie, 3
- Graph, 1
- Greedy-Algorithmus, 1
  
- Heuristik, 3
  
- Kante, 1
- Knoten, 1
  - adjazenter, 1
  - inzidenter, 1
- Knotengrad, 1
- Knuth-Morris-Pratt-Algorithmus, 5
- Kreissuche, 1
- Kruskal-Algorithmus, 1
  
- Las-Vegas-Verfahren, 8
- Last-Verschiebung, 6
  
- Miller-Rabin-Algorithmus, 8
- Monte-Carlo-Verfahren, 8
  
- Optimierung, 2
  
- Parallel Random Access Machine, *siehe* PRAM
- Präfix, 5
- PRAM, 8
  - Concurrent Read, 8
  - Concurrent Write, 8
  - Exclusive Read, 8
  - Exclusive Write, 8
- Prim-Algorithmus, 1
  
- Rucksackproblem, 2
  
- Scan-Line-Prinzip, 4
- Simulated Annealing, 4
- Spanning-Tree-Heuristik, 3
- Speedup, 9
- Suffix, 5
- Suffix Compression, 7
- Suffix-Verschiebung, 6
  
- Trie, 6
  - Indexed Trie, 7
  - Linked Trie, 7
  - Packed Trie, 7
  - Radix Trie, 6
  
- Zusammenhang, 1
- Zusammenhangskomponente, 1