Inhaltsverzeichnis

1 Prädikatenlogik **1**

1.1 Definition 1

1.1.1 Syntax der Prädikatenlogik 1

1.1.2 Definition einer PL-Formel 1

1.1.3 Vereinfachte Schreibweise einer Formel 2

1.1.4 Definition der prädikatenlogischen Semantik 2

1.1.5 Definition der Semantik, Teil zwei 4

2 Datentypen **6**

2.1 Abstrakte Datentypen 6

2.1.1 Sigma-Algebra 7

2.1.2 Bool 7

2.1.3 Keller 7

2.1.4 Schlange 8

2.1.5 Liste 8

3 Gierig-Algorithmen **10**

3.1 Einführung 10

3.2 Minimale, zusammenhängende Bäume 12

3.2.1 Einführung 12

3.2.2 Kruskalscher Algorithmus 14

3.2.3 Primscher Algorithmus 17

3.3 Kürzeste Pfade 19
3.4 Zeitplanerstellung (Scheduling) 22
3.4.1 Zeitplanerstellung ohne Schlußtermine 22
3.4.2 Zeitplanerstellung mit Schlußterminen 23
3.5 Greedy-Heuristik 26
3.5.1 Einfärben eines Graphen 26
3.5.2 Das Handlungsreisende-Problem 27

4 Teile-und-Herrsche (Divide and Conquer) 28
4.1 Einführung 28
4.2 Bestimmung des Grenzwertes 31
4.3 Selektion und Median 32
4.4 Langzahlarithmetik 37
4.5 Matrixmultiplikation 38

5 Dynamisches Programmieren 41
5.1 Einführung 41
5.2 Optimale binäre Suchbäume 42
5.3 Das Problem des Handlungsreisenden 47
5.4 Verkettete Matrixmultiplikation 50

6 Probabilistische Algorithmen 53
6.1 Einführung 53
6.2 Macao Algorithmen 55
6.2.1 Allgemeine Kommentare 55
6.2.2 Nähestes-Paar-Algorithmus 56
6.3 Monte Carlo Algorithmen 58
6.4 Las Vegas Algorithmen 60

7 Vorbestimmung und Vorberechnung 62
7.1 Vorbestimmung 62

7.1.1 Einführung 62
7.1.2 Vorgänger in einem Wurzelaum 65
7.1.3 Wiederholte Auswertung eines Polynoms 66
7.2 Vorberechnung für Zeichenreihe-Suchprobleme 68

Kapitel 1

Prädikatenlogik

1.1 Definition

1.1.1 Syntax der Prädikatenlogik

Definition (Syntax der Prädikatenlogik)
Eine *Variable* hat die Form x_i mit $i = 1, 2, 3 \ldots$. Ein *Prädikatensymbol* hat die Form P_i^k mit $i = 1, 2, 3 \ldots$ und ein *Funktionssymbol* hat die Form f_i^k mit $i = 1, 2, 3 \ldots$. Hierbei heißt i jeweils der *Unterscheidungsindex* und k die *Stelligkeit*. Wir definieren nun die *Terme* durch einen Induktiven Prozess:

1. Jede Variable ist ein Term.

2. falls f ein Funktionssymbol ist mit Stelligkeit k, und falls $t_1, \ldots, t_k$ Terme sind, so ist auch $f(t_1, \ldots, t_k)$ ein Term.

Hierbei sind auch *Konstanten* als Funktionssymbole mit Stellenzahl 0 eingeschlossen. Bei Konstanten werden die Klammern weggelassen.

1.1.2 Definition einer PL-Formel

Nun können wir wiederum induktiv *Formeln* definieren.

1. Falls P ein Prädikatensymbol der Stelligkeit k ist und falls $t_1, \ldots, t_k$ Terme sind, dann ist $P(t_1, \ldots, t_k)$ eine Formel.

2. Für jede Formel F ist auch $\neg F$ eine Formel.

3. Für alle Formeln F und G sind auch $(F \wedge G)$ und $(F \vee G)$ Formeln.

4. Falls x eine Variable ist und F eine Formel, so sind auch $\exists xF$ und $\forall xF$ Formeln.

Atomare Formeln nennen wir genau die Formeln, welche nach 1. aufgebaut sind.

Falls F eine Formel ist und F als Teil einer Formel G auftritt, so heißt F *Teilformel* von G.

Eine Variable x, welche in einer Formel F vorkommt heißt *gebunden*, falls x in einer Teilformel von F in der Form $\exists xG$ oder $\forall xG$ vorkommt. Andernfalls heißt dieses Vorkommen von x frei.

Ein Beispiel für ein PL-Formel

$$F = (P_1^1(x_1) \wedge \forall x P_2^1(x_1))$$

ist eine Formel in der x_1 sowohl frei als auch gebunden vorkommt.

Eine Formel ohne Vorkommen einer freien Variable heißt *geschlossen* oder eine *Aussage*. Das Symbol $\exists$ wird *Existenzquantor* und $\forall$ *Allquantor* genannt. Die *Matrix* einer Formel F ist diejenige Formel, die man aus F erhält, indem jedes Vorkommen von $\exists$ und $\forall$, samt der dahinterstehenden Variablen gestrichen wird. Symbolisch bezeichnen wir die Matrix der Formel F mit F^*.

1.1.3 Vereinfachte Schreibweise einer Formel

Wir vereinbaren eine vereinfachende Schreibweise wie folgt:

u, v, w, x, y, z	stehen für Variablen
a, b, c	stehen für Konstanten
f, g, h	stehen für Funktionssymbole
P, Q, R	stehen für Prädikatensymbole
$(F \rightarrow G)$	steht für $(\neg F \vee G)$
$(F \leftrightarrow G)$	steht für $((F \wedge G) \vee (\neg F \wedge \neg G))$

1.1.4 Definition der prädikatenlogischen Semantik

Definition (Semantik der Prädikatenlogik)
Eine *Struktur* ist eine Paar $\mathcal{A}\,(U_{\mathcal{A}}, I_{\mathcal{A}})$, wobei $U_{\mathcal{A}}$ eine beliebige aber nicht lee-

re Menge ist, das *Universum* (auch *Grundmenge* oder *Grundbereich*). Ferner ist $I_\mathcal{A}$ eine Abbildung die

- jedem k stelligen Prädikatsymbol P ein k stelliges Prädikat über $U_\mathcal{A}$ zuordnet.
- jedem k stelligen Funktionssymbol f eine k stellige Funktion über $U_\mathcal{A}$ zuordnet.
- jeder Variablen x eine Variable über der Grundmenge $U_\mathcal{A}$ zuordnet.

Wir schreiben abkürzend $P^\mathcal{A}$ statt $I_\mathcal{A}(P)$, sowie $F_\mathcal{A}$ statt $I_\mathcal{A}(F)$ und für $I_\mathcal{A}(x)$ einfacher $x^\mathcal{A}$.

Sei F eine Formel und $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$ eine Struktur. $\mathcal{A}$ heißt zu F *passend*, falls $I_\mathcal{A}$ für alle in F vorkommenden Symbole und freien Variablen definiert ist.

Ein Beispiel für eine Struktur

$$F = \forall x P(x, f(x)) \wedge Q(g(a, z))$$

ist eine Formel. Eine zu F passende Struktur ist z.B.:

$$U_\mathcal{A} = \{1, 2, 3, \ldots\},$$

$$I_\mathcal{A}(P) = \{(m, n) \mid m, n \in U_\mathcal{A} \text{ und } m < n\}$$

$$I_\mathcal{A}(Q) = \{n \in U_\mathcal{A} \mid n \text{ ist Primzahl}\},$$

$$I_\mathcal{A}(f) = f^\mathcal{A}(n) = n + 1,$$

$$I_\mathcal{A}(g) = g^\mathcal{A}(m, n) = m + n,$$

$$I_\mathcal{A}(a) = 2,$$

$$I_\mathcal{A}(x) = x \in U_\mathcal{A},$$

Beachte: x kommt frei vor!

1.1.5 Definition der Semantik, Teil zwei

Definition (Semantik der Prädikatenlogik - Fortsetzung)
Sei F eine Formel und $\mathcal{A}$ eine zu F passende Struktur. Für jeden Term t, den man aus den Bestandteilen von F bilden kann, definieren wir nun den *Wert* von t ind der Struktur $\mathcal{A}$, den wir mit $\mathcal{A}(t)$ bezeichnen.

1. Falls t eine Variable ist, so ist $\mathcal{A}(t) = x^{\mathcal{A}}$.
2. Falls t die Form hat $t = f(t_1, \ldots, t_k)$, wobei $t_1 \ldots t_k$ Terme und f ein k-stelliges Funktionssymbol ist, so ist

$$\mathcal{A}(t) \quad = \quad f^{\mathcal{A}}\left(\mathcal{A}(t_1), \ldots, \mathcal{A}(t_k)\right).$$

Wahrheitswert einer Formel

Auf analoge Weise definieren wir den *(Wahrheits-)Wert* der Formel F, wobei wir ebenfalls die Bezeichnung $\mathcal{A}(F)$ verwenden.

1. Falls F die Form $F \;=\; P(t_1, \ldots, t_k)$ mit den Termen $t_1, \ldots t_k$, so ist

$$\mathcal{A}(F) = \begin{cases} 1, & \text{falls } (\mathcal{A}(t_1), \ldots, \mathcal{A}(t_k)) \in P^{\mathcal{A}} \\ 0, & \textit{sonst} \end{cases}$$

2. Falls F die Form $F \;=\; \neg G$ hat, so ist

$$\mathcal{A}(F) = \begin{cases} 1, & \text{falls } \mathcal{A}(G) \;=\; 0 \\ 0, & \textit{sonst} \end{cases}$$

3. Falls F die Form $F \;=\; (G \wedge H)$ hat, so ist

$$\mathcal{A}(F) = \begin{cases} 1, & \text{falls } \mathcal{A}(G) \;=\; 1 \text{ und } \mathcal{A}(H) \;=\; 1 \\ 0, & \textit{sonst} \end{cases}$$

4. Falls F die Form $F \;=\; (G \vee H)$ hat, so ist

$$\mathcal{A}(F) = \begin{cases} 1, & \text{falls } \mathcal{A}(G) \;=\; 1 \text{ oder } \mathcal{A}(H) \;=\; 1 \\ 0, & \textit{sonst} \end{cases}$$

5. Falls F die Form $F \;=\; \forall x G$ hat, so ist

$$\mathcal{A}(F) = \begin{cases} 1, & \text{falls für alle } d \in U_{\mathcal{A}} \text{ gilt: } \mathcal{A}_{[x/d]}(G) \;=\; 1 \\ 0, & \textit{sonst} \end{cases}$$

6. Falls F die Form $F = \exists x G$ hat, so ist

$$\mathcal{A}(F) = \begin{cases} 1, & \text{falls es ein } d \in U_{\mathcal{A}} \text{ gibt mit: } \mathcal{A}_{[x/d]}(G) = 1 \\ 0, & sonst \end{cases}$$

Hierbei bedeutet $\mathcal{A}_{[x/d]}$ diejenige Struktur $\mathcal{A}'$, die überall mit $\mathcal{A}$ identisch ist, bis auf die Definition von $x^{\mathcal{A}}$: Hier ist $x^{\mathcal{A}'} = d$, wobei $d \in U_{\mathcal{A}} = U_{\mathcal{A}'}$.

Bemerkungen

1. F ist gültig genau dann, wenn $\neg F$ unerfüllbar ist.
   Beweis: Es gilt:
   F ist gültig
   gdw. jede zu F passende Belegung ist ein Modell für F
   gdw. jede zu F und damit auch zu $\neg F$ passende Belegung ist kein Modell für $\neg F$
   gdw. $\neg F$ besitzt kein Modell.
   gdw. $\neg F$ ist unerfüllbar □

2. Nicht jede mathematische Aussage kann im Rahmen der Prädikatenlogik formuliert werden. Erst wenn auch Quantoren über Prädikaten- und Funktionssymbole erlaubt werden ist dies möglich. Dies ist dann die *Prädikatenlogik der zweiten Stufe.* Die obige Definition ist die *Prädikatenlogik der ersten Stufe* (auf die wir uns hier beschränken wollen).

Kapitel 2

Datentypen

Ein wichtiger Aspekt bei der Implementierung von Algorithmen ist die Wahl eines geeigneten Datentyps. Dieses Kapitel stellt eine Auswahl verschiedener Datentypen vor.

2.1 Abstrakte Datentypen

Bestimmte Datentypen lassen sich leicht auf eine algebraische Art und Weise beschreiben.

2.1.1 Sigma-Algebra

2.1.2 Bool

Bezeichnung des Datentyps:	`Bool`

Signaturen:	`True : Bool`
	`False : Bool`
	`not : Bool -> Bool`
	`and : Bool -> Bool -> Bool`
	`or : Bool -> Bool -> Bool`

Axiome:	`not False = True`
	`not x = False`
	
	`and False y = False`
	`and x y = y`
	
	`or True y = True`
	`or x y = False`

Konstruktoren sind hier `True` und `False`, die anderen Operatoren `and`, `or` und `not` sind Destruktoren.

2.1.3 Keller

Ein Keller oder Stapel (engl. stack) ist eine polymorphe Datenstruktur.

Bezeichnung des Datentyps:	`Stack a`

Signaturen:	`⊥ : Stack a`
	`Push : a -> Stack a -> Stack a`
	`pop : Stack a -> Stack a`
	`top : Stack a -> a`
	
	`empty : Stack a -> Bool`

Axiome:	`pop (Push x s) = s`
	
	`top (Push x s) = x`
	
	`empty ⊥ = True`
	`empty s = False`

Hier sind $\perp$ und `Push` Konstruktoren, `pop` und `top` sind Destruktoren und `empty` ist ein Verhalten.

2.1.4 Schlange

Auch die Schlange (engl. queue) ist ein polymorpher Datentyp. Als Vorbild dient hier die Warteschlange: Wer zuerst angekommen ist, wird auch zuerst bedient.

Bezeichnung des Datentyps:	`Queue a`

Signaturen:	`⊥ : Queue a`
	`EnQueue : a -> Queue a -> Queue a`
	`deQueue : Queue a -> Queue a`
	`first : Queue a -> a`
	
	`empty : Queue a -> Bool`

Axiome:	`dequeue (EnQueue x ⊥) = ⊥`
	`deQueue (Enqueue x q) = EnQueue x (deQueue q)`
	
	`first (EnQueue x ⊥) = x`
	`first (EnQueue x q) = first q`
	
	`empty ⊥ = True`
	`empty q = False`

2.1.5 Liste

Der hier vorgestellte Datentyp Liste (engl. list) ist polymorph und unterscheidet sich nur namentlich vom ADT Keller. Allerdings könnte man zusätzlich Operatoren zum Verbinden von Listen (`concat`) oder auch für den Zugriff auf das letzte Element (`last`) aufnehmen.

Bezeichnung des Datentyps:	`List a`

Signaturen:	`[] : List a`
	`Cons : a -> List a -> List a`
	`head : List a -> a`
	`tail : List a -> List a`
	
	`empty : List a -> Bool`

Axiome:	`tail (Cons x l) = l`
	
	`head (Cons x l) = x`
	
	`empty [] = True`
	`empty l = False`

Als Konstruktoren sind hier `Cons` und `[]` zu vermerken. `tail` und `head` sind Destruktoren und `empty` ist eine Verhalten der Liste.

Kapitel 3

Gierig-Algorithmen

3.1 Einführung

Greedy[1]-Algorithmen sind normalerweise sehr einfach. Typischerweise werden sie benutzt, um Optimierungsprobleme zu lösen. Zum Beispiel, um die beste Reihenfolge bei der Ausführung bestimmter Mengen von Jobs auf einem Rechner zu finden, ...

In einer typischen Situation haben wir :

- Eine Menge von Kandidaten (auszuführende Jobs, Knoten eines Graphen, ...).
- Eine Menge von Kandidaten, die schon benutzt worden sind.
- Eine Funktion, die feststellt, ob eine bestimmte Menge von Kandidaten eine Lösung zu diesem Problem ist (Optimalität bezüglich Zeit wird ignoriert).
- Eine Funktion, die feststellt, ob eine Menge von Kandidaten eine zulässige Menge ist, d.h., ob es möglich ist, die Menge so zu vervollständigen, daß mindestens eine Lösung gefunden wird.
- Eine Wahlfunktion (selection function), die in beliebiger Zeit den geeignetsten Kandidaten aus den unbenutzten Kandidaten bestimmt.
- Eine Zielfunktion, die den Wert einer Lösung angibt. Dies ist die Funktion, die optimiert werden soll.

[1] wörtlich übersetzt: “gierig”

Beispiel : Wechselgeldausgabe an einen Kunden, die aus möglichst wenigen Geldstücken besteht.

- Kandidaten: Eine Menge von Geldstücken (1,5,10,...), wobei jede Sorte aus mindestens einem Geldstück besteht.
- Lösung: Der Gesamtbetrag der gewählten Geldstücke entspricht dem eingezahlten Betrag.
- Zulässige Menge: Eine Menge, deren Gesamtbetrag den eingezahlten Betrag nicht überschreitet.
- Wahlfunktion: Wähle das am höchsten bewertete Geldstück, das noch in der Menge der Kandidaten übrig ist.
- Zielfunktion: Die Anzahl der in der Lösung benutzten Geldstücke.

Um das Optimierungsproblem zu lösen, betrachtet man die Menge der Kandidaten, aus denen eine Lösung besteht, die den Wert der Zielfunktion optimiert (minimiert oder maximiert). Zu bemerken ist, daß Wahl- und Zielfunktion identisch sein könnten.

Ein Greedy-Algorithmus arbeitet schrittweise.

1. Zu Beginn ist die Menge der Kandidaten leer.
2. Bei jedem Schritt versucht man, mit Hilfe der Wahlfunktion den besten, übriggebliebenen Kandidaten zu dieser Menge hinzuzufügen.
3. Falls die erweiterte Menge der gewählten Kandidaten nicht mehr zulässig ist, entfernen wir den gerade hinzugefügten Kandidaten. Ein solcher Kandidat wird später nicht mehr berücksichtigt.
4. Falls die gewählte Menge noch zulässig ist, gehört der gerade gewählte Kandidat dieser Menge für immer an.
5. Nachdem wir die Menge erweitert haben, überprüfen wir, ob die Menge eine Lösung des gegebenen Problems ist.

Solch ein Algorithmus wird "greedy" genannt, da in jedem Schritt nur der möglichst einfach zu bestimmende Kandidat gewählt wird, ohne die nächsten Schritte zu berück-sichtigen. Wird einmal entschieden, einen Kandidaten einzuschliessen oder auszuschliessen, kann diese Entscheidung nicht mehr revidiert werden.

Ein Greedy-Algorithmus läßt sich abstrakt folgendermaßen definieren:

```
Algorithmus Greedy(C)    {C: Menge aller Kandidaten}
S ← ∅    {S: Lösungsmenge}
while not Lösung(S) and C≠∅ do
        x ← ein Element aus C, das Wahl(x) maximiert
        C ← C\{x}
        if zulässig(S∪{x}) then S ← S∪{x}
if Lösung(S)
        then return S
        else return "keine Lösung"
```

Wir beschreiben nun diese Technik anhand einiger Beispiele.

3.2 Minimale, zusammenhängende Bäume

3.2.1 Einführung

Sei G=<N,A> ein zusammenhängender, ungerichteter Graph mit N: Menge von Knoten, A: Menge von Kanten, wobei jeder Kante eine nichtnegative Länge zugeordnet wird.

Problem : Finde eine Teilmenge T von A, so daß alle Knoten zusammenhängend bleiben, wenn man nur die Kanten aus T benutzt. Dabei soll die Summe der Kantenlänge aus T so klein wie möglich gehalten werden.

