Zusammenfassung

Effiziente Algorithmen

Michael Gregorius

Version vom 16. September 2003

http://www.michaelgregorius.de/

Inhaltsverzeichnis

1 Grundlegende Graphalgorithmen **4**
1.1 Tiefensuche und topologisches Sortieren 4
1.2 Starke Zusammenhangskomponenten in gerichteten Graphen 6
1.3 Maximale zweifach zusammenhängende Komponenten von ungerichteten Graphen 9
1.4 Der Algorithmus von Dijkstra zur Berechnung kürzester Wege 16

2 Einige Beispiele zur Analyse von Algorithmen **19**
2.1 Amortisierte Rechenzeit, Buchhaltermethode und Potenzialfunktionen 19
2.2 Eine Analyse der Pfadkomprimierung bei UNION-FIND-Datenstrukturen . . 21
2.3 String Matching - ein Mustererkennungsproblem 24

3 Dynamische Programmierung **29**
3.1 Ein Rückblick auf bekannte Beispiele 29
3.2 Ein weiterer pseudopolynomieller Algorithmus für das Rucksackproblem . . . 31
3.3 Bioinformatik - die Ähnlichkeit zweier Sequenzen 32
3.4 Hidden Markoff Modelle 36

4 Branch-and-Bound Algorithmen **40**
4.1 Die Vorgehensweise bei Branch-and-Bound Algorithmen 40
4.2 Ein Branch-and-Bound Algorithmus für das Rucksackproblem 42
4.3 Branch-and-Bound Algorithmen für das TSP 43

5 Approximationsalgorithmen **49**
5.1 Approximationsalgorithmen für das metrische TSP 49
5.2 Ein echt polynomielles Approximationsschema für das Rucksackproblem . . . 52
5.3 Methoden zum Entwurf polynomieller Approximationsschemata 54
5.4 PTAS durch Glättung des Suchraums 54
5.5 PTAS durch Glättung des Lösungsraums 58
5.6 FPTAS durch Glättung der Zwischenlösungen eines langsamen Algorithmus . 59

6 Lineare Optimierung, Randomisiertes Runden und Derandomisierung **64**
6.1 Lineare Optimierungsprobleme 64
6.2 Ein einfacher randomisierter Algorithmus für die Probleme MAXSAT und MAXkSAT 67
6.3 Randomisiertes Runden 68
6.4 Derandomisierung und Methode der bedingten Wahrscheinlichkeiten 71

7 Randomisierte Suchheuristiken **75**
7.1 Motivation 75
7.2 Eine allgemeine obere Schranke für die erwartete Optimierungszeit des (1+1)EA 76
7.3 Populationen oder Multistarts? 81

8 Maximierung von Flüssen in Netzwerken **83**
8.1 Problemstellung und Beispiele 83
8.2 Der Algorithmus von Ford und Fulkerson 84
8.3 Der Algorithmus von Edmonds und Karp 88
8.4 Der Algorithmus von Dinic 92
8.5 Ein verbesserter Algorithmus zur Sperrflussberechnung 96

9 Algorithmen für Probleme der elementaren Zahlentheorie **101**
9.1 Kryptographische Systeme 101
9.2 Potenzen, multiplikative Inverse und größte gemeinsame Teiler 103
9.3 Der chinesische Restklassensatz 108
9.4 Grundlagen aus der Zahlentheorie 109
9.5 Das Jacobi-Symbol 112
9.6 Der probabilistische Primzahltest von Solovay und Strassen 112

Abbildungsverzeichnis **117**

Tabellenverzeichnis **117**

Liste der Algorithmen **117**

Literatur **118**

Index **119**

1 Grundlegende Graphalgorithmen

1.1 Tiefensuche und topologisches Sortieren

Frage 1.1: Wie funktioniert die Tiefensuche?

Die Tiefensuche (Depth-First Search) kann sowohl auf ungerichtete als auch auf gerichteten Graphen durchgeführt werden. Der Graph sei dabei als eine Menge von Knoten mit ihren entsprechenden Adjazenzlisten gegeben, wobei die Listen entweder als Array oder als lineare Liste gegeben sein kann. Im Laufe der Tiefensuche erhält jeder Knoten eine DFS-Nummer, die aufsteigend vergeben werden. Zu Beginn wird die DFS-Nummer für jeden Knoten mit Null initialisiert wird. Wird im Laufe des Algorithmus ein Knoten v erreicht, der noch keine Nummer hat, bekommt er eine Nummer zugeteilt und die Tiefensuche wird von ihm aus fortgesetzt, d.h. es werden alle Knoten $w \in Adj(v)$ untersucht. Der Algorithmus läuft also erstmal in die Tiefe, so lange er noch nicht besuchte Knoten findet. Aufgrund dieser charakteristischen Vorgehensweise hat die Tiefensuche auch ihren Namen.

In einem ungerichteten Graphen werden die Kanten im Laufe der Tiefensuche in die Menge der Tree- und Backkanten eingeteilt. Wird eine Kante betrachtet, die zu einem noch nicht besuchten Knoten führt, wird die Kante eine Tree-Kante ansonsten eine Backkante. Im resultierenden Ergebnisgraphen aus Tree- und Backkanten gibt es zwischen zwei Knoten immer nur eine gerichtete Kante, die entweder Tree- oder Backkante ist. Oder anders gesagt: Eine ungerichtete Kante wird zwar zweimal betrachtet, jedoch immer nur einmal klassifiziert. Dies liegt daran, dass sich bei jedem DFS-Aufruf für einen Knoten v gemerkt wird, von welchem Knoten w aus der Knoten v erreicht wurde. Der Aufruf hat also die Form $DFS(v, w)$.

Frage 1.2: Wann enthält ein ungerichteter Graph einen Kreis?

Ein ungerichteter Graph $G = (V, E)$ enthält genau dann einen Kreis, wenn die Menge der Back-Kanten nicht leer ist.

Frage 1.3: Ist die DFS-Nummerierung eindeutig?

Nein, die DFS-Nummerierung ist nicht eindeutig. Dies liegt daran, dass die Ordnung der Knoten in der Adjazenzliste eines Knotens nicht eindeutig festgelegt ist und der Algorithmus die adjazenten Knoten dementsprechend unterschiedlich nummerieren kann.

Frage 1.4: In welche Klassen werden die Kanten bei einer Tiefensuche in einem gerichteten Graphen eingeteilt?

Bei einem gerichteten Graphen werden die Kanten in vier verschiedene Klassen unterteilt. Dies muss gemacht werden, da bei einem gerichteten Graphen die Symmetrie $v \in Adj(w) \Leftrightarrow w \in Adj(v)$ nicht mehr gilt. Dies bedeutet, dass es nicht mehr selbstverständlich ist, dass wenn man über die Kante (v, w) von v zu w kommt, man auch über eine Kante (w, v) zurück von w nach v kann.

Der Algorithmus für ungerichtete Graphen kann auch nicht ohne weiteres für gerichtete Graphen übernommen werden. Bei ungerichteten Graphen entsteht ein Wald von Bäumen, die sich durch die Tree-Kanten ergeben und welche die Zusammenhangskomponenten repräsentieren. Allerdings sind Zusammenhangskomponenten für gerichtete Graphen anders definiert und somit kann man in diesem Fall mit den Tree-Kanten nicht so viel anfangen. Desweiteren würde bei der Anwendung des Algorithmus für ungerichtete Graphen auch nicht automatisch gelten, dass der Graph einen Kreis enthält, wenn die Menge der Back-Kanten leer ist. Aus

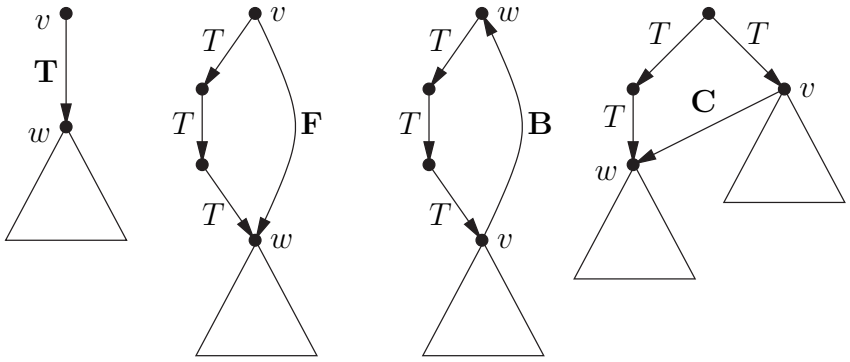


Abbildung 1: Beispiele für Tree-, Forward-, Back- und Cross-Kanten

diesem Grund werden vier verschiedene Kanten-Arten definiert, wobei diese grob danach unterschieden werden können, ob ein betrachteter Knoten schon eine DFS-Nummer hat (er also schon vorher besucht wurde), ob diese größer oder kleiner ist, als die DFS-Nummer des Knotens, von dem man kommt und ob der DFS-Aufruf des betrachteten Knotens noch läuft oder schon beendet ist. Die Kanten werden disjunkt die in die folgenden Kanten-Arten zerlegt, wobei die Kante (v, w), also der Aufruf $DFS(w, v)$ betrachtet wird:

Tree: Der Knoten w hat noch keine DFS-Nummer, also $num(w) = 0$. $(v, w) \in T$

Forward: Der Knoten w hat eine größere Nummer als v. Es gilt also $num(w) > num(v)$. Dementsprechend ist die Bearbeitung des Knotens w auch schon abgeschlossen, denn ansonsten würden wir ja nicht am Knoten v weiterarbeiten. $(v, w) \in F$

Back: Der Knoten w hat eine kleinere Nummer als v und die Bearbeitung von w ist noch nicht abgeschlossen. Während der Bearbeitung des Aufrufs $DFS(w, \cdot)$ wurde also der Knoten v über T-Kanten erreicht.

Cross: Der Knoten w hat eine kleinere Nummer als der Knoten v und die Bearbeitung von w ist abgeschlossen. Es gibt also entweder einen Weg von einem „späteren“ in einen „früheren“ Teilbaum bzw. von einem „späteren“ zu einem „früheren“ Baum.

Abbildung 1 gibt einige Beispiele für die entsprechenden Einteilungen. Es wird in der Abbildung jeweils davon ausgegangen, dass an der Wurzel der Beispielbäume zuerst der linke Teilbaum abgearbeitet wird.

Frage 1.5: Welche Laufzeit hat der DFS-Algorithmus?
Beim DFS-Algorithmus wird jeder Knoten und jede Kante behandelt. Daher beträgt die Laufzeit für n Knoten und m Kanten $O(n + m)$.

Frage 1.6: Wie ist eine partielle Ordnung definiert? Wie kann eine topologische Numerierung auf einer partiellen Ordnung berechnet werden?
Eine partielle Ordnung ist eine Relation, die reflexiv, antisymmetrisch und transitiv ist. Es sei $M \neq \emptyset$ und R eine Relation auf M. Damit es sich um eine partielle Ordnung handelt, muss gelten:

Reflexivität: $\forall x \in M : (x, x) \in R$

Antisymmetrie: $\forall x, y \in M : \ (x, y) \in R \ \wedge \ x \neq y \Rightarrow (y, x) \notin R$

Transitivität: $\forall x, y, z \in M : \ (x, y) \in R \ \wedge \ (y, z) \in R \Rightarrow (x, z) \in R$

Beispiele für partielle Ordnung sind z.B. $(P(M), \subseteq)$ oder $(\mathbb{Z}, \leq)$. Eine partielle Ordnung lässt sich durch einen gerichteten Graphen darstellen. Wird die Reflexivität für jedes Element angenommen, muss dabei gelten:

$$(x, y) \in R \wedge x \neq y \ \Leftrightarrow \ (x, y) \in E$$

Damit es sich auch wirklich um eine partielle Ordnung handelt, muss der Graph aufgrund der Transitivitätseigenschaft und der Antisymmetrie kreisfrei sein. Ein praktisches Beispiel für eine partielle Ordnung können verschiedene Aufgaben sein, bei denen einige nicht eher begonnen werden können, bevor andere Aufgaben erledigt wurden. Dann gibt eine topologische Nummerierung eine mögliche Reihenfolge der Bearbeitung an, welche die partielle Ordnung, in diesem Beispiel also die Reihenfolgebedingung, respektiert. Die Aufgaben sollen, entsprechend ihrer Nummerierung, in aufsteigender Reihenfolge behandelt werden. Damit dies klappt, muss also $num(v) \leq num(w)$ gelten, wenn $(v, w) \in E$. Es dürfen also nur Kanten von „früheren“ Knoten zu „späteren“ Knoten führen, d.h. die Menge der Back- und Cross-Kanten müssen leer sein.

Die Nummerierung wird im Algorithmus wie folgt vergeben. Der DFS-Algorithmus wird gestartet und jeder Knoten bekommt seine topologische Nummer, wenn sein DFS-Aufruf beendet ist. Die Nummern werden dabei rückwärts von n bis 1 vergeben. Um die Korrektheit dieses Algorithmus zu zeigen, ist zu beweisen, dass bei jeder Kante $(v, w) \in E$ stets der Aufruf $DFS(w)$ vor $DFS(v)$ beendet wird. Es können zwei Fälle unterschieden werden, bei denen dies nicht der Fall wäre:

1. Der Aufruf $DFS(v)$ wird zuerst gestartet und beendet, bevor $DFS(w)$ aufgerufen wird. Dies ist wegen des Vorhandenseins der Kante (v, w) unmöglich, da während des Aufrufs $DFS(v)$ auch $DFS(w)$ aufgerufen werden muss.

2. Der Aufruf $DFS(w)$ wird zuerst aufgerufen und innerhalb diesem wird $DFS(v)$ aufgerufen und beendet. Dies ist ebenfalls nicht möglich, da es in diesem Fall einen gerichteten Weg von w nach v gäbe, der zusammen mit der Kante (v, w) einen Kreis schließen würde. Dies ist ein Widerspruch zur Kreisfreiheit des Graphen.

1.2 Starke Zusammenhangskomponenten in gerichteten Graphen

Frage 1.7: Wie sind starke Zusammenhangskomponenten in gerichteten Graphen definiert?

In ungerichteten Graphen hängen zwei Knoten v und w zusammen, wenn es einen Weg zwischen den beiden gibt. Aufgrund der Definition eines ungerichteten Graphen, gibt es dann sowohl einen Weg von v zu w als auch von w zu v. Es liegt daher nahe, zwei Knoten v und w in einem gerichteten Graphen ebenfalls als zusammenhängend zu bezeichnen, wenn sowohl ein Weg von v zu w als auch von w zu v existiert. Zur besseren Unterscheidung werden zusammenhängende Knoten in gerichteten Graphen als stark zusammenhängend bezeichnet.

Die Relation starker Zusammenhang ($v \leftrightarrow w$) ist eine Äquivalenzrelation, wie schnell einzusehen ist:

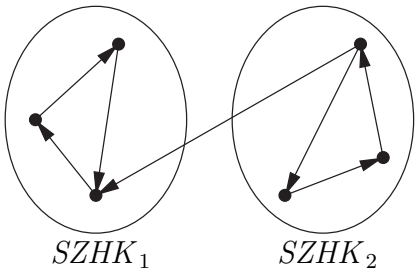


Abbildung 2: Beispiel für starke Zusammenhangskomponenten

Reflexivität: Es gilt $v \leftrightarrow v$, da es für jeden Knoten einen Weg der Länge 0 gibt, über den er sich selbst erreicht.

Symmetrie: Es gilt $v \leftrightarrow w \Leftrightarrow w \leftrightarrow v$, da die starken Zusammenhangskomponenten eben so definiert sind.

Transitivität: $v \leftrightarrow w, w \leftrightarrow z \Rightarrow v \leftrightarrow z$ gilt, da die entsprechenden Wege von v nach w und von w nach z bzw. von z nach w und von w nach v nur konkateniert werden müssen.

Da diese Relation eine Äquivalenzrelation ist, werden die Knoten also in Äquivalenzklassen $V_1, \ldots, V_k$ partitioniert. Die Knotenmengen $V_1, \ldots, V_k$ werden starke Zusammenhangskomponenten genannt. Oft werden auch die durch die Knotenmengen induzierten Teilgraphen als starke Zusammenhangskomponenten bezeichnet, also die Graphen $G_i = (V_i, E_i)$ mit $E_i = E \cap (V_i \times V_i)$ für $i = 1, \ldots, k$. Im Gegensatz zu den ungerichteten Graphen können bei gerichteten Graphen auch Kanten zwischen den Zusammenhangskomponenten verlaufen, was in Abbildung 2 verdeutlicht wird.

Frage 1.8: Wie können starke Zusammenhangskomponenten in gerichteten Graphen berechnet werden?

Zur Berechnung von starken Zusammenhangskomponenten existiert ein Algorithmus, der grob betrachtet aus zwei DFS-Durchläufen besteht:

1. Führe eine DFS-Traversierung durch, wobei die Knotennummern rückwärts von n bis 1 vergeben werden, und jeder Knoten seine Nummer bei Beendigung seines DFS-Aufrufs bekommt.

2. Bilde den rückwärts gerichteten Graph $G_r = (V, E_r)$, wobei $(v, w) \in E_r$ genau dann gilt, wenn $(w, v) \in E$.

3. Führe eine DFS-Traversierung auf G_r durch, wobei die Knoten als Liste gemäß der in Schritt 1 berechneten Nummerierung vorliegt. Die Zusammenhangskomponenten sind dann die Knotenmengen $V_1, \ldots, V_k$ der Bäume, die durch die konstruierten Tree-Kanten gebildet werden.

Die Laufzeit dieses Algorithmus beträgt $O(n + m)$ bei einem Graph mit n Knoten und m Kanten. In Schritt 1 wird eine DFS-Traversierung durchgeführt, die $O(n + m)$ dauert. Danach wird der rückwärts gerichtete Graph gebildet, was ebenfalls $O(m)$ dauert. Anschließend wird eine zweite DFS-Traversierung durchgeführt (wieder $O(n + m)$ und die Knotenmengen

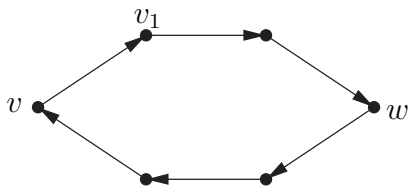


Abbildung 3: Der Grund, warum $v_1 \notin V_1 \cup \cdots \cup V_{i-1}$ gilt

ermittelt. Der letzte Schritt dauert $O(n)$, da der Durchlauf durch einen Baum mit k Knoten eine Laufzeit von $O(k)$ hat.

Frage 1.9: Beweise, dass der Algorithmus zur Berechnung der starken Zusammenhangskomponenten in gerichteten Graphen korrekt ist!

Im folgenden wird nun gezeigt:

$$v \leftrightarrow w \quad \Leftrightarrow \quad v \text{ und } w \text{ liegen nach der zweiten DFS-Traversierung in einem } T\text{-Baum.}$$

Es wird nun zuerst die Richtung „⇒“ gezeigt, also dass zwei Knoten bei der zweiten DFS-Traversierung in einem Baum liegen, wenn sie stark zusammenhängend sind.

„⇒“

Da $v \leftrightarrow w$ gilt, gibt es also einen Weg von v zu w sowie einen Weg von w zu v. Dies gilt natürlich immer noch im Graph G_r, wenn die Kanten also umgedreht werden. Denn in diesem Fall sind die umgedrehten Kanten des Weges von v nach w in G nun die Kanten des Weges von w nach v in G_r. Dasselbe gilt für die Kanten des Weges von w nach v in G. Es wird nun die zweite DFS-Traversierung betrachtet, wobei o.B.d.A. davon ausgegangen wird, dass $DFS(v)$ vor $DFS(w)$ ausgeführt wird. Der Knoten v wird während des Algorithmus zu einer Knotenmenge V_i hinzugefügt. Dies bedeutet, dass die Knotenmengen $V_1, \ldots, V_{i-1}$ nur Knoten enthalten, von denen es keinen Weg zum Knoten v oder zum Knoten w gibt. Denn ansonsten wäre der Knoten v während einer „früheren“ Bearbeitung direkt in eine der Mengen $V_1, \ldots, V_{i-1}$ eingeordnet worden oder er wäre über den Knoten w während dessen Aufruf $DFS(w)$ entdeckt worden und wie der Knoten w in eine der Mengen $V_1, \ldots, V_{i-1}$ eingeordnet worden. In diesem Fall wäre jedoch $DFS(w)$ vor $DFS(v)$ ausgeführt worden, was wir etwas weiter oben jedoch eh nicht angenommen haben. Da $v \leftrightarrow w$ gilt, existiert ein Weg zwischen v und w. Dies sei der Weg $(v, v_1, \ldots, v_l, w)$. Im Algorithmus werde nun die Kante (v, v_1) betrachtet. Da $v_1 \notin V_1 \cup \cdots \cup V_{i-1}$ gilt, ist v_1 also schon in V_i enthalten oder es wird zu V_i hinzugefügt (dann ist die Kante $(v, v_1) \in T$). Dasselbe Argument gilt dann für die Kante (v_1, v_2) und die weiteren Kanten bis w, so dass induktiv $w \in V_i$ folgt. Somit liegen v und w in einem Baum.