Bemerkung : Die Länge einer Kante kann auch als "Kosten" interpretiert werden. Das Problem besteht nun darin, den Pfad mit den "minimalen Kosten" zu finden. Man kann diese Technik anwenden, um z.B. das billigste Straßennetz zur Verbindung mehrerer Städte zu konstruieren.

Die theoretische Grundlage für minimale, zusammenhängende Bäume läßt sich folgendermaßen zusammenfassen:

Lemma 1 : Der partielle Graph <N,T> ist ein Baum, genannt der minimale, zusammenhängende Baum.

Die Terminologie für dieses Problem:

1. Eine Menge von Kanten ist eine Lösung, wenn sie einen zusammenhängenden Baum bildet.

2. Sie ist zulässig, wenn sie keinen Zyklus enthält.

3. Eine zulässige Menge von Kanten heißt günstig, wenn sie zu einer optimalen Lösung vervollständigt werden kann (Insbesondere ist eine leere Menge günstig, da G zusammenhängend ist).

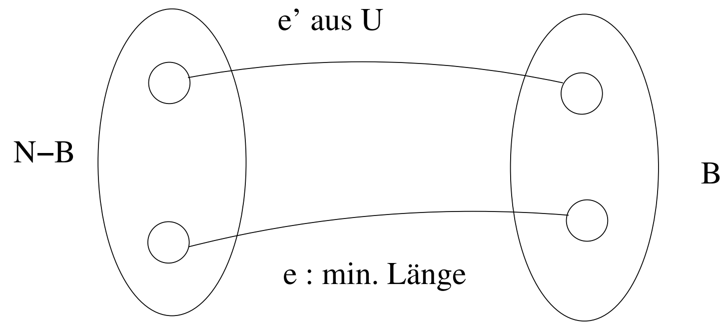
4. Eine Kante berührt eine gegebene Menge von Kanten, wenn genau ein Ende der Kante ein Element aus dieser Menge ist.

Die Richtigkeit der folgenden Algorithmen läßt sich durch das nachstehende Lemma beweisen:

Lemma 2 : Seien $G=<N,A>$ ein zusammenhängender, ungerichteter Graph, wobei die Länge jeder Kante gegeben sei. Sei $B \subset N$ eine echte Teilmenge der Knoten aus G. Sei $T \subseteq A$ eine günstige Menge von Kanten, so daß keine Kante aus T die Knotenmenge B berührt. Sei e die kürzeste Kante, die B berührt. Dann ist $T \cup \{e\}$ günstig.

Beweis : Sei U ein minimaler, zusammenhängender Baum von G, so daß $T \subseteq U$ (dieser minimale, zusammenhängende Baum existiert, da T nach Annahme günstig ist). Falls $e \in U$, gibt es nichts zu beweisen. Sonst fügen wir die Kante e zu U hinzu. Dabei entsteht ein Zyklus (dies ist die Eigenschaft eines Baumes). In diesem Zyklus muß mindestens eine Kante e' existieren, die e berührt, da e B berührt (sonst ist der Zyklus nicht abgeschlossen, wie er von der folgenden Abbildung beschrieben wird).

Entfernen wir e', so verschwindet der Zyklus, und wir erhalten einen neuen Baum U', der G ausspannt. Da gemäß Definition die Länge von e nicht größer als die Länge von e' ist, so überschreitet die Gesamtlänge der Kanten in U' nicht die Gesamtlänge in U. Daher ist auch U' ein minimaler, zusammenhängender Baum, der e einschließt. Um den Beweis zu vervollständigen, merken wir, daß $T \subseteq U'$, da die Kante e', die B berührt, nicht aus T sein kann.

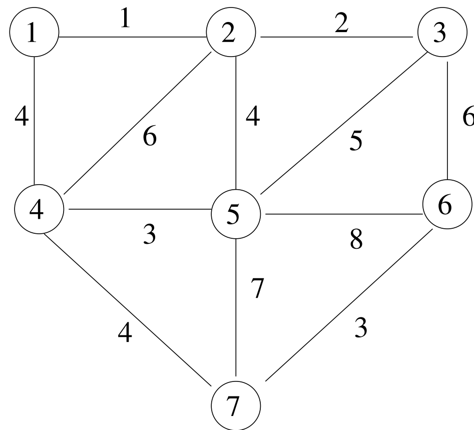


3.2.2 Kruskalscher Algorithmus

Zu Beginn ist T leer. Während des Ablaufs des Algorithmus werden Kanten zu T hinzugefügt. In jedem Schritt besteht der von den Knoten aus G und den Kanten aus T gebildete Graph aus zusammengebundenen Elementen. Die Elemente von T, die in einer gegebenen, zusammengebundenen Komponente eingeschlossen sind, bilden einen minimalen, zusammenhängenden Baum für die Knoten dieser Komponente. Am Ende des Algorithmus verbleibt nur noch eine zusammengebundene Komponente, so daß T nun ein minimaler, zusammenhängender Baum für alle Knoten aus G ist. Um immer größer werdende, zusammengebundene Komponenten zu bilden, prüfen wir die Kantenlänge von G in aufsteigender Reihenfolge. Wenn eine Kante zwei Knoten in verschiedenen Komponenten verbindet, dann fügen wir sie zu T hinzu, um daraus nur eine Komponente zu erhalten. Sonst wird die Kante nicht angenommen, da sie zwei Knoten innerhalb einer zusammengebundenen Komponente verbindet und einen Zyklus erzeugt, wenn sie hinzugefügt wird. Der Algorithmus terminiert, wenn nur eine zusammengebundene Komponente verbleibt.

Beispiel : Ein Graph mit 7 Knoten. Die aufsteigende Reihenfolge der Kantenlänge ist: {1,2}, {2,3}, {4,5}, {6,7}, {1,4}, {2,5}, {4,7}, {3,5}, {2,4}, {3,6}, {5,7}, {5,6}. Dies sei durch das folgende Bild verdeutlicht:

Schritt	Berücksichtigte Kanten	Zusammengebundene Komponenten
Initialisierung	—	{1} {2} {3} {4} {5} {6} {7}
1	{1,2}	{1,2} {3} {4} {5} {6} {7}
2	{2,3}	{1,2,3} {4} {5} {6} {7}
3	{4,5}	{1,2,3} {4,5} {6} {7}
4	{6,7}	{1,2,3} {4,5} {6,7}



5	{1,4}	{1,2,3,4,5} {6,7}
6	{2,5}	nicht angenommen
7	{4,7}	{1,2,3,4,5,6,7}

T enthält die Kanten {1,2}, {2,3}, {4,5}, {6,7}, {1,4} und {4,7}. Diese Kanten werden im obigen Bild als dicke Linien dargestellt.

Die Richtigkeit läßt sich durch die Anwendung von Lemma 2 und Induktion über die Anzahl der bisher gewählten Kanten beweisen.

Der Algorithmus

Man braucht die folgenden Teilalgorithmen:

- find(x), der feststellt, in welcher Komponente der Knoten x zu finden ist.
- merge(A,B): Mischen von zwei disjunkten Mengen.

Algorithmus Kruskal(G=<N,A>)

||Input: N: Menge von Knoten, A: Menge von Kanten mit Längenangabe
Output: Eine Menge von Kanten||

1. Initialisierung

1.1. Sortiere A nach aufsteigender Länge
1.2. n ← #N ||Anzahl der Knoten||
1.3. T ← $\emptyset$ ||Lösungsmenge||
1.4. Initialisiere n disjunkte Mengen, wobei jede Menge ein Element aus N enthält.

2. Greedy-Schleife

repeat
2.1. {u,v} $\leftarrow$ noch nicht berücksichtigte, kürzeste Kante
2.2. ucomp $\leftarrow$ find(u) ||in den Mengen der bereits verbundenen Kanten||
vcomp $\leftarrow$ find(v)
2.3. **if** ucomp $\neq$ vcomp
then merge(ucomp,vcomp)
T $\leftarrow$ T$\cup${{u,v}}

until #T=n-1

3. return T

Komplexitätsanalyse

Die Anzahl der Operationen für einen Graphen mit n Knoten und a Kanten ist (wenn man die Ergebnisse für Sortieren und Mischen benutzt):

- O(a·log(a)), um die Kanten zu sortieren. Da $n-1 \leq a \leq \frac{n(n-1)}{2}$, ist dies äquivalent zu O(a·log(n).
- O(n), um die n disjunkten Mengen zu initialisieren.
- Für "find-" und "merge-" Operationen gibt es höchstens $2 \cdot a$ find-Operationen und $(n-1)$ merge-Operationen. Daher ist der worst-case $O((2a+n-1)\log^* n)$. $\log^*$ wird induktiv folgendermaßen definiert.

  Definition:

  1. $\log^{(0)}(n) = n$
  2. $\log^{(k)}(n) = \log(\log^{k-1}(n))$, $k \geq 1$.

  $\rightarrow \log^* n = \min\{i | \log^{(i)}(n) \leq 1\}$, somit erhält man

n	:	$log^*(n)$
1	:	0
2	:	1
3, 4	:	2
$5 \rightarrow 16$	:	3
$17 \rightarrow 65536$	:	4

  $\log^*$ wächst sehr langsam. Es ist leicht zu erkennen, daß $n \in o(n\log^* n)$, $\frac{n}{log^* n} \in o(n)$ gilt.

- Höchstens O(a) für die restlichen Operationen.

Für einen zusammenhängenden Graphen ist a≥(n-1), da O(log*(n))≤O(log(n)). Also ist die Gesamtkomplexität für den Algorithmus O(a·log(n)).

Bemerkung : Werden die Kanten in einem "Heap" organisiert (eine bestimmte Baumart), so wird die Initialisierung in O(a) ausgeführt. Aber die Suche nach einem minimalen Element in der Schleife benötigt $O(\log(a)) = O(\log(n))$. Dies bringt einen Vorteil, wenn der minimale, zusammenhängende Baum schon gefunden ist, und noch viele ungewählte Kanten übrig geblieben sind. Die Zeitverschwendung, unnötige Kanten zu untersuchen, wird dabei verhindert. Dies verdeutlicht die Bedeutung der Datenstruktur.

3.2.3 Primscher Algorithmus

Im Kruskalschen Algorithmus werden Kanten unter der Bedingung gewählt, daß sie keinen Zyklus bilden. Als Ergebnisse bekommen wir einen Wald von Bäumen, der irgendwann außer Kontrolle wächst. Ein zusammenhängender Baum wächst in natürliche Weise aus einer bestimmten Wurzel. Der Algorithmus läßt sich informell folgendermaßen beschreiben:

Algorithmus Prim(G=<N,A>)

1. Initialisierung
    1.1. T ← ∅ ||Lösungsmenge von Kanten||
    1.2. B ← {ein willkürliches Element aus N}

2. Greedy-Schleife

    while B≠N **do**
        2.1. Finde {u,v} von minimaler Länge, so daß u∈N\B und v∈B
        2.2. T ← T∪{{u,v}}
        2.3. B ← B∪{u}

3. return T

Dieser Algorithmus wird anhand des gleichen Beispiels wie für den Kruskalschen Algorithmus verdeutlicht :

Schritt	{u,v}	B
Initialisierung	—	{1}
1	{2,1}	{1,2}
2	{3,2}	{1,2,3}
3	{4,1}	{1,2,3,4}
4	{5,4}	{1,2,3,4,5}
5	{7,4}	{1,2,3,4,5,7}
6	{6,7}	{1,2,3,4,5,6,7}

T enthält die gewählten Kanten: {2,1}, {3,2}, {4,1}, {5,4}, {7,4} und {6,7}

Implementierung

Es ist möglich, eine implementierbare Version anhand der informellen Beschreibung zu entwerfen. Seien N={1,2,...,n} eine Menge und L eine symmetrische Matrix, die die Länge jeder Kante darstellt (mit L[i,j]=∞, wenn die Kante$\{v_i, v_j\}$ im Graphen nicht existiert). Man verwendet zwei Arrays. Für jeden Knoten i∈N\B liefert nearest[i] den Knoten aus B, der zu i am nächsten liegt. Wir setzen ferner mindist[i]=-1. Die Menge B wird mit {1} initialisiert. Daher werden nearest[1] und mindist[1] nicht benutzt.

```
function Prim(L[1..n,1..n])
    ||Initialisierung: nur Knoten 1 ist in B||
    T ← ∅
    for i ← 1 to n do
        nearest[i] ← 1
        mindist[i] ← L[i,1]
    ||Greedy-Schleife||
    repeat n-1 times
        min ← ∞
        for j ← 2 to n do
            if 0 ≤ mindist[j] < mind
                then min ← mindist[j]
                     k ← j
        T ← T∪{{k,nearest[k]}}
        mindist[k] ← -1 ||füge k zu B hinzu||
        for j ← 2 to n do
            if L[k,j]<mindist[j]
                then mindist[j] ← L[k,j]
                     nearest[j] ← k
```

return T

Komplexitätsanalyse und Vergleich

Die Hauptschleife von Prim wird (n-1) mal ausgeführt. Bei jeder Iteration benötigen die for-Schleifen eine Zeit von O(n). Daher ist der Primsche Algorithmus von O(n^2).

Kruskalscher Algorithmus: O(a·log(n)). Für einen sehr dicht besetzten Graphen nähert sich a $\frac{n\cdot(n-1)}{2}$. In diesem Fall erhält man O(n^2log(n)) und der Primsche Algorithmus ist wahrscheinlich besser. Für einen sehr dünn besetzten Graphen nähert sich a n, daher bekommt man O(n·log(n)) für den Kruskalschen Algorithmus und der Primsche Algorithmus ist wahrscheinlich weniger effizient.

Kommentar : Dies verdeutlicht, daß wenn für ein gegebenes Problem zwei verschiedene Algorithmen existieren, nicht immer behauptet werden kann, daß einer stets effizienter ist.

3.3 Kürzeste Pfade

Die Datenstruktur ist die gleiche wie im Teilkapitel 2: G=<N,A>, mit der zusätzlichen Annahme, daß G ein gerichteter Graph sei. Ein Knoten wird als Ursprungsknoten bezeichnet.

Problem : Bestimme die Länge (Kosten) des kürzesten Pfades vom Ursprungsknoten zu jedem anderen Knoten im Graphen G.

Eine mögliche Lösung bietet der Greedy-Algorithmus, der hier oft Dijkstrascher Algorithmus genannt wird. C: Menge der vorhandenen Kandidaten (Knoten); S: Menge der bereits gewählten Kandidaten. In jedem Schritt enthält S diejenigen Knoten, deren minimale Entfernung vom Ursprungsknoten bekannt ist, und C enthält alle anderen Knoten. Bei jedem Schritt wählt man den Knoten aus C, dessen Entfernung zu dem Ursprungsknoten am kürzesten ist, und wir fügen ihn zu S hinzu.

- Wir sagen, daß ein Pfad vom Ursprungsknoten zu irgendeinem anderen Knoten speziell ist, wenn alle dazwischenliegenden Knoten Elemente aus S sind.

- In jedem Schritt des Algorithmus enthält ein Array (D) die Länge des kürzesten speziellen Pfades zu jedem Knoten des Graphen.
- Man beweist, daß in dem Moment, in dem wir einen neuen Knoten v zu S hinzufügen, der kürzeste, spezielle Pfad zu v auch der insgesamt kürzeste Pfad zu v ist.
- Terminiert der Algorithmus, so enthält S alle Knoten des Graphen, und daher sind alle Pfade vom Ursprungknoten zu irgendeinem anderen Knoten speziell. Daher ergibt der Wert in D die Lösung zu unserem Problem.

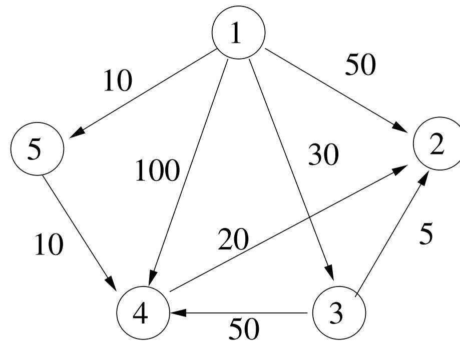
Einfachheitshalber numerieren wir die Knoten von 1 bis n. N={1,2,...,n}. Wir nehmen an, daß der Knoten 1 der Ursprung ist und die Länge jeder gerichteten Kante in der Matrix L zu finden ist. $L[i,j] \geq 0$, wenn die Kante (i,j) existiert; $L[i,j]=\infty$ sonst.

Die Menge $S=N \setminus C$ ist nur implizit bekannt.

```
Algorithmus Dijkstra(L[1..n,1..n]):array[2..n]
    1. Initialisierung
        C ← [2,3,...,n]
        for i ← 2 to n do D[i] ← L[1,i]
    2. Greedy-Schleife
        repeat n-2 times
            v ← ein Element aus C, das D[v] minimiert
            C ← C\{v}
        for each w∈C do
            D[w] ← min(D[w], D[v]+L[v,w])
    3 return D
```

Beispiel

Schritt	v	C	D
Initialisierung	–	{2,3,4,5}	[50,30,100,10]
1	5	{2,3,4}	[50,30,20,10]
2	4	{2,3}	[40,30,20,10]
3	3	{2}	[35,30,20,10]



Es ist leicht zu erkennen, daß D sich nicht verändert, wenn noch eine Iteration durchgeführt wird, um das letzte Element von C zu entfernen. Das ist der Grund, warum die Hauptschleife nur (n-2)-mal wiederholt wird.

Bemerkung : Um zu wissen, wohin die kürzesten Pfade führen, betrachtet man P[2..n], wobei P[v] dem Knoten entspricht, der sich vor v im kürzesten Pfad befindet. Im Algorithmus: man nimmt die folgenden Änderungen vor:

- Initialisiere P[i]=1 für i=2,3,...,n
- In der For-Schleife in Schritt 2: Ersetze den Rumpf durch:
  if D[w] > D[v]+L[v,w] **then**
  D[w] ← D[v]+L[v,w]; P[w] ← v

Beweis der Korrektheit : Durch Induktion.

1. Wenn ein Knoten Element aus S ist, dann ergibt D[i] die Länge des kürzesten Pfades vom Ursprung zu i.
2. Wenn ein Knoten i kein Element aus S ist, dann ergibt D[i] die Länge des kürzesten, speziellen Pfades vom Ursprung zu i.

Komplexitätsanalyse : Wir nehmen an, daß wir diesen Algorithmus auf einen Graphen mit n Knoten und a Kanten anwenden. Die Kosten werden in einer Matrix L[1..n,1..n] angegeben.

- Die Initialisierung benötigt O(n).
- "Repeat-Schleife": Alle Elemente aus C müssen untersucht werden, um v zu wählen. Daher untersuchen wir (n-1), (n-2), ..., 2 Werte von D bei sukzessiven Iterationen. Die Gesamtzeit ist daher $O(n^2)$.

- Die innere "for"-Schleife führt (n-1), (n-2), ..., 1 Iterationen aus. Daraus resultiert eine Gesamtzeit von $O(n^2)$.

Infolgedessen benötigt dieser Algorithmus eine Zeit von $O(n^2)$ für die oben gegebene Implementation. Eine verbesserte Analyse zeigt, daß diese Implementation nur für dicht besetzte Graphen geeignet ist. Wenn der Graph dünn besetzt und zusammenhängend ist, kann die Zeitkomplexität durch die Verwendung eines Heaps auf $O(a \cdot \log(n))$ reduziert werden.

3.4 Zeitplanerstellung (Scheduling)

3.4.1 Zeitplanerstellung ohne Schlußtermine

Ein Server (ein Prozessor, ein Kassierer in einer Bank, ...) habe n Kunden in einem gegebenen System zu bedienen. Die für jeden Kunden benötigte Bedienzeit sei im voraus bekannt: Diese sei t_i für Kunde i ($1 \leq i \leq n$). Wir möchten

$$T = \sum_{i=1}^{n} (\text{Gesamtzeit für Kunde } i)$$

minimieren. Da die Anzahl der Kunden schon festgelegt ist, ist die Minimierung der Gesamtzeit äquivalent zur Minimierung der Durchschnittszeit.

Beispiel : $n = 3$, $t_1 = 5$, $t_2 = 10$, $t_3 = 3$. Dann sind 6 Reihenfolgen möglich. Die erste Möglichkeit: Kunde 1 wird zunächst bedient; Kunde 2 wartet, während Kunde 1 noch bedient wird, und wird anschliessend bedient; Kunde 3 wartet, während Kunde 1 und Kunde 2 bedient werden, und wird anschliessend bedient.

Reihenfolge	T
1 2 3	5 + (5+10) + (5+10+3) = 38
1 3 2	5 + (5+3) + (5+3+10) = 31
2 1 3	10 + (10+5) + (10+5+3) = 43
2 3 1	10 + (10+3) + (10+3+5) = 41
3 1 2	3 + (3+5) + (3+5+10) = 29 ← optimal
3 2 1	3 + (3+10) + (3+10+5) = 34

Wir suchen einen Algorithmus, der den optimalen Zeitplan (schedule) schrittweise berechnet. Angenommen, wir bedienen Kunde j nach den Kunden

$i_1, ..., i_m$. Der Anstieg von T in dieser Phase ist gegeben durch

$$t_{i_1} + ... + t_{i_m} + t_j \ .$$

Zur Minimierung dieses Anstiegs brauchen wir nur t_j zu minimieren. Dies ergibt unverkennbar einen Greedy-Algorithmus: Füge ans Ende des Zeitplanes $t_{i_1} + ... + t_{i_m}$ den Kunden ein, der am wenigsten Zeit benötigt. Dieser triviale Algorithmus liefert die korrekte Antwort für das Beispiel: 3, 1, 2.

Theorem : Dieser Algorithmus ist immer optimal.

Sei I=$(i_1, i_2, ..., i_n)$ eine Permutation der ganzen Zahlen {1,2,...,n}. Wenn die Kunden in der Reihenfolge I bedient werden, ist die Gesamtzeit für alle Kunden in dem System :

$$\begin{aligned} T(I) &= t_{i_1} + (t_{i_1} + t_{i_2}) + ... \\ &= n \cdot t_{i_1} + (n-1)t_{i_2} + ... \\ &= \textstyle\sum_{k=1}^{n} (n-k+1)t_{i_k} \end{aligned}$$

Wir nehmen nun an, daß es für I zwei ganze Zahlen a, b gibt mit a<b und $t_{i_a} > t_{i_b}$. Vertauschen wir die Positionen von a und b, so erhalten wir eine neue Bedienreihenfolge I', die günstiger als I ist, da:

$$T(I') = (n-a+1)t_{i_b} + (n-b+1)t_{i_a} + \sum_{k=1,\ k \neq a,b}^{n} (n-k+1)t_{i_k}$$

$$\begin{aligned} T(I)-T(I') &= (n-a+1)(t_{i_b} - t_{i_a}) + (n-b+1)(t_{i_a} - t_{i_b}) \\ &= (b-a)(t_{i_a} - t_{i_b}) > 0 \end{aligned}$$

Auf diese Weise können wir einen Zeitplan verbessern, indem der Kunde, füer den die geringste Bedienzeit benötigt wird, vor allen anderen Kunden bedient wird. Solche Zeitplanungen erfordern, daß wir die Kunden in der Reihenfolge aufsteigender Bedienzeiten betrachten.

3.4.2 Zeitplanerstellung mit Schlußterminen

(Englisch: Scheduling with deadline)

Problem : n Aufträge sind zu verarbeiten, wobei zur Verarbeitung eines Auftrags eine Zeiteinheit benötigt wird. Zu jedem Zeitpunkt t=1,2,...

kann nur ein Auftrag verarbeitet werden. Der Gewinn g_i eines Auftrags i wird genau dann erzielt, wenn der Auftrag bis zum Schlußtermin d_i erledigt wird.

Beispiel :

n=4;	i	1	2	3	4
	g_i	50	10	15	30
	d_i	2	1	2	1

Die Reihenfolge (3,2) wird zum Beispiel nicht berücksichtigt, da dann Auftrag 2 zum Zeitpunkt t=2 nach seinem Schlußtermin t=1 verarbeitet wird.

Reihenfolge	Gewinn
1	50
2	10
3	15
4	30
(1,3)	65
(2,1)	60
(2,3)	25
(3,1)	65
(4,1)	80
(4,3)	45

Die Auftragsfolge (4,1) ist optimal und kann in der Reihenfolge (1,4) nicht verarbeitet werden. Es ist zu beweisen, daß es nicht notwendig ist, alle n! mögliche Auftragsfolgen für n Aufträge zu untersuchen. Es genügt eine Auftragsfolge in der Reihenfolge aufsteigender Schlußtermine zu untersuchen [(4,1) aber nicht (1,4) zum Beispiel].

Eine einfache Implementierung ist durch den folgenden Algorithmus gegeben (mit den Annahmen $g_1 \geq g_2 \geq ... \geq g_n$, $n > 0$ und $d_i > 0$, $0 \leq i \leq n0$).

Algorithmus Sequence(d[0..n])
    array j[0..n]
1. Initialisierung
    d[0], j[0] $\leftarrow$ 0

```
        k, j[1] ← 1 {Auftrag 1 wird immer gewählt}
2. Greedy-Schleife
        for i ← 2 to n do
                r ← k
                while d[j[r]] > max(d[i],r) do r ← r-1
                if d[j[r]]≤d[i] and d[i]>r then
                        for l ← k step -1 to r+1 do j[l+1] ← j[l]
                        j[r+1] ← i
                        k ← k+1
        return k, j[1..k]
```

Übung: Wende diesen Algorithmus auf das obige Beispiel an.

Es ist möglich, einen schnellen Algorithmus mit Hilfe von disjunkten Mengenstrukturen zu entwerfen. Die grundlegenden Schritte sind dabei:

1. Initialisierung: Wir beginnen mit l Mengen {1}, ..., {l}. Die Funktion F liefert von jeder Menge das kleinste Element.
2. Addition eines Auftrags mit Schlußtermin d
   - Finde die Menge K, die min(n,d) enthält.
   - Wenn F(K)=0, lehne den Auftrag ab.
   - Wenn F(K)≠0 :
     - Weise dem Auftrag die Position F(K) zu.
     - Finde die Menge L, die F(K)-1 enthält (L und K sind unterschiedliche Mengen).
     - Mische K und L; der Wert von F für diese neue Menge ist der alte Wert von F(L)

Übung : Prüfe, wie der schnelle Algorithmus für folgendes Beispiel funktioniert:

i	1	2	3	4	5	6
g_i	20	15	10	7	5	3
d_i	3	1	1	3	1	3

Die Analyse der Komplexität des schnellen Algorithmus zeigt, daß sie von der Ordnung O(n·log(n)) ist. Dies ist der Sortieraufwand zur Bestimmung der Anfangsreihenfolge.

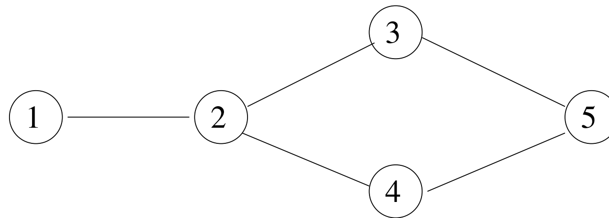
3.5 Greedy-Heuristik

Da Greedy-Methoden sehr einfach sind, werden sie oft in bestimmten Situationen als Heuristik angewendet, wobei eine approximierte Lösung anstatt einer optimalen Lösung gesucht wird. Prototypische Beispiele sind NP-Probleme. Wir behandeln kurz zwei solche Probleme.

3.5.1 Einfärben eines Graphen

G=<N,A>: ungerichteter Graph, dessen Knoten zu färben sind. Wenn zwei Knoten durch eine Kante verbunden sind, müssen sie mit verschiedenen Farben gefärbt werden. Ziel: Möglichst wenige, verschiedene Farben zu benutzen.

Beispiel :



Zwei Farben werden benötigt: Eine für 1,3,4 und eine andere für 2,5

Greedy-Algorithmus

- Wähle eine Farbe und einen willkürlichen Anfangsknoten.
- Berücksichtige jeden anderen Knoten der Reihe nach, und färbe ihn mit dieser Farbe, falls möglich.
- Wenn kein weiterer Knoten gefärbt werden kann, wähle eine neue Farbe und einen neuen Anfangsknoten, der noch nicht gefärbt worden ist.
- Färbe soviele Knoten wie möglich mit der zweiten Farbe und wähle danach eine dritte Farbe und einen neuen Anfangsknoten ...
- Und so weiter.

Dieser Algorithmus ist nicht immer optimal. Für das obige Beispiel führt die Reihenfolge 1,2,3,4,5 zu einem optimalen Ergebnis; aber nicht die Reihenfolge 1,5,2,3,4. Auf diese Weise findet der Algorithmus offensichtlich keine gute Lösung.

3.5.2 Das Handlungsreisende-Problem

Ein Greedy-Algorithmus ist :

Wähle in jedem Schritt die kürzeste Kante aus den restlichen Kanten, die die folgenden Bedingungen erfüllt :

1. Zusammen mit den bereits gewählten Kanten bildet sie keinen Zyklus (die letzte Kante, die die Tour vervollständigt, ist eine Ausnahme).
2. Wenn eine Kante gewählt wird, darf sie nicht die dritte gewählte Kante sein, die von irgendeinem Knoten anfängt.

Beispiel :

Von-Nach	2	3	4	5	6
1	3	10	11	7	25
2		6	12	8	26
3			9	4	20
4				5	15
5					18

Die Kanten werden in der Reihenfolge {1,2}, {3,5}, {4,5}, {2,3}, {4,6}, {1,6} gewählt. Die Gesamtlänge des Zyklus ist 58 und die Tour ist (1,2,3,5,4,6,1). {1,5} wird nicht gewählt, da sie sonst die dritte Kante vom Knoten 5 wäre, und außerdem bildet (1,2,3,5,1) einen Zyklus. In diesem Beispiel ist der Greedy-Algorithmus nicht optimal, da die Tour (1,2,3,6,4,5,1) eine Gesamtlänge von nur 56 hat.

Kapitel 4

Teile-und-Herrsche (Divide and Conquer)

4.1 Einführung

Die klassische Teile-und-Herrsche-Methode besteht darin, die zu lösende Instanz in eine bestimmte Anzahl von kleineren Teilinstanzen zu zerlegen. Jede dieser Teil-instanzen wird sukzessiv und unabhängig gelöst. Die Teilergebnisse werden kombiniert, um die Lösung der Originalinstanz zu erhalten. Die Effizienz der Teile-und-Herrsche-Methode liegt darin, daß die Teilinstanzen wahrscheinlich effizient gelöst werden können.

Um einige Aspekte dieser Methode zu zeigen, betrachten wir ein fiktives Beispiel:

1. Sei A ein gegebener Algorithmus, dessen Komplexität quadratisch ist, und c eine Konstante, so daß eine bestimmte Implementation von A eine Zeit von $t_A(n) \leq cn^2$ benötigt, um eine Instanz der Größe n zu lösen.

2. Angenommen, es gibt eine Möglichkeit, die Anfangsinstanz in drei Teilinstanzen der Größe $\lceil \frac{n}{2} \rceil$ zu teilen und deren Ergebnisse zu kombinieren. Sei d eine Konstante, so daß die zur Zerlegung einer Instanz in Teilinstanzen und zum Kombinieren der Ergebnisse benötigte Zeit $t(n) \leq d \cdot n$ ist. Benutzt man den ursprünglichen Algorithmus und diese Zerlegung, so bekommt man

   $t_B(n) = 3t_A(\lceil \frac{n}{2} \rceil) + t(n)$

$$\leq 3c(\frac{(n+1)}{2})^2 + dn = \frac{3}{4}cn^2 + (\frac{3}{2}c + d)n + \frac{3}{4}c$$

Der Term $\frac{3}{4}cn^2$ dominiert, wenn n groß genug ist. Daher ist Algorithmus B 25% schneller als Algorithmus A. Aber B benötigt noch eine quadratische Zeit: Die Verbesserung ist begrenzt.

3. Um diese Methode zu verbessern, betrachten wir die Lösung der Teilinstanzen.

   - Wenn sie klein sind, ist Algorithmus A wahrscheinlich der beste, wie in (2) gezeigt wurde.
   - Wenn sie groß genug sind, ist es wahrscheinlich besser, unseren neuen Algorithmus rekursiv zu benutzen. Auf diese Weise erhalten wir Algorithmus C, der eine Zeit

   $$t_c(n) = \begin{cases} t_A(n) & : \; falls\ n \leq n_0 \\ 3t_c(\lceil \frac{n}{2} \rceil) + t_n & : \; sonst \end{cases}$$

   benötigt (n_0 ist der sogenannte Grenzwert).

   Es bleibt nun $T(n) = 3T(\frac{n}{2}) + dn$ mit der Annahme $n = 2^k > 1$ zu lösen. Dies ergibt: $T(2^k) = 3T(2^{k-1}) + d2^k$ oder $t_k = 3t_{k-1} + d2^k$

   Benutzt man die Methode der charakteristischen Gleichung, so erhält man: $(x-3)(x-2) = 0$ und damit:

   $$\begin{aligned} t_k &= c_1 3^k + c_2 2^k \\ T(n) &= c_1 3^{log_2 n} + c_2 n \\ &= c_1 n^{log_2 3} + c_2 n \end{aligned}$$

   Da $n^{log_2 3}$ ungefähr $n^{1.59}$ ist, so ist die Verbesserung wesentlich. Die zwei Hauptprobleme bestehen darin, den Grenzwert n_0 zu bestimmen und die "verborgene" Konstante c_1 so klein wie möglich zu halten.

Die allgemeinen Schritte der Teile-und-Herrsche-Methode lassen sich folgendermaßen beschreiben:

Algorithmus DQ(x)

1. **if** (x ist klein genug oder einfach) **then** **return** ADHOC(x)

2. Teile x in kleinere Teilinstanzen $x_1, x_2, ..., x_k$

3. **for** i ← 1 **to** k **do** $y_1 \leftarrow \mathrm{DQ}(x_i)$
4. Kombiniere die y_i's, um eine Lösung y für x zu erhalten.
5. **return** y

ADHOC ist der Grundalgorithmus zur Lösung von Teilinstanzen des Problems.

Spezialfälle

1. Wenn k=1 ist, wird diese Technik Vereinfachung statt Teile-und-Herrsche genannt.
2. Manchmal funktioniert ein DQ-Algorithmus nicht genau, wie oben beschrieben wird, aber es wird gefordert, daß die erste Teilinstanz gelöst wird, bevor die zweite Subinstanz formuliert wird (siehe Teil 3).

Damit diese Methode nützlich wird, müssen einige Bedingungen berücksichtigt werden:

- Es ist möglich, eine Instanz in Teilinstanzen zu teilen.
- Es ist möglich, die Teilergebnisse effizient zu kombinieren.
- Die Größe der Teilinstanzen soll möglichst gleich sein.
- Es muß gut überlegt werden, wann man den Grundalgorithmus benutzen soll, anstatt weiter rekursiv zu arbeiten.
- Es muß gut überlegt werden, wie man den Grenzwert wählt.

Dieses Kapitel ist folgendermaßen organisiert. Wir machen zunächst einen Überblick, wie man den Grenzwert bestimmt. Im Anschluß daran geben wir einige Beispielanwendungen dieser Methode. Wir behandeln keine Grundalgorithmen, da wir annehmen, daß sie schon bekannt sind, zum Beispiel:

- Binäres Suchen
- Sortieren durch Mischen (mergesort)
- Quicksort

Sie werden zum Beispiel in Horowitz - Sahni (Algorithmen, Originalbuch: Fundamentals of Computer Algorithms) beschrieben.

4.2 Bestimmung des Grenzwertes

Ein DQ-Algorithmus muß mit dem Teilen aufhören, wenn die Größe der Teilinstanzen eine bestimmte Größe unterschreitet. In diesem Fall ist es besser, den Grundalgorithmus anzuwenden. Um dies zu beschreiben, betrachten wir wieder das vorherige Beispiel für den Algorithmus C.

$$t_c(n) = \begin{cases} t_A(n) & : \ falls\ n \leq n_0 \\ 3t_c(\lceil \frac{n}{2} \rceil) + t_n & : \ sonst \end{cases}$$

Um den Grenzwert, der $t_c(n)$ minimiert, zu bestimmen, ist es genug zu wissen, daß $t_A \in \Theta(n^2)$ und $t(n) \in \Theta(n)$. Zum Beispiel: wir betrachten eine Implementation, so daß $t_A(n) = n^2$ Millisekunden, $t(n) = 16n$ Millisekunden, und wir haben eine Instanz der Größe 1024, die gelöst werden muß.

- Wenn der Algorithmus rekursiv bis $n = 1$ arbeitet (Instanz der Größe 1), d.h. $n_0 = 1$, würde es 30 Minuten dauern, bis die Lösung gefunden wird.
- Wenn man den Subalgorithmus direkt anwendet, dauert es 15 Minuten (d.h wir setzen $n_0 = \infty$)

Dies würde bedeuten, daß man eine Zeit von O($n^{log(3)}$) erhält, wenn die verborgene Konstante sehr groß wird. Infolgedessen ist der resultierende Algorithmus nie effizient.

Glücklicherweise ist dies nicht richtig: mit einem gut gewählten n_0 in unserem Beispiel, kann eine Instanz der Größe 1024 in weniger als 8 Minuten gelöst werden (Dies wird für $n_0 = 2^k$ und bestimmte Werte von k erreicht, die man dadurch erhält, daß man die Rekurrenzgleichung aus der Komplexitätsanalyse löst).

- Der beste Wert für den Grenzwert hängt nicht nur von dem Algorithmus sondern auch von der Implementation ab.
- Änderungen des Grenzwerts beeinflussen die Effizienz des Algorithmus nicht, wenn nur Instanzen spezifischer Größe berücksichtigt werden (dies wird anhand des vorherigen Beispiels beschrieben).
- Im allgemeinen gibt es keinen besten Grenzwert. In unserem Beispiel: ein Grenzwert größer als 66 ist optimal für Instanzen der Größe 67, und es ist besser, einen Grenzwert zwischen 33 und 65 für Instanzen der Größe 66 zu benutzen.

Wie wählt man n_0? Die allgemeinen Richtlinien sind:

1. Man muß $n_0 \geq 1$ wählen, um die unendliche Rekursion zu vermeiden, die für Instanzen der Größe 1 auftreten kann.

2. Wenn eine bestimmte Implementation gegeben ist, kann der optimale Grenzwert empirisch bestimmt werden. Dies bedeutet, daß man die Ergebnisse der Untersuchungen auflistet, wobei verschiedene Werte der Instanzen und Grenzwerte benutzt werden.

3. Eine bessere Lösung ist oft eine hybride Methode:

   (i) Man bestimmt theoretisch die Form der Rekurrenzgleichungen,

   (ii) Man findet empirisch die Werte der Konstante, die in diesen Gleichungen für die vorhandene Implementation benutzt werden.

Dies verdeutlichen wir anhand des vorherigen Beispiels. Der optimale Grenzwert kann durch das Lösen der folgenden Gleichung gefunden werden:

$t_A(n) = 3t_A(\lceil \frac{n}{2} \rceil) + t(n)$
da $t_c(\lceil \frac{n}{2} \rceil) = t_A(\lceil \frac{n}{2} \rceil)$, wenn $\lceil \frac{n}{2} \rceil \leq n_0$.

Vernachlässigt man die Dachfunktion, d.h. man betrachtet $\frac{n}{2}$, so erhält man $n = 64$. Auf der anderen Seite, wenn wir $\lceil \frac{n}{2} \rceil$ durch $\frac{(n+1)}{2}$ ersetzen, erhalten wir $n \simeq 70$. Eine mögliche gute Lösung (wenn man berücksichtigt, daß der Mittelwert von $\lceil \frac{n}{2} \rceil$ $\frac{(2n+1)}{4}$ ist) besteht darin, daß man $n_0 = 67$ für den Grenzwert wäht.