Bleibt nur noch zu klären, warum im obigen Beweis $v_1 \notin V_1 \cup \cdots \cup V_{i-1}$ angenommen werden kann. Abbildung 3 versucht dies zu verdeutlichen. Der Knoten v liegt in der Menge V_i und somit im i-ten Baum, der erzeugt wird. Wäre der Knoten v_1 nun vorher entdeckt worden und läge somit in einem der Bäume $V_1, \ldots, V_{i-1}$, so könnte man über den Kreis, der sich durch den Weg von v zu w und den Weg von w zu v ergibt, einen Weg von T-Kanten zu v konstruieren und v wäre somit ebenfalls vorher entdeckt worden. Es könnte nun argumentiert werden, dass es nicht automatisch einen Weg aus T-Kanten vom Knoten v_1 zum Knoten v geben muss. Dies wäre der Fall, wenn einer der Knoten auf dem Weg entdeckt worden wäre, bevor v_1 entdeckt worden wäre. Wie leicht einzusehen ist, verschiebt dies das Problem jedoch nur und der Knoten v wäre erst recht nicht in die Menge V_i eingeordnet worden.

Es bleibt nun also nur noch die Richtung „⇐“ zu zeigen, also dass die Knoten v und w zusammenhängend sind, wenn sie nach der zweiten DFS-Traversierung in einem T-Baum liegen.

„⇐“ Es wird nun also angenommen, dass v und w im selben Baum landen. Dies sei der T-Baum T_i auf V_i. In diesem Baum gibt es dann also einen Weg von der Wurzel r zu v sowie einen Weg von r zu w. Es wird im Folgenden nun gezeigt, dass $r \leftrightarrow v$ und $r \leftrightarrow w$ gilt, weshalb durch die Transitivität auch $v \leftrightarrow w$ gelten muss. Da die Fälle symmetrisch sind, reicht es $r \leftrightarrow v$ zu beweisen. Wir wissen, dass es in G_r einen Weg von r zu v gibt, woraus folgt, dass es in G einen Weg von v zu r gibt. Dieses Wissen wird gleich auch noch benötigt werden. Zu zeigen ist also noch, dass es in G auch einen Weg von r zu v gibt. Da r und v beide im Baum T_i liegen, wurden sie also vor dessen Konstruktion noch nicht behandelt. Da r die Wurzel von T_i ist, muss dieser Knoten während der DFS-Traversierung in Schritt 1 also eine kleinere Nummer als der Knoten v erhalten haben. Dies ist gleichbedeutend damit, dass der DFS-Aufruf von v eher beendet wurde als der von r. Es sind zwei Fälle möglich, bei denen dies auftreten kann:

1. Der DFS-Aufruf von v wird innerhalb des DFS-Aufrufs von r behandelt. In diesem Fall muss es also einen Weg von r zu v geben. Genau dies wollen wir zeigen.

2. Der DFS-Aufruf von v wurde beendet, bevor der DFS-Aufruf von r gestartet wird. Es wird sich gleich zeigen, dass dieser Fall nicht auftreten kann.

Es wird nun also gezeigt, dass $DFS(v)$ nicht beendet werden kann, bevor $DFS(r)$ beginnt. Wie etwas weiter oben schon festgestellt wurde, gibt es einen Weg von v zu r. Dieser Weg sei der Weg $(v, v_1, \ldots, v_l, r)$. Zuerst werden die Knoten $v_1, \ldots, v_l$ näher betrachtet. Sie müssen alle in V_i bzw. T_i liegen, denn ansonsten wäre der Knoten r nicht die Wurzel von T_i geworden. Würde nämlich einer der Knoten in $V_1, \ldots, V_{i-1}$ liegen, so wäre r in dieselbe Menge eingeordnet worden, da ein Weg zu ihm existiert. Die Knoten müssen also alle entdeckt werden, während der Baum T_i aufgebaut wird. Irgendwann wird während dieses Vorgangs also der Knoten v erreicht und $DFS(v)$ aufgerufen. Was passiert nun, wenn die Knoten $v_1, \ldots, v_l$ zu diesem Zeitpunkt alle noch nicht entdeckt wurden? In diesem Fall würde $DFS(r)$ noch während $DFS(v)$ aufgerufen werden, da er spätestens über den Weg $(v, v_1, \ldots, v_l, r)$ entdeckt werden würde. Dies widerspricht jedoch der Annahme, dass der DFS-Aufruf von v beendet ist, bevor der Aufruf von r gestartet wird. Also muss es einen Knoten v^* auf dem Weg geben, für den $DFS(v^*)$ vor $DFS(v)$ aufgerufen wurde. Damit wurde sein DFS-Aufruf auch vor $DFS(r)$ gestartet. Andererseits gibt es einen Weg von ihm zu r also muss $DFS(r)$ auch vor $DFS(v^*)$ beendet werden. Abbildung 4 zeigt diese Situation graphisch. Da $DFS(v^*)$ nach $DFS(r)$und $DFS(v)$ beendet wird, bekommt v^* in Schritt 1 des Algorithmus also auch eine kleinere Nummer als r. Wenn in Schritt 3 nun jedoch der Baum T_i aufgebaut werden soll, sind sowohl der Knoten v^* als auch der Knoten r noch nicht erreicht worden. Da der Algorithmus unter den noch nicht erreichten Knoten einen mit kleinster Nummer wählt, kann dies nicht r sein, da v^* eine kleinere Nummer hätte. Dann könnte r jedoch auch nicht die Wurzel von T_i werden, was zu einem Widerspruch führt.

1.3 Maximale zweifach zusammenhängende Komponenten von ungerichteten Graphen

Frage 1.10: Wie ist ein Teilgraph für einen ungerichteten Graphen definiert?

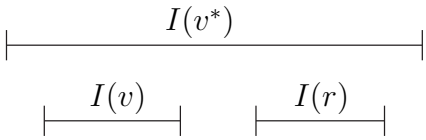


Abbildung 4: Die Situation mit dem Knoten v^*

Es sei ein ungerichteter Graph $G = (V, E)$ gegeben. Dann ist der Teilgraph auf $V' \subseteq V$ als Graph $G' = (V', E')$ gegeben, wobei E' nur Kanten enthält, deren beide Knoten beide zu V' gehören. Eine andere Vorstellungsweise ist die, dass aus dem Graph G alle Knoten $V - V'$ sowie die zu diesen Knoten adjazenten Kanten entfernt werden.

Frage 1.11: Wann heißt ein ungerichteter Graph zweifach zusammenhängend? Was ist eine maximale zweifach zusammenhängende Komponente?

Ein zusammenhängender ungerichteter Graph $G = (V, E)$ heißt zweifach zusammenhängend, wenn für jeden Knoten v der Teilgraph auf $V - \{v\}$ immer noch zusammenhängend ist. Ein Teilgraph auf $V' \subseteq V$ heißt maximale zweifach zusammenhängende Komponente, wenn er zweifach zusammenhängend ist und für jede echte Obermenge V'' von V' der Teilgraph nicht mehr zweifach zusammenhängend ist.

Frage 1.12: Was ist ein Schnittpunkt?

Sei $G = (V, E)$ ein zusammenhängender ungerichteter Graph. Dann ist ein Knoten v, für den der Teilgraph auf $V - \{v\}$ nicht mehr zusammenhängend ist, ein Schnittpunkt von G.

Frage 1.13: Zeige, dass die Berechnung aller maximal zweifach zusammenhängender Komponenten eine „vernünftige Aufgabe“ ist!

Mit der Formulierung „vernünftige Aufgabe“ ist gemeint, dass es nicht exponentiell viele Lösungen gibt bzw. dass die Ausgabe nicht exponentiell zur Eingabe ist. Es wird sich etwas später zeigen, dass jede Kante eines Graphen in nur einer ZZK (zweifach zusammenhängende Komponente) liegen kann. Damit besteht die Aufgabe der Berechnung aller ZZKs quasi in einer Partitionierung der Kanten eines Graphen und somit ist die Ausgabe linear zu Anzahl der Kanten des Graphen.

Frage 1.14: Zeige, dass ein Knoten, der zu zwei ZZKs gehört, ein Schnittpunkt ist!

Die ZZKs werden über ihre Knotenmengen identifiziert. Es seien also zwei ZZKs V_1 und V_2 gegeben und ein Knoten $v \in V_1 \cap V_2$ sei in beiden enthalten. Damit ist der Graph auf $V_1 \cup V_2$ zusammenhängend, nicht jedoch zweifach zusammenhängend. Ansonsten wären V_1 und V_2 ja nicht maximale ZZKs. Es muss also einen Schnittpunkt w geben, so dass $(V_1 \cup V_2) - \{w\}$ nicht zusammenhängend ist. Die Graphen $V_1 - \{w\}$ und $V_2 - \{w\}$ sind jedoch weiterhin zusammenhängend, da V_1 und V_2 ja ZZKs sind und ein Knoten ruhig entfernt werden kann. Damit muss es also Knoten $v_1 \in V_1 - \{w\}$ und $v_2 \in V_2 - \{w\}$ geben, so dass es im Teilgraph $(V_! \cup V_2) - \{w\}$ keinen Weg zwischen v_1 und v_2 gibt. Nun wird der Knoten w etwas genauer betrachtet. Wenn $w \neq v$ ist, gibt es in $V_1 - \{w\}$ noch einen Weg von v_1 zu v und in $V_2 - \{w\}$ noch einen Weg von v_2 zu v. Somit gäbe es aber noch einen Weg zwischen v_1 und v_2 und es ergäbe sich ein Widerspruch, da $(V_1 \cup V_2) - \{w\}$ nach herausnehmen des Schnittpunkts immer noch zusammenhängend wäre. Also muss $w = v$ gelten und v ist somit ein Schnittpunkt. Abbildung 5 versucht die Idee etwas zu verdeutlichen.

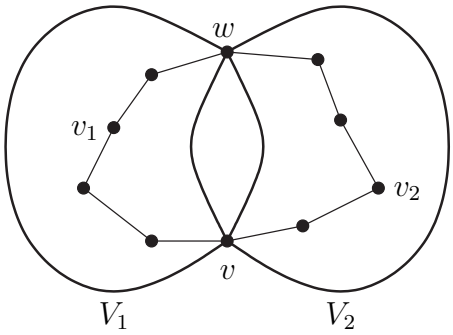


Abbildung 5: Warum zwei ZZKs V_1 und V_2 maximal einen Schnittpunkt gemeinsam haben können

Frage 1.15: Zeige, dass zwei verschiedene ZZKs maximal einen Schnittpunkt gemeinsam haben können!
Die wurde im Prinzip schon im Beweis zuvor gezeigt und kann sich auch schnell an Abbildung 5 klar gemacht werden. Würde es zwei ZZKs V_1 und V_2 geben, die zwei Schnittpunkte v und w gemeinsam hätten, so gäbe es selbst nach der Herausnahme einer der beiden Knoten „Ersatzweg“ zwischen jedem Knoten $v_1 \in V_1$ und $v_2 \in V_2$, der über den jeweils anderen Knoten führen würde. Dann wären V_1 und V_2 jedoch von vornherein keine ZZKs gewesen, da $V_1 \cup V_2$ ebenfalls eine ZZK wäre.

Frage 1.16: Zeige, dass jede Kante zu genau einer ZZK gehört!
Zuerst wird gezeigt, dass *jede* Kante zu einer ZZK gehört. Sei $\{v, w\} \in E$. Dann ist $\{v, w\}$ zweifach zusammenhängend, da nach Herausnahme von v oder w der andere Knoten immer noch erreichbar ist (über einen Weg der Länge 0). Wenn diese Kante nun in zwei ZZKs V_1 und V_2 liegen würde, müsste somit $v, w \in V_1 \cap V_2$ gelten. Wie wird jedoch schon gesehen haben, ist ein Knoten, der in zwei ZZKs liegt, ein Schnittpunkt. Somit müssten v und w Schnittpunkte sein. Dies steht jedoch widerum im Widerspruch dazu, dass zwei ZZKs höchstens über einen Schnittpunkt miteinander verbunden sind. Also kann jede Kante nur in genau einer ZZK liegen.

Frage 1.17: Zeige, dass Schnittpunkte in mindestens zwei ZZKs liegen!
Es sei ein Graph $G = (V, E)$ und ein Schnittpunkt v gegeben. Dann ist der Teilgraph auf $V - \{v\}$ nicht mehr zusammenhängend. Dies bedeutet, dass es zwei Knoten w_1 und w_2 in G geben muss, so dass nach Herausnahme von v alle Wege zwischen diesen zerstört werden. Daraus folgt, dass also alle Wege zwischen w_1 und w_2 im ursprünglichen Graph G über v laufen müssen. Im Folgenden wird nun ein solcher Weg betrachtet. Dieser läuft über v sowie über die zwei Nachbarn von v, die u_1 und u_2 genannt werden. Da nach dem Herausnehmen von v auch alle Wege zwischen u_1 und u_2 zerstört werden (ansonsten würde es ja noch einen Weg zwischen w_1 und w_2 geben), müssen diese beiden Knoten also in zwei verschiedenen ZZKs liegen. Somit liegen auch die Kanten $\{u_1, v\}$ und $\{v, u_2\}$ in verschiedenen ZZKs und v in zwei ZZKs.

Frage 1.18: Was ist die Grundidee des Algorithmus zur Berechnung der ZZKs?
ZZKs „kleben“ über Schnittpunkte zusammen. Dies liegt daran, dass Schnittpunkte in mindestens zwei ZZKs liegen und zwei verschiedene ZZKs nur über einen Schnittpunkt miteinander verbunden sein können. Abbildung 6 gibt ein Beispiel, wie dies aussehen könnte.

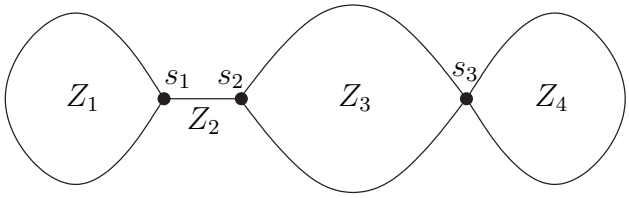


Abbildung 6: Beispiel für Schnittpunkte und ZZKs

Der Algorithmus zur Berechnung der ZZKs ist eine modifizierte Tiefensuche. Anhand von Abbildung 6 soll nun überlegt werden, was bei der Ausführung der Tiefensuche passiert. Angenommen der Algorithmus arbeitet gerade in der ZZK Z_3 und kommt zum Schnittpunkt s_3. Dieser wird nun per $DFS(s_3, \cdot)$ bearbeitet. Dabei kann natürlich auch zuerst weiter in die ZZK Z_3 hereingelaufen werden. Irgendwann jedoch wird die erste Kante bearbeitet, die in Z_4 hineinläuft. Wenn wir an dieser Stelle nun erkennen würden, dass wir über den Schnittpunkt s_3 in eine neue ZZK hineingelaufen sind, könnten wir alle Kanten, die während dieses Aufrufs gefunden werden, als zur ZZK Z_4 zugehörig in die Ausgabe schreiben. Denn die ZZK muss ja aufgrund der Arbeitsweise der Tiefensuche komplett abgearbeitet werden, bevor der Algorithmus über die weitere Bearbeitung von s_3 eventuell wieder in Z_3 weiter arbeitet. Was ist nun das Besondere an der „ersten“ Kante einer ZZK? Das Besondere ist, dass wir vor Beendigung der Bearbeitung der ZZK keine Kante finden werden, die auf einen Knoten zeigt, der eine kleinere DFS-Nummer als der Schnittpunkt s hat. Für alle anderen Kanten (v, w) muss es aufgrund des zweifachen Zusammenhangs eine Kante von einem Nachfolger von v zu einem Vorgänger von v geben. Dies wird sich als effizient überprüfbar herausstellen.

Es gibt dabei jedoch noch ein Problem. Und zwar kann es ja passieren, dass in lauter neue ZZKs hineinlaufen wird, bevor die vorherigen überhaupt komplett abgearbeitet wurden. Aus diesem Grund werden die neu gefundenen Kanten alle auf einen Stack gelegt. Irgendwann werden wir dann in die „letzte“ ZZK (wie in Abbildung 6 z.B. die ZZK Z_4) hineinlaufen und diese komplett abarbeiten. Die ZZK wird dann am Ende ihrer Bearbeitung hoffentlich als eine solche erkannt und alle Kanten können vom Stack heruntergenommen werden. Wenn anschließend weitergearbeitet wird, wird entweder wieder in eine neue ZZK hereingelaufen, bei der entsprechend vorgegangen wird, oder aber die vorherige ZZK weiter bearbeitet.

Was passiert dabei in der ersten ZZK, in der der Algorithmus gestartet wird? Diese muss ja nicht zwangsläufig an einem Schnittpunkt starten. Da alle weiteren ZZKs bearbeitet und gefunden werden, bevor der erste Aufruf terminiert, müssen bei der Terminierung dieses Aufrufs also alle Kanten, die noch auf dem Stack liegen, zu dieser ZZK gehören.

Frage 1.19: Anhand welchen Kriteriums können Schnittpunkte erkannt werden?

Zuerst wird nun die Funktion $l(v)$ (Lowpoint von v) definiert, die im darauf folgenden Kriterium benutzt wird. Es sei ein ungerichteter Graph $G = (V, E)$ gegeben und $num : V \to \{1, \ldots, n\}$, die vom DFS-Algorithmus berechnete Knotennummerierung. Dann wird $l(v)$ definiert als:

$$l(v) = \min\{num(x) \mid \quad \exists \text{ Pfad } v \to x \text{ aus } T\text{-Kanten und am Ende eventuell einer } B\text{-Kante}\}$$

Es wird nun das Kriterium vorgestellt. Es sei ein zusammenhängender, ungerichteter Graph

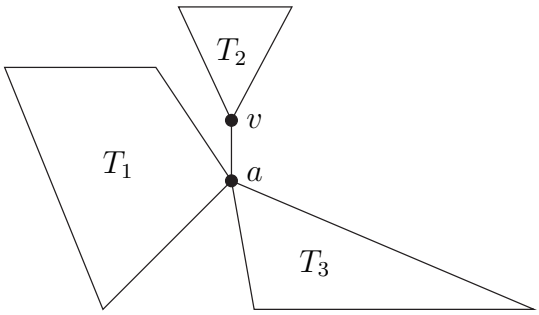


Abbildung 7: Schematische Darstellung für die Richtung „$2 \Rightarrow 1$“

$G = (V, E)$ gegeben, auf dem die Tiefensuche die Nummerierung *num* und die Kanteneinteilung $E = T \cup B$ liefert. Dann sind die beiden folgenden Aussagen äquivalent:

1. a ist Schnittpunkt.
2. $\exists v, w \in V$:
   a) $(a, v) \in T$,
   b) $w \neq a$,
   c) w ist kein T-Nachfolger von v,
   d) $l(v) \geq num(a)$.

„$2 \Rightarrow 1$“ Es wird nun zuerst die Richtung „$2 \Rightarrow 1$“ gezeigt. G ist zusammenhängend und somit bildet T einen Baum auf allen Knoten in G. Da $(a, v) \in T$ gilt, muss also während der Tiefensuche $DFS(v, a)$ aufgerufen worden sein. Der bei der Tiefensuche entstandene T-Baum wird nun in verschiedene Komponenten zerlegt. Zum einen in den Teilbaum T_1, der vor dem Aufruf von $DFS(v, a)$ erzeugt wurde, der Kante (a, v), dem Teilbaum T_2, der während des Aufrufs $DFS(v, a)$ erzeugt wurde sowie dem Teilbaum T_3, der nach Beendigung von $DFS(v, a)$ erzeugt wurde. Das Ziel ist es nun, zu zeigen, dass einen Knoten w in T_1 oder T_3 gibt, für den es nach Herausnahme von a keinen Weg mehr zum Knoten v gibt. Oder umgangssprachlicher gesagt: Es gibt einen Knoten im bisher konstruierten oder im später konstruierten Baum, der v nur über a erreichen kann. Da der Graph vorher zusammenhängend war, muss a dann also ein Schnittpunkt sein. Die Existenz eines solchen Knotens wird durch die Bedingungen 2b) und 2c) gewährleistet. Abbildung 7 zeigt die Situation schematisch. Dabei muss der Teilbaum T_3 nicht unbedingt an a hängen, sondern kann auch leer sein oder an T_1 mit dranhängen.