Bemerkung : In der Praxis gibt es noch eine andere Schwierigkeit. Sei $t_A(n) = an^2 + bn + c$. Da der Grundalgorithmus gerade auf Instanzen von kleineren Größen angewendet wird, kann man $(bn+c)$ nicht ganz vernachlässigen. Eine mögliche Lösung besteht darin, $t_A(n)$ für mehrere Werte von n zu messen und die Konstanten mit Hilfe von Regressionstechniken abzuschätzen.

4.3 Selektion und Median

Sei $T[1..n]$ eine Reihung der ganzen Zahlen. m ist der Median von T genau dann, wenn

1. $m \in T$

2. $\#\{i \in [1..n] \mid T[i] < m\} < \frac{n}{2}$ und
   $\#\{i \in [1..n] \mid T[i] \leq m\} \geq \frac{n}{2}$

Bei dieser Definition wird die Möglichkeit berücksichtigt, daß m nicht gerade ist oder nicht alle Elemente von T verschieden sind.

Naiver Algorithmus : Die Reihung ist in aufsteigender Ordnung zu sortieren und man erhält das $\lceil \frac{n}{2} \rceil$-te Element. Mit mergesort benötigt man eine Zeit von O(nlog(n)), um den Median zu bestimmen.

Eine bessere Lösung ist möglich, wenn man das folgende Problem studiert:

Selektion-Problem : T : Reihung der Größe n, k: ganze Zahl zwischen 1 und n. Das k-te kleinste Element von T ist das Element m, so daß
$\#\{i \in [1..n] \mid T[i] < m\} <$ k während
$\#\{i \in [1..n] \mid T[i] \leq m\} \geq$ k.

Dies bedeutet, daß es das k-te Element aus T ist, wenn die Reihung in aufsteigender Ordnung sortiert ist. Zum Beispiel, der Median von T ist das $\lceil \frac{n}{2} \rceil$-te kleinste Element.

Analog zum Quicksort-Algorithmus, ist es möglich, den folgenden Algorithmus, den wir später diskutieren werden, zu entwerfen.

```
Algorithmus selection (T[1..n],k)
{Finde das k-te kleinste Element von T; 1 ≤ k ≤ n}
    if n ist klein then
        sort(T)
        return T[k]
    p ← irgendein Element aus T[1..n]    {Es wird später spezifiziert. Man nennt es häufig Pivot.}
    u ← #{i ∈ [1..n] | T[i] < p}
    v ← #{i ∈ [1..n] | T[i] ≤ p}
    if k ≤ u then
        array U[1..u]
        U ← Die Elemente aus T kleiner als p
        {Das k-te kleinste Element aus T ist auch das k-te kleinste Element aus U}
        return selection(U,k)
    if k ≤ v
        then return p {Das ist die Lösung!}
        else  array V[1..(n-v)]
              V ← die Elemente aus T größer als p
              {das k-te kleinste Elemente aus T ist auch das (k-v)-te kleinste Element aus V}
              return selection(V,k-v)
```

Welches Element aus T sollen wir als Pivot p benutzen? Die beste Wahl ist sicherlich der Median von T, so daß die Größen von U und V möglichst gleich sind.

(i) Nehmen wir an, daß wir eine Methode zur Berechnung des Medians mit einer Kosteneinheit hätten (Dies bedeutet, daß es sich um eine Vereinfachung statt um Teile-und-Herrsche handelt). Da per Definition $u < \lceil \frac{n}{2} \rceil$ und $v \geq \lceil \frac{n}{2} \rceil$ ist, haben wir : $n - v \leq \lfloor \frac{n}{2} \rfloor$. Wenn es eine Rekursion gibt, enthalten die Reihungen U und V höchstens $\lfloor \frac{n}{2} \rfloor$ Elemente. In diesem Fall benötigen die restlichen Operationen eine Zeit von O(n). Sei $t_m(n)$ die Zeit, die in dieser Methode im schlechtesten Fall benötigt wird, um das k-te kleinste Element einer Reihung von höchstens n Elementen zu finden. Auf diese Weise erhalten wir :

$$t_m(n) \in O(n) + max\{t_m(i) \mid i \leq \lfloor \frac{n}{2} \rfloor\}$$

Es ist leicht durch Induktion zu beweisen (der Beweis ist aber lang), daß der Algorithmus linear ist.

$$t_m(n) \in O(n)$$

(ii) Was werden wir tun, wenn wir zur Bestimmung des Medians in einer bestimmten Zeiteinheit keinen einfachen Weg haben, und wenn wir bereit sind, eine höhere Zeitkomplexität im schlechtesten Fall in Kauf zu nehmen, um eine einigermaßen schnelle mittlere Laufzeit zu erreichen ? Wie in Quicksort kann man einfach wählen

$$p \leftarrow T[1]$$

Wenn wir das tun, nehmen wir an, daß die Größen von U und V nicht sehr unbalanziert sind. Jedoch benötigt der schlechteste Fall des resultierenden Algorithmus eine quadratische Zeit, obwohl die mittlere Laufzeit noch linear ist.

(iii) Um diesen quadratischen schlechtesten Fall zu vermeiden,versucht man, einen guten Näherungswert des Medians so schnell wie möglich zu finden. Annahme: $n \geq 5$

```
Algorithmus pseudomed (T[1..n])
    {Finde einen Näherungswert des Medians von Reihung T}
    s ← n div 5      {div: ganzzahlige Division}
    array S[1..s]
    for i ← 1 to s do S[i] ← adhocmed5(T[5i-4..5i])
    return selection(S,(s+1) div 2).
```

---

adhocmed5 ist irgendein spezieller Algorithmus, der den Median von genau 5 Elementen berechnet. Die benötigte Zeit ist dabei konstant.

Sei m der Näherungswert des Medians in “pseudomed” . Dieser ist der exakte Median von S. Daher:

$$\#\{i \in [1..s] \mid S[i] \leq m\} \geq \lceil \frac{s}{2} \rceil$$

Jedes Element aus S ist jedoch der Median von 5 Elementen aus T. Infolgedessen gibt es für i mit S[i] $\leq$ m exakt 3 i_1, i_2, i_3 zwischen (5i-4) und 5i, so daß gilt:

$$T[i_1] \leq T[i_2] \leq T[i_3] = S[i] \leq m$$

Folglich:

$$\#\{i \in [1..n] \mid T[i] \leq m\} \geq 3\lceil \frac{s}{2} \rceil = 3\lceil \frac{\lfloor \frac{n}{5} \rfloor}{2} \rceil \geq \frac{(3n-12)}{10}$$

Ähnlich:

$$\#\{i \in [1..n] \mid T[i] < m\} \leq \frac{(7n-3)}{10}$$

Daher liegt m näherungsweise zwischen $\frac{3n}{10}$ und $\frac{7n}{10}$.

Komplexitätsanalyse von "Selection" (Zusammenfassung der Analyse)

Wir führen ein: p ← pseudomed(T)

1. pseudomed(T) benötigt O(n) + t($\lfloor \frac{n}{5} \rfloor$), da die Reihung S in linearer Zeit konstruierbar ist.

2. Die Berechnungen von u und v benötigen lineare Zeit.

3. Wir haben oben gezeigt, daß $u \leq \frac{(7n-3)}{10}$ und $v \geq \frac{(3n-12)}{10}$ gilt. Daraus folgt: $(n - v) \leq \frac{(7n+12)}{10}$. Der nun folgende rekursive Aufruf benötigt daher eine Zeit, die nach oben durch

$$max\{t(i) \mid i \leq \frac{(7n+12)}{10}\}$$

   beschränkt ist.

4. Initialisierung der Reihungen U und V: lineare Zeit.

5. Folglich existiert eine Konstante c, so daß für jedes genügend großes n gilt:

$$t(n) \leq t(\lfloor \frac{n}{5} \rfloor) + max\{t(i) \mid i \leq \frac{(7n+12)}{10}\} + cn$$

Wir benutzen ein allgemeines Ergebnis der Komplexitätstheorie :
Seien p,q zwei positive reelle Konstanten mit p + q < 1; n_0 eine positive ganze Zahl; b eine positive reelle Konstante. Sei $f : N \rightarrow R^*$ eine Funktion mit

$$f(n) = f(\lfloor p^n \rfloor) + f(\lfloor q^n \rfloor) + bn; \qquad n > n_0$$

Dann: $f(n) \in \Theta(n)$

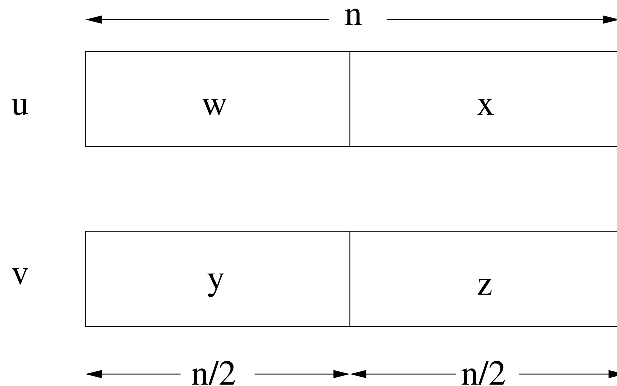
Der beweis läßt sich fortsetzen, indem gezeigt wird, daß f(n) existiert mit

$$t(n) \leq f(n)$$

Dies impliziert : $t(n) \in O(n)$

4.4 Langzahlarithmetik

Wir betrachten hier das Beispiel der Multiplikation zweier ganzen Zahlen von n Dezimalziffern, wobei n sehr groß sein kann. Teile-und-Herrsche suggeriert, daß jeder Operand in zwei Teile möglichst gleicher Größe zerlegt wird.



$u = 10^s w + x$; $v = 10^s y + z$ mit
$0 \leq x \leq 10^s$, $0 \leq z \leq 10^s$, $s = \lfloor \frac{n}{2} \rfloor$
Wir interessieren uns für das Produkt

$$uv = 10^{2s} wy + 10^s (wz + xy) + xz$$

Dies führt zu dem folgenden Algorithmus :

Algorithmus mult(u,v)
  n ← die kleinste ganze Zahl, so daß u und v von Größe n sind
  if n klein **then** **return** classic-product(u,v)
  s ← n **div** 2
  w ← u **div** 10^s; x ← u **mod** 10^s
  y ← v **div** 10^s; z ← v **mod** 10^s
  return
    mult(w,y) × 10^{2s}
    + (mult(w,z) - mult(x,y)) × 10^s
    + mult(x,z)

Eine triviale Verbesserung wird dadurch erreicht, daß man den letzten Schritt durch

```
r ← mult(w+x, y+z)
p ← mult(w,y)
q ← mult(x,z)
```
return $10^{2s}p + 10^s(r - p - q) + q$

ersetzt.

Bemerkung1: Die Komplexitätsanalyse zeigt, daß der Algorithmus eine Zeit von $O(n^{log_2 3}) = O(n^{1.59})$ benötigt. Benutzt man die "Schnelle-Fourier-Transformation" und Teile-und-Herrsche zum Entwurf eines Algorithmus, so kann die Komplexität auf $O(n \cdot \log n \cdot \log \log n)$ reduziert werden.

Bemerkung2: Eine spezielle Version dieses Algorithmus ist als "Karatsuba-Algorithmus" bekannt. Seien n gerade mit $n = 2m$, u und v ganze Zahlen der Länge n Bits.

$$u = a2^m + b; v = c2^m + d$$

Folglich: $w = uv = y2^{2m} + (x - y - z)2^m + z$, wobei $x = (a + b)(c + d), y = ac, z = bd$

Die Komplexitätsanalyse läßt sich einfacher durchführen. Wir haben nur drei Multiplikationen von ganzen Zahlen der Länge m und Operationen (Addition, Shiften), die linear sind. Infolgedessen existiert eine Konstante k, so daß gilt:

$$t(n) = \begin{cases} k & : \; falls \;\; m = 1 \\ 3t(m) + kn & : \; falls \;\; m > 1 \end{cases}$$

Seien n eine positive ganze Zahl und t die kleinste ganze Zahl, so daß

$$2^t \geq n, \quad somit \quad t(n) \leq t(2^t)$$

Benutzt man diese Relation für $2^t, 2^{t-1}, ..., 2$, so erhält man

$$t(2^t) = O(3^t)$$

Dies impliziert: $t(n) = O(n^\alpha)$, wobei $\alpha = \frac{log3}{log2} \simeq 1.585$.

Bemerkung3: Eine sehr gute Spezialisierung der Multiplikation auf das Potenzieren ist eine der interessantesten Anwendungen der Teile-und-Herrsche-Methode. Darauf werden wir später eingehen.

4.5 Matrixmultiplikation

A, B: Zwei n×n-Matrizen; $C = A \cdot B$

$$C_{ij} = \sum_{k=1}^{n} A_{jk} B_{kj}$$

Jede Eintragung in C wird in $\Theta(n)$ berechnet, wenn man annimmt, daß skalare Addition und Multiplikation elementare Operationen sind. Es gibt n^2 Eintragungen in C. Daher wird das Produkt $A \cdot B$ in $\Theta(n^3)$ berechnet. Eine Verbesserung ist auf eine Idee von Strassen zurückzuführen. Die Idee ist ähnlich wie Teile-und-Herrsche.

Zunächst zeigt man, daß es möglich ist, 2×2-Matrizen mit weniger als acht skalaren Operationen zu multiplizieren.

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}; \quad B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

$$\begin{aligned}
m_1 &= (a_{21} + a_{22} - a_{11})(b_{22} - b_{12} + b_{11}) \\
m_2 &= a_{11}b_{11} \\
m_3 &= a_{12}b_{21} \\
m_4 &= (a_{11} - a_{21})(b_{22} - b_{12}) \\
m_5 &= (a_{21} - a_{22})(b_{12} - b_{11}) \\
m_6 &= (a_{12} - a_{21} + a_{11} - a_{22})b_{22} \\
m_7 &= a_{22}(b_{11} + b_{22} - b_{12} - b_{21})
\end{aligned}$$

$$\Rightarrow AB = \begin{pmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 - m_7 & m_1 + m_2 + m_4 + m_5 \end{pmatrix}$$

Auf diese Weise benötigt man nur sieben skalare Multiplikationen. Des Hauptziel liegt darin, diesen Algorithmus auf 2n×2n Matrizen zu erweitern. Dies ist trivial, da kein Schritt in dieser Methode auf die Komutativität von skalaren Multiplikationen basiert.

Komplexitätsanalyse

Es ist immer noch ein Forschungsthema, einen asymptotisch schnellen Algorithmus für Matrixmultiplikation zu finden. Die Situation läßt sich folgendermaßen beschreiben:

1. Die Methode von Strassen benötigt $O(n^{2.81})$. 18 Additionen und Subtraktionen (nicht 24, wie in unserer einfachen Beschreibung) und 7 Multiplikationen werden benötigt. Um dies zu erreichen, vermeidet man Terme wie $(m_1 + m_2 + m_4)$ neu zu berechnen.

2. Das letzte Ergebnis ist auf Coppersmith und Winograd (1986) zurückzuführen. Sie entwarfen einen Algorithmus, der eine Zeit von $O(n^{2.376})$ benötigt.

3. Die Werte der Konstante bei den verbesserten Algorithmen liegen so, daß die Methode von Strassen in der Praxis immer noch am günstigsten ist.

Bemerkung: Die Methode von Strassen kann auf 3×3-Matrix erweitert werden. Man kann zeigen, daß die minimale Anzahl der benötigten Multiplikationen 23 ist (Algorithmus von Laderman). Ein offenes Problem besteht darin, die minimale Anzahl von Multiplikationen zur Multiplikation von zwei n×n Matrizen für $n > 3$ zu finden.

Kapitel 5

Dynamisches Programmieren

5.1 Einführung

Dynamisches Programmieren ist eine Methode zum Entwurf von Algorithmen, die man dann anwenden kann, wenn die Lösung eines Problems als Ergebnis einer Folge von Entscheidungen angesehen werden kann.

Wie wir gesehen haben, ist Teile-und-Herrsche eine "top-down"-Methode: Man betrachtet direkt die Lösung einer globalen Instanz, die in weitere kleinere Teile zerlegt wird.

Dynamisches Programmieren ist eine "bottom-up"-Methode. Man beginnt in der Regel mit den kleinsten oder einfachsten Teilinstanzen. Kombiniert man die einzelnen Lösungen, so erhält man die Ergebnisse der Teilinstanzen zunehmender Größe, bis wir zum Schluß die Lösung der Originalinstanz erreichen.

Außer dieser Definition gibt es ein weiteres Grundprinzip des dynamischen Programmierens: Vermeide, die gleichen Berechnungen zweimal durchzuführen ,indem die bekannten Ergebnisse in einer Tabelle gehalten werden, in die jede gelöste Teilinstanz eingetragen wird.

Dynamisches Programmieren ist besonders interessant für Probleme, für die es keine Möglichkeit gibt, eine Folge schrittweiser Entscheidungen zu bestimmen, die zu einer optimalen Entscheidungsfolge führt. Zum Beispiel, wenn man alle möglichen Lösungen aufzählen kann und dann die beste davon auswählt. Dynamisches Programmieren reduziert oft drastisch die Anzhl der Lösungen, die nicht optimal sein können. Eine optimale Folge von Entscheidungen wird erreicht durch die Anwendung des

Optimalitätsprinzips: Eine optimale Folge von Entscheidungen besitzt die Eigenschaft, daß unabhängig vom Anfangszustand und von der Anfangsentscheidung die übrigen Entscheidungen eine optimale Entscheidungsfolge bilden müssen unter Berück-sichtigung des aus der ersten Entscheidung resultierenden Zustands.

Basierend auf das Prinzip können wir den Unterschied zwischen dynamischem Programmieren und Greedy-Algorithmen feststellen: Beim dynamischen Programmieren können verschiedene Entscheidungsfolgen erzeugt werden, während bei Greedy-Algorithmen überhaupt nur eine einzige Entscheidungsfolge erzeugt wird. Jedoch können Folgen nicht optimal sein, obwohl das Optimalitätsprinzip gilt.

5.2 Optimale binäre Suchbäume

Definition: Ein binärer Suchbaum T ist ein binärer Baum, der entweder leer ist oder bei dem jeder Knoten einen Bezeichner enthält und

(i) Alle Bezeichner im linken Teilbaum von T sind (numerisch oder lexikographisch) kleiner als der Bezeichner im Wurzelknoten T,

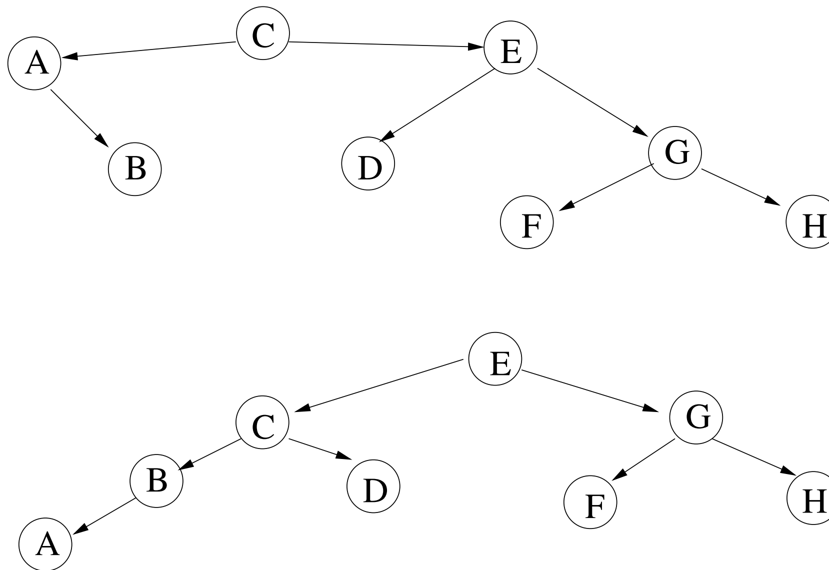
(ii) Alle Bezeichner im rechten Teilbaum von T sind größer als der Bezeichner im Wurzelknoten T,

(iii) Der linke und der rechte Teilbaum von T sind ebenfalls binäre Suchbäume.