Wie schon gesagt, sorgen die Bedingungen 2b) und 2c) dafür, dass es einen Knoten w in T_1 oder T_3 gibt. Der Knoten v ist die Wurzel von T_2. Wenn wir nun zeigen, dass es keine T- und B-Kanten zwischen T_1 und T_2 bzw. T_3 und T_2 geben kann, dann zeigen wir, dass der Knoten a für den Knoten w unbedingt notwendig ist, um den Knoten v zu erreichen. Wenn es eine solche Kante nämlich geben würde, könnten wir nach dem Herausnehmen von Knoten a immer noch vom Knoten w über diese Kante in den Baum T_2 hereinlaufen und von dort zu Knoten v gelangen. Dies ginge aufgrund der Zusammenhangseigenschaft von G.

Zuerst wird gezeigt, dass keine T-Kanten existieren können. Zwischen T_1 und T_2 sowie zwischen T_2 und T_3 kann es aufgrund der Baumeigenschaft keine T-Kanten geben. Andersherum ist dies natürlich erst recht nicht möglich, da die Teilbäume mit den kleineren Nummern von den Teilbäumen mit größeren Nummern erzeugt wurden (also z.B. T_2 vor T_3) und somit

ein Knoten in einem Teilbaum mit einer kleineren Nummer nicht noch einmal von einem Teilbaum mit einer größeren Nummer aus neu entdeckt werden könnte. Nun bleibt noch zu zeigen, dass keine B-Kanten existieren können. Im Falle einer B-Kante zwischen T_3 und T_2 kann dies recht schnell widerlegt werden. Wenn es zwischen dem Baum T_3 und dem Baum T_2 eine B-Kante geben würde, so wäre diese beim Aufbau von T_2 entdeckt worden und der entsprechende Knoten in T_3 wäre in T_2 gelandet. Im Falle einer B-Kante zwischen T_2 und T_1 kann leider nicht so einfach argumentiert werden. Dies liegt daran, dass T_2 auf jedem Fall komplett abgearbeitet ist, wenn T_3 aufgebaut wird. Im Falle des Aufbaus von T_2 muss jedoch nicht unbedingt T_1 komplett aufgebaut sein. Das heißt, es könnte also noch Kanten geben, die von T_2 zu einem Knoten in T_1 laufen, für den der DFS-Aufruf noch nicht beendet wurde, so dass sich eine B-Kante vom Baum T_2 zu T_1 ergäbe. Aus diesem Grund geht nun die Bedingung 2d) mit in den Beweis ein. Es sei y der Knoten in T_1 zu dem die B-Kante führen soll. Im Falle, dass $num(y) > num(a)$ gilt, wurde $DFS(y, \cdot)$ schon komplett abgearbeitet, da der Knoten während der Bearbeitung von a entdeckt wurde. In diesem Fall gilt jedoch wieder dasselbe Argument wie eben. Die Kante zu y wäre keine B-Kante geworden sondern eine T-Kante. Für $num(y) < num(a)$ ergibt sich jedoch ebenfalls ein Widerspruch. Denn dann gäbe es für v einen Weg über einige T-Kanten und eine B-Kante zu y und somit würde $l(v) \leq num(y) < num(a)$ gelten, also $l(v) < num(a)$. Ein Widerspruch zu Bedingung 2d), die $l(v) \geq num(a)$ fordert.

Es ergibt sich also insgesamt, dass nach Herausnahme von a keine Weg mehr zwischen w und v existiert und a somit ein Schnittpunkt ist.

Nun wird die Richtung „$1 \Rightarrow 2$“ gezeigt, wobei zwei Fälle unterschieden werden können. Der Knoten a kann die Wurzel von T sein oder mitten in T liegen (nicht jedoch ein Blatt sein). „$1 \Rightarrow 2$“

Fall 1: Im ersten Fall ist a die Wurzel von T. Dabei sei (a, v) die erste vom DFS-Algorithmus gehandelte Kante. Wenn es nun keine zweite T-Kante von a geben würde, so wäre der Baum T nach der Herausnahme von a immer noch zusammenhängend, was ein Widerspruch zur Annahme ist, dass a ein Schnittpunkt ist. Also muss es noch eine zweite Kante $(a, w) \in T$ geben, mit $w \neq v$. Damit gelten die Bedingungen 2a) - 2c). Bedingung 2d) gilt ebenfalls, da $l(v) \geq 1 = num(a)$.

Fall 2: In diesem Fall ist a nicht die Wurzel von T. Ein Blatt kann a jedoch auch nicht sein, da ein Baum nach Herausnahme eines Blattes immer noch zusammenhängend ist. Somit muss a also „mitten im Baum“ liegen und es gibt Kanten $(w, a) \in T$ und $(a, v) \in T$. Damit wären die Bedingungen 2a) - 2c) wieder erfüllt. Wenn a in T genau s Söhne hat, dann zerfällt T nach der Herausnahme von a in genau $s+1$ Zusammenhangskomponenten: Den Teilbaum Z_0, der auch den Knoten w enthält, sowie die s Teilbäume $Z_1, \ldots, Z_s$ mit der Wurzel a. Dies sind jedoch die Zusammenhangskomponenten, in die der Baum T zerfällt. Da es in G noch mehr Kanten geben kann, sind die Zusammenhangskomponenten, in die G zerfällt, Vereinigungen der Komponenten Z_i. Es muss dabei jedoch mindestens eine Komponente Z_j geben, die nicht mit Z_0 vereinigt wird, denn ansonsten könnte a ja kein Schnittpunkt sein. Wir wählen die Wurzel dieser Komponente Z_j als Knoten v, für den wir nun nur noch zeigen müssen, dass $l(v) \geq num(a)$ gilt. Da jeder Knoten einen $l(v)$-Wert bekommt, muss nur gezeigt werden, dass $l(v) < num(a)$ nicht gelten kann, woraus automatisch $l(v) \geq num(a)$ folgt. Angenommen $l(v) < num(a)$ würde gelten. Dann gäbe es also einen Knoten x mit $num(x) = l(v)$, der über einen Pfad von T-Kanten und einer B-Kante von v aus erreichbar ist. Da $num(x) < num(a)$

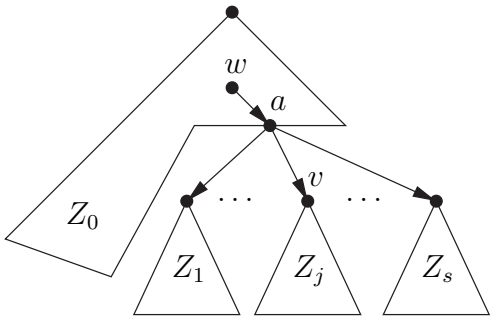


Abbildung 8: Fall 2 der Richtung „1 ⇒ 2“

gilt, muss der Knoten in Z_0 liegen, da alle Knoten, die vor dem Knoten a behandelt wurden, in Z_0 liegen. Da $num(x) < num(a)$ gilt, ist $x \neq a$, d.h. der Knoten x ist auch noch nach der Herausnahme von a vorhanden. Der Knoten a liegt auch nicht auf diesem Weg, da er von v aus nicht über T-Kanten erreichbar ist. Also bleibt dieser Weg auch nach der Herausnahme von a noch bestehen und die Komponenten Z_0 und Z_j wären im Widerspruch zur Konstruktion noch zusammenhängend.

Frage 1.20: Wie können die l(v)-Werte effizient berechnet werden?

Zwar führt die ursprüngliche Definition zu einem polynomiellen Algorithmus, jedoch können die $l(v)$-Werte auch noch alternativ charakterisiert werden, was zu einem rekursiven, effizienteren Algorithmus führt. Dieser kann sofort während des Durchlaufs durch den Graph bei der Berechnung der ZZKs verwendet werden. Die Definition von $l(v)$ kann auch geschrieben werden als:

$$l(v) = \min(\ \{num(v)\}\ \cup\ \{num(x)|(v,x) \in B\}\ \cup\ \{l(x)|(v,x) \in T\}\ )$$

Frage 1.21: Gebe den Algorithmus zur Berechnung der ZZKs an!

Algorithmus 1 beschreibt die Berechnung der ZZKs. Es wird ein Stack benutzt, der zu Beginn leer initialisiert ist. Da wir eine modifizierte Tiefensuche durchführen, muss der *num*-Wert eines jeden Knotens mit 0 initialisiert sein, da wir ansonsten nicht wüssten, ob ein Knoten schon gefunden und behandelt wurde oder nicht.

Im Prinzip wird eine Tiefensuche durchgeführt, welche die etwas weiter oben beschriebenen Erkenntnisse benutzt, um die ZZKs zu identifizieren. Für jeden Knoten wird mit Hilfe der rekursiven Definition zum richtigen Zeitpunkt sein l-Wert berechnet. Es wird die Adjazenzliste eines jeden Knoten, für den $ZZK(v,u)$ aufgerufen wird, durchlaufen und getestet, ob eine Tree- oder Backkante gefunden wurde. In beiden Fällen wird die Kante auf den Stack gepackt. Handelt es sich um eine Treekante, wird die *ZZK*-Prozedur rekursiv aufgerufen. Läuft die Rekursion aus diesem Aufruf zurück, schauen wir, ob wir den l-Wert aktualisieren müssen. Im Anschluss wird geprüft, ob die betrachtete Kante, die ersten Kante einer ZZK ist. Dies wird nur für Treekanten getestet, da eine solche Kante stets Treekante wird. Aufgrund des Zweizusammenhangs werden zu diesen Zeitpunkt auch schon alle Kanten der ZZK auf den Stack gepackt worden sein. Insbesondere sind alle anderen Kanten, die zum Schnittpunkt inzident sind, als Backkanten klassifiziert worden, so dass eine Ausgabe des Stacks für einen Schnittpunkt auch nur einmal stattfinden wird. Bei der Bearbeitung der Backkanten werden diese lediglich auf den Stack gepackt und der l-Wert aktualisiert.

Algorithmus 1 Der rekursive Algorithmus zur Berechnung von $\mathbf{ZZK}(v, u)$

```
i = i + 1; num(v) = i; l(v) = i;
for w ∈ Adj(v) do
  if num(w) = 0 then
    (v, w) → S; ZZK(w, v);
    l(v) = min{l(v), l(w)};
    if l(w) ≥ num(v) then
      entferne Kanten vom Stack bis einschließlich (v, w). Die Kanten bilden eine ZZK
      und werden in die Ausgabe geschrieben.
    end if
  else
    if num(w) < num(v) und w ≠ u then
      (v, w) → S; l(v) = min(l(v), num(w))
    end if
  end if
end for
```

1.4 Der Algorithmus von Dijkstra zur Berechnung kürzester Wege

Frage 1.22: Wie ist die grobe Vorgehensweise beim Algorithmus von Dijkstra?

Das Ziel des Algorithmus von Dijkstra ist es, für einen Graphen $G = (V, E, c)$ mit $c : E \to \mathbb{R}_0^+$ die kürzesten Wege von einem Quellknoten s (source) zu allen anderen Knoten zu berechnen. Dabei sollen die folgenden Dinge für jeden Knoten i gespeichert werden: seine Entfernung $d(i)$ zum Knoten s sowie den Vorgängerknoten $V(i)$ auf dem kostenminimalen Weg zu i. Auf diese Art und Weise kann in linearer Zeit ein kostenminimaler Weg für jeden Knoten berechnet werden, da es einsichtig ist, dass für den Weg bis zum Vorgängerknoten ebenfalls der minimale Weg genommen werden sollte. Also wird wiederum der Vorgängerknoten des Vorgängers betrachtet usw. bis wir beim Knoten s sind.

Der Algorithmus von Dijkstra benutzt sowohl einen greedy-ähnlichen Ansatz als auch Ideen aus der dynamischen Programmierung. Greedy ist dabei, dass zuerst der zu s am nächsten liegende Knoten berechnet wird, dann der zweitnächste und so weiter, bis alle Knoten des Graph erfasst wurden.

Es wird im folgenden nun die folgende Schreibweise benutzt. Nach dem k-ten Schritt des Algorithmus kennen wir die Menge $A(k)$ der k Knoten mit der kleinsten Distanz von s, wobei die Distanzen zweier verschiedener Knoten auch gleich sein können. Ebenso kennen wir für jeden Knoten $i \notin A(k)$ die Länge $d_k(i)$ eines kostengünstigen (aber nicht zwingenderweise kostengünstigsten) Weges von s nach i, wobei jedoch nur Knoten aus der Menge $A(k)$ als Zwischenknoten benutzt werden dürfen. $V_k(i)$ ist in diesem Fall der vorletzte Knoten eines solchen Weges, weshalb $V_k(i) \in A(k)$ gilt. Im folgenden soll nun überlegt werden, wie aus der Menge $A(k)$ die Menge $A(k+1)$ berechnet werden kann.

Da s zu sich selbst am nächsten liegt, liegt zu Beginn die folgende Situation vor:

- $A(1) = \{s\}$, $d(s) = 0$, $V(s) = \emptyset$.
- Für alle $i \notin A(1)$: $d_k(i) = c(s, i)$, $V_k(i) = s$.

Es wird nun angenommen, dass die $A(k)$, $d(i)$ und $V(i)$ für $i \in A(k)$ sowie die $d_k(i)$ und $V_k(i)$ für $i \notin A(k)$ bekannt sind. Welcher Knoten sollte nun als nächstes zur Menge $A(k)$

hinzugefügt werden? Wir werden nun zeigen, dass dies der Knoten $j \notin A(k)$ sein sollte, der bezüglich $d_k(j)$ die niedrigsten Kosten hat. Es sei $(s = v_0, v_1, \ldots, v_l = j)$ ein kostengünstigster Weg von s zu j. Da $s \in A(k)$ und $j \notin A(k)$ gilt, muss es auf diesem Weg also eine Kante (v_i, v_{i+1}) geben, für die erstmalig $v_i \in A(k)$ und $(v_{k+1}) \notin A(k)$ gilt. Da alle Knoten von von v_0 bis v_i in $A(k)$ liegen und nur der Knoten v_{i+1} nicht, beträgt die Länge dieses Teilstücks also $d_k(v_{i+1})$. Da der Knoten j jedoch so gewählt wurde, dass seine Kosten unter allen d_k minimal sind, muss $d_k(v_{i+1}) \geq d_k(j)$ gelten. Damit haben wir also gezeigt, dass es keinen günstigeren Weg von s zu j gibt, als innerhalb von $A(k)$ bis zum Knoten $V_k(j)$ zu laufen und von dort zu j. Analog kann noch gezeigt werden, dass für alle anderen $j' \notin A(k)$ folgendes gilt:

$$d(j') \geq \min\{d_k(m) | m \notin A(k)\} \geq d_k(j)$$

Es wurde also gezeigt, dass der Knoten j unter allen Knoten $i \notin A(k)$ die kürzeste Distanz zum Knoten s besitzt (insgesamt also die $k+1$-kürzeste Distanz). Damit kann er zur Menge $A(k)$ hinzugefügt werden, um diese zur Menge $A(k+1)$ zu erweitern. Es werden also die folgenden Aktualisierungen für j vorgenommen:

$$A(k+1) = A(k) \cup \{j\}, \quad d(j) = d_k(j), \quad V(j) = V_k(j)$$

Damit sind wir jedoch noch nicht fertig. Bisher wurden für alle $i \notin A(k)$ die günstigsten Wege gespeichert, wenn nur Knoten aus $A(k)$ benutzt wurden. Jetzt ist jedoch der Knoten j neu hinzugekommen und es könnte damit notwendig sein, die Werte d_k und V_k für die Knoten $i \notin A(k+1)$ zu aktualisieren (j wurde ja schon aktualisiert). Es ist recht schnell einzusehen, dass in diesem Fall zwei Möglichkeiten bestehen. Entweder wird der neue Knoten j genutzt oder nicht. In letzterem Fall ändert sich an den d_k- und V_k-Werten nichts. Wird der Knoten j benutzt, so sollte der kürzeste Weg bis zu diesem Knoten gewählt werden und danach sofort zum Knoten i gelaufen werden. Es würde ja nichts bringen vorher noch zu einem anderen Knoten $j' \in A(k)$ (bzw. $j' \in A(k+1)$) zu laufen, da $d(j') \leq d(j)$ gilt und man also auch sofort den kürzesten Weg zu j' hätte wählen können. Damit ergibt sich die Bellmannsche Optimalitätsgleichung:

$$d_{k+1}(i) = \min\{d_k(i),\ d(j) + c(j,i)\}$$

Gilt $d_k(i) \leq d(j) + c(j,i)$, dann wird $V_{k+1}(i) = V_k(i)$, ansonsten $V_{k+1}(i) = j$.

Frage 1.23: Welche Laufzeit hat der Algorithmus von Dijkstra?
Im Prinzip müssen beim Algorithmus von Dijkstra die folgenden Schritte effizient implementiert werden:

- Finden des kleinsten d_k-Wertes.
- Aktualisierung von $A(k+1)$, $d(j)$ und $V(j)$.
- Aktualisierung von $d_k(i)$ und $V_k(i)$ für alle $i \notin A(k+1)$.

Es wird nun auf die einzelnen Schritte eingegangen, wobei der erste Schritt zuletzt behandelt wird, da dieser Verbesserungen zulässt. Der Graph sei durch seine Adjazenzlisten gegeben. Das Vergrößern der Menge $A(k)$ sowie die Aktualisierung von $d(j)$ und $V(j)$ gehen jeweils in $O(1)$. Wenn dies also für alle n Knoten durchgeführt wird, ergibt dies eine Laufzeit von $O(n)$. Der dritte Schritt, also die Aktualisierung der d_k- und V_k-Werte für alle Knoten, die sich nicht in $A(k+1)$ befinden ist in $O(m)$ möglich. Dies liegt daran, dass diese Werte nur für die Knoten

neu berechnet werden müssen, die adjazent zum neuen Knoten j sind. Insgesamt muss dies für m Kanten gemacht werden. Der teuerste Schritt scheint die Ermittlung des Elementes mit dem kleinsten d_k-Wert zu sein. Wenn eine doppelt verkettete Liste benutzt wird, über welche durch ein Array auch direkt auf das Element i zugegriffen werden kann, ist das Löschen eines Elementes in $O(1)$ möglich, das Suchen benötigt im k-ten Schritt jedoch $n - k - 1$ Schritte, wobei $1 \leq k \leq n - 1$ gilt. Insgesamt scheint die Suche also $\Theta(n^2)$ zu dauern. Durch den Einsatz von besseren Datenstrukturen kann diese Laufzeit jedoch verbessert werden. Die d_k-Werte werden im folgenden nun in einem *min*-Heap gespeichert. Der Aufbau eines solchen Heaps benötigt Zeit $O(n)$. Damit ist das kleinste Element immer an der Wurzel des Heaps verfügbar. Wird ein $d_k(i)$-Wert durch einen $d_k(i)$-Wert überschrieben, so ist dieser Wert höchstens kleiner als der vorherige Wert (sonst wäre ja nicht aktualisiert worden). Dann lässt sich in Zeit $O(\log n)$ die Heapeigenschaft wieder herstellen. Da dies höchstens m Mal passieren kann, ist die Gesamtlaufzeit also $O(n + m \log n)$.

2 Einige Beispiele zur Analyse von Algorithmen

2.1 Amortisierte Rechenzeit, Buchhaltermethode und Potenzialfunktionen

Frage 2.1: Was versteht man unter dem Begriff der amortisierten Rechenzeit?

Zuerst soll eine Datenstruktur auf n Daten eingeführt werden, welche die Operation U unterstützt. Die worst-case-Laufzeit dieser Operation betrage $t(n)$. Wenn nun m Aufrufe von U gemacht werden, wird bei einer normalen worst-case-Abschätzung eine obere Schranke von $m \cdot t(n)$ herauskommen. In diesem Fall würde also angenommen werden, dass *jeder* Aufruf von U auch tatsächlich die worst-case-Zeit benötigt. Dies muss jedoch nicht der Fall sein, da die Datenstruktur z.B. so beschaffen sein kann, dass eine teure Operation zur Folge hat, dass verschiedene andere Operationen wiederum billiger werden. Die amortisierte Rechenzeit wird im Folgenden nun mit $T(m, n)$ bezeichnet, wobei darunter die Rechenzeit einer ununterbrochenen Folge von m Aufrufen von U vestanden werden soll.

Frage 2.2: Gebe ein Beispiel für eine schlechte Abschätzung einer oberen Schranke (mit worst-case-Analyse)!

Als Beispiel wird hier die UNION-FIND-Datenstruktur angeführt, bei der die Objekte, die von $1, \ldots, n$ benannt sind, in einem Array verwaltet werden. Der Eintrag eines jeden Objekts ist die Menge, in der es sich momentan befindet. Die Mengen sind ebenfalls von $1, \ldots, n$ benannt und werden in einem Array verwaltet, wobei für jeden Eintrag die Größe der Menge gespeichert wird sowie eine Liste mit den Elementen in der Menge. Die Operation FIND kann in $O(1)$ durchgeführt werden und ist daher für die weitere Betrachtung uninteressant. Ein UNION ist dagegen schon interessanter. Die Arrayeinträge einer der Mengen müssen geändert werden, sowie die Mengengrößen und die Listen aktualisiert werden. Da die Kosten der Aktualisierung proportional zur Anzahl der Elemente der zu aktualisierenden Menge sind, werden bei dieser Implementierung die Einträge der kleineren Menge aktualisiert.

Zur Berechnung einer oberen Schranke für das worst-case-Verhalten könnte man sich nun zuerst überlegen, wie teuer ein UNION maximal werden kann. Es können offensichtlich höchstens die Kosten $\frac{n}{2}$ entstehen. Da maximal $n-1$ UNIONs möglich sind, ergibt dies eine worst-case-Abschätzung von $(n-1) \cdot \frac{n}{2} = \Theta(n^2)$. Andererseits wird das erste UNION nicht Kosten von $\frac{n}{2}$ sondern von Eins verursachen. Es könnte also sein, dass die Abschätzung der oberen Schranke sehr grob ist. Und tatsächlich kann auch z.B. ein Fall angegeben werden, bei dem die Laufzeit $\Theta(n \log n)$ beträgt.

Die durchgeführte Operationenfolge wird durch einen vollständigen, balancierten Baum mit $n = 2^k$ Knoten repräsentiert, wobei die inneren Knoten jeweils die Vereinigungen darstellen. Es gibt also $\frac{n}{2}$ Operationen mit Kosten Eins, $\frac{n}{4}$ mit Kosten 2, usw. Allgemein gibt es also $\frac{n}{2^m}$ Operationen mit Kosten 2^{m-1}, $1 \leq m \leq k$. Somit betragen die Gesamtkosten:

$$\sum_{m=1}^{k} \frac{n}{2^m} \cdot 2^{m-1} = \sum_{m=1}^{k} \frac{n}{2} = k \cdot \frac{n}{2} = \frac{1}{2} n \log n$$

Im Folgenden wird nun auf zwei Arten (Buchhaltermethode, Potenzialfunktion) gezeigt werden, dass die obere Schranke sich ebenfalls im Bereich von $n \log n$ bewegt.

Frage 2.3: Wie funktioniert die Buchhaltermethode?

Bei der Buchhaltermethode wird versucht, die Rechenschritte auf andere Kostenträger umzubuchen. Werden die Kosten der einzelnen Kostenträger dann summiert, so erhält man eine

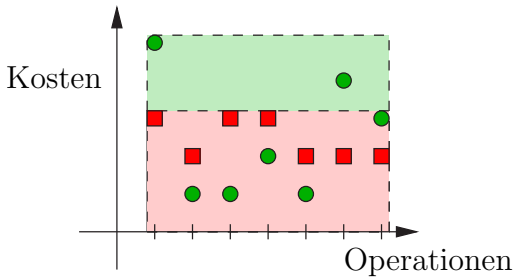


Abbildung 9: Prinzip der Buchhaltermethode

obere Schranke der Kosten. Diese obere Schranke kann auch abgeschätzt werden, indem die maximalen Kosten, die pro Kostenträger anfallen können, multipliziert werden mit der Anzahl der Kostenträger. Dabei wird versucht, die Kostenträger möglichst gleichmäßig zu belasten, damit die Abschätzung möglichst gut ist. Es wird also quasi versucht, den Effekt, dass einige Operationen teuer sein können und andere billig, möglichst gleichmäßig über alle Operationen zu „verschmieren“. Abbildung 9 versucht diese Idee zu verdeutlichen. Die Operationen sind an der x-Achse aufgetragen, die Kosten jeder Operation an der y-Achse.

Im UNION-FIND-Beispiel werden die Objekte die Kostenträger. Wird die Menge A mit der Menge B verschmolzen, und gilt $|A| \leq |B|$, dann verursacht dies Kosten $|A|$. Verursacher dieser Kosten sind die Objekte in der Menge A, deren Arrayeinträge ja geändert werden müssen. Daher bekommt jedes Objekt eine Kosteneinheit. Wir versuchen nun, abzuschätzen, wieviele Kosteneinheiten ein Objekt bekommen kann. Wenn ein Objekt in eine neue Menge kommt, dann ist diese Menge mindestens doppelt so groß, wie die Menge in der es vorher war. Das bedeutet, dass sich ein Objekt nach t Verschmelzungen in einer Menge befindet, die aus mindestens 2^t Objekten besteht. Da es am Ende höchstens in einer Menge mit n Elementen sein kann, muss also $2^t \leq n$ gelten und somit $t \leq \log n$. Da t ganzzahlig ist, gilt sogar $t \leq \lfloor \log n \rfloor$. Dies gilt für alle n Objekte, also können die Kosten höchstens $n \lfloor \log n \rfloor$ betragen, was schon eine bessere obere Schranke als $(n-1)\frac{n}{2}$ darstellt.

Frage 2.4: Wie funktioniert der Beweis mit der Potenzialfunktion?

Bei der Methode der Potenzialfunktion wird der Fortschritt des Algorithmus an einer vorgegebenen Funktion gemessen. Im UNION-FIND-Beispiel wird das Potenzial nach m UNION-Operationen wie folgt berechnet:

$$\Phi = \sum_{i=1}^{n} \varphi(i) - \sum_{j=1}^{m} c(j)$$

Dabei ist $\varphi(i)$ die größte Zahl t, so dass die das Objekt i enthaltende Menge mindestens 2^t Elemente hat. Die Funktion $c(j)$ gibt die Kosten der j-ten Operation an. Zu Beginn startet das Potenzial mit 0, denn $\varphi(i)$ ist zu Beginn das größte t, so dass die das Objekt i enthaltende Menge mindestens $2^0 = 1$ Elemente enthält. Desweiteren wird das Potenzial nicht gesenkt, denn für alle Objekte der kleineren Menge erhöht sich der $\varphi(i)$-Werte um mindestens 1. Die Kosten in diesem Schritt entsprechen ebenfalls der Anzahl der „betroffenen“ Objekte. Also gilt $\Phi \geq 0$. Dies impliziert jedoch, dass die Kosten für alle Operationen nicht höher sein können als die Summe der maximalen $\varphi(i)$-Werte für jedes Objekt. Der maximale φ-Wert eines jeden Objektes beträgt $\lfloor \log n \rfloor$, nämlich genau am Ende, wenn alle Objekte sich in einer Menge befinden. Also ist die Summe höchstens $n \lfloor \log n \rfloor$.

2.2 Eine Analyse der Pfadkomprimierung bei UNION-FIND-Datenstrukturen

Frage 2.5: Welche Datenstruktur sollte für schnelle UNIONs benutzt werden? Wie kann bei den FINDs gespart werden?

Bei der UNION-FIND-Datenstruktur mit den schnellen UNIONs werden die Mengen als wurzelgerichtete Bäume dargestellt. An den Wurzeln der Bäume steht jeweils die Mächtigkeit und der Name der Menge. Bei einem UNION wird einfach die Wurzel des Baumes der Menge mit der kleineren Mächtigkeit auf die Wurzel des Baumes mit der größeren Mächtigkeit gerichtet und die Mächtigkeit in dieser Wurzel entsprechend aktualisiert. Ein UNION kann bei dieser Vorgehensweise in $O(1)$ durchgeführt werden.

Bei einem FIND(x) muss vom entsprechenden Objekt x bis zur Wurzel gelaufen werden. Werden dabei die Knoten $x = v_k, \ldots, v_1, v_0$ besucht, so dauert dies $O(k+1)$. Die Laufzeit ist also abhängig davon, wie weit das Objekt von der Wurzel entfernt ist. Es ist nun jedoch so, dass bei lang dauernden UNIONs nebenbei auch sehr viele Informationen über die anderen Objekte in der Menge gefunden werden. So wissen wir nach dem obigen FIND, dass sich die Objekte $v_{k-1}, \ldots, v_1$ ebenfalls in der Menge v_0 befinden. Diese Objekt befinden sich auch immer noch in dieser Menge, wenn wir diese Knoten direkt auf die Wurzel zeigen lassen. Bei zukünftigen FINDs wird dann Zeit gespart.

Frage 2.6: Wieviele Elemente hat ein UNION-FIND-Baum der Höhe h mindestens?

Ein UNION-FIND-Baum der Höhe h, d.h. der längste Weg von einem Knoten zur Wurzel hat die Länge h, besitzt mindestens 2^h Knoten. Der Beweis wird per Induktion geführt. Für $h = 0$ besitzt ein Baum genau $2^0 = 1$ Knoten und die Behauptung stimmt somit. Es sei nun T der Baum, der unter allen Bäumen der Höhe h die minimale Anzahl an Knoten hat. Dieser Baum muss aus dem UNION eines Baumes T_1 und T_2 entstanden sein, wobei nun o.B.d.A. angenommen wird, dass $size(T_1) \leq size(T_2)$ gelte, d.h. der Baum T_1 wurde unter die Wurzel des Baumes T_2 gehängt. Es wird nun zuerst überlegt, wie hoch der Baum T_1 sein muss. Wenn seine Höhe mindestens h betragen würde, müsste der Baum T ebenfalls höher als h sein, da T_1 ja unter dessen Wurzel gehängt wurde. Somit gilt also Höhe$(T_1) \leq h - 1$. Die Höhe kann jedoch auch nicht echt kleiner als $h - 1$ sein, da ansonsten der Baum T_2 schon vorher die Höhe h gehabt haben müsste. In diesem Fall könnte der Teilbaum T_1 jedoch in T weggelassen werden und T wäre somit nicht unter allen Bäumen der mit der geringsten Knotenzahl. Also muss Höhe$(T_1) = h - 1$ gelten. Nach Induktionsvoraussetzung gilt damit $size(T_1) \geq 2^{h-1}$. T_1 wurde unter T_2 gehängt und somit gilt also $size(T_2) \geq size(T_1) \geq 2^{h-1}$. Also gilt auch $size(T_2) + size(T_1) \geq 2 \cdot 2^{h-1} = 2^h$.

Frage 2.7: Was wird im Folgenden unter dem Rang eines Elementes verstanden?

Im Folgenden werden zwei Datenstrukturen betrachtet und miteinander verglichen. Auf diesen Datenstrukturen werde jeweils eine Befehlsfolge r ausgeführt, die aus $n - 1$ UNIONs und m FINDs bestehe. Dabei ist U_r^{ohne} die Datenstruktur, die ohne Pfadkomprimierung nach der Befehlsfolge r entsteht, und U_r^{mit} die Datenstruktur, die mit Pfadkomprimierung entsteht. Allgemein seien U_i^{ohne} und U_i^{mit} die Datenstrukturen nach i Befehlen. Zu Beginn sind beide Datenstrukturen U_0^{ohne} und U_0^{mit} gleich, da sie beide aus den n einelementigen Mengen bestehen. Es lassen sich nun die folgenden zwei Beobachtungen machen:

1. Ein FIND-Befehl mit Pfadkomprimierung ändert zwar das Aussehen des Baumes, nicht jedoch die Anzahl der Knoten oder die Wurzel.

2. Die Entscheidung, welcher Baum bei einem UNION in den andern gehangen wird, hängt nur von der Größe der beteiligten Bäume ab.

Dies bedeutet, dass wenn es in U_i^{ohne} einen Baum mit Wurzel w gibt, der die Menge S darstellt, es auch in U_i^{mit} einen Baum mit Wurzel w gibt, der ebenfalls die Menge S darstellt. Am Ende bestehen also auch beide Datenstrukturen U_r^{ohne} und U_r^{mit} aus einem Baum.

Der Rang ist nun über den Baum U_r^{ohne} definiert. Die Größe Rang(x) gibt für einen Knoten x die Länge des längsten Weges von einem Blatt zum Knoten x in der Datenstruktur U_r^{ohne} an. Wie relativ leicht ersichtlich ist, gilt in U_r^{ohne} die „Rangwachstumseigenschaft“:

$$v \text{ ist Vater von } w \Rightarrow \text{Rang}(v) > \text{Rang}(w)$$

Frage 2.8: Zeige, dass die Rangwachstumseigenschaft auch in Bäumen mit Path-Compression gilt!

Die „Rangwachstumseigenschaft“ wird in einer späteren Phase des Beweises als Argument benutzt, daher muss nun bewiesen werden, dass sie auch in beliebigen Mengen der Datenstruktur U_i^{mit}, für die wir ja eine Abschätzung haben möchten, gilt.

Zu Beginn (U_0^{mit}) gilt die Rangwachstumseigenschaft sicher, da alle Mengen einelementig sind und damit keinen Vater besitzen. Es muss also einen späteren Befehl geben, bei dem die Rangwachstumseigenschaft das erste Mal nicht mehr gilt. Es gibt also ein i, so dass U_i^{mit} nicht mehr die Eigenschaft erfüllt, diese jedoch in U_{i-1}^{mit} noch gilt. Es soll nun überlegt werden, während welcher Operation die Eigenschaft verletzt worden sein könnte. Ein FIND kann dies nicht gewesen sein, da in diesem Fall nur Knoten direkt an die Wurzel gehangen werden. Da in U_{i-1}^{mit} die Rangwachstumseigenschaft jedoch noch galt, muss sie dann in U_i^{mit} auch noch gelten. Also ist die Eigenschaft anscheinend durch ein UNION verletzt worden. Die Datenstruktur U_i^{mit} ist durch ein UNION in der Datenstruktur U_{i-1}^{mit} entstanden. Es wird nun angenommen, dass der Baum mit der Wurzel t_2 an die Wurzel t_1 des anderen Baumes gehangen wurde. Da die Rangwachstumseigenschaft verletzt ist, muss also $\text{Rang}(t_1) < \text{Rang}(t_2)$ gelten, da sich nur die Vater-Sohn-Verhältnisse der beiden Knoten t_1 und t_2 geändert haben. Wie jedoch vorhin gezeigt wurde, würde auch in U_i^{ohne} der Knoten t_1 der neue Vater von t_2 werden. Da jedoch ohne die Pfadkomprimierung die Vater-Sohn-Eigenschaften nicht mehr geändert werden, ist t_1 also auch noch am Schluss (U_r^{ohne}) der Vater von t_2. Nach Definition des Rangs gilt dann jedoch $\text{Rang}(t_1) \geq \text{Rang}(t_2)$, was im Widerspruch zur Annahme steht, dass $t_1 < t_2$ gilt.

Frage 2.9: Wieviele Elemente mit Rang r kann es höchstens geben?

Es kann höchstens $\frac{n}{2^r}$ Elemente mit Rang r geben. Der Beweis ist relativ simpel. Es wurde schon gezeigt, dass ein UNION-FIND-Baum, dessen längster Weg h beträgt mindestens 2^h Knoten besitzt. Ist der Rang eines Knotens r, ist die Tiefe seines Teilbaum also r. Damit besitzt der Teilbaum mindestens 2^r Knoten. Da es (in U_r^{ohne}) keinen Knoten mit Rang r gibt, dessen Vorgänger ebenfalls den Rang r besitzt, werden also auch keine Knoten doppelt gezählt. Damit kann es höchstens $\frac{n}{2^r}$ Knoten mit Rang r geben, da es ansonsten mehr als n Knoten geben würde ($\frac{n}{2^r} \cdot 2^r = n$).

Frage 2.10: Wie ist die Zweierturm-Funktion definiert? Warum wird sie eingeführt?

Die Zweierturm-Funktion $F(i)$ und ihre „Umkehrfunktion“ $\log^* n$ sind wie folgt definiert:

$$F(0) = 1,\ F(i) = 2^{F(i-1)} \text{ und } \log^* n = \min\{k | F(k) \geq n\}$$

Da $F(i)$ einen Turm von Exponentationen zur Basis 2 bildet, wächst die Funktion rasend schnell. Dementsprechend wächst $\log^* n$ nur sehr langsam. Die Funktion $\log^* n$ kann auf zwei Arten interpretiert werden. Zum einen, entsprechend ihrer Definition, als kleinstes k, so dass die Zweierturm-Funktion gerade einen Wert größer oder gleich n ergibt. Oder aber als die Anzahl, wie oft die Zahl n logarithmiert werden muss, bis $a \leq 1$ gilt.

Das Ziel besteht nun darin, zu zeigen, dass es zwar FIND-Befehle geben kann, die Kosten von $\Theta(\log n)$ verursachen, bei einer Folge von $n-1$ UNIONs und m FINDs jedoch nur Kosten von $O((m+n)\cdot \log^* n)$ entstehen. Dabei wird die Buchhaltermethode verwendet. Die Aufgabe der Zweierturm-Funktion bzw. der $\log^* n$-Funktion besteht in diesem Zusammenhang darin, eine günstige Aufteilung der Kosten zu ermöglichen. Damit dies möglich ist, werden nun verschiedene Ränge jeweils verschiedenen Ranggruppen zugeordnet. Ein Knoten mit Rang r wird dabei der Ranggruppe $\log^* r$ zugeordnet. Die Knoten mit den Rängen $\{5, \ldots, 16\}$ werden also z.B. der Ranggruppe 3 zugeordnet.

Frage 2.11: Wie werden die Kosten umgebucht?
Im Folgenden wird jeweils der Aufruf FIND(x) betrachtet, bei dem der Weg $x = v_k \to v_{k-1} \to \cdots \to v_1 \to v_0$ abgelaufen wird. Die Kosten des Aufrufs werden vollständig auf verschiedene Konten verteilt. Zuerst einmal wird jedem FIND-Befehl ein Konto zugeordnet, womit es es m solcher Konten gibt. Ebenso erhält jeder Knoten ein Konto. Also gibt es n solcher Konten.

Die Kosten von $k+1$ des Aufrufs FIND(x) müssen vollständig auf die Konten verteilt werden. Dabei wird nun die folgende Regel benutzt. Eine Kosteneinheit wird quasi für den Aufruf von FIND auf dessen Konto eingezahlt. Danach wird für jede Kante $v_i \to v_{i-1}$ eine Kosteneinheit vergeben. Dabei wird die Einheit für die Kante $v_1 \to v_0$ immer auf das FIND-Konto eingezahlt. Für alle anderen Kanten $v_i \to v_{i-1}$ hängt das Konto, auf dem die Einheit eingezahlt wird, von den Ranggruppen der beteiligten Knoten ab:

Ranggruppe verschieden: Sind die Ranggruppen des Knoten v_i und v_{i-1} verschieden, so wird die Einheit auf das Konto des jeweiligen FIND-Befehls eingezahlt.

Ranggruppe gleich: Im Falle, dass die Ranggruppen gleich sind, wird auf das Konto des Knotens v_i eingezahlt.

Die Gesamtkosten entstehen, wie es bei der Buchhaltermethode üblich ist, indem die Beträge alle Konten aufaddiert werden. Da dies nicht trivial ist, werden wir nun versuchen, die Belastung eines jeden einzelnen Kontos möglichst genau abzuschätzen. Zuerst wird die Abschätzung für ein FIND-Konto vorgenommen und danach für ein Knoten-Konto.

Jedes FIND verursacht durch seinen Aufruf schon die Einzahlung einer Einheit auf sein Konto. Wird die Kante $v_1 \to v_0$ abgelaufen, verursacht dies ebenfalls eine Einzahlung. Weitere Einzahlungen entstehen durch den Wechsel von Ranggruppen. Wieviele Ranggruppen kann es bei n Elementen nun höchstens geben? Es kann höchstens $\log^* n + 1$ Ranggruppen geben. Da wir in einer dieser Gruppen starten und die Rangwachstumseigenschaft gilt, können wir also nur noch maximal $\log^* n$-mal wechseln. Also beträgt die maximale Einzahlung auf ein FIND-Konto $\log^* n + 2$. Für alle m Konten belaufen sich die maximal entstehenden Kosten also auf $m \cdot (\log^* n + 2) = O(m \cdot \log^* n)$.