In der Definition wird gefordert, daß alle Bezeichner im Baum voneinander verschieden sind. Zu einer gegebenen Menge von Bezeichnern sind mehrere verschiedene binäre Suchbäume möglich. Zum Beispiel:

Wir möchten feststellen, ob ein Bezeichner X (auch Schlüssel genannt) im Baum vorkommt. Zunächst vergleichen wir X mit dem Schlüssel R der Wurzel.

- Wenn $X = R$ ist : Wir haben gefunden.
- Wenn $X < R$ ist : Suche im linken Teilbaum.
- Wenn $X > R$ ist : Suche im rechten Teilbaum.



```
procedure search (T,X)
    {Durchsuche den binären Suchbaum T nach X}
    {Jeder Knoten von T hat Felder LCHILD, RCHILD, IDENT}
    if T=nil then return "nicht gefunden"
        else if X = IDENT(ROOT(T)) then return ROOT(T)
            else if IDENT(ROOT(T)) < X then search(RCHILD(T),X)
                else search(LCHILD(T),X)
```

Wenn alle Schlüssel mit gleicher Wahrscheinlichkeit gesucht werden, benötigt das obige Beispiel durchschnittlich

$$\frac{(2+3+1+3+2+4+3+4)}{8} = \frac{22}{8} \quad \text{beziehungsweise}$$

$$\frac{(4+3+2+3+1+3+2+3)}{8} = \frac{21}{8}$$

Vergleiche, um einen Schlüssel zu finden. In Wirklichkeit möchten wir jedoch ein allgemeineres Problem lösen.

Angenommen, wir haben eine geordnete Menge $c_1 < c_2 < \cdots < c_n$ von n verschiedenen Schlüsseln. Sei p_i die Wahrscheinlichkeit dafür, daß wir nach c_i suchen für $i = 1, 2, 3, ..., n$. Wir nehmen an, daß $\sum_{i=1}^{n} p_i = 1$ gilt (d.h. man

sucht nur nach Schlüsseln, die in T enthalten sind). Die Stufe (Tiefe) der Wurzel ist 0, die Stufe der Söhne ist 1 und so fort. Wenn ein Schlüssel auf der Stufe d_i zu finden ist, dann werden $(d_i + 1)$ Vergleiche benötigt, um den Schlüssel zu finden. Die mittlere Anzahl von Vergleichen für einen gegebenen Baum ist:

$$C = \sum_{i=1}^{n} p_i(d_i + 1)$$

Das ist die gesuchte Funktion, die zu minimieren ist. Betrachten wir die Folge von Schlüsseln $c_i, c_{i+1}, ..., c_j \quad (j \geq i)$. Angenommen, ein optimaler Baum enthält alle n Schlüsseln. Die Folge von $(j - i + 1)$ Schlüssel enthält die Knoten eines Teilbaums. Ist der Schlüssel c_k $(i \leq k \leq j)$ in einem Knoten der Stufe d_k^* in diesem Teilbaum enthalten, so erhält man die mittlere Anzahl der Vergleiche in diesem Teilbaum

$$C^* = \sum_{k=i}^{j} P_k(d_k^* + 1),$$

wenn wir einen Schlüssel im Hauptbaum suchen (allerdings muß dieser Schlüssel nicht im Teilbaum sein).

Wir beobachten :

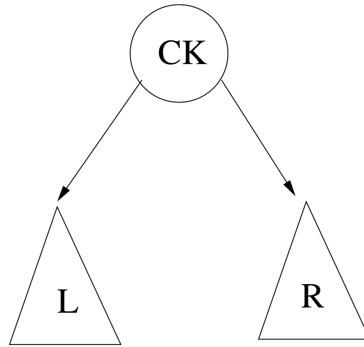
(i) Dieser Ausdruck hat dieselbe Form wie der Ausdruck für C,

(ii) Eine Änderung im Teilbaum beeinflußt die Mitwirkung anderer Teilbäume des Hauptbaums zu C nicht.

Auf diese Weise erreichen wir das Optimalitätsprinzip: In einem optimalen Baum müssen alle Teilbäume bezüglich der enthaltenen Schlüssel auch optimal sein.

Seien $m_{ij} = \sum_{k=i}^{j} P_k$ und C_{ij} die mittlere Anzahl von erforderlichen Vergleichen in einem optimalen Teilbaum, der die Schlüssel $c_i, c_{i+1}, ..., c_j$ enthält, wenn ein Schlüssel im Hauptbaum gesucht wird. Wir definieren $C_{ij} = 0$, wenn $j = (i - 1)$ gilt. Einer dieser Schlüssel c_k ist in der Wurzel des Baumes

L und R sind optimale Teilbäume. $c_i, ..., c_{k-1} \in$ L und $c_{k+1}, ..., c_j \in$ R. Die Wahrscheinlichkeit dafür, daß ein Schlüssel in $\{C_{k+1}, ..., C_j\}$ liegt, ist m_{ij}, wenn wir den Schlüssel im Hauptbaum suchen. Ein Vergleich mit c_k wird



ausgeführt und andere Vergleiche können in L oder in R ausgeführt werden. Die mittlere Anzahl von erforderlichen Vergleichen ist damit:

$$C_{ij}^k = m_{ij} + C_{i,k-1} + C_{k+1,j}$$

Um ein Schema des dynamischen Programmierens zu erreichen, muß die Wurzel k so gewählt werden, daß C_{ij} minimiert wird.

$$C_{ij} = m_{ij} + min_{i \leq k \leq j}(C_{i,k-1} + C_{k+1,j})$$

Insbesondere: $C_{ii} = P_i$

Beispiel: 5 Schlüssel $c_1, ..., c_5$ mit den folgenden Wahrscheinlichkeiten:

i	1	2	3	4	5
P_i	0.30	0.05	0.08	0.45	0.12

Zunächst berechnet man die Matrix m:

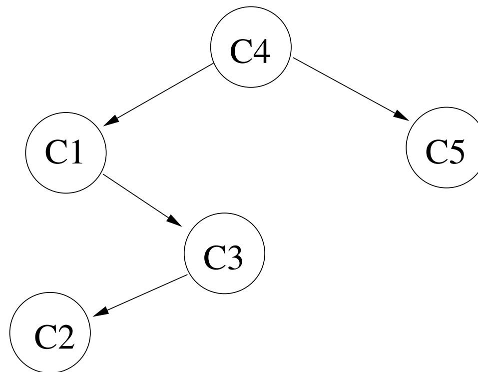
$$m = \begin{pmatrix} 0.30 & 0.35 & 0.43 & 0.88 & 1.00 \\ & 0.05 & 0.13 & 0.58 & 0.70 \\ & & 0.08 & 0.53 & 0.65 \\ & & & 0.45 & 0.57 \\ & & & & 0.12 \end{pmatrix}$$

Mit $C_{ii} = P_i$ und $C_{ij} = 0$ falls $j = i - 1$ lassen sich die anderen Werte von C_{ij} folgendermaßen berechnen:

$C_{12} = m_{12} + min(C_{10} + C_{22}, C_{11} + C_{32})$

$= 0.35 + min(0 + 0.05, 0.30 + 0) = 0.40$
$C_{23} = 0.18$, $C_{34} = 0.61$, $C_{45} = 0.69$
Dann :
$C_{13} = m_{13} + min(C_{10} + C_{23}, C_{11} + C_{33}, C_{12} + C_{43}) = 0.61$
$C_{24} = m_{24} + min(C_{21} + C_{34}, C_{22} + C_{44}, C_{23} + C_{54}) = 0.76$
$C_{35} = m_{35} + min(C_{32} + C_{45}, C_{33} + C_{55}, C_{34} + C_{65}) = 0.85$
$C_{14} = m_{14} + min(C_{10} + C_{24}, C_{11} + C_{34}, C_{12} + C_{44}, C_{13} + C_{54}) = 1.49$
$C_{25} = m_{25} + min(C_{21} + C_{35}, C_{22} + C_{45}, C_{23} + C_{55}, C_{24} + C_{65}) = 1.00$
$C_{15} = m_{15} + min(C_{10} + C_{25}, C_{11} + C_{35}, C_{12} + C_{45}, C_{13} + C_{55}, C_{14} + C_{65}) = 1.73$

Folglich erfordert der optimale Suchbaum für diese Schlüssel 1.73 Vergleiche im mittleren Fall, um einen Schlüssel zu finden.



Dieses Bild zeigt den optimalen binären Suchbaum

Komplexitätsanalyse :

Wir berechnen C_{ij} für $(j-i) = 1, 2, ..., n$. Wenn $j - i = m$ gilt, gibt es $(n-m+1)C_{ij}$'s, die zu berechnen sind. Bei der Ermittlung eines C_{ij} ist es erforderlich, unter m Größen das Minimum zu finden. Also wird jedes C_{ij} in der Zeit O(m) berechnet. Daher benötigt man für die $(n-m+1)C_{ij}$'s: $O(nm - m^2)$. Die Gesamtzeit zur Auswertung aller möglichen Wurzeln ist daher:

$$\sum_{1 \leq m \leq n} (nm - m^2) = O(n^3)$$

Bemerkung : Nach Knuth ist eine Verbesserung möglich, indem man sich bei der Suche auf den Bereich

$$R(i, j-1) \leq k \leq R(i+1, j)$$

beschränkt. In diesem Fall wird die Rechenzeit $O(n^2)$. Einen solchen Algorithmus findet man in Horowitz-Sahni.

5.3 Das Problem des Handlungsreisenden

{Dieses Problem ist auch für die Untersuchung der Bewegung des Armes eines Roboters geeignet.}
Sei G = <N,A> ein gerichteter Graph, N = {1,2,....,n}. Die Längen der Kanten sind L_{ij} mit $L_{ii} = L[i,i] = 0$, $L[i,j] \geq 0$ für $i \neq j$, $L[i,j] = \infty$ falls die Kante (i,j) nicht existiert. Ohne Beschränkung der Allgemeinheit nehmen wir an, daß die Tour bei Knoten 1 beginnt und endet. Sie besteht daher aus einer Kante $(1,j)$, $j \neq 1$ und aus einem Pfad von j zu 1, der genau einmal durch jeden Knoten $N \setminus \{1,j\}$ führt. Ist die Tour optimal, dann ist der Pfad von j nach 1 ein kürzester Pfad. Also gilt das Optimalitätsprinzip.

Wir betrachten eine Menge von Knoten $S \subseteq N \setminus \{1\}$ und einen Knoten $i \in N \setminus S$, wobei $i = 1$ zugelassen wird, wenn $S = N \setminus \{1\}$ ist. Sei $g(i,S)$ die Länge eines kürzesten Pfades, der bei Knoten i beginnt, durch alle Knoten in S geht und bei Knoten 1 endet. Nach dieser Definition ist $g(1, N \setminus \{1\})$ die Länge einer optimalen Tour. Aus dem Optimalitätsprinzip folgt :

$$g(1, N \setminus \{1\}) = min_{2 \leq j \leq n}(L_{1j} + g(j, N \setminus \{1,j\})) \tag{5.1}$$

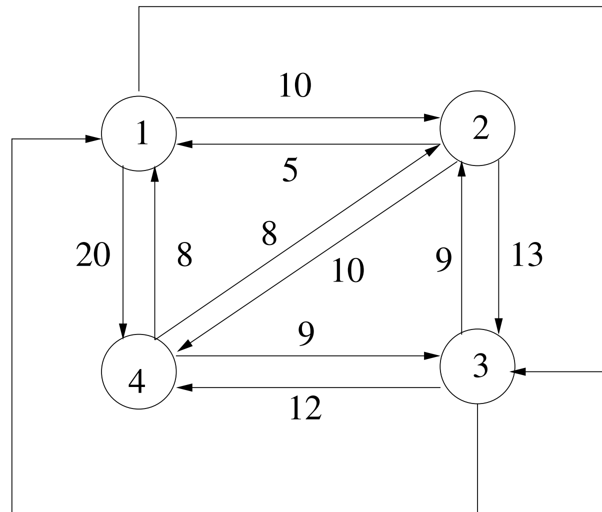
Allgemeiner: Falls $i \neq 1, S \neq \emptyset, S \neq N \setminus \{1\}$ und $i \notin S$ gilt :

$$g(i,S) = min_{j \in S}(L_{ij} + g(j, S \setminus \{j\})) \tag{5.2}$$

Es gilt: $g(i, \emptyset) = L_{i1}$, $i = 2, 3, ..., n$

Die Werte von $g(i,S)$ sind daher bekannt, wenn S leer ist. Wir können (3.2) anwenden, um g für alle Mengen, die genau einen Knoten ($\neq 1$) enthalten, zu berechnen, und wieder auf diese Weise für alle Mengen mit zwei Knoten ($\neq 1$) anwenden und so weiter. Sind alle Werte bekannt, so können wir (3.1) anwenden, um $g(1, N \setminus \{1\})$ zu berechnen und das Problem zu lösen.

Beispiel : Wir haben den folgenden Graphen:



$$L = \begin{pmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{pmatrix}$$

Wir initialisieren: $g(2, \emptyset) = 5$, $g(3, \emptyset) = 6$, $g(4, \emptyset) = 8$.

Durch die Anwendung von (3.2) erhält man: $g(2, \{3\}) = L_{23} + g(3, \emptyset) = 15$
$g(2, \{4\}) = L_{24} + g(4, \emptyset) = 18$

Ähnlich: $g(3, \{2\}) = 18, g(3, \{4\}) = 20, g(4, \{2\}) = 13, g(4, \{3\}) = 15$

Nun wenden wir (3.2) auf Mengen von zwei Knoten an

$$g(2, \{3, 4\}) = min(L_{23} + g(3, \{4\}), L_{24} + g(4, \{3\})) = min(29, 25) = 25$$
$$g(3, \{2, 4\}) = min(L_{32} + g(2, \{4\}), L_{34} + g(4, \{2\})) = 25$$
$$g(4, \{2, 3\}) = min(L_{42} + g(2, \{3\}), L_{43} + g(3, \{2\})) = 23$$

Zum Schluß:
$$g(1, \{2, 3, 4\}) = min(L_{12} + g(2, \{3, 4\}), L_{13} + g(3, \{2, 4\}, L_{14} + g(4, \{2, 3\})))$$
$$= min(35, 40, 43) = 35$$

Daher hat die optimale Tour in diesem Beispiel die Länge 35.

Um zu wissen, wohin die Tour führt, brauchen wir eine zusätzliche Funktion: $J(i, S)$ ist der Wert des gewählten j zur Minimierung von g zum Zeitpunkt, wo wir (3.1) und (3.2) anwenden, um $g(i, S)$ zu berechnen.

In diesem Beispiel: $J(2,\{3,4\}) = 4$
$J(3,\{2,4\}) = 4$
$J(4,\{2,3\}) = 2$
$J(1,\{2,3,4\}) = 2$

Die optimale Tour ist: $1 \to J(1,\{2,3,4\}) = 2 \to J(2,\{3,4\}) = 4 \to J(4,\{3\}) = 3 \to 1$.

Berechnungszeit:

- Zur Berechnung von $g(j,\emptyset)$: $n-1$ Konsultationen einer Tabelle.
- Zur Berechnung aller $g(i,S)$ für $1 \le \#S = k \le n-2$:

  $$(n-2)\binom{n-2}{k}k$$

  Additionen insgesamt (wobei die "Spalte" den binomischen Koeffizienten entspricht).
- Zur Berechnung von $g(1,N\setminus\{1\})$: $n-1$ Additionen.

Die Zeitkomplexität ist daher:

$$\Theta\left(2(n-1)+\sum_{k=1}^{n-2}(n-1)k\binom{n-2}{k}\right) = \Theta(n^2 2^n)$$

da gilt:

$$\sum_{k=1}^{r} k\binom{r}{k} = r2^{r-1}$$

Dies ist besser als $\Omega(n!)$, jedoch kein praktischer Algorithmus. Darüber hinaus kann man prüfen, daß der Platzaufwand für die Werte von g und J $\Omega(n2^n)$ ist. Die folgende Tabelle verdeutlicht, daß diese Werte so sind, daß der Algorithmus nicht sehr praktisch ist.

n	Zeit Direkte Methode	Zeit Dynamisches Programmieren	Platz Dynamisches Programmieren
	$n!$	$n^2 2^n$	$n2^n$
5	120	800	160

10	3628800	102400	10240
15	1.31×10^{12}	7372800	491520
20	2.43×10^{18}	419430400	20971520

Dies zeigt, daß $20^2 2^{20}$ Mikrosekunden weniger als 7 Minuten, aber 20! Mikrosekunden mehr als 77.000 Jahre sind.

5.4 Verkettete Matrixmultiplikation

Wir möchten $M = M_1 M_2 ... M_n$ berechnen. Da die Matrixmultiplikation assoziativ ist, haben wir mehrere Möglichkeiten, das Produkt zu berechnen. Zum Beispiel:

$$\begin{aligned} M &= (M_1(M_2(M_3.....(M_{n-1}M_n).....))) \\ &= ((.....(M_1M_2)M_3).....M_n) \\ &= ((M_1M_2)(M_3M_4).....) \text{ u. s. w.} \end{aligned}$$

Die Auswahl einer Berechnungsmethode kann einen wesentlichen Einfluß auf die Laufzeit haben.

Beispiel : Wenn A eine $p \times q$-Matrix und B eine $q \times r$-Matrix ist, kann man zeigen, daß die direkte Methode zur Berechnung von AB pqr skalare Multiplikationen erfordert.

Wenn A: 13×5, B: 5×89, C: 89×3, D: 3×34 ist, erhält man die folgende Anzahl der Operationen:

$((AB)C)D$	10582
$(AB)(CD)$	54201
$(A(BC))D$	2856
$A((BC)D)$	4055
$A(B(CD))$	26418

Was sind die Kosten zur Bestimmung der besten Lösung, wenn man alle möglichen Lösungen untersucht? Sei $T(n)$ die Anzahl der verschiedenen Möglichkeiten, ein Produkt von n Matrizen in Klammern zu setzen. Trennen wir das Matrizenprodukt an der i-ten Stelle

$$M = (M_1......M_i)(M_{i+1}.....M_n)$$

so gibt es $T(i)$ Möglichkeiten, den linken Term in Klammern zu setzen und $T(n-i)$ Möglichkeiten für den rechten Term. Da i zwischen 1 und $(n-1)$ liegt, erhalten wir:

$$T(n) = \sum_{i=1}^{n-1} T(i)T(n-i)$$

Zusätzlich hat man $T(1) = 1$. Somit kann man die Werte von $T(n)$ für verschiedenen n berechnen.

n	1	2	3	4	5	10	15
$T(n)$	1	1	2	5	14	4862	2674440

$T(n)$ sind in Wirklichkeit wohl bekannte Zahlen aus Zahlentheorie. Man nennt sie Catalansche Zahlen und sie sind definiert durch:

$$T(n) = \frac{1}{n} \binom{2n-2}{n-1}$$

Man kann zeigen, daß $T(n)$ aus $\Omega(\frac{4^n}{n^2})$ ist. Um die Anzahl der erforderlichen skalaren Multiplikationen zur Setzung jedes möglichen Klammerpaares zu zählen, wird eine Zeit von $\Omega(n)$ benötigt. Daher benötigt die direkte Methode eine Zeit von $\Omega(\frac{4n}{n})$ und nicht praktikabel für ein großes n.