Im Falle der Knoten-Konten ist es etwas komplizierter. Ein Knoten bekommt eine Kosteneinheit auf sein Konto eingezahlt, wenn er sich auf dem Weg von FIND(x) befindet. Danach wird er an die Wurzel v_0 gehängt. Da er vorher nicht unter diesem Knoten hing und die Rangwachstumseigenschaft gilt, muss der Rang seines neuen Vaters also mindestens um Eins

größer sein als der Rang des alten Vaters. Dieser Knoten kann bezüglich seines Kontos jedoch nur so lange Kosten verursachen, so lange sich sein neuer Vater nicht in einer anderen Ranggruppe befindet. Befindet sich das Element in Ranggruppe g, so liegt sein Rang in der Menge $\{F(g-1)+1, \ldots, F(g)\}$. Die Anzahl der Elemente in dieser Menge kann mit $F(g) - (F(g-1)+1) + 1 = F(g) - F(g-1) \leq F(g)$ Elementen abgeschätzt werden. Also kann sich der Rang des Vaters dieses Elementes auch nur höchstens so oft erhöhen. Danach muss der Vater in einer anderen Ranggruppe liegen. Damit haben wir die Kosten, die jedes Element in einer Ranggruppe höchstens verursachen kann. Doch wieviele Elemente $N(g)$ hat eine Ranggruppe g. Dies wird wie folgt abgeschätzt:

$$\begin{aligned} N(g) &\leq \sum_{F(g-1)+1 \leq r \leq F(g)} n2^{-r} \leq n2^{-F(g-1)} \underbrace{\left(\frac{1}{2} + \frac{1}{4} + \ldots\right)}_{\text{Harm. Reihe}} \\ &\leq n2^{-F(g-1)} = \frac{n}{2^{F(g-1)}} = \frac{n}{F(g)} \end{aligned}$$

Somit gibt es in jeder Ranggruppe $N(g)$ Elemente die jeweils höchstens Kosten von $F(g)$ verursachen. Insgesamt also:

$$N(g) \cdot F(g) \leq \frac{n}{F(g)} \cdot F(g) = n$$

Somit werden in einer Ranggruppe also höchstens Kosten von n verursacht. Es gibt höchstens $\log^* n + 1$ Ranggruppen, womit schlussendlich maximale Kosten von $n \cdot (\log^* n + 1)$ entstehen.

Jetzt sind die maximalen Belastungen aller Konten klar. Insgesamt ergibt sich also folgendes Ergebnis: Eine Folge von $n-1$ UNION-Befehlen und m FINDs kann in Zeit $O((m+n) \cdot \log^* n)$ durchgeführt werden.

2.3 String Matching - ein Mustererkennungsproblem

Frage 2.12: Worum geht es beim String Matching?

Das String Matching Problem besteht darin, einem String $S = (s_1, \ldots, s_n)$ darauf zu untersuchen, ob er ein Muster (Pattern) $P = (p_1, \ldots, p_m)$ enthält. Für dieses Problem werden nun drei Lösungen vorgestellt. Eine naive Lösung mit schlechter Laufzeit, eine Lösung mit einer schnellen Laufzeit, aber unter Umständen hoher Preprocessing-Zeit und zuletzt der Algorithmus von Knuth, Morris und Pratt, der ein lineare Preprocessing- und Laufzeit aufweist.

Frage 2.13: Wie funktioniert ein einfacher Algorithmus für das String Matching Problem? Welche Laufzeit hat dieser?

Ein naiver Algorithmus überprüft für jede Startposition $i \in \{0, \ldots, n-m\}$, ob für alle $j \in \{1, \ldots, m\}$ die Beziehung $p_j = s_{i+j}$ gilt. Die Laufzeit dieser Methode beträgt $O((n-m) \cdot m) = O(nm - m^2) = O(nm)$. Ein Beispiel, in dem die worst-case-Rechenzeit erreicht wird, ist der String $S = (a, a, \ldots, a)$, in dem das Pattern $P = (a, a, \ldots, b)$ gesucht wird.

Frage 2.14: Wie funktioniert der NFA für das String Matching Problem? Wie kann er genutzt werden? Welche Laufzeit entsteht dabei?

Das String Matching Problem kann auch als das Wortproblem für die reguläre Sprache $L = \Sigma^* p_1 \ldots p_m \Sigma^*$ aufgefasst werden. Für diese Sprache kann ein nichtdeterministischer Automat

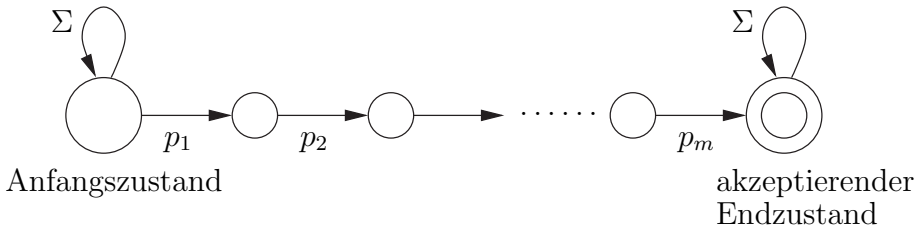


Abbildung 10: Der NFA A zur Erkennung des Musters $P = (p_1, \ldots, p_m)$

entworfen werden, der diese Sprache erkennt. Dieser Automat ist in Abbildung 10 dargestellt.

Dieser nichtdeterministische Algorithmus kann also in n Schritten entscheiden, ob $S \in L$ ist. Da ein nichtdeterministischer Algorithmus nicht implementiert werden kann, muss aus dem NFA A ein DFA A' erzeugt werden. Dieser DFA kann jedoch unter Umständen exponentiell viele Zustände (in m) benötigen, wodurch die Preprocessing-Zeit zur Erstellung des DFA ebenfalls exponentiell lange dauern kann. Der Algorithmus von Knuth, Morris und Pratt stellt einen guten Kompromiss dar und wird als nächstes vorgestellt.

Frage 2.15: Wie funktioniert der KMP-Algorithmus?
Der Algorithmus von Knuth, Morris und Pratt, auch KMP-Algorithmus genannt, besteht aus zwei Schritten. Der erste Schritt besteht aus einem Preprocessing, in dem sich das zu suchende Muster etwas genauer angeschaut wird. Warum dies gemacht wird, wird gleich erläutert. Es sei schonmal festgehalten, dass dieser Schritt eine Laufzeit von $O(m)$ hat. Im zweiten Schritt wird das Muster dann im String gesucht. Aufgrund der Preprocessing-Phase muss die Suche bei einer Nichtübereinstimmung nicht immer wieder ganz von vorne begonnen werden. Diese Phase benötigt Zeit $O(n)$. Der Algorithmus ist mit einer Gesamtlaufzeit von $O(n + m)$ also ein guter Kompromiss zwischen dem naiven Algorithmus und dem nichtdeterministischen Algorithmus.

Frage 2.16: Was wird im Preprocessing-Schritt des KMP-Algorithmus gemacht?
Im Preprocessing-Schritt werden die sogenannten Fehlerkanten für das Suchmuster berechnet. Bevor dies erläutert wird, soll zuerst einmal das Ziel vorgestellt werden. Wir wollen nach dem Lesen des i-ten Buchstaben des Strings S wissen, ob $(s_1, \ldots, s_i)$ das Muster schon enthält. Wenn dies nicht der Fall ist, möchten wir gerne wissen, wieviel des Patterns denn bis jetzt schon gelesen wurde. Oder anders gesagt: Wir wollen den längsten Suffix von $(s_1, \ldots, s_i)$ kennen, der auch Präfix von P ist. Dieser Suffix kann auch die Länge Null haben. Um dies zu speichern wird ein Graph $V = \{0, \ldots, m\}$ verwendet, bei dem der Zustand $j \geq 1$ die größte Zahl bezeichnen soll, so dass folgendes gilt:

$$(s_{i-j+2}, \ldots, s_i) = (p_1, \ldots, p_{j-1})$$

Wenn wir uns in diesem Graphen also z.B. im Zustand 4 befinden, so bedeutet dies, dass die letzten drei Buchstaben des bisher gelesenen Strings mit den ersten drei Buchstaben übereinstimmen. Bei einem Wechsel in den (nicht vorhandenen) Zustand $m + 1$ ist das Pattern gefunden worden. Wurde das Pattern bis jetzt noch nicht gefunden, wird der Buchstabe s_{i+1} mit der aktuellen Position j im Pattern verglichen. Das Problem besteht bei einer Nichtübereinstimmung in der Berechnung des neuen j. In der Preprocessing-Phase werden daher die

Fehlerkanten berechnet, die angeben, wie sich das neue j aus dem alten ergibt. Das ganze soll an einem Beispiel verdeutlicht werden. Gegeben sei das folgende Pattern P und die Beispielstrings S_1, S_2 und S_3:

P: EINMALEINENEFFIZIENTENALGORITHMUSENTWERFEN

S_1: ESWAREINMALEINEHEXE

S_2: ESWAREINMALEINMENSCH

S_3: ESWAREINMALEINRIESE

Alle Strings fangen mit „ESWAREINMALEIN“ an. Wenn dieser Teilstring gelesen ist, ist in allen drei Fällen „EINMALEIN“ der längste Suffix, der auch Präfix von P ist. In allen drei Fällen wird also der nächste Buchstabe betrachtet. Im String S_1 passt auch dieser überein, nicht jedoch in den Fällen S_2 und S_3. Allerdings soll nun nicht wieder ganz von vorn angefangen werden. Schließlich haben wir in allen drei Fällen zuletzt den Teilstring „EIN“ gelesen, der ein Präfix von P ist. Vielleicht fängt ja hier jetzt der „richtige“ Teilstring an, der das Pattern enthält. Suffixe von S, die Präfixe von P sind, werden gute Suffixe genannt. Im Algorithmus wird nun Schritt für Schritt geschaut, ob ein guter Suffix auch mit dem nächsten Stringbuchstaben ein guter Suffix bleibt. Ist dies nicht der Fall liegt es also nahe, zu testen, ob der zweitlängste gute Suffix sich mit dem aktuellen Buchstaben im String zu einem guten Suffix erweitern lässt, usw.

Wenn wir für j nicht erfolgreich sind, bedeutet dies, dass der Suffix $(s_{i-j+2}, \ldots, s_i)$ zwar mit dem Präfix $(p_1, \ldots, p_{j-1})$ übereinstimmt, der Buchstabe s_{i+1} jedoch nicht mehr mit p_j. Wenn wir im String nun den nächsten guten Suffix suchen, ist dieser kürzer und muss ebenfalls am Schluss von $(s_{i-j+2}, \ldots, s_i)$ liegen. Wie wir gesehen haben, ist der String $(s_{i-j+2}, \ldots, s_i)$ jedoch identisch mit $(p_1, \ldots, p_{k-1})$. Es reichen also die Informationen, die im Pattern P enthalten sind und die Betrachtungen können völlig unabhängig vom String S geschehen. Es wird also das größte $k < j$ gesucht, für das $(p_1, \ldots, p_{k-1})$ mit $(p_{i-k+1}, \ldots, p_{j-1})$ übereinstimmt. Oder anders gesagt: Es wird der längste Suffix von $(p_1, \ldots, p_{j-1})$ gesucht, der auch Präfix ist.

Frage 2.17: Wie sind die Fehlerkanten definiert?

Wie etwas weiter oben schon erläutert wurde, möchten wir gerne in Abhängigkeit von der aktuellen Position j im Pattern wissen, welchen Buchstaben des Pattern wir bei einer Nichtübereinstimmung als nächstes mit dem aktuellen Stringbuchstaben vergleichen können. Implizit ist dadurch also der nächstbeste gute Suffix gegeben. Dies wird über die Fehlerkanten gemacht, die ihren Namen deshalb haben, da sie im Fall benötigt werden, wenn ein guter Suffix mit dem nächsten Buchstaben des Strings nicht weiter zu einem guten Suffix ausgebaut werden kann. Sie sind wie folgt definiert:

$$F(j) = \max\{0 \leq k \leq j-1 \mid (p_1, \ldots, p_{k-1}) = (p_{j-k+1}, \ldots, p_{j-1})\}$$

Zuerst muss überlegt werden, ob diese Definition für alle Werte sinnvoll ist. Für $k = 0$ und $k = 1$ sind beide Vektoren leer und stimmen somit überein. Somit gilt also $F(1) = 0$ und $F(2) = 1$. Für alle anderen $j \geq 2$ gilt $F(j) \geq 1$ und $F(j) < j$. Für Knoten 1 wird der leere Suffix untersucht, ob er verlängert werden kann. Ist dies nicht der Fall, stimmt also der aktuell untersuchte Buchstabe des String nicht mit dem ersten Buchstaben des Pattern überein und es muss beim nächsten Buchstaben von vorne angefangen werden. Der Zustand 0 ist dazu da, um dies zu erkennen.

Frage 2.18: Gebe den Query-Algorithmus für den KMP-Verfahren an! Welche Laufzeit hat er?

Als Eingabe benötigt der Algorithmus den String $S = (s_1, \ldots, s_n) \in \Sigma^n$ und das Pattern $P = (p_1, \ldots, p_m) \in \Sigma^m$. Weiter werden deren Längen n und m benutzt, sowie die Fehlerkanten $F(1), \ldots, F(m)$. Die Ausgabe ist „Erfolg“ oder „Kein Erfolg“, je nachdem, ob der String S das Pattern P enthält. Der Algorithmus ist in Algoritmus 2 beschrieben.

Algorithmus 2 Der Query-Algorithmus für das KMP-Verfahren

1: $i = 1$; $j = 1$;
2: **while** $i \leq n$ **do**
3: Solange $j \neq 0$ und $p_j \neq s_i$, setze $j = F(j)$.
4: Falls $j = m$: Gebe „Erfolg“ aus und **STOP**.
5: Falls $j \neq m$, setze $i = i + 1$ und $j = j + 1$.
6: **end while**
7: Gebe „Kein Erfolg“ aus und **STOP**.

Die Laufzeit des Algorithmus ist $O(n)$. Die Variable i wird nur erhöht und nie erniedrigt. Also kann sie in Schritt 5 nur n-mal erhöht werden, bevor die Schleife nicht mehr ausgeführt wird und der Algorithmus stoppt. Bleibt nur die Frage, wie oft die while-Schleife in Schritt 3 ausgeführt werden kann. Die Variable j wird ebenfalls nur in Schritt 5 erhöht. Also kann sie ebenfalls maximal n-mal erhöht werden. Da j jedoch auch nicht negativ werden kann, kann Schritt 3 ebenfalls nur höchstens $(n + 1)$-mal ausgeführt werden. Damit ergibt sich eine Gesamtlaufzeit von $O(n)$.

Frage 2.19: Wie können die Fehlerkanten effizient berechnet werden? Welche Laufzeit ergibt sich?

Die Fehlerkanten werden inkrementell berechnet, wobei unter Umständen vorherige Ergebnisse benutzt werden. Es seien $F(1), \ldots, F(i-1)$ schon bekannt. Dann ist für $j = F(i-1)$ der Teilstring $(p_1, \ldots, p_{j-1}) = (p_{i-j}, \ldots, p_{i-2})$ der längste Suffix von $(p_1, \ldots, p_{i-2})$, der auch Präfix ist. Zur Berechnung von $F(i)$ wird das Teilpattern $(p_1, \ldots, p_{i-1})$ näher betrachtet. Dabei gibt es nun zwei Fälle zu unterscheiden:

Fall 1: Es gilt $p_{i-1} = p_j$. Damit ist also $(p_1, \ldots, p_j) = (p_{i-j}, \ldots, p_{i-1})$ der längste passende Suffix und es gilt $F(i) = F(i-1) + 1$.

Fall 2: Es gilt $p_{i-1} \neq p_j$. Die Situation sieht dann wie folgt aus:

$$\begin{array}{lccccccc}
 & p_1 & \cdots & p_{j-j'+1} & \cdots & p_{j-1} & p_j \\
 & \| & & \| & & \| & \nparallel \\
 & p_{i-j} & \cdots & p_{i-j'} & \cdots & p_{i-2} & p_{i-1} \\
 & & & \| & & \| & ? \\
\text{Gesucht:} & & & p_1 & \cdots & p_{j'-1} & p_j
\end{array}$$

Der Suffix und Präfix stimmen also für $F(i-1)$ überein. Lediglich die nächsten Buchstaben p_j und p_{i-1} stimmen nicht überein. Gesucht ist also der nächstlängste Suffix $(p_1, \ldots, p_{j'-1})$ dessen Nachfolger $p_{j'}$ wir gegen p_{i-1} vergleichen können. Dieser Nachfolger ist jedoch gerade durch $j' = F(j)$ gegeben. Wenn auch dieser nicht klappt, finden wir den wiederum nächstlängeren als $j'' = F(j')$, usw.

Algorithmus 3 Der Algorithmus zur Berechnung der Fehlerkanten

1: Initialisierung: $F(1) = 0$; $F(2) = 1$; $i = 3$;
2: **while** $i \leq m$ **do**
3: $\quad j = F(i-1)$;
4: $\quad$ Solange $j \neq 0$ und $p_j \neq p_{i-1}$, setze $j = F(j)$;
5: $\quad F(i) = j + 1$; $i = i + 1$;
6: **end while**

Mit diesen Erkenntnissen ergibt sich Algorithmus 3.

Die Laufzeit des Algorithmus beträgt $O(m)$. Die Schleife wird $m - 2$ mal aufgerufen, da für jedes m eine Fehlerkante berechnet werden muss und i nur in Schritt 5 jedesmal um Eins erhöht wird. Es bleibt also auch hier nur noch die Frage, wie sich die Schleife in Zeile 4 auf die Laufzeit auswirken kann. Zur Analyse wird die Buchhaltermethode benutzt, bei der in diesem Fall zwei Konten verwaltet werden. In eines wird für jeden Erfolg eingezahlt, d.h. wenn $p_j = p_{i-1}$ oder $j = 0$ gilt. Das andere Konto zählt die Misserfolge, wenn also $p_j \neq p_{i-1}$ und $j \neq 0$ gilt. Da für jedes i spätestens bei $j = 0$ ein Erfolg auftritt, werden also höchstens $m - 2$ Einheiten auf das „Erfolgs-Konto“ eingezahlt. Weiter kann beobachtet werden, dass j bei $j = F(2) = 1$ startet und bei jedem Misserfolg mindestens um Eins erniedrigt wird, wobei es jedoch nicht kleiner als Null werden kann. Erhöht werden kann es nur in Zeile 3, wenn ihm der Wert $F(i-1)$ zugewiesen wird. Es kann sich also nur höchstens $(m-2)$-mal erhöhen. Es bleibt nur noch die Frage, um wieviel es maximal erhöht werden kann. Nachdem der Variable j in Zeile 4 der Wert $F(j)$ zugewiesen wird, wird $F(i)$ der Wert $j+1$ zugewiesen und i um Eins auf $i + 1$ erhöht. Da j danach in Zeile 3 den Wert $F(i-1)$ zugewiesen bekommt, entspricht dies dem Wert $F(i) = j + 1$, da i ja vorher erhöht wurde. Somit wächst das j bei jedem mal höchstens um Eins. Insgesamt kann es also nur um $m-2$ wachsen und dementsprechend kann auch nur höchstens so oft etwas abgezogen werden. Damit ergibt sich dann die Gesamtlaufzeit von $O(m)$.

3 Dynamische Programmierung

3.1 Ein Rückblick auf bekannte Beispiele

Frage 3.1: Motiviere die dynamische Programmierung!

Bei rekursiven Algorithmen kann es schnell passieren, dass Teilprobleme mehrfach gelöst werden. Ein sehr einfaches Beispiel ist dabei die rekursiv definierte Formel zur Berechnung der Fibonacci-Zahlen. Sie ist wie folgt definiert:

$$\text{Fib}(0) = 0;\ \text{Fib}(1) = 1;\ \text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$$

Wird in der Formel für $\text{Fib}(n)$ der Term $\text{Fib}(n-1)$ entsprechend der Definition ersetzt, so ergibt sich:

$$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2) = \underbrace{\text{Fib}(n-2) + \text{Fib}(n-3)}_{\text{Fib}(n-1)} + \text{Fib}(n-2)$$

$\text{Fib}(n-2)$ wird also mehrmals berechnet. Wird die Fibonacci-Zahl $\text{Fib}(n)$ naiv mit einem rekursiven Algorithmus berechnet, so ergeben sich $\text{Fib}(n)$ nicht triviale Additionen (kein Summand ist 0). Die Rechenzeit ist damit exponentiell, da $\text{Fib}(n)$ auch berechnet werden kann als:

$$\text{Fib}(n) = \frac{1}{\sqrt{5}} \cdot \left(\left(\frac{1+\sqrt{5}}{2} \right)^{n+1} - \left(\frac{1-\sqrt{5}}{2} \right)^{n+1} \right)$$

Mit etwas Nachdenken kommt man jedoch auf einen sehr viel effizienteren Algorithmus, indem die Werte $\text{Fib}(2), \ldots, \text{Fib}(n)$ einfach iterativ berechnet werden. In diesem Fall genügen $n-1$ Additionen und konstanter Speicherplatz, da sich nur die letzten beiden Ergebnisse gemerkt werden müssen.

Bei der dynamischen Programmierung werden diese Ideen nun auf Optimierungsprobleme übertragen. Sie kann immer dann angewendet werden, wenn sich die Lösung größerer Probleme aus einer optimalen Verbindung der Lösung kleinerer Probleme ergibt. Der Zusammenhang zwischen dem Problem und den Teilproblemen wird durch die bellmansche Optimalitätsgleichung ausgedrückt. Bei jedem Problem muss natürlich bewiesen werden, dass dieser Zusammenhang auch tatsächlich gilt. Desweiteren ist wichtig, dass sich aus suboptimalen Teillösungen keine optimale Lösung zusammensetzen lassen darf.

Frage 3.2: Wann ergibt die dynamische Programmierung effiziente Algorithmen?

Damit sich die Eigenschaften der dynamischen Programmierung effizient nutzen lassen, müssen für das Problem die folgenden Bedingungen gelten:

1. Die Anzahl der betrachteten Teilprobleme ist klein.
2. Die Anzahl der betrachteten Kombinationen von Teilproblemen ist klein.
3. Die Berechnung des Wertes eines Problems aus einer Kombination von Teilproblemen ist einfach.

Frage 3.3: Bei Problemen welchen Typs lässt sich die dynamische Programmierung besonders leicht anwenden?

Die dynamische Programmierung lässt sich besonders leicht auf Problemen des „Intervalltyps“ anwenden. Dies sind Probleme auf n Daten $D_1, \ldots, D_n$, welches als Problem vom Typ $(1, n)$ bezeichnet wird. Die Teilprobleme sind Probleme auf den Daten $D_i, \ldots, D_j$, also Probleme vom Typ (i, j) mit $1 \leq i \leq j \leq n$. Die Anzahl der Teilprobleme ist schonmal relativ klein. Sie beträgt:

$$\binom{n}{2} + n = \frac{n \cdot (n-1)}{2} + n = \frac{1}{2}n^2 + \frac{n}{2} \leq n^2$$

Bei dieser Analyse wurden die Probleme in die Klassen der Probleme des Typ (i, j) mit $i \neq j$ und die Probleme des Typs (i, i) für $1 \leq i \leq n$ aufgeteilt. Damit ist also die erste Forderung, dass die Anzahl der Teilprobleme klein ist, erfüllt. Wie sieht es mit der Anzahl der möglichen Kombinationen aus? Ein Teilproblem (i, j) kann in die beiden kleineren Teilprobleme (i, k) und $(k+1, j)$ für $i \leq k \leq j-1$ zerlegt werden. Dabei wird $j-i$ als die Größe des Teilproblems (i, j) betrachtet. Wenn sich die Lösung eines größeren Problems also als die optimale Wahl des k herausstellt, dann kann die dynamische Programmierung Verwendung finden. Mit $Z(n)$ sei der Aufwand bezeichnet, aus den Lösungen der Teilprobleme (i, k) und $(k+1, j)$ die Lösung für das Problem (i, j) zu berechnen. Dann beträgt der Gesamtaufwand $O(n^3 \cdot Z(n))$. Denn es gibt n^2 Teilprobleme, für welche es höchsten n Zerlegungspunkte k gibt.

Frage 3.4: Welches weitere typische Vorgehen gibt es bei der dynamischen Programmierung?

Ein weiteres typisches Vorgehen besteht darin, den Suchbereich einzuschränken, indem bestimmte Teile des Suchbereichs „verboten“ werden. Beispiele für diese Vorgehensweise sind der Algorithmus zur Berechnung aller kürzesten Wege in einem Graphen (All-Pairs-Shortest-Path) oder das Rucksackproblem. Beim ersterem Problem werden im k-ten Teilproblem jeweils nur die Knoten $\{1, \ldots, k\}$ als Zwischenknoten erlaubt. Für das Rucksackproblem gibt es zwei Varianten mit pseudopolynomieller Laufzeit, bei denen im k-ten Teilproblem jeweils nur die Objekte $\{1, \ldots, k\}$ betrachtet werden. Sie unterschieden sich in der weiteren Einschränkung. Bei einer Variante werden verschiedene Gewichtsschranken betrachtet, bei der anderen verschiedene Nutzen, die genau erreicht werden müssen. Beide Algorithmen werden im Folgenden vorgestellt.

Frage 3.5: Worin besteht das Rucksackproblem?

Beim Rucksackproblem ist eine Menge von n Objekten $\{1, \ldots, n\}$ gegeben, wobei jedem Objekt i zwei Werte zugeordnet sind: ein Gewichtswert g_i und ein Nutzenswerte v_i. Die Aufgabe besteht nun darin, eine Teilmenge dieser Objekte so in einen Rucksack zu packen, dass der Nutzen maximiert wird, wobei jedoch eine Gewichtsschranke G eingehalten werden muss.

Frage 3.6: Wie funktioniert die Variante des Rucksackproblems mit verschiedenen Gewichtsschranken?

Bei dieser Variante des Rucksackproblems wird das Problem auf zweifache Art und Weise eingeschränkt. Zum einen werden im k-ten Teilproblem nur die Objekte $\{1, \ldots, k\}$ betrachtet. Desweiteren werden verschiedene Gewichtsschranken g betrachtet. Das Teilproblem (k, g) ist also eine optimale Bepackung, bei der nur die ersten k Objekte betrachtet werden und die Gewichtsschranke g gegeben ist. Der Wert $V(k, g)$ beschreibe den Nutzen einer optimalen Lösung für dieses Teilproblem. Wir sind also am Wert $V(n, G)$ interessiert. Im Algorithmus werden alle Teilprobleme $(k, g),\ 1 \leq k \leq n,\ 1 \leq g \leq G$ betrachtet.

Wir beschäftigen uns nun mit der Frage, wie sich der Wert für $V(k, g)$ berechnet. Es wird bei diesem Problem das k-te Objekt mit der Gewichtsschranke g betrachtet. Für das Objekt bestehen nur zwei Möglichkeiten. Entweder wird es eingepackt oder nicht. Wenn es eingepackt wird, gewinnen wir den Nutzen v_k und für die Objekte $\{1, \ldots, k-1\}$ muss ein optimaler Nutzen mit der Gewichtsschranke $g - g_k$ erreicht werden, da das k-te Objekt eingepackt wurde und schon das Gewicht g_k beansprucht. Wird es nicht eingepackt, bleibt für die Objekte $\{1, \ldots, k-1\}$ eine Gewichtsschranke von g, um einen optimalen Nutzen zu erzielen. Da wir für das Teilproblem (k, g) den Nutzen maximieren wollen, ergibt sich also die folgende bellmansche Optimalitätsgleichung:

$$V(k, g) = \max\{V(k-1, g-g_k) + v_k,\ V(k-1, g)\}$$

Der Algorithmus baut nun eine Tabelle mit n Zeilen und G Spalten auf, wobei diese zeilenweise gefüllt wird. Es werden nun noch einige sinnvolle Randwerte definiert, da es z.B. vorkommen kann, dass $g - g_k$ negativ wird. Die Randwerte sind:

- $V(k, g) = -\infty$ für $g < 0$
- $V(0, g) = 0$ für $g \geq 0$
- $V(k, 0) = 0$

Der Algorithmus beginnt damit, die erste Zeile zu füllen, wobei aufgrund der Tatsache, dass $V(0, g) = 0$ gilt, sich $V(0, g) + v_1$ stets bei der Maximierung durchsetzt. Der erste Nutzen ungleich 0 (genauer: der Nutzen v_1) steht in der ersten Zeile dabei in der Spalte für die $g = g_1$ gilt. Danach werden, jeweils mit Hilfe der Vorgängerzeile, die restlichen Zeile gefüllt.

Da jedes Element der Tabelle in $O(1)$ gefüllt werden kann, benötigt der Algorithmus eine Laufzeit von $O(nG)$. Diese Laufzeit ist pseudopolynomiell. Dies bedeutet der Algorithmus ist nur für polynomiell kleine G effizient. Gilt z.B. $G = 2^n$ so ist die Eingabelänge zwar polynomiell begrenzt, die Laufzeit beträgt jedoch $O(n \cdot 2^n)$ und ist somit exponentiell. Würde jedoch z.B. gelten $G = n^2$, so würde die Laufzeit $O(n \cdot n^2) = O(n^3)$ betragen und der Algorithmus wäre relativ effizient.

3.2 Ein weiterer pseudopolynomieller Algorithmus für das Rucksackproblem

Frage 3.7: Wie funktioniert die Variante des Rucksackproblems mit verschiedenen Nutzenswerten?

Der maximale Nutzen, der beim Rucksackproblem erreicht werden kann, ist $v_{sum} = \sum_{i=1}^{n} v_i$. Bei dieser Variante des Rucksackproblems werden nun $n \cdot v_{sum}$ Teilprobleme betrachtet, die jeweils mit (k, v) bezeichnet werden. Dabei stellt das Teilproblem (k, v) die Fragestellung dar, ob mit den ersten k Objekten genau der Nutzen v erreicht werden kann, wobei natürlich die Gewichtsschranke G eingehalten werden muss. Ist dies der Fall, so wird für dieses Teilproblem der Wert (g, a) gespeichert, also $T(k, v) = (g, a)$. Dabei gibt g das minimale Gesamtgewicht an, mit dem genau der Nutzen v erreicht werden kann. In a wird sich gemerkt, ob das k-te Objekt an dieser Lösung beteiligt ist oder nicht. Die optimale Lösung ergibt sich damit als das größte v in Zeile n, so dass (n, v) nicht leer ist.

Wie kann die optimale Bepackung nun berechnet werden? Ist das maximale v gefunden, für das (n, v) nicht leer ist, so schauen wir nach, ob das n-te Objekt an dieser Lösung beteiligt

ist oder nicht. Ist es beteiligt, so wird als nächstes der Eintrag $T(n-1, v-v_n)$ betrachtet, ansonsten $T(n-1, v)$. Dieses Verfahren wird bis zur ersten Zeile iteriert, wobei für jede Zeile wie eben beschrieben vorgegangen wird.

Der Algorithmus baut also, genau wie der vorher beschriebene, eine Tabelle mit $n \cdot v_{sum}$ Elementen auf. Diese Tabelle wird ebenfalls Zeile für Zeile gefüllt, wobei bei der ersten Zeile begonnen wird, da sie sofort gefüllt werden kann. Es sind nämlich lediglich die Einträge $(1, 0)$ und $(1, v_1)$ gefüllt und zwar mit $(0, 0)$ und $(g_1, 1)$. Letzterer ist jedoch nur gefüllt, wenn $g_1 \leq G$ gilt, was jedoch vernünftigerweise angenommen werden kann. Ansonsten werden in einem Vorverarbeitungsschritt alle Elemente i mit $g_i > G$ entfernt, da diese nicht an einer Lösung beteiligt sein können.

Wie werden nun die restlichen Zeilen gefüllt? Wir betrachten den Eintrag $T(k, v)$. Für das k-te Objekt gibt es wieder zwei Möglichkeiten. Entweder es wird eingepackt und ist an der Lösung beteiligt oder nicht. Wenn es eingepackt wird, bleibt das Teilproblem $T(k-1, v-v_k)$, ansonsten $T(k-1, v)$. Es gibt nun vier Fälle für $T(k, v)$, wobei im Folgenden ein Feld $T(k, v)$ mit $v < 0$ als leer angenommen wird:

- $T(k-1, v-v_k)$ und $T(k-1, v)$ sind beide leer. Also kann der Nutzen v nicht mit den ersten k Objekten erreicht werden, egal, ob wir k einpacken oder nicht. $T(k, v)$ bleibt somit leer.

- $T(k-1, v-v_k) = (g, a)$ und $T(k-1, v)$ ist leer. Mit dem Objekt k kann also der Nutzen v genau erreicht werden. Dabei muss jedoch $g + g_k \leq G$ gelten. Ist dies der Fall, so gilt $T(k, v) = (g + g_k, 1)$.

- $T(k-1, v-v_k)$ ist leer und $T(k-1, v) = (g, a)$. Der Nutzen v kann also mit den ersten k Objekten genau erreicht werden, jedoch nur, wenn das k-te Objekt nicht eingepackt wird. Also gilt $T(k, v) = (g, 0)$.

- $T(k-1, v-v_k) = (g, a)$ und $T(k-1, v) = (g', a')$. Der Nutzen kann also unabhängig davon erreicht werden, ob wir k einpacken oder nicht. Jedoch möchten wir den Nutzen mit so wenig Gewicht wie möglich erreichen. Wenn wir also k einpacken und das Gewicht bleibt kleiner als wenn wir k nicht einpacken ($g+g_k \leq g'$), dann setzen wir $T(k, v) = (g+g_k, 1)$. Ansonsten gilt $T(k, v) = (g', 0)$.

Das bellmansche Optimalitätsprinzip steckt in Punkt vier, wo eine Entscheidung getroffen werden muss. Da diese vier Fälle in $O(1)$ getestet werden können, ergibt sich eine Laufzeit von $O(n \cdot v_{sum})$. Auch diese Laufzeit ist wieder psyeudopolynomiell. Ist die Summe der Nutzwerte nicht polynomiell begrenzt, so ist der Algorithmus ineffizient. Ist mindestens einer der Nutzwerte nicht polynomiell beschränkt, so kann auch der maximale Nutzwert $v_{max} = \max\{v_1, \ldots, v_n\}$ nicht polynomiell beschränkt sein. Die Laufzeit kann daher als $O(n \cdot (n \cdot v_{max})) = O(n^2 \cdot v_{max})$ abgeschätzt werden.

3.3 Bioinformatik - die Ähnlichkeit zweier Sequenzen

Frage 3.8: Wie sieht das Grundproblem aus?

Das Grundproblem besteht darin, dass zwei Sequenzen $x = (x_1, \ldots, x_n)$ und $y = (y_1, \ldots, y_m)$ über einem Alphabet Σ gegeben sind. Im Zusammenhang mit der Bioinformatik könnte dies z.B. das Alphabet $\{A, C, G, T\}$ sein. Es soll nun die Ähnlichkeit zwischen diesen Sequenzen

gemessen werden. Das heißt, es soll herausgefunden werden, wie „wahrscheinlich“ es ist, dass die Sequenzen identisch sind, wobei jedoch Mutationen, Auslassungen und Einfügungen aufgetreten sein können. Um dies zu modellieren wird ein weiterer Buchstabe eingeführt, der jedoch nicht zu Σ gehört. Dies ist das Leerzeichen bzw. der Lückenbuchstabe „$-$“. Weiter ist eine Bewertungsfunktion (score function) $s : \Sigma \times \Sigma \to \mathbb{R}$ gegeben, mit der die Ähnlichkeit zweier Buchstaben gemessen werden kann.

Gesucht sind nun die Erweiterungen $x^* = (x_1^*, \ldots, x_k^*)$ und $y^* = (y_1^*, \ldots, y_k^*)$ von x und y, so dass diese gleich lang sind und die Bewertung

$$s(x^*, y^*) = \sum_{i=1}^{k} s(x_i^*, y_i^*)$$

maximal ist. Die Erweiterung eines Strings erfolgt durch die Einfügung von Leerzeichen. Damit die Erweiterungen nicht unendlich lang werden können, nehmen wir an, dass die Score-Funktion für zwei Leerzeichen einen negativen Wert liefert, also $s(-,-) < 0$ gilt. Durch diese Regelung wird keine Erweiterung gebildet, in der zwei Leerzeichen nebeneinander stehen, da dieselbe Erweiterung ohne dieses Leerzeichenpaar eine höhere Score erreichen würde. Also gilt $k \leq m+n$ und das Problem wird endlich. Damit nicht zuviele Paare der Art $(-, a)$ oder $(b, -)$ entstehen, wird eine einheitliche Lückenstrafe d eingeführt und es gilt: $s(-, a) = s(b, -) = -d$. Das beschriebene Problem ist das Problem des “globalen Alignment“.

Frage 3.9: Wie funktioniert der Algorithmus von Needleman und Wunsch?
Der Algorithmus von Needleman und Wunsch löst das oben beschriebene Problem des globalen Alignment in Zeit $O(nm)$ und Platz $O(\min\{n, m\})$. Wird der Platz $O((n+1)(m+1))$ investiert kann auch ein globales Alignment berechnet werden. Er funktioniert wie folgt. Wenn $S(i, j)$ der Wert des globalen Alignment für $(x_1, \ldots, x_i)$ und $(y_1, \ldots, y_j)$ ist, dann gibt es für die letzte Stelle des Alignment nur drei Möglichkeiten, da ein Leerzeichenpaar ja mit Hilfe des negativen Score-Wert verhindert wird:

$$\begin{bmatrix} x_i \\ y_j \end{bmatrix} \text{ oder } \begin{bmatrix} x_i \\ - \end{bmatrix} \text{ oder } \begin{bmatrix} - \\ y_j \end{bmatrix}$$

Im ersten Fall wird die letzte Stelle mit der Score $s(x_i, y_j)$ eingehen und in den beiden anderen Fällen mit der Lückenstrafe $-d$. Es bleiben also die Teilprobleme $(i-1, j-1)$, $(i-1, j)$ und $(i, j-1)$, die eine möglichst hohe Score erreichen sollen. Also ergibt sich die bellmansche Optimalitätsgleichung als:

$$S(i, j) = \max\{S(i-1, j-1) + s(x_i, y_i),\ S(i-1, j) - d,\ S(i, j-1) - d\}$$

Wir sind an $S(n, m)$ interessiert. Wenn wir die Werte $S(i, j)$ in einer Tabelle speichern, dann können wir diese zeilen- bzw. spaltenweise auffüllen. Dazu benötigen wir jedoch noch die Randwerte $S(0, \cdot)$ und $S(\cdot, 0)$. Da z.B. $S(i, 0)$ der Sequenz $(x_1, \ldots, x_i)$ nur gepaart mit Leerzeichen entspricht, berechnen sich die Randwerte als $S(i, 0) = -id$ und $S(0, j) = -jd$.

Die Rechenzeit ergibt sich, da $(n+1)(m+1) = O(nm)$ Tabelleneinträge gefüllt werden müssen. Es stellt sich jedoch die Frage, warum der Platzbedarf nicht genauso hoch ist. Dies liegt daran, dass für jedes zu berechnende Tabellenelement nur die Werte der Elemente links und oben sowie links oben davon benötigt werden. Werden die Werte nun zeilen- bzw. spaltenweise berechnet, dann wird nur die vorherige und die aktuelle Zeile bzw. Spalte benötigt. Also

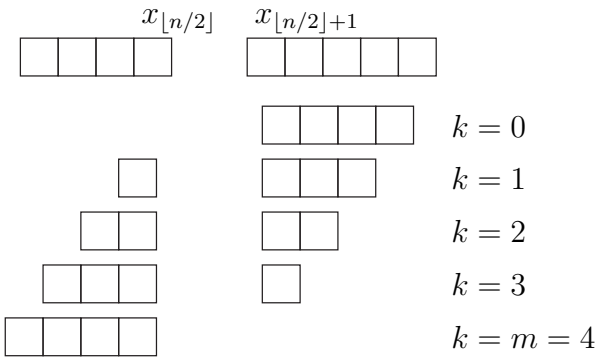


Abbildung 11: Die Idee des Algorithmus von Hirschberg

stets 2 Zeilen oder Spalten. Somit ergibt sich ein Platzbedarf von $O(\min\{n, m\})$. Mit diesen Zeit- und Platzbedingungen kann jedoch nur der Wert eines globalen Alignment berechnet werden.

Soll auch das Alignment selbst berechnet werden, so steigt der Platzbedarf wieder auf $O(nm)$ Es wird sich dann für jedes der $(n+1)(m+1)$ Elemente in einem Zeiger gemerkt, für welches Element in „seiner“ bellmanschen Optimalitätsgleichung das Maximum angenommen wurde. Beim Element (n, m) beginnend kann dann ein Weg bis zum einem Element $(0, \cdot)$ bzw. $(\cdot, 0)$ verfolgt werden und so das optimale Alignment berechnet werden. Allerdings kann der Platzbedarf von $O(nm)$ für $n = m \approx 10^6$ schon kritisch werden. Es gibt jedoch auch eine Möglichkeit ein global optimales Alignment mit dem Verfahren von Hirschberg in Zeit $O(nm)$ und Platz $O(n + m)$ zu berechnen.