Zum Glück gilt in diesem Fall das Optimalitätsprinzip und man kann daher einen besseren Algorithmus durch die Anwendung des dynamischen Programmierens entwerfen. Man konstruiert eine Tabelle m_{ij} $(1 \leq i \leq j \leq n)$, wobei m_{ij} die optimale Lösung (sie ist die Anzahl von skalaren Multiplikationen) für den Teil $M_i M_{i+1} ... M_j$ des Produkts ergibt. Daraus ergibt sich die Lösung des Originalproblems zu m_{1n}

Angenommen die Dimension der Matrizen M_i ist durch einen Vektor d_i $(0 \leq i \leq n)$ gegeben, so daß Matrix M_i von Dimension $d_{i-1} \times d_i$ ist. Wir bilden die Tabelle m_{ij} diagonalenweise: Diagonale s enthält diejenigen Elemente m_{ij}, so daß $j - i = s$. Sukzessiv erhalten wir:

$s = 0$: $m_{ii} = 0$, $i = 1, 2, ..., n$
$s = 1$: $m_{i,i+1} = d_{i-1} d_i d_{i+1}$, $i = 1, 2, ..., n-1$
$1 < s < n$: $m_{i,i+s} = min_{i \leq k \leq i+s} \; (m_{ik} + m_{k+1,i+s} + d_{i-1} d_k d_{i+s})$
$i = 1, 2, ..., n-s$

Der dritte Fall beschreibt das Verfahren zur Berechnung von $M_i...M_{i+s}$, indem man alle Möglichkeiten $(M_i...M_k)(M_{k+1}...M_{i+s})$ untersucht und die beste davon für $i \leq k \leq i+s$ auswählt.

Beispiel : Mit dem vorherigen Beispiel: $d = (13, 5, 89, 3, 34)$

Für $s = 1$:
$m_{12} = 5785$, $m_{23} = 1335$, $m_{34} = 9078$.

Für $s = 2$:
m_{13} =min$(m_{11}+m_{23}+13\times5\times3, m_{12}+m_{33}+13\times89\times3)$ =min$(1530, 9256) =$ 1530
m_{24} =min$(m_{22}+m_{34}+5\times89\times34, m_{23}+m_{44}+5\times3\times34)$ =min$(24208, 1845) =$ 1845.

Schließlich für $s = 3$:
m_{14} =min$(\{k = 1\}m_{11}+m_{24}+13\times3\times34, \{k = 2\}m_{12}+m_{34}+13\times89\times34, \{k = 3\}m_{13} + m_{44} + 13 \times 3 \times 34)$ =min$(4055, 54201, 2856) = 2856$.

Dies ergibt die Lösung: Die Reihung m wird daher gegeben durch:

i/j	1	2	3	4	
1	0	5785	1530	2856	
2		0	1335	1845	$s = 3$
3			0	9078	$s = 2$
4				0	$s = 1$
					$s = 0$

Komplexität : Für $s > 0$ gibt es in der Diagonale s $(n - s)$ Elemente, die zu berechnen sind. Für jedes Element haben wir s Möglichkeiten (die verschiedenen möglichen Werte von k). Die Laufzeit:

$$\sum_{s=1}^{n-1}(n-s)s = n\sum_{s=1}^{n-1} s - \sum_{s=1}^{n-1} s^2 = \frac{n^2(n-1)}{2} - \frac{n(n-1)(2n-1)}{6} = \frac{(n^3-n)}{6}$$

Daher ist die Ausführungszeit $\Theta(n^3)$, besser als die direkte Suche.

Kapitel 6

Probabilistische Algorithmen

Der Inhalt dieses Kapitels stammt aus Forschungspublikationen von Rabin und nicht aus Büchern. Probabilistische Algorithmen wurden zunächst in der Numerik eingeführt (Siehe zum Beispiel I.M. Sobol, "The Monte Carlo Method", Univ. of Chicago Press, 1974), wobei der bekannteste sich mit der Integration befasst. Um $\int_0^1 \cdots \int_0^1 f(x_i)\Pi_i^n dx_i$ zu berechnen, wählt man zufällig Probepunkte aus dem Integrationsbereich. Danach wurden viele Techniken zur Verbesserung der Genauigkeit dieser Methode entworfen, die als Monte-Carlo-Integration bekannt sind. Erst im Jahre 1976 wurde das Konzept von Rabin effektiver formuliert.

6.1 Einführung

Um probabilistische Algorithmen zu definieren, beginnen wir mit der Definition eines deterministischen Algorithmus nach Knuth:

1. Eine Berechnungsmethode ist ein Quadrupel (Q, I, O, f) mit $I \subset Q$, $O \subset Q$, $f : Q \to Q$ und $f(p) = p', \forall p \in I$. Q ist der Zustand der Berechnung, I die Eingabe, O die Ausgabe und f die Berechnungsregeln.
   - Jedes $x \in I$ ergibt eine Folge $x_0, x_1, ...$, die durch $x_{k+1} = f(x_k), k \geq 0, x_0 = x$ definiert ist.
   - Die Folge terminiert nach K Schritten (K: kleiner als die k's, sodaß gilt $x_k \in O$)
2. Ein Algorithmus ist eine Berechnungsmethode, die in endlich vielen Schritten für alle $x \in I$ terminiert.

3. Ein deterministischer Algorithmus ist eine formale Beschreibung für eine endliche, definite Prozedur, deren Ausgaben zu beliebigen Eingaben eindeutig sind.

Die Hauptmotivation zur Einführung von probabilistischen Algorithmen stammt aus der Komplexitätsanalyse. Man unterscheidet zwischen dem Verhalten im schlechtesten Fall und dem Verhalten im mittleren Fall eines Algorithmus. Diese Fälle sind festgelegt, sobald das Problem und die Daten bestimmt sind. Mit anderen Worten: Die effektive Berechnungszeit eines gegebenen Problems hängt von den gewählten Eingabedaten ab. Diese Situation läßt sich dadurch beschreiben, daß durch eine zufällige Auswahl der Eingabe die Berechnungszeit des Algorithmus bestimmt wird. Darüber hinaus wird die obere Schranke durch den schlechtesten Fall festgelegt. Gibt es eine Möglichkeit, diese Situation zu verbessern? Die Lösung, die durch probabilistische Algorithmen erreicht werden kann, besteht darin, daß der Zufall in die Prozedur aufgenommen wird, in der Hoffnung, daß der schlechteste Fall so vermieden wird. Mit anderen Worten: Die mittlere Berechnungszeit für deterministische Algorithmen wird mit der Tatsache verbunden, daß jede mögliche Instanz einer gegebenen Größe als fast gleich berücksichtigt wird, während die erwartete Berechnungszeit eines probabilistischen Algorithmus nur für individuelle Instanzen definiert wird. Dies verweist auf die mittlere Zeit, die durchschnittlich zur Lösung der gleichen Instanz benötigt wird.

Die unterstrichenen Wörter in den Definitionen eines deterministischen Algorithmus sind die Schlüssel zur Definition von drei Arten von probabilistischen Algorithmen.

1. Macao Algorithmen

   - Mindestens bei einem Schritt der Prozedur werden einige Zahlen zufällig ausgewählt (nicht definit).
   - Sonst: deterministisch

   Diese Algorithmen liefern immer eine korrekte Antwort. Sie werden benutzt, wenn irgendein bekannter Algorithmus zur Lösung eines bestimmten Problems im mittleren Fall viel schneller als im schlechtesten Fall läuft. Diese Algorithmen sind auch as "Sherwood"-Algorithmen genannt werden sollen.

2. Algorithmen zweiter Art (Monte-Carlo-Algorithmen)

   - Gleich wie Algorithmen erster Art (nicht definit)

- Zusätzlich: Ausgabe ist korrekt mit einer Wahrscheinlichkeit von $(1-\epsilon)$, wobei ϵ sehr klein ist (nicht eindeutig).

Der Algorithmus liefert immer eine Antwort, wobei die Antwort nicht unbedingt richtig ist. Die Wahrscheinlichkeit dafür, daß eine Antwort richtig ist, ist proportional zu der dem Algorithmus zur Verfügung stehenden Zeit ($\epsilon \to 0$, falls $t \to \infty$). Das Problem bei solchen Algorithmen liegt darin, daß im allgemeinen schwer zu entscheiden ist, ob die Antwort korrekt ist.

3. Algorithmen dritter Art (Las-Vegas-Algorithmen)

   - Gleich wie Algorithmen erster Art (nicht definit)
   - Eine Folge von zufälligen Wahlen kann unendlich sein (mit einer Wahrscheinlichkeit $\epsilon \to 0$) (nicht endlich).

   Diese Algorithmen liefern nie eine unkorrekte Antwort, jedoch besteht die Möglichkeit, daß sie überhaupt keine Antwort finden können. Es sei bemerkt, daß die erwartete Berechnungszeit dennoch endlich ist.

   Im folgenden geben wir einige Beispiele von probabilistischen Algorithmen.

6.2 Macao Algorithmen

6.2.1 Allgemeine Kommentare

Sei A ein deterministischer Algorithmus und sei $t_A(X)$ die benötigte Zeit zur Lösung irgendeiner Instanz X. Für jede ganze Zahl n sei X_n die Menge von Instanzen der Größe n. Wenn man annimmt, daß jede Instanz einer gegebenen Größe gleich wahrscheinlich ist, ist die vom Algorithmus zur Lösung einer Instanz der Größe n benötigte mittlere Zeit:

$$\bar{t}_A(n) = \frac{\sum_{x \in X_n} t_A(x)}{\# X_n}$$

Dies schließt jedoch die Möglichkeit nicht aus, daß eine Instanz x der Größe n existiert, sodaß $t_A(x) >> \bar{t}_A(n)$ gilt. Wir möchten versuchen, einen probabilistischen Algorithmus zu entwerfen, sodaß gilt $t_B(x) \approx \bar{t}_A(n) + s(n)$ für jede Instanz der Größe n, wobei $s(n)$ die zusätzlich zum Erreichen dieser Einheitlichkeit benötigten Kosten darstellt. Es sei angemerkt, daß es möglich ist,

daß Algorithmus B wegen der vom Algorithmus ausgeführten probabilistischen Wahlen (jedoch nicht wegen der selektierten Instanz) gelegentlich für eine Instanz der Größe n mehr Zeit als $\bar{t}_A(n) + s(n)$ benötigt. Folglich gibt es keine Instanzen im schlechtesten Fall mehr sondern nur noch Ausführungen im schlechtesten Fall. Wir definieren die mittlere erwartete Zeit $\bar{t}_B(n)$ durch:

$$\bar{t}_B(n) = \frac{\sum_{x \in X_n} t_B(x)}{\# X_n}$$

Es ist leicht erkennbar, daß $\bar{t}_B(n) \approx \bar{t}_A(n) + s(n)$ gilt. Ist $s(n)$ vernachlässigbar, so ist der Zuwachs in der mittleren Berechnungszeit klein. Dies ist eine Bedingung, die man beachten muß, wenn man solche Algorithmen anwenden will.

6.2.2 Nähestes-Paar-Algorithmus

- **Problem:** $x_1, ..., x_n$ seien n Punkte im k-dimensionalen Raum R^k. Wir möchten das näheste Paar (oder ein davon) x_i, x_j finden, sodaß gilt

  $$d(x_i, x_j) = min\ d(x_p, x_q) \qquad 1 \leq p < q \leq n,$$

  wobei d die gewöhnliche Abstandsfunktion aus R^k bezeichnet.

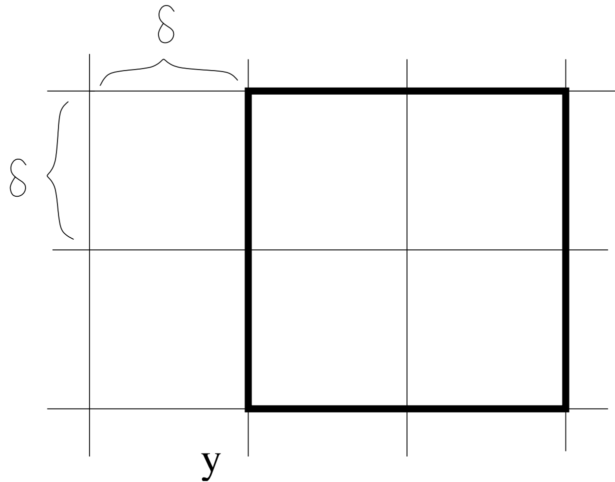
- "Brutale Gewalt"-Methode: Man wertet alle $\frac{n(n-1)}{2}$ relevanten, gegenseitigen Abstände aus und ermittelt daraus den minimalen Abstand. Dieser Algorithmus ist daher von der Ordnung $O(n^2)$.

- Deterministische Algorithmen sind von Ordnung $O(n \cdot log(n))$. Die Idee besteht darin, daß man Hüllen der Punkte $S = \{x_1, ..., x_n\}$ berücksichtigt und das näheste Paar innerhalb dieser Hüllen sucht.

- Die Schlüsselidee von Rabin besteht darin, daß man eine Teilmenge von Punkten zufällig auswählt. Er entwarf einen Algorithmus der Ordnung $O(n)$ mit einer sehr günstigen Konstante. Die Hauptschritte des Algorithmus lassen sich für $S \subset R^2$ (Die Erweiterung auf R^k ist klar) folgendermaßen beschreiben:

  1. Wähle zufällig $S_1 = \{x_{i_1}, ..., x_{i_m}\}, m = n^{\frac{2}{3}}$. Dies erfolgt dadurch, daß man die Indizes $i_1, ..., i_m$ aus $\{1, ..., n\}$ auswählt, wobei m die Kardinalität von S_1 und $m = c(S_1)$ ist.

2. Berechne $\delta(S_1) = min(x_p, x_q)$ für $x_p, x_q \in S_1$. Dies kann in einer Zeit von $O(n)$ ausgeführt werden: Wir iterieren einmal den gleichen Algorithmus für S_1, indem man $S_2 \subset S_1$ mit $c(S_2) = m^{\frac{2}{3}} = n^{\frac{4}{9}}$ zufällig auswählt. Um $\delta(S_2)$ zu finden, berücksichtigen wir einfach alle Paare $x_p, x_q \in S_2$. Die Anzahl solcher Paare ist kleiner als $(n^{\frac{4}{9}})^2 < n$, sodaß die erwartete Berechnungszeit für $\delta(S_1)$ nun $O(n)$ wird.
3. Konstruiere einen quadratischen Verband Γ mit Netzgröße (mesh size)$\delta = \delta(S_1)$ und berücksichtige die vier Verbände $\Gamma_1, ..., \Gamma_4$; leite aus Γ ab durch Verdoppelung der Netzgröße auf 2δ.



Lemma: Gilt $\delta(S) \leq \delta$, wobei δ die Netzgröße von Γ ist, so existiert ein Verbandpunkt y aus Γ, sodaß das näheste Paar im Quadrupel von Quadraten aus Γ direkt über und rechts von y liegt.

Dieses Lemma garantiert, daß das näheste Paar x_i, x_j aus S innerhalb eines gleichen Quadrats aus Γ_i liegt, $1 \leq i \leq 4$.

4. Finde für jedes Γ_i die Dekomposition

$$S = S_1^i \cup ..... \cup S_{k_i}^{(i)}, 1 \leq i \leq 4,$$

wobei $S_j^{(i)}$ die nicht leere Durchschnittmenge von S mit einem Quadrat aus Γ_i ist; folglich gilt $k_i \leq 4$. Im allgemeinen ist es nicht einfach, diese Dekomposition zu finden. Der Algorithmus arbeitet anders als auf Hashing-Techniken basierende Algorithmen. Man bezeichnet das Maß einer Dekomposition mit $N(D)$

$$N(D) = \sum_{i=1}^{k} \frac{n_i(n_i - 1)}{2}; n_i = c(S_i)$$

Das Ziel ist nun, einen Verband zu finden, für den $N(\Gamma) = O(n)$ gilt. Hierzu beweist man, daß der erwartete Wert von $N(\Gamma_i)$ kleiner als $c'n$ (d.h. $O(n)$) ist, wobei c' eine Konstante ist. Dieser Beweis erfordert Ergebnisse anderer Beweise, die hier nicht behandelt werden.

5. Für jedes $x_p, x_q \in S_j^{(i)}$ berechne $d(x_p, x_q)$. Das näheste Paar ist unter diesen Paaren zu finden, sodaß gilt

$$\delta(S) = min\ d(x_p, x_q); \qquad x_p, x_q \in S_j^{(i)}, 1 \leq i \leq 4, 1 \leq j \leq 4$$

Die Anzahl der Abstände wird in $O(n)$ berechnet und diese ist auch die Anzahl der benötigten Vergleiche zur Berechnung des Minimums.

Literaturhinweise:

1. Deterministic Algorithm : Yuval, *"Finding Nearest Neighbours"* , Information Processing Letters, 1976.

2. Probabilistic Algorithm : Rabin, in *"Algorithms and Complexity, Recent Results and New Directions"*, J.F. Traub (Editor), Academic Press, 1976, pp. 21 — 40.

6.3 Monte Carlo Algorithmen

Primzahltest

Es ist ein Problem von theoretischem Interesse in der Zahlentheorie und von praktischem Interesse in Domänen wie der Kryptologie, zu testen, ob eine Zahl prim ist oder nicht. Dieses Problem ist viel einfacher als das Faktorisierungsproblem von ganzen Zahlen in ihre Primfaktoren. Es gibt keinen deterministischen Algorithmus, der dieses Problem in einer vertretbaren Zeit löst, wenn die Länge der zu testenden Zahl größer als einige hundert Dezimalstellen ist.

Zur Zeit ist es möglich, Zahlen bis zu 700 Ziffern mit Hilfe des verteilten Programmierens (10 SUN Workstations, Berechnungszeit: Eine Woche) festzustellen, ob sie prim sind. Mit Hilfe von besseren Methoden (der Primtest von Atkin) wurde ein Weltrekord erzielt, um zu zeigen, daß $(2^{3539}+1)/3$ prim ist. Diese Zahl hat 1065 Ziffern. Dafür benötigt man 12 SUN Workstations und eine Berechnungszeit von anderthalb Monaten.

Rabin entwarf als einer der ersten Wissenschaftler einen sehr einfachen und effizienten probabilistischen Algorithmus für dieses Problem. (*Journal of Number Theory*, vol 12, s. 123 — 138, 1980).

Der Algorithmus stellt fest, ob eine Zahl n prim ist. In diesem Algorithmus werden m Zahlen $1 \leq b_1, ..., b_m < n$ zufällig ausgewählt. Falls für eine gegebene Zahl n und irgendein $\epsilon > 0 \quad log_2(\frac{1}{\epsilon}) \leq m$ gilt, dann wird der Algorithmus die korrekte Antwort mit einer Wahrscheinlichkeit größer als $(1 - \epsilon)$ liefern.

Die Grundidee läßt sich folgendermaßen erklären: Sei $W_n(b)$ die folgende Bedingung für eine ganze Zahl b:

(i) $1 \leq b < n$

(ii) (a) $b^{n-1} \not\equiv 1 \; mod \; n$, oder

(b) $\exists i$, so daß $2^i \mid (n-1)$ und $1 < ggT(b^{\frac{(n-1)}{2^i}} - 1, n) < n$ gilt.

[Wir bezeichnen: $(n-1)/2^i = m$]

Eine ganze Zahl b, die diese Bedingung erfüllt, wird Zeuge (witness) für die Teilbarkeit von n genannt. Wenn (ii-a) gilt, dann ist der Fermatsche Satz verletzt. (ii-b) bedeutet, daß n einen echten Teiler hat. Folglich: Wenn $W_n(b)$ gilt, ist n teilbar, d.h. n ist keine Primzahl. Es stellt sich heraus, daß es aufgrund des folgenden Theorems viele Zeugen gibt, wenn n teilbar ist.