Frage 3.10: Wie funktioniert das Verfahren von Hirschberg?

Hirschberg hat gezeigt, dass ein global optimales Alignment in Zeit $O(nm)$ und Platz $O(n + m)$ berechnet werden kann. Zuerst wird die Notation etwas erweitert. Es sei $S(i, j)$ weiterhin der Wert eines global optimalen Alignments von $(x_1, \ldots, x_i)$ und $(y_1, \ldots, y_j)$. Mit $S^r(i, j)$ sei im Folgenden der Wert eines global optimalen Alignment von $(x_{n-i+1}, \ldots, x_n)$ und $(y_{n-j+1}, \ldots, y_m)$ bezeichnet. Der Wert $S^r(i, j)$ kann analog zu den $S(i, j)$-Werten berechnet werden.

Wir betrachten nun ein global optimales Alignment für $(x_1, \ldots, x_n)$ und $(y_1, \ldots, y_m)$. Wird die Folge $(x_1, \ldots, x_n)$ in die Teilfolgen $x_l = (x_1, \ldots, x_{\lfloor n/2 \rfloor})$ und $x_r = (x_{\lfloor n/2 \rfloor+1}, \ldots, x_n)$ aufgeteilt, dann gibt es ein k mit $0 \leq k \leq m$, so dass $(y_1, \ldots, y_k)$ zu einem optimalen Alignment der Folge x_l gehört und $(y_{k+1}, \ldots, y_m)$ zu einem optimalen Alignment der Folge x_r. Ein global optimales Alignment kann also aus optimalen Alignments des linken und des rechten Teils „zusammengeklebt“ werden. Dies funktioniert aufgrund der Additivität der Bewertungsfunktion. Es gilt also:

$$S(n, m) = \max\{S(\lfloor n/2 \rfloor, k) + S^r(\lceil n/2 \rceil, m - k) \mid 0 \leq k \leq m\}$$

Die Idee des Algorithmus von Hirschberg besteht nun darin, zuerst das k zu berechnen, welches die Bewertung optimiert. Dies wird für jedes $0 \leq k \leq m$ mit Hilfe des Algorithmus von Needleman und Wunsch gemacht. Abbildung 11 verdeutlicht diese Idee. Ist das optimale k^* nach $O(m)$ Vergleichen ermittelt, werden die entstehenden Teilfolgen $(x_1, \ldots, x_{\lfloor n/2 \rfloor})$ und $(y_1, \ldots, y_{k^*})$ sowie $(x_{\lfloor n/2 \rfloor+1}, \ldots, x_n)$ und $(y_{k^*+1}, \ldots, y_m)$ rekursiv behandelt. Die Rekursion

endet in den trivialen Fällen, dass eine der Folgen leer ist oder nur noch ein Element enthält. In letzterem Fall wird die Lösung direkt berechnet. In diesen Fällen wird die Ergebnisfolge auch ausgegeben. Da die linken Folgen vor den rechten ausgegeben werden, wird das Ergebnis korrekt berechnet. Hier der Algorithmus komplett:

1. In Zeit $O(nm)$ und Platz $O(m)$ werden die $S(\lfloor n/2 \rfloor, \cdot)$-Werte berechnet und abgespeichert.
2. In Zeit $O(nm)$ und Platz $O(m)$ werden die $S^r(\lceil n/2 \rceil, \cdot)$-Werte berechnet und abgespeichert.
3. In Zeit $O(m)$ wird das k^* berechnet, welches die Bewertung maximiert. Es wird der Wert $(\lfloor n/2 \rfloor, k^*)$ abgespeichert und die bisher berechneten Ergebnisse gelöscht.
4. Löse die Probleme rekursiv für die Folgen $(x_1, \ldots, x_{\lfloor n/2 \rfloor})$ und $(y_1, \ldots, y_{k^*})$ sowie für die Folgen $(x_{\lfloor n/2 \rfloor + 1}, \ldots, x_n)$ und $(y_{k^*+1}, \ldots, y_m)$ und konkateniere die Lösungen. Die Rekursion endet in den oben beschriebenen trivialen Fällen.

Frage 3.11: Welchen Zeit- und Platzbedarf hat der Algorithmus von Hirschberg?

Der Algorithmus von Hirschberg löst das Problem des globalen Alignment in Zeit $O(nm)$ und Platz $O(n+m)$. In Schritt 1 und 2 wird jeweils Platz $O(m)$ benötigt, um ein Element in Platz $O(1)$ abzuspeichern. Es werden $O(m)$ Elemente abgespeichert, die in Schritt 3 miteinander verglichen werden, um ein Element zu ermitteln, welches in Platz $O(1)$ abgespeichert wird. In Schritt 3 werden also höchstens $O(n)$ Elemente abgespeichert. Insgesamt ergibt sich also ein Platzbedarf von $O(n+m)$.

Die Berechnung des Zeitbedarfs ist etwas komplizierter. Gegeben seien n und m. Dann wählen wir eine Konstante c derart, dass die Schritte 1, 2 und 3 zusammen in Zeit cnm durchführbar sind. Die Konstante soll auch so gewählt werden, dass Probleme mit $n = 1$ in Zeit cm und Probleme mit $m = 1$ in Zeit cn gelöst werden können. Im Folgenden wird angenommen, dass n eine Zweierpotenz ist. Dann gilt für die Rechenzeit $T(n, m)$:

$$T(n,m) \leq cnm + \max\{T(n/2,k) + T(n/2, m-k) \mid 0 \leq k \leq m\}$$

Um das Problem (n, m) zu lösen, veranschlagen wir also Zeit cnm für die ersten drei Schritte und nehmen weiterhin die schlimmste Aufteilung der Teilprobleme in Schritt vier an. Es wird nun die Lösung $T(n, m) \leq 2cnm$ „geraten“ und induktiv gezeigt. Für $n = 1$ und $m = 1$ gilt die geratene Lösung, da $cn \leq 2cn$ bzw. $cm \leq 2cm$. Es gilt dann:

$$\begin{aligned}
T(n,m) &\leq cnm + \max\{2c\frac{n}{2}k + 2c\frac{n}{2}(m-k) \mid 0 \leq k \leq m\} \\
&= cnm + \max\{cnk + cn(m-k) \mid 0 \leq k \leq m\} \\
&= cnm + \max\{cn \cdot (k + (m-k)) \mid 0 \leq k \leq m\} \\
&= cnm + \max\{cnm \mid 0 \leq k \leq m\} \\
&= cnm + cnm \\
&= 2cnm
\end{aligned}$$

Da c eine Konstante ist, beträgt der Zeitbedarf also $O(nm)$.

Frage 3.12: Was ist ein lokales Alignment? Wie kann es berechnet werden?

Beim Problem des lokalen Alignment sind wieder zwei Sequenzen $x = (x_1, \ldots, x_n)$ und $y = (y_1, \ldots, y_m)$ gegeben. Es sollen nun jeweils die Teilsequenzen $(x_i, \ldots, x_k)$ und $(y_j, \ldots, y_l)$ gefunden werden, deren globales Alignment unter allen Teilsequenzpaaren die maximale Bewertung hat. Auch hier betrachten wir wieder Teilprobleme $S^*(i, j)$, wobei $S^*(i, j)$ der Wert eines lokal optimalen Alignment für $(x_1, \ldots, x_i)$ und $(y_1, \ldots, y_j)$ ist. Damit wir die Lösung eines solches Teilproblem auch zur Lösung weiterer Teilprobleme benutzen können, müssen wir jedoch eine weitere Einschränkung machen. Und zwar muss in der Lösung dieses Teilproblems entweder x_i oder y_i oder beide Variablen vorkommen. Dies bedeutet, dass bei der Berechnung des globale Aligment entweder x_i neben einem Leerzeichen steht, y_j neben einem Leerzeichen steht oder x_i mit y_j aligt wird. Desweiteren wird diese Forderung unter anderem benötigt, damit das lokal optimale Alignment selbst auch berechnet werden kann. Nun kann es natürlich passieren, dass alle drei Lösungen unter diesen Voraussetzungen sehr schlecht sind, d.h. eine negative Score ergeben. In diesem Fall wählen wir die vierte Option, zwei leere Sequenzen mit Wert 0 zu wählen. Die anderen drei Lösungswerte ergeben sich analog wie bei Needleman und Wunsch. Wenn beide Variablen x_i und y_i in der Lösung nebeneinander stehen, müssen wir zum Wert von $S^*(i-1, j-1)$ die Score $s(x_i, y_j)$ addieren. Ansonsten müssen wir von $S^*(i-1, j)$ bzw. $S(i, j-1)$ die Lückenstrafe d subtrahieren. Somit ergibt sich folgende bellmansche Optimalitätsgleichung:

$$S^*(i,j) = \begin{cases} S^*(i-1, j-1) + s(x_i, y_j) \\ S^*(i-1, j) - d \\ S^*(i, j-1) - d \\ 0 \end{cases}$$

Den Wert des lokal optimalen Alignment bekommen wir, indem wir den maximalen $S^*(i, j)$-Wert aus der Tabelle suchen, also als:

$$\max\{S^*(i,j) | 1 \leq i \leq n \text{ und } 1 \leq j \leq m\}$$

Das Alignment selbst kann berechnet werden, indem sich für jedes $S^*(i, j)$ gemerkt wird, für welchen der vier Fälle es maximiert wurde. Wird dies gemacht, können wir uns, bei dem maximalen Wert $S^*(i'', j'')$ beginnend, so lange über diese Zeiger „durchgehangeln“, bis wir das erste Mal auf den Wert 0 stoßen. Dies wird spätestens für $S^*(0, \cdot)$ bzw. $S^*(\cdot, 0)$ der Fall sein. Es sei $S^*(i', j')$ der letzte von 0 verschiedene Wert, bevor wir zu einer 0 weitergehen. Dann sind (x', x'') und (y', y'') die lokal optimalen Alignments.

Frage 3.13: Was sind affine Lückenkosten? Wie kann ein global optimales Alignment berechnet werden?

3.4 Hidden Markoff Modelle

Frage 3.14: Was ist ein Hidden Markoff Modell?

Ein Hidden Markoff Modell ist eine Markoffkette mit verborgenen Zuständen. Die Markoffkette besteht aus einer Menge von Zuständen Q, auf der ein Markoffprozess abläuft. Dies bedeutet, dass die Kette zeitdiskret die Zustände wechselt. Die Wahrscheinlichkeit, dass die Kette von einem Zustand k in den Zustand l wechselt, hängt dabei von einer gegebenen Übergangswahrscheinlichkeit a_{kl} ab. Die Summe der Übergangswahrscheinlichkeiten muss dabei

für jeden Zustand k genau 1 ergeben, d.h.: $\sum_l a_{kl} = 1$. Bei einem Hidden Markoff Modell ist es nun so, dass die Folge der Zustände nicht direkt beobachtet werden kann. Stattdessen kann eine Folge von Buchstaben aus einem Alphabet Σ beobachtet werden, wobei bei jedem Zustandswechsel genau ein Buchstabe ausgegeben wird. Für jeden Zustand sind nun wiederum die Wahrscheinlichkeiten $e_k(b)$ definiert, dass ein Buchstabe $b \in \Sigma$ ausgegeben wird, wenn die Markoffkette in den Zustand k wechselt. Im Folgenden werden die Parameter a_{kl} und $e_k(b)$ als gegeben vorausgesetzt. Ebenso wird angenommen, dass die Kette sich zum Zeitpunkt 0 im Zustand 0 befindet.

Frage 3.15: Wie groß ist die Wahrscheinlichkeit, dass zu einer gegebenen Zustandsfolge eine gegebener Buchstabenfolge ausgegeben wird?

Es sei $\pi = (\pi_1, \ldots, \pi_n)$ eine Zustandsfolge und $x = (x_1, \ldots, x_n)$ eine Folge von Buchstaben. Dann beträgt die Wahrscheinlichkeit, dass π durchlaufen und dabei x ausgegeben wird:

$$P(x, \pi) = \prod_{i=1}^{n} a_{\pi_{i-1}\pi_i} \cdot e_{\pi_i}(x_i)$$

Die Wahrscheinlichkeit setzt sich also aus den Produkten der Wahrscheinlichkeiten zusammen, dass von einem Zustand π_{i-1} in den Nachfolgerzustand π_i gewechselt wird, jeweils multipliziert mit der Wahrscheinlichkeit, dass im Zustand π_i überhaupt der Buchstabe x_i ausgegeben wird.

Frage 3.16: Was ist das Decodierungsproblem?

Das Decodierungsproblem besteht in der Aufgabe, zu einer beobachteten Buchstabenfolge $x = (x_1, \ldots, x_n)$ die Zustandsfolge $\pi = (\pi_1, \ldots, \pi_n)$ zu finden, die am wahrscheinlichsten durchlaufen wurde. Die Wahrscheinlichkeit, dass wenn x beobachtet wurde, die Folge π durchlaufen wurde, ergibt sich als die bedingte Wahrscheinlichkeit $P(\pi|x)$. Allgemein gilt für die bedingte Wahrscheinlichkeit $P(B|A)$ zweier Ereignisse A und B:

$$P(B|A) = \frac{P(B \cap A)}{P(A)}$$

Die Wahrscheinlichkeit des Schnitts der Ereignisse π und x ist die oben schon erwähnte Wahrscheinlichkeit $P(x, \pi)$. Denn der Schnitt besteht eben genau darin, dass π durchlaufen wird und dabei x ausgegeben wird.

Wenn also die Folge π gesucht wird, die am wahrscheinlichsten durchlaufen wurde, wenn x beobachtet wird, wird also das π gesucht, welches $P(\pi|x)$ maximiert. Es gilt nach der Formel für die bedingte Wahrscheinlichkeit:

$$P(\pi|x) = \frac{P(x, \pi)}{P(x)}$$

$P(x)$ ist die Wahrscheinlichkeit, dass unter allen Möglichkeiten überhaupt x ausgegeben wird. Da diese Wahrscheinlichkeit demnach unabhängig von einer bestimmten Folge π ist, reicht es also $P(x, \pi)$ zu maximieren, um $P(\pi|x)$ zu maximieren.

Frage 3.17: Wie funktioniert der Viterbi-Algorithmus?

Der Viterbi-Algortihmus dient dazu, die Zustandsfolge π zu finden, welche $P(x, \pi)$ maximiert. Dazu wird zuerst die Notation $v_k(i)$ eingeführt, welche die maximale Wahrscheinlichkeit eines Paares (x', π') bezeichnen soll, so dass $x' = (x_1, \ldots, x_i)$ und $\pi' \in Q^i$ mit $\pi'_i = k$. Also

bezeichnet $v_k(i)$ die größte Wahrscheinlichkeit, dass man sich nach i Schritte im Zustand k befindet und x' ausgegeben wurde. Von welchen Faktoren ist diese Wahrscheinlichkeit nun abhängig? Zum einen natürlich von der Wahrscheinlichkeit, dass der Buchstabe x'_i im Zustand k ausgegeben wird. Zum anderen muss man jedoch aus irgendeinem Vorgängerzustand m heraus in den Zustand k gewechselt sein. Genau hier besteht also eine Wahlmöglichkeit, über die wir $v_k(i)$ maximieren können. Welcher Vorgängerzustand $v_k(i)$ maximiert, hängt wiederum von zwei Faktoren ab:

1. Die Wahrscheinlichkeit, sich nach $i-1$ Schritten in diesem Vorgängerzustand zu befinden.
2. Die Übergangswahrscheinlichkeit a_{mk} vom Zustand m in den Zustand k zu wechseln.

Damit ergibt sich die folgende bellmansche Optimalitätsgleichung:

$$v_k(i) = e_k(x_i) \cdot \max\{v_m(i-1) \cdot a_{mk} \mid m \in Q\}$$

Die Randwerte $v_0(0) = 1$ und $v_k(0) = 0$ sind bekannt, da man sich nach 0 Schritten nach Voraussetzung im Zustand 0 befindet und dementsprechend die Wahrscheinlichkeiten für die anderen Zustände $k \neq 0$ gleich 0 sein müssen. Somit können die Werte $v_k(n)$ für alle Zustände $k \in Q$ inkrementell berechnet werden. Die größtmögliche Wahrscheinlichkeit einer Zustandsfolge beträgt dann:

$$v_{max} = \max\{v_k(n) \mid k \in Q\}$$

Wenn sich nun für jeden Wert $v_k(i)$ gemerkt wird, für welchen Vorgängerzustand das Maximum angenommen wird, kann so die wahrscheinlichste Zustandsfolge rekonstruiert werden.

Der Platzbedarf beträgt $O(n \cdot |Q|)$, da sich für jedes $k \in Q$ und $i \in \{0, \ldots, n\}$ der Wert $v_k(i)$ und der maximierende Vorgängerzustand gemerkt werden muss. Der Zeitbedarf beträgt $O(n \cdot |Q|^2)$ da für jeden Eintrag der $O(n \cdot |Q|)$ großen Tabelle ein Wert berechnet werden muss. Zur Berechnung eines solchen Wertes $v_k(i)$ müssen alle Vorgängerwerte $v_m(i-1)$ mit $m \in Q$ mit der dazugehörigen Übergangswahrscheinlichkeit a_{mk} betrachtet werden und das Maximum berechnet werden. Somit müssen für einen Eintrag $O(|Q|)$ Operationen durchgeführt werden. Für alle Einträge ergibt dies $O(n \cdot |Q| \cdot |Q|) = O(n \cdot |Q|^2)$ Operationen.

Frage 3.18: Wie groß ist die Wahrscheinlichkeit, dass der i-te Zustand der Zustand k war, wenn x ausgegeben wird?

Gesucht ist also die folgende bedingte Wahrscheinlichkeit:

$$P(\pi_i = k|x) = \frac{P(x, \pi_i = k)}{P(x)}$$

Es müssen also getrennt die Wahrscheinlichkeiten $P(x, \pi_i = k)$ sowie $P(x)$ ausgerechnet werden. Letztere Wahrscheinlichkeit gibt die Wahrscheinlichkeit an, dass überhaupt die Buchstabenfolge $P(x)$ ausgegeben wird und wird mit dem Forward-Algorithmus berechnet. $P(x, \pi_i = k)$ wird mit dem Backward-Algorithmus berechnet werden.

Frage 3.19: Wie funktioniert der Forward-Algorithmus?

Der Forward-Algorithmus wird benutzt, um die Wahrscheinlichkeit $P(x)$ zu berechnen. Wir definieren zuerst $f_k(i)$ als die Wahrscheinlichkeit, dass wir uns nach dem i-ten Schritt im Zustand k befinden und die ersten i Buchstaben von x ausgegeben haben:

$$f_k(i) = P(x_1, \ldots, x_i, \pi_i = k)$$

Die Wahrscheinlichkeit, überhaupt am Ende x ausgegeben zu haben, ist dann die Summe der Wahrscheinlichkeiten $f_k(n)$ für alle $k \in Q$. Da auch hier wieder im Zustand 0 gestartet wird, gilt $f_0(0) = 1$ (wir befinden uns nach 0 Schritten auf jeden Fall im Zustand 0) und $f_k(0) = 0$ für $k \neq 0$. Es soll nun $f_k(i+1)$ ausgerechnet werden. Es wird sich zeigen, dass wir dafür nur alle $f_k(i)$-Werte sowie die schon gegebenen a_{kl}- und $e_k(b)$-Werte benötigen. Im Skript findet sich eine etwas längere Rechnung, die hier jedoch weggelassen wird, da die Idee auch intuitiv recht gut zu verstehen ist. Zuerst die Formel:

$$f_k(i+1) = e_k(x_{i+1}) \cdot \sum_m a_{mk} \cdot f_m(i)$$

Die Wahrscheinlichkeit, dass wir nach $i+1$ Schritten $(x_1, \ldots, x_{i+1})$ ausgegeben haben und uns im Zustand k befinden, hängt grob von zwei Faktoren ab. Zuerst einmal müssen wir im Zustand k den Buchstaben x_{i+1} ausgeben. Die Wahrscheinlichkeit dafür beträgt $e_k(x_{i+1})$ und geht entsprechend in die Formel ein. Desweiteren müssen wir von einem Vorgängerzustand m in den Zustand k gewechselt sein. Es wird nun also die Situation einen Schritt vor $f_k(i+1)$ betrachtet. In dieser muss gelten, dass wir uns im Vorgängerzustand m befinden und $(x_1, \ldots, x_i)$ ausgegeben wurde. Dies entspricht genau $f_m(i)$. Desweiteren geht die Wahrscheinlichkeit a_{mk} ein, dass wir überhaupt vom Zustand m in den Zustand k wechseln. Die Gesamtwahrscheinlichkeit für $f_k(i+1)$ ergibt sich als die Summe der Vorgängerwahrscheinlichkeiten.