Theorem: Wenn $n > 4$ teilbar ist, dann gilt

$$\frac{3(n-1)}{4} \leq c(\{b \mid 1 \leq b < n, W_n(b) gilt\}),$$

wobei $c(S)$ die Anzahl der Elemente der Menge S ist (Kardinalität). Da $b < n$, bedeutet dieses Theorem, daß nicht mehr als $\frac{1}{4}$ der Zahlen $1 \leq b < n$ keine Zeugen sind (Bemerkung: $\frac{3}{4}$ scheint ein optimaler Faktor zu sein). Die Idee von Rabin liegt darin, daß eine probabilistische Methode den Test beschleunigen würde, da viele Zeugen für Teilbarkeiten existieren.

Der Algorithmus läßt sich fortsetzen, indem man m Zahlen $1 \leq b_1, ..., b_m < n$ zufällig auswählt und für jedes b_i testet, ob $W_n(b_i)$ gilt. Aufgrund des obigen Theorems ist die Wahrscheinlichkeit für eine falsche Antwort kleiner als $\frac{1}{4^m}$. Da die Möglichkeit besteht, eine falsche Antwort zu bekommen, ist es nicht

möglich festzustellen, ob der Algorithmus Primzahlen entdeckt. Man sagt auch, daß der Algorithmus "strenge Pseudo-Primzahlen" identifiziert. In der Praxis kann $m = 5$ gewählt werden. Der Algorithmus zum Testen, ob $W_n(b)$ gilt, ist einfach:

Eingabe:	n, ungerade ganze Zahl > 1
Ausgabe:	$b = \pm 1$, falls entschieden ist, daß n prim ist
	$b = 0$, falls entschieden ist, daß n teilbar ist
Schritt 1:	Wähle zufällig a aus: $1 \leq a < n$
Schritt 2:	Faktorisiere $(n-1)$ zu $2^l m$
	$n - 1 = 2^l m$, m ungerade
Schritt 3:	(Teste) $b = a^m \bmod n$, $i = 1$
	while $b \neq -1$ **and** $b \neq 1$ **and** $i < l$
	do { $b = b^2 \bmod a$, $i = i + 1$}
Schritt 4:	(Entscheide) **if** $b = 1$ **or** $b = -1$ **then** n prime
	else n composite $(b = 0)$.

Dieser Algorithmus benötigt $m(2 + l)log_2(n)$ Schritte. Daher ist er sehr effizient, obwohl er sehr einfach ist. Die Beweise werden nicht ausgeführt , sind jedoch nicht schwierig. Sie basieren auf sehr einfachen Ergebnissen aus der Zahlentheorie. Vollständigkeitshalber kann man erwähnen, daß einer der bekanntesten Algorithmen zum Testen, ob eine Zahl prim ist, aus einige tausend Zeilen (Pascal-Programm) und aus höherem, nichttrivialen mathematischen Wissen besteht (selbstverständlich außerhalb des Rahmens dieser Vorlesung).

6.4 Las Vegas Algorithmen

Wir betrachten wieder ein Beispiel nach Rabin (*SIAM J. Computing*, vol. 9, S. 273, 1980). In Wirklichkeit gibt Rabin Algorithmen zur Berechnung der Nullstellen und zur Faktorisierung von Polynomen, die über endlichen Körper $GF(p^n)$ definiert sind, wobei p eine Primzahl und n eine ganze Zahl ist. (GF bedeutet Galois-Körper; $GF(p)$ wird oft als Z_p geschrieben). Diese Probleme sind sehr wichtig in der Computer Algebra und in der Kodierungstheorie. Sie werden in der Vorlesung Computer Algebra ausführlich behandelt. Wir beschränken uns auf das Problem des Auffindens von irreduziblen Polynomen in endlichen Körpern. Das Prinzip des Algorithmus ist einfach.

Algorithmus Irreduzibles Polynom

Eingabe: Primzahl p und ganze Zahl n

repeat 1. Generiere ein zufälliges Polynom $g \in GF(p)[x]$ der Ordnung n

2. Teste die Irreduzibilität

until test is true

Dieser Algorithmus könnte nicht terminieren (d.h. er ist nicht endlich), da unendlich viele Folgen von zufälligen Polynomen über $GF(p)$ existieren, die alle reduzibel sind. Die Wahrscheinlichkeit geht jedoch mit wachsender Anzahl von Versuchen gegen 0. Die maximale Zeit könnte $t_{max} = \infty$ sein. In Wirklichkeit beweist man, daß die mittlere zu erwartende Zeit $\bar{t} = O(n^3 log(p))$ ist. Der Test auf Irreduzibilität basiert auf dem folgenden Theorem:

Theorem: Seien $l_1, ..., l_k$ alle Primteiler von n. Bezeichne $\frac{n}{l_i} = m_i$. Ein Polynom $g(x) \in GF(p)[x]$ vom Grad n ist irreduzibel in $GF(p)$ g.d., w.

(a) $g(x) \mid (x^{p^n} - x)$

(b) $ggT(g(x), x^{p^{m_i}} - x) = 1, \quad 1 \leq i \leq k$

Beweis : Angenommen $g(x)$ ist irreduzibel über $GF(p)$, dann liegt jede Nullstelle α von $g(x) = 0$ in $GF(p^n)$. Daraus folgt, daß $(\alpha^{p^n} - \alpha) = 0$ gilt und $(x - \alpha)$ teilt $x^{p^n} - x$. Da $g(x)$ keine vielfache Nullstellen hat, folgt (a).

Da $g(x)$ irreduzibel und vom Grad n ist, hat es keine Nullstellen im Körper $GF(p^m) \quad m < n$. Dies impliziert direkt (b).

Angenommen es gelten (a) und (b). Aus (a) folgt, daß alle Nullstellen von $g(x) = 0$ aus $E = GF(p^n)$ sind. Angenommen g hat einen irreduziblen Faktor $g_1(x)$ vom Grad $m < n$. Die Nullstellen von $g_1(x)$ liegen in $GF(p^m)$. Dieser Körper wird über $GF(p)$ durch Adjunktion irgendeiner dieser Nullstellen erzeugt. Folglich gilt $GF(p^m) \subseteq E$ und $m \mid n$. Ferner gilt $m \mid m_i$ für einen der maximalen Teiler m_i von n und alle Nullstellen von $g_i(x)$ liegen in $GF(p^{m_i})$. Dann ist $ggT(g(x), x^{p^{m_i}} - x)$ durch $g_1(x)$ teilbar. Ein Widerspruch zu (b). Daher muß $g(x)$ irreduzibel sein. □

In der Praxis wird der zweite Schritt des Algorithmus in zwei Teile zerlegt:

(2.1) Falls $g(x) \mid (x^{p^n} - x)$, dann ist Test1 erfolgreich

(2.2) Falls $ggT(g(x), x^{p^{m_i}} - x) = 1$ für alle m_i, dann ist Test2 erfolgreich.

Kapitel 7

Vorbestimmung und Vorberechnung

- Falls wir Instanzen des gleichen Problems zu lösen haben, ist es manchmal sinnvoll, Zeit zur Berechnung von Hilfsergebnissen zu investieren. Sie können danach benutzt werden, um die Lösung jeder Instanz zu beschleunigen. Dies nennt man *Vorbestimmung* [1].
- Auch wenn es nur eine zu lösende Instanz gibt, kann eine *Vorberechnung* [2] von Hilfstabellen (oder Ergebnissen) zu einem effizienteren Algorithmus führen.

7.1 Vorbestimmung

7.1.1 Einführung

Sei I eine Menge von Instanzen eines gegebenen Problems. Angenommen jede Instanz $i \in I$ kann in zwei Komponenten $j \in J$ und $k \in K$ aufgeteilt werden, d.h. $I \subseteq J \times K$. Ein "Vorbestimmungsalgorithmus" für dieses Problem ist ein Algorithmus A, der irgendein Element j aus J als Eingabe akzeptiert und einen neuen Algorithmus B_j als Ausgabe liefert. Der Algorithmus B_j muß folgende Bedingung erfüllen: Wenn $k \in K$ und $< j, k > \in I$ gilt, dann liefert die Anwendung von B_j auf k die Lösung zu der Instanz $< j, k >$ des Originalproblems.

[1] preconditioning
[2] precomputation

Beispiel: Sei J eine Menge von Grammatiken für eine Familie von Programmiersprachen. J kann zum Beispiel eine Menge von Grammatiken in Backus-Naur-Form für Sprachen wie Algol, Pascal, Simula,.... sein. Sei K eine Menge von Programmen. Das allgemeine Problem ist nun die Feststellung, daß ein gegebenes Problem bezüglich einer gegebenen Sprache syntaktisch korrekt ist. In diesem Fall ist I die Menge von Instanzen des Typs

"Ist $k \in K$ ein gültiges Programm in der Sprache, die durch die Grammatik $j \in J$ definiert ist ?"

Ein möglicher Vorbestimmungsalgorithmus für dieses Beispiel ist ein Compiler-Generator: Angewendet auf die Grammatik $j \in J$, generiert er einen Compiler B_j für die entsprechende Sprache. Danach, um festzustellen, ob $k \in K$ ein Programm in der Sprache j ist, wenden wir einfach den Compiler B_j auf k an.

Es seien $a(j)$ = benötigte Zeit, um B_j für ein gegebenes j zu produzieren,
$b_j(k)$ = benötigte Zeit, um B_j auf k anzuwenden,
$t(j,k)$ = benötigte Zeit, um $< j,k >$ direkt zu lösen.

Normalerweise gilt $b_j(k) \leq t(j,k) \leq a(j) + b_j(k)$. Es ist Zeitverschwendung, Vorbestimmung zu benutzen, wenn $b_j(k) > t(j,k)$ gilt. Andererseits besteht eine Möglichkeit zur Lösung von $< j,k >$ darin, daß man B_j von j produziert und B_j auf k anwendet.

Vorbestimmung kann in zwei Situationen sinnvoll sein

(a) Man muß irgendeine Instanz $i \in I$ sehr schnell lösen, um eine schnelle Antwort in Echtzeitanwendungen zu erhalten. In diesem Fall ist es manchmal unpraktikabel, die $\#I$ Lösungen zu allen relevanten Instanzen im voraus zu berechnen und zu speichern. Auf der anderen Seite könnte es möglich sein, $\#J$ vorbestimmte Algorithmen im voraus zu berechnen und zu speichern. Zwei prototypische Beispiele sind:

  (i) Einen laufenden Kern-Reaktor zu stoppen.

  (ii) Die von einem Studenten verbrauchte Zeit zur Vorbereitung einer Prüfung.

(b) Wir müssen eine Reihe von Instanzen $< j,k_1 >, < j,k_2 >, ..., < j,k_n >$ mit dem gleichen j lösen. In diesem Fall ist die zur Lösung aller Instanzen benötigte Zeit ohne Vorbestimmung:

$$t_1 = \sum_{i=1}^{n} t(j, k_i)$$

und mit Vorbestimmung:

$$t_2 = a(j) + \sum_{i=1}^{n} b_j(k_i)$$

Wenn n groß genug ist, passiert es oft, daß t_2 viel kleiner als t_1 ist.

Beispiel: Sei J eine Menge von Schlüsselwörtermengen:

$$J = \{\{\text{if, then, else, endif}\}, \{\text{for, to, by}\}, ...\}$$

Sei K eine Menge von Schlüsselwörtern : $K = \{\text{begin, function, ...}\}$. Wir müssen eine große Anzahl von Instanzen des nachfolgenden Typs lösen:

"Gehört das Schlüsselwort $k \in K$ der Menge $j \in J$ an?"

Wenn wir jede Instanz direkt lösen, erhalten wir

$$t(j, k) \in \Theta(n_j)$$

im schlechtesten Fall, wobei n_j die Anzahl der Elemente in der Menge j ist. Wenn wir aber zunächst mit dem Sortieren von j beginnen (dies ist Vorbestimmung), dann können wir nachträglich $< j, k >$ mit einem binären Suchalgorithmus lösen:

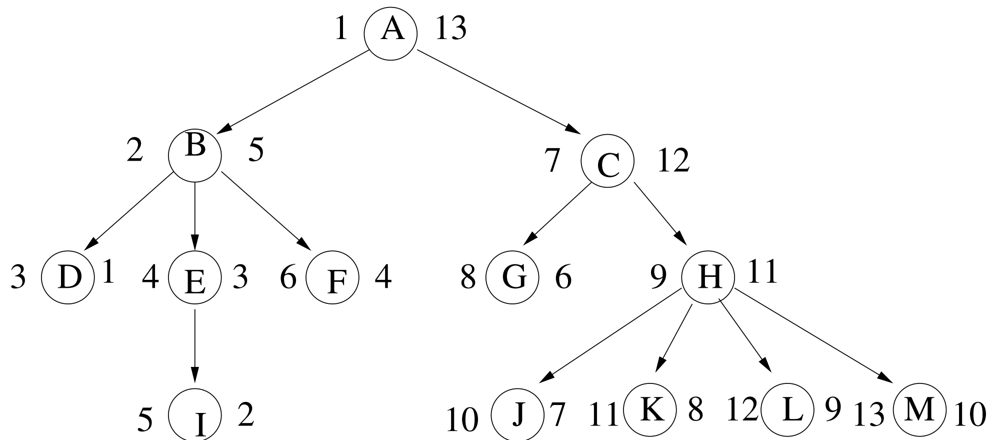
$$a(j) \in \Theta(n_j log(n_j)) \text{ für das Sortieren}$$
$$b_j(k) \in \Theta(log(n_j)) \text{ für das Suchen}$$

Falls es viele zu lösende Instanzen für das gleiche j gibt, dann ist die zweite Technik vorzuziehen.

7.1.2 Vorgänger in einem Wurzelaum

Sei J die Menge aller Bäume und sei K die Menge von Knotenpaaren $< v, w >$. Für ein gegebenes Paar $k =< v, w >$ und einen gegebenen Baum j möchten wir feststellen, ob Knoten v der Vorgänger vom Knoten w im Baum j ist. (Per Definition ist jeder Knoten sein eigener Vorgänger). Enthält der Baum j n Knoten, so benötigt eine direkte Lösung dieser Instanz eine Zeit in $\Omega(n)$ im schlechtesten Fall.

Es ist immer noch möglich, Vorbestimmung für den Baum in einer Zeit von $\Theta(n)$ durchzuführen, so daß wir nachträglich eine bestimmte Instanz in einer Zeit von $\Theta(1)$ lösen können. Um diese Methode zu beschreiben, betrachten wir den Baum im folgenden Bild.



Er enthält 13 Knoten. Um den Baum vorzubestimmen, durchlaufen wir ihn zunächst in Preorder und dann in Postorder, wir numerieren die Knoten sequentiell, wie sie durchlaufen werden. Für einen Knoten v seien prenum$[v]$ und postnum$[v]$ die dem Knoten während der Preorder- beziehungsweise Postorder- Durchläufe zugewiesenen Zahlen. Sie erscheinen links beziehungsweise rechts vom Knoten.

Es seien v und w zwei Knoten aus dem Baum. In Preorder numerieren wir zunächst einen Knoten und dann die Teilbäume von links nach rechts. Folglich gilt

$$\text{prenum}[v] \leq \text{prenum}[w] \Leftrightarrow v \text{ ist ein Vorgänger von w } \textbf{or } v \text{ ist links von } w \text{ in Baum}$$

In Postorder numerieren wir zunächst die Teilbäume eines Knotens von links nach rechts und dann numerieren wir den Knoten. Folglich gilt

postnum$[v] \geq$ postnum$[w] \Leftrightarrow v$ ist ein Vorgänger von w **or** v ist rechts von w im Baum

Daraus folgt, daß:

prenum$[v] \leq$ prenum$[w]$ **and** postnum$[v] \geq$ postnum$[w] \Leftrightarrow v$ ist ein Vorgänger von w.

Wenn die Werte von prenum und postnum einmal in $\Theta(n)$ berechnet worden sind, so kann die Bedingung in $\Theta(1)$ geprüft werden.

7.1.3 Wiederholte Auswertung eines Polynoms

Sei J die Menge der Polynome in einer Variablen x und sei K die Menge von Werten dieser Variable. Das Problem besteht darin, daß man ein gegebenes Polynom an einer gegebenen Stelle auswertet. Einfachheitshalber beschränken wir uns auf ganzzahlige Koeffizienten und normierte Polynome (der führende Koeffizient ist 1 (monic)) vom Grad $n = 2^k - 1$ für eine ganze Zahl $k \geq 1$. Die Messung der Effizienz eines Algorithmus ist durch die Anzahl der auszuführenden, ganzzahligen Multiplikationen gegeben.

Beispiel: $p(x) = x^7 - 5x^6 + 4x^5 - 13x^4 + 3x^3 - 10x^2 + 5x - 17$

Naive Methode: Berechne zunächst die Reihe von Werten $x^2, x^3, ..., x^7$, von denen man $5x, -10x^2, ...$ und zum Schluß $p(x)$ berechnen kann. Diese Methode benötigt 12 Multiplikationen und 7 Additionen (wir zählen eine Subtraktion als eine Addition). Sie kann leicht verbessert werden. Wenn wir $p(x)$ als

$$p(x) = ((((((x-5)x+4)x-13)x+3)x-10)x+5)x-17$$

auswerten, brauchen wir nur 6 Multiplikationen und 7 Additionen. Noch besser ist es, p(x) folgendermaßen auszuwerten

$$p(x) = (x^4+2)[(x^2+3)(x-5)+(x+2)] + [(x^2-4)x+(x+9)]$$

Dabei brauchen wir nur 5 Multiplikationen (2 zur Berechnung von x^2 und x^4) und 9 Additionen.

(Bemerkung: Ein typisches normiertes Polynom vom Grad 7 würde 5 Multiplikationen aber 10 Additionen benötigen.)

Wenn $p(x)$ ein normiertes Polynom vom Grad $n = 2^k - 1$ ist, drücken wir zunächst das Polynom in der nachfolgenden Form aus:

$$p(x) = (x^{\frac{(n+1)}{2}} + a)q(x) + r(x)$$

wobei a eine Konstante, $q(x)$ und $r(x)$ normierte Polynome vom Grad $(2^{k-1} - 1)$ ist. Danach wenden wir die gleiche Prozedur rekursiv auf $q(x)$ und $r(x)$ an. Schließlich wird $p(x)$ als Polynom der Form $(x^i + c)$, wobei i eine Zweier-Potenz ist, ausgedrückt.

Aus dem vorherigen Beispiel erhält man zunächst:

$$p(x) = (x^4 + a)(x^3 + q_2x^2 + q_1x + q_0) + (x^3 + r_2x^2 + r_1x + r_0)$$

Vergleicht man die Koeffizienten, so erhält man: $q_2 = -5, q_1 = 4, q_0 = -13, a = 2, r_2 = 0, r_1 = -3, r_0 = 9$.

Folglich: $p(x) = (x^4 + 2)(x^3 - 5x^2 + 4x - 13) + (x^3 - 3x + 9)$

Ähnlich: $x^3 - 5x^2 + 4x - 13 = (x^2 + 3)(x - 5) + (x - 2)$
$x^3 - 3x + 9 = (x^2 - 4)x + (x + 9)$

Daraus erhält man das im Beispiel gegebene Ergebnis.

Analyse der Methode: Sei $M(k)$ die Anzahl der benötigten Multiplikationen zur Auswertung von $p(x)$, wobei $p(x)$ ein vorbestimmtes, normiertes Polynom vom Grad $n = 2^k - 1$ ist. Sei $\hat{M}(k) = M(k) - k + 1$ die Anzahl der benötigten Multiplikationen, wenn wir die Operationen zur Berechnung von $x^2, ..., x^{\frac{(n+1)}{2}}$ nicht dazu zählen. Wir erhalten die Rekurrenzrelation:

$$\hat{M}(k) = \begin{cases} 0 & k = 1 \\ 2\hat{M}(k-1) + 1 & k \geq 2 \end{cases}$$

Folglich gilt $\hat{M}(k) = 2^{k-1} - 1$ für $k \geq 1$ und daraus folgt: $M(k) = 2^{k-1} + k - 2$. Mit anderen Worten wir brauchen nur

$$\left\lceil \frac{(n-3)}{2} + log(n+1) \right\rceil$$

Multiplikationen, um ein vorbestimmtes Polynom vom Grad $n = 2^k - 1$ auszuwerten.

Bemerkung: Diese Methode wurde von Belaga (im “Problemi Kibernitiki”, vol 5, pp 7 — 15, 1961) vorgeschlagen.

7.2 Vorberechnung für Zeichenreihe-Suchprobleme

Problem: Gegeben seien $S = s_1 s_2 .....s_n$ eine aus n Zeichen bestehende “Zielzeichenreihe” und $P = p_1 p_2 .....p_m$ ein aus m Zeichen bestehendes Muster. Wir möchten wissen, ob P eine Teilzeichenreihe von S ist, und gegebenenfalls, wo P in S vorkommt. Ohne Beschränkung der Allgemeinheit sei $n \geq m$.

In der Analyse des Algorithmus betrachten wir die Anzahl der Vergleiche zwischen Zeichenpaaren als ein Effizienzmaß.

Ein naiver Algorithmus ist selbstverständlich. Er liefert r, falls das erste Vorkommen von P in S in der Position r beginnt (r ist die kleinste ganze Zahl, so daß $s_{r+i-1} = p_i$, $i = 1, 2, 3, ..., m$ gilt), und 0, falls P keine Teilzeichenreihe von S ist.

```
for i ← 0 to n − m do
ok ← true
j ← 1
while ok and j ≤ m do
        if p[j] ≠ s[i + j] then ok ← false
                else j ← j + 1
if ok then return i + 1
return 0
```

Jede Position in S wird geprüft. Im schlechtesten Fall macht der Algorithmus m Vergleiche bei jeder Position. Die Gesamtzahl von Vergleichen ist daher in $\Omega(m(n-m))$ und in $\Omega(mn)$, falls n viel größer als m ist. Tatsächlich ist eine bessere Ausführung möglich, indem man verschiedene Methoden anwendet.

1. Signaturen

Angenommen S kann in einer natürlichen Weise in Teilzeichenreihen zerlegt werden: $S = S_1 S_2 .....S_n$, und falls P in S vorkommt, muß P vollkommen in einer der Teilzeichenreihen vorkommen. (Zum Beispiel: S_i sind die Zeilen in einer Textdatei und man sucht die Zeilen, die P enthalten.)

Die Grundidee besteht darin, daß man eine Boolesche Funktion $T(P, S_i)$ benutzt, die sehr schnell berechnet werden kann, um einen vorläufigen Test

durchzuführen. Falls $T(P, S_i)$ falsch ist, dann gilt $P \notin S_i$, sonst ist es möglich, daß P eine Teilzeichenreihe von S_i sein könnte. Dies muß geprüft werden, indem man zum Beispiel den naiven Algorithmus anwendet.

Mit Signaturen findet man ein einfaches Verfahren, um einen solchen Algorithmus zu implementieren. Angenommen das verwendete Zeichenvorrat für die Zeichenreihen S und P ist $\{a, b, c, ..., y, z, other\}$, wobei "*other*" allen möglichen nicht-alphabetischen Zeichen entspricht. Angenommen wir arbeiten auf einem 32-Bit Rechner. Eine mögliche Definition einer Signatur ist:

Def.:

(i) Definiere val("*a*") = 0, val("*b*") = 1, ..., val("*z*") = 25, val("*other*") = 26

(ii) Falls c_1 und c_2 Zeichen sind, definiere:

$$B(c_1, c_2) = (27val(c_1) + val(c_2))\ mod\ 32$$

(iii) Definiere die Signatur sig(C) einer Zeichenreihe $C = c_1c_2.....c_r$ als ein 32-Bit-Wort, wobei die Bits mit den Nummern $B(c_1, c_2), B(c_2, c_3), ..., B(c_{r-1}, c_r)$ auf 1 und sonst auf 0 gesetzt sind.

Beispiel: $C =$ "computers".

$B(\text{"c"},\text{"o"}) = (27 \times 2 + 14)\ mod\ 32 = 4$
$B(\text{"o"},\text{"m"}) = (27 \times 14 + 12)\ mod\ 32 = 6$ ....
$B(\text{"r"},\text{"s"}) = (27 \times 17 + 18)\ mod\ 32 = 29$

Wenn die Bit des Wortes von links nach rechts mit 0 bis 31 numeriert werden, ist die Signatur dieser Zeichenreihe :

0000 1110 0100 0001 0001 0000 0000 0100

Nur 7 Bit sind auf 1 gesetzt, da $B(\text{"e"},\text{"r"}) = B(\text{"r"},\text{"s"}) = 29$ ist.

Wir berechnen die Signatur für jedes S_i und für das Muster P. Wenn S_i das Muster P enthält, sind alle Bit, die in der Signatur von P auf 1 gesetzt sind, auch in der Signatur S_i auf 1 gesetzt. Dies gibt uns die Funktion $T(P, S_i)$, die wir brauchen:

$$T(P, S_i) = [(Sig(P) \text{ and } Sig(S_i)) = Sig(P)]$$

"*and*" entspricht der bitweisen Konjunktion von zwei ganzen Wörtern. T kann sehr schnell berechnet werden, wenn alle Signaturen bereits berechnet worden sind. Dies ist ein anderes Beispiel sowohl von Vorbestimmung als auch von Vorberechnung. Die Berechnung von Signaturen für S benötigt eine Zeit in $O(n)$. Für jedes gegebene Muster P, brauchen wir eine zusätzliche Zeit in $O(m)$, um die entsprechende Signatur zu berechnen. Aber von da an hoffen wir, daß der vorläufige Test die Suche für P beschleunigen wird.

2. Der Knuth-Morris-Pratt-Algorithmus

Wir geben nur eine informelle Beschreibung des KMP-Algorithmus, der die Vorkommen von P in S in einer Zeit in $O(n)$ bestimmt. Die Einzelheiten findet man in (KMP, SIAM J. on Computing, Vol. 6, S. 240 — 267, 1977). Der Algorithmus wird anhand eines Beispiels beschrieben.

Beispiel:

Um P in S zu finden, lassen wir P unter S von links nach rechts schieben und prüfen die übereinander stehenden Zeichen. Anfangs erhalten wir

1	*S*	*b*	*a*	*b*	*c*	*b*	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*c*	*a*	*b*	*c*
	P	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*c*	*a*	*b*																
		↑																									

Wir prüfen die Zeichen von P von links nach rechts. Die Pfeile zeigen die ausgeführten Vergleiche, bevor wir ein Zeichen, das nicht übereinstimmt, finden. In diesem Fall gibt es einen Vergleich. Nach diesem Fehler versuchen wir

2	*S*	*b*	*a*	*b*	*c*	*b*	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*b*	*c*	*a*	*c*	*a*	*b*	*c*
	P		*a*	*b*	*c*	*a*	*b*	*c*	*a*	*c*	*a*	*b*															
			↑	↑	↑	↑																					

Dieses Mal stimmen die ersten 3 Zeichen mit den darüber stehenden Zeichen überein, jedoch nicht das vierte Zeichen. Was wir bisher gemacht haben, ist nichts andere als der naive Algorithmus. Jedoch wissen wir, daß die letzten 4 geprüften Zeichen in S $abcx$ sind, wobei $x \neq$ "a" ist. Ohne weitere Vergleiche mit S durchzuführen, können wir schliessen, daß es sinnlos ist, P um eine, zwei oder drei Zeichen zu schieben: ein solches Argument kann nicht richtig sein. Nun versuchen wir P um 4 Zeichen zu schieben:

```
3 S  b a b c b a b c a b c a a b c a b c a b c a c a b c
  P            a b c a b c a c a b
               ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
```

Verfolgt man diesen Mismatch, so wissen wir, daß die 8 untersuchten Zeichen in S $abcabcax$ sind, wobei $x \neq$ "c" ist. Es kann nicht richtig sein, wenn man P um eine oder zwei Stellen verschiebt. Es funktioniert jedoch, wenn man P um 3 Stellen verschiebt.

```
4 S  b a b c b a b c a b c a a b c a b c a b c a c a b c
  P                  a b c a b c a c a b
                             ↑
```

Es ist unnötig, die ersten 4 Zeichen von P nochmals zu prüfen: Um sicher zu sein, wählen wir die Bewegung von P so, daß die Zeichen notwendigerweise übereinstimmen. Es genügt damit zu beginnen, daß man an der jetzigen Position des Zeigers prüft. Wir haben wieder einen Mismatch. Dieses Mal kann man P um 4 Stellen verschieben. (Eine 3-Stellen-Verschiebung genügt nicht: wir wissen, daß die letzten geprüften Zeichen in S ax sind, wobei x ungleich "b" ist)

```
5 S  b a b c b a b c a b c a a b c a b c a b c a c a b c
  P                          a b c a b c a c a b
                             ↑ ↑ ↑ ↑ ↑ ↑ ↑ ↑
```

Wir haben wieder einen Mismatch und dieses Mal ist eine 3-Stellen-Verschiebung notwendig

```
6 S  b a b c b a b c a b c a a b c a b c a b c a c a b c
  P                                a b c a b c a c a b
                                           ↑ ↑ ↑ ↑ ↑ ↑
```

Wir vervollständigen die Verifikation, indem wir an der Stelle des Zeigers beginnen. Die Übereinstimmung ist nun vollständig.

Um den in diesem Beispiel geschilderten Algorithmus zu implementieren, brauchen wir eine Reihung next$[1..m]$, die uns informiert, was zu tun ist, wenn

ein Mismatch an der Position j in dem Muster vorkommt. Falls next$[j] = 0$ gilt, ist es sinnlos, weitere Zeichen des Musters mit der Zielzeichenreihe an der jetzigen Position zu vergleichen. Hierzu müssen wir P unter das erste Zeichen von S, das noch nicht berücksichtigt worden ist, schieben; und wir beginnen wieder am Anfang von P zu prüfen. Wenn next$[j] = i > 0$ gilt, sollen wir das i-te Zeichen von P auf das jetzige Zeichen von S richten und wieder an dieser Position prüfen. In beiden Fällen verschieben wir P um ($j-$next$[j]$) Zeichen nach rechts bezüglich S.

Aus dem vorherigen Beispiel haben wir:

j	1	2	3	4	5	6	7	8	9	10
$p[j]$	a	b	c	a	b	c	a	c	a	b
next$[j]$	0	1	1	0	1	1	0	5	0	1

$next[j]$ ist leicht aus dem vorherigen Beispiel zu berechnen. Betrachten wir folgende zwei Fälle:

(i) $j = 5$ (Bild 4), $p[j] = b$, wie der Pfeil zeigt. Bei der nächsten Bewegung beginnt der Vergleich mit dem ersten linken Zeichen von P (linker Pfeil im Bild 5). Folglich gilt $next[5] = 1$

(ii) Im Bild 5 ist der rechte Pfeil unter dem achten Zeichen. Folglich gilt $j = 8, p[j] = c$. Die nächste Bewegung beginnt mit einem Vergleich des fünften Zeichens von P (linker Pfeil im Bild 6). Folglich gilt $next[j] = 5$.

Wenn man diese Reihung einmal berechnet hat, sieht der Algorithmus zum Auffinden von P in S folgendermaßen aus:

```
Algorithmus KMP
    j, k ← 1
    while j ≤ m and k ≤ n do
        while j > 0 and s[k] ≠ p[j] do
            j ←next[j]
        k ← k + 1
        j ← j + 1
    if j > m then return k − m
        else return 0
```

Dieser Algorithmus liefert entweder die Position des ersten Vorkommens von P in S oder 0, falls P keine Teilzeichenreihe von S ist.

Analyse: Nach jedem Vergleich von 2 Zeichen bewegen wir entweder den Zeiger (Pfeil im Beispiel oder k im Algorithmus) oder das Muster P. Beide können höchstens n mal bewegt werden. Die Laufzeit für den Algorithmus ist daher in $O(n)$. Vorberechnung von next$[j]$ kann in $O(m)$ durchgeführt werden und ist vernachlässigbar, da $n >> m$ ist. Daher ist die gesamte Zeit in $O(n)$.

Bem.: Dies ist Vorberechnung für alle möglichen Fälle, die man betrachten kann. Zusätzlich ist dies auch Vorbestimmung, wenn das gleiche Muster in mehreren verschiedenen Zielzeichenreihen gesucht wird.

3. Der Boyer-Moore-Algorithmus

Literatur: BM, Communication ACM, vol. 20, 762 — 772 (1977). Der BM-Algorithmus findet die Vorkommen von P in S in $O(n)$ im schlechtesten Fall. Im Gegensatz zu dem KMP-Algorithmus, der mindestens n Vergleiche braucht, braucht BM weniger als n Vergleiche. Daher ist BM effizienter, wenn m (Anzahl der Zeichen in P) wächst. Im besten Fall ist BM in $O(m + \frac{n}{m})$.

Wie für KMP schieben wir P unter S von links nach rechts. Dieses Mal werden jedoch die Zeichen von P von rechts nach links nach jeder Bewegung des Musters P geprüft. Wir benutzen zwei Regeln, um zu entscheiden, wie weit wir P nach einem Mismatch bewegen sollen.

(i) Wenn es nach der Bewegung von P einen Mismatch gibt: Sei c das über $p[m]$ stehende Zeichen, dann gilt $c \neq p[m]$. Falls c anderswo im Muster vorkommt, verschieben wir P so, daß das letzte Vorkommen von c in P mit dem c in S auf der gleichen Stelle steht. Erscheint c nicht in P, können wir so anordnen, daß P unmittelbar nach dem Vorkommen von c in S angeordnet wird.

(ii) Wenn eine Anzahl von Zeichen am Ende von P mit den Zeichen in S übereinstimmen, dann (wie in KMP) benutzen wir dieses Teilwissen von S, um P in eine neue Position, die mit dieser Information kompatibel ist, zu schieben.

Beispiel: S = “This is a delicate topic”; P = “cat”

```
1    S  T h i s   i s   a   d e l i c a t e   t o p i c
     P  c a t
            ↑
```

Da “i” nicht in P vorkommt, schieben wir P zu der rechten Seite des Pfeils

```
2    S   T h i s   i s   a   d e l i c a t e   t o p i c
     P         c a t
                   ↑
```

Die gleiche Situation wie im Schritt 1

```
3    S   T h i s   i s   a   d e l i c a t e   t o p i c
     P               c a t
                         ↑
```

“a”ist in P: Wir ordnen “a” in S und P an:

```
4    S   T h i s   i s   a   d e l i c a t e   t o p i c
     P                 c a t
                           ↑
```

Es gibt einen Mismatch und “ ” kommt in P nicht vor. Nach einem weiteren Mismatch (Schritt 5) erhalten wir:

```
6    S   T h i s   i s   a   d e l i c a t e   t o p i c
     P                           c a t
                                     ↑
```

Ordnen wir beide a an, so bekommen wir die Lösung

```
7    S   T h i s   i s   a   d e l i c a t e   t o p i c
     P                             c a t
                                   ↑   ↑
```

Wir haben 9 Vergleiche gemacht. $n = 24, m = 3$. Trotzdem ist dies besser als $m + \frac{n}{m} = 3 + 8 = 11$.

Übung: Man zeige, daß KMP für das gegebene Beispiel mehr als 9 Vergleiche benötigt.

Um den Algorithmus zu implementieren, braucht man zwei Reihungen $d_1[\{Zeichenmenge\}]$ und $d_2[1..m-1]$, um Regel (i) beziehungsweise (ii) zu implementieren.

d_1 ist einfach zu berechnen. Für jedes Zeichen c:

$d_1[c] \leftarrow$ **if** c in $p[1..m]$ nicht auftritt **then** m
else $m-\max\{i \mid p[i] = c\}$.

Dies ist der Abstand, um P nach der Regel (i) zu bewegen, wenn wir einen direkten Mismatch haben.

Es ist komplizierter, d_2 zu berechnen. Die Interpretation läßt sich folgendermaßen erklären: Nach einem Mismatch an der Position i des Musters beginnt man an der Position m von P $d_2[i]$ weitere Zeichen von S wieder zu prüfen. Statt eines Algorithmus zur Berechnung von d_2 geben wir nun Beispiele:

Beispiel: $P =$ "*assesses*"

(1) Angenommen wir haben einen Mismatch in der Position $p[7]$.

```
S    ? ? ? ? ? ? x s ? ? ? ? ? ? ? ?
P    a s s e s s e s
                 ↑ ↑
```

$x \neq$ "e"

Wir können das "s" in $p[6]$ und s aus S an der gleichen Stelle anordnen.

```
S    ? ? ? ? ? ? x s ? ? ? ? ? ? ? ?
P        a s s e s s e s
                     ↑
```

$x \neq$ "e"

Wir beginnen wieder am Ende von P zu prüfen. Es sind 3 Zeichen mehr aus S als beim vorherigen Vergleich. Folglich gilt $d_2[7] = 3$

(2) Angenommen wir haben in der Position $p[6]$ einen Mismatch. Daher:

```
S    ? ? ? ? ? x e s ? ? ? ? ? ? ? ?
P    a s s e s s e s
               ↑ ↑ ↑
```

$x \neq$ "s"

Da $x \neq$ "s" ist, können wir "e" in $p[4]$ und "s" in $p[5]$ mit "e" und "s" aus S nicht an der gleichen Stelle anordnen. Wir müssen P so anordnen, daß P unter den Zeichen aus S, die noch nicht berücksichtigt sind, steht.

S	?	?	?	?	?	x	e	s	?	?	?	?	?	?	?	?
P									a	s	s	e	s	s	e	s
																↑

Wir beginnen wieder am Ende von P zu prüfen. Es sind 10 Zeichen mehr aus S als beim vorherige Vergleich. Folglich gilt $d_2[6] = 10$.

(3) Angenommen der Mismatch ist in der Position $p[4]$:

S	?	?	?	x	s	s	e	s	?	?	?	?	?	?	?	?
P	a	s	s	e	s	s	e	s								
				↑	↑	↑	↑	↑								

$x \neq$ "e"

Es ist möglich, P um 3 Stellen zu verschieben.

S	?	?	?	x	s	s	e	s	?	?	?	?	?	?	?	?
P				a	s	s	e	s	s	e	s					
											↑					

$x \neq$ "e"

Wir beginnen am Ende von P zu prüfen, 7 Zeichen an der rechten Seite des vorherigen Vergleichs. Folglich gilt $d_2[4] = 7$.

Aus diesem Beispiel erhalten wir:

i	1	2	3	4	5	6	7	8
$p[i]$	a	s	s	e	s	s	e	s
$d_2[i]$	15	14	13	7	11	10	3	

Damit erhalten wir: d_1["s"] $= 0$, d_1["e"] $= 1$, d_1["a"] $= 7$ und d_1["*any other character*"] $=$ 8.

Der Algorithmus:

```
Algorithmus BM
    j, k ← m
    while k ≤ n and j > 0 do
        while j > 0 and s[k] = p[j] do
            k ← k − 1
            j ← j − 1
        if j ≠ 0 then
            if j = m then k ← k + d1[s[k]]
                else k ← k + d2[j]
                    j ← m
    if j = 0 then return k + 1
        else return 0
```

Eine direkte Verbesserung ist möglich, wenn man daran denkt, daß dieser Algorithmus nicht prüft, ob das Zeichen aus S, wo ein Mismatch entdeckt wird, in P auftritt. Dies kann leicht dadurch korrigiert werden, daß man $d_2[m] = 1, d_1[p[m]] = 0$ definiert und die drei Zeilen

```
if j = m then k ← k + d1[s[k]]
    else k ← k + d2[j]
        j ← m
```

durch

```
k ← k+ max(d1[s[k]], d2[j])
j ← m
```

ersetzt.

Dies ist die richtige Form des BM-Algorithmus.

Schlußbemerkung:
Es ist auch möglich, einen probabilistischen Algorithmus für das Suchen in Zeichenreihen zu entwerfen.