Frage 3.20: Wo wird der Backward-Algorithmus benötigt und wie funktioniert er?

Der Backward-Algorithmus wird zur Berechnung von $P(x, \pi_i = k)$ benötigt. Dies ist die Wahrscheinlichkeit, dass x ausgegeben wird und der i-te Zustand der Zustand k ist. Diese Wahrscheinlichkeit kann jedoch auch als Produkt zweier anderer Wahrscheinlichkeiten ausgedrückt werden. Zum einen als die Wahrscheinlichkeit nach i Schritten im Zustand k zu sein und $(x_1, \ldots, x_i)$ ausgegeben zu haben. Diese Wahrscheinlichkeit haben wir etwas weiter oben schon mit dem Forward-Algorithmus berechnet. Die zweite Wahrscheinlichkeit ist dann die Wahrscheinlichkeit $P(x_{i+1}, \ldots, x_n | \pi_i = k)$, also dass vom Zustand k startend die Buchstabenfolge $(x_{i+1}, \ldots, x_n)$ ausgegeben wird. Diese Wahrscheinlichkeit wird nun mit $b_k(i)$ bezeichnet. Auch hier wird nicht auf die Umrechnung eingegangen, sondern nur das Ergebnis intuitiv erklärt. Dies Formel zur Berechnung von $b_k(i)$ lautet:

$$b_k(i) = \sum_m a_{km} \cdot b_m(i+1) \cdot e_m(x_{i+1})$$

Wenn wir uns nach i Schritten im Zustand k befinden, haben wir m Möglichkeiten in einen Nachfolgerzustand zu wechseln. Die Wahrscheinlichkeit von k in m zu wechseln beträgt a_{km}. In diesem Zustand wird mit Wahrscheinlichkeit $e_m(x_{i+1})$ der „gewünschte“ Buchstabe x_{i+1} ausgegeben. Desweiteren wird mit Wahrscheinlichkeit $b_m(i+1)$ vom Zustand m started die Restfolge $(x_{i+2}, \ldots, x_n)$ ausgegeben. Über die Summe wird dafür gesorgt, dass alle potentiellen Durchläufe, die bei k beginnen, betrachtet werden. Da $b_k(n) = 1$ gilt, kann $b_k(i)$ also mit den Methoden der dynamischen Programmierung berechnet werden.

4 Branch-and-Bound Algorithmen

4.1 Die Vorgehensweise bei Branch-and-Bound Algorithmen

Frage 4.1: Was ist ein Branch-and-Bound Algorithmus?

Ein Branch-and-Bound Algorithmus ist ein heuristisches Optimierungsverfahren, welches in jedem Fall eine optimale Lösung berechnet. Es kann bei diesen Verfahren jedoch keine gute Garantie für die Rechenzeit gegeben werden. Meistens werden Branch-and-Bound Verfahren auf NP-harte Probleme angesetzt, weshalb auch nicht mit polynomiellen worst-case-Laufzeiten gerechnet werden kann. Es wird jedoch gehofft, dass diese Verfahren in vielen Fällen relativ schnell sind bzw. eine akzeptable Rechenzeit haben.

Ein Vorteil der Branch-and-Bound-Methode liegt darin, dass im Fall einer Überschreitung der maximalen Rechenzeit die Güte der Lösung auf jeden Fall angegeben werden kann. Diese Eigenschaft kann natürlich auch dazu genutzt werden, einfach so lange zu rechnen, bis die Lösung eine gewisse Güte besitzt.

Im Folgenden werden nur Probleme betrachtet, bei denen die Anzahl der Lösungen endlich ist und der Wert einer Lösung effizient berechnet werden kann. Das allgemeine Vorgehen wird anhand von Maximierungsproblemen erklärt. Es muss sich lediglich klar gemacht werden, dass obere Schranken bei Maximierungsproblemen unteren Schranken bei Minimierungsproblemen entsprechen und umgekehrt.

Frage 4.2: Aus welchen Modulen besteht ein Branch-and-Bound Algorithmus?

Ein Branch-and-Bound Algorithmus besteht aus vier Modulen:

- Upper Bound Modul
- Lower Bound Modul
- Branching Modul
- Search Modul

Die Aufgabe des Upper Bound Modul ist es, eine möglichst gute obere Schranke für eine Lösung zu berechnen. Dies ist also eine „Besser als so gehts unter den gegebenen Bedingungen nicht“-Schranke. Damit dies effizient möglich ist, wird ein Problem oft „relaxter“ betrachtet. Dies bedeutet, dass Nebenbedingungen, die das Problem oft erst schwer machen, etwas gelockert werden. Diese Lockerung, im Folgenden Relaxation genannt, soll dafür sorgen, dass ein effizient zu lösendes Optimierungsproblem entsteht. Dabei ist natürlich wichtig, dass sich die Lösungen des nicht gelockerten Problems alle in der Menge der Lösungen des gelockerten Problems befinden. Ist dies der Fall, dann ist die Lösung des gelockerten Problems also auf jeden Fall eine gültige obere Schranke für das ursprüngliche Problem.

Das Lower Bound Modul soll eine untere Schranke für die optimale Lösung des Problems berechnen. Umgangssprachlich ist dies eine „So gut gehts es unter den gegebenen Bedingungen auf jeden Fall“-Schranke. Oft werden dazu Greedy-Algorithmen benutzt, welche eine gültige Lösung effizient berechnen. Diese Lösung muss nicht die optimale Lösung sein, stellt jedoch auf jeden Fall eine gültige Lösung und somit eine untere Schranke für die optimale Lösung dar.

Wenn eine untere Schranke L und eine obere Schranke U gegeben ist, muss die Lösung also irgendwo zwischen L und U liegen. Damit kann die optimale Lösung höchstens $U - L$ von

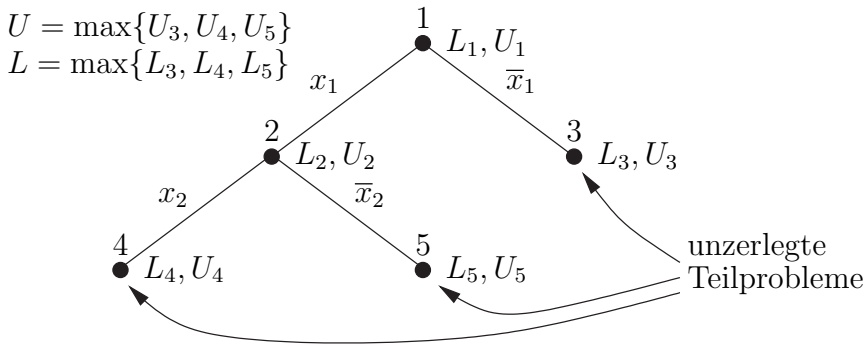


Abbildung 12: Beispiel für eine Branch-and-Bound Zerlegung

der bis jetzt berechneten Lösung entfernt sein. Es kann jedoch auch sein, dass die optimale Lösung schon gefunden wurde, dann muss nur noch gezeigt werden, dass dies auch die optimale Lösung ist. Sicher ist dies erst, wenn $U = L$ gilt. Wenn für die Schranken $U - L > 0$ gilt, dann kann die Berechnung also abgebrochen werden, wenn die Lösung gut genug erscheint, oder aber weiter gerechnet werden bis $U = L$ gilt.

Die Aufgabe des Branching Modul besteht darin, aus einem (Teil-)Problem Teilprobleme vom selben Typ zu erzeugen. Ein Problem P' ist ein Teilproblem von P, wenn gilt:

- Die Menge der zulässigen Lösungen von P' ist eine *echte* Teilmenge der in P zulässigen Lösungen: $L(P') \subset L(P)$.
- Jede zulässige Lösung in P' hat denselben Wert wie in P.

Es wird im Folgenden nun von zerlegten und unzerlegten Teilproblemen gesprochen. Durch die Zerlegungen entsteht ein Baum von Problemen und den dazu gehörigen Teilproblemen. Die unzerlegten Probleme sind in diesem Baum die Blätter während alle anderen Knoten die zerlegten Probleme darstellen. Zu jedem unzerlegten Teilproblem i wird die obere Schranke U_i und die untere Schranke L_i berechnet. Die gesamte obere Schranke U ergibt sich dann als das Maximum aller U_i. Der Grund dafür liegt darin, dass das Upper Bound Modul ja mit einer Relaxation arbeitet und man also nicht wirklich weiß, ob die berechnete Lösung auch eine gültige Lösung für das tatsächliche Problem ist. Wird nach einem Branch eine Lösung $U_i > U$ berechnet (z.B. weil das Upper Bound Modul sehr „schlampig“ arbeitet), so wird der Wert für U_i auf U gesetzt. Denn wir wissen ja schon, dass dies eine obere Schranke für die Gesamtlösung ist. Die untere Schranke ist ebenfalls das Maximum aller L_i. Dies ist die beste Lösung, die auch auf jeden Fall eine korrekte Lösung für das Gesamtproblem ist. Abbildung 12 zeigt, wie eine Zerlegung zu einem bestimmten Zeitpunkt aussehen könnte. Es gilt also stets:

$$L_{alt} \leq L_{neu} \leq U_{neu} \leq U_{alt}$$

Die untere und die obere Schranke bewegen sich also langsam aber sicher aufeinander zu und es gilt, dass $U_{neu} - L_{neu}$ nicht größer ist als $U_{alt} - L_{alt}$. Das Verfahren ist endlich, da es irgendwann für jedes Teilproblem nur noch eine Lösung gibt. In diesem Fall ist $U_i = L_i$ und das Teilproblem i kann als gelöst betrachtet werden. Dabei ist zu beachten, dass eine optimale Lösung für das Gesamtproblem erst gefunden ist, wenn $U = L$ gilt.

Wenn irgendwann alle möglichen Teilprobleme gelöst sind, dann entspricht dies einer erschöpfenden Suche nach einer Lösung und man hat nichts gewonnen. Teilprobleme i mit

$U_i = L_i$ sind jedoch nicht die einzigen Probleme, die nicht mehr betrachtet werden müssen. Wenn für ein Teilproblem $U_i < L$ gilt, dann muss dieses Teilproblem auch nicht mehr weiter betrachtet werden, da schon eine Lösung gefunden wurde, die alle Lösungen im Teilproblem i übertrifft. Dieser Zweig kann somit also „abgeschnitten“ werden, d.h. wir müssen ihn nicht mehr weiter betrachten. Das Ziel ist es, möglichst viele solcher Zweige abzuschneiden und somit Arbeit zu sparen. Dies geht entweder über Zerlegungen, die die oberen Schranken möglichst schnell senken oder indem möglichst optimale Lösungen berechnet und damit L gehoben wird. Die verwendete Strategie wird im Endeffekt durch das Search Modul bestimmt.

Das Search Modul hat die Aufgabe, zu entscheiden, welches unzerlegte Teilproblem als nächstes zerlegt wird. Dabei können zwei grobe Strategien unterschieden werden. Wenn die berechneten unteren Schranken gut sind, also recht nahe am Optimum liegen, muss die obere Schranke U möglichst schnell gesenkt werden. Denn es kann sein, dass die optimale Lösung schon gefunden wurde und nur noch gezeigt werden muss, dass diese tatsächlich optimal ist. Dazu wird das Teilproblem i mit $U_i = U$ zerlegt. Das Upper Bound Modul hat für dieses Teilproblem den maximalen U-Wert geliefert. Wenn dieses Problem zerlegt wird, besteht die Hoffnung, dass die zerlegten Teilprobleme j alle eine obere Schranke $U_j < U$ liefern, so dass der Wert der oberen Schranke gesenkt werden kann. Diese Strategie entspricht also quasi einer Greedy-Strategie. Sind die unteren Schranken jedoch nicht so gut, dann sind wir überhaupt erst einmal an einer guten Lösung interessiert, um Zweige mit $U_i < L$ abschneiden zu können. Dazu wird ein Zweig so lange zerlegt, bis das zu ihm gehörige Teilproblem gelöst ist.

4.2 Ein Branch-and-Bound Algorithmus für das Rucksackproblem

Frage 4.3: Beschreibe die Branch-and-Bound Module für das Rucksackproblem!

Gegeben sind n Objekte $\{1, \ldots, n\}$, wobei das Objekt i das Gewicht g_i und den Nutzen v_i besitzt. Es soll nun eine Bepackung für einen Rucksack gefunden werden, welche die Gewichtsschranke G des Rucksacks respektiert und den Nutzen dabei maximiert. Das Rucksackproblem kann als Problem der binären Optimierung (bzw. ganzzahligen Optimierung, siehe Kapitel 6.1) beschrieben werden:

$$\begin{array}{ll} & v_1 x_1 + \cdots + v_n x_n \to \max \\ \text{wobei} & g_1 x_1 + \cdots + g_n x_n \leq G \\ \text{und} & x_1, \ldots, x_n \in \{0, 1\} \end{array}$$

Der Lösungsraum besitzt 2^n Elemente und das Problem ist NP-hart. Wie kann nun eine Relaxation für das Problem aussehen? Das Problem wird unter anderem dadurch schwer, dass wir ein Objekt entweder ganz einpacken können oder garnicht. Wir lockern also $x_1, \ldots, x_n \in \{0, 1\}$ auf und ersetzen diese Forderung durch:

$$0 \leq x_1, \ldots, x_n \leq 1$$

Objekte können jetzt also auch zerschnitten bzw. teilweise in den Rucksack gepackt werden. Dadurch wird der Lösungsraum zwar überabzählbar unendlich, jedoch kann eine Lösung nun mit einem Greedy Algorithmus in linearer Zeit berechnet werden. Damit das Upper Bound und das Lower Bound Modul richtig arbeiten können, bringen wir die Objekte noch in eine bestimmte Reihenfolge. Wir berechnen zu jedem Objekt i seine Effizienz $e_i = v_i/g_i$ und sortieren die Objekte nach fallender Effizienz. Entsprechend werden die Objekte umnummeriert,

Algorithmus 4 Der Algorithmus des Upper Bound Modul

1.) Bestimme das größte i mit $g_1 + \cdots + g_i \leq G$ und berechne $G^* = G - (g_1 + \cdots + g_i)$
2.) Falls $i = n$, dann setze $U = v_1 + \cdots + v_n$
3.) Falls $i \neq n$, berechne $x_{i+1} = G^*/g_{i+1}$ und setze $U = v_1 + \cdots + v_i + v_{i+1}x_{i+1}$

d.h. es gilt:

$$e_1 \geq e_2 \geq ... \geq e_n$$

Das Upper Bound Modul packt die Objekte der Reihe nach ein, bis die Gewichtsschranke G erreicht ist. Dabei wird das letzte Objekt unter Umständen zerschnitten. Algorithmus 4 beschreibt die Vorgehensweise genauer. Der Algorithmus arbeitet korrekt, da wir uns vorstellen können, dass das Objekt i in g_i Teilobjekte mit Gewicht 1 und Nutzen e_i zerlegt wird. Der Nutzen wird in diesem Fall maximiert, wenn die G nützlichsten Objekte eingepackt werden. Da die Objekte nach Effizienz sortiert sind, müssen die Objekte also der Reihe nach eingepackt werden, bis eines unter Umständen zerschnitten wird oder alle Objekte eingepackt sind.

Beim Lower Bound Modul werden die Objekte ebenfalls nach fallender Effizienz betrachtet. Es wird eine Greedy Algorithmus benutzt der die Objekte der Reihe nach betrachtet und dabei jedes Objekt, das noch passt, in den Rucksack packt. Dabei werden die unzerschnittenen Objekte $1, \ldots, i$, welche das Upper Bound Modul eingepackt hat, also auf jeden Fall eingepackt. Wird bei der Berechnung der oberen Schranke ein Objekt zerschnitten, wird dies bei der Berechnung der unteren Schranke natürlich nicht mehr eingepackt. Danach können aber eventuell noch Objekte eingepackt werden, die zwar nicht mehr so effizient sind, jedoch unter Umständen noch leicht genug sind und somit passen. Es gilt für die untere Schranke L also in jedem Fall:

$$L \geq v_1 + \cdots + v_i$$

Damit gilt für die Differenz der unteren und oberen Schranke $U - L \leq v_{i+1}$, wobei i der in Algorithmus 4 berechnete Wert ist.

Das Search Modul benutzt die Greedy Strategie, da die L-Werte, wie wir etwas weiter oben gesehen haben, recht gut sind.

Beim Branching wird jeweils ein Objekt j ausgewählt, welches in einem Fall komplett eingepackt ($x_j = 1$) und im anderen Fall ignoriert ($x_j = 0$) wird. Dadurch wird der Lösungsraum disjunkt in zwei gleich große Teile zerlegt. Realisiert wird die Zerlegung in beiden Fällen, indem das Objekt aus der Menge der Objekte entfernt wird. Zur Inklusion wird dabei jedoch die neue Gewichtsgrenze $G - g_j$ benutzt und der Nutzen v_j des Objekts zu allen Schranken hinzu addiert. Es bleibt nur noch die Frage, welches Objekt j ausgewählt wird. Da die Exklusion sehr effizienter Objekte und die Inklusion sehr ineffizienter Objekte wahrscheinlich schlechte Ergebnisse liefert, wählen wir das Objekt, welches bei der Berechnung der Upper Bound zerschnitten wurde. Denn es scheint eine kritische Entscheidung zu sein, ob es eingepackt wird oder nicht.

4.3 Branch-and-Bound Algorithmen für das TSP

Frage 4.4: Was muss bei dem Branch-and-Bound Algorithmus für das TSP beachtet werden?

Beim Traveling Salesman Problem (TSP) ist ein Graph mit Kantenbewertung gegeben, für den ein Hamiltonkreis, also eine Rundreise, mit minimalen Kosten berechnet werden soll. Es handelt sich um ein Minimierungsproblem und dementsprechend sind die Rollen für das Upper Bound und das Lower Bound Modul vertauscht. Mit Hilfe des Lower Bound Moduls wird eine relaxierte Lösung berechnet, also eine Lösung, welche die Kosten unter Umständen unterschätzt. Das Upper Bound Modul berechnet eine gültige Lösung mit Hilfe eines Greedy Algorithmus. Es wird versucht, die untere Schranke nach oben zu bewegen, wobei L das Minimum aller L_i ist, und die obere Schranke nach unten zu bewegen. Auch hier ist das Minimum aller U_i die globale obere Schranke U.

Frage 4.5: Was ist die Idee des Lower Bound Moduls beim symmetrischen TSP?

Beim symmetrischen TSP gilt für zwei Kanten (i, j) und (j, i), dass sie die gleichen Kosten haben. Es ist also genau so teuer, von i nach j zu kommen, wie von j zu i. Oder etwas formaler: $c(i, j) = c(j, i)$. Existiert zwischen zwei Knoten i und j keine Kante, so setzen wir die Kosten auf ∞.

Wie schon erwähnt, wird beim symmetrischen TSP mit einer Relaxation im Lower Bound Modul gearbeitet. Wie können wir das Problem des TSP nun vereinfachen bzw. die Kosten einer Rundreise nach unten abschätzen? Eine Rundreise ist ein Hamiltonkreis, d.h. ein Kreis, bei dem jeder Knoten genau einmal besucht wird. Wird in diesem Kreis nun eine Kante entfernt, so entsteht ein Spannbaum, dessen Kosten geringer sind als die Kosten der Rundreise. Dieser Spannbaum hat jedoch ein sehr spezielles Aussehen und es ist sehr wahrscheinlich, dass es einen Spannbaum mit geringeren Kosten gibt. Mit Hilfe des Algorithmus von Kruskal kann ein minimaler Spannbaum berechnet werden. Die Kosten dieses Spannbaums stellen dann also auf jeden Fall eine untere Schranke für die Kosten einer Rundreise dar. Allerdings haben wir das Problem, dass Touren keine Spannbäume sind, d.h. wenn wir Spannbäume als Relaxation benutzen, dann bewegen sich die Probleme nicht aufeinander zu. Dies kann man sich an einem Beispiel recht schnell klar machen. Gegeben sei ein Graph mit n Knoten, in dem eine Rundreise exisitert. Die Kosten einer jeden Kante betragen 1. Die Kosten eines minimalen Spannbaums belaufen sich dann auf $n - 1$, während die Kosten einer jeden Rundreise n betragen. Die Kosten fallen also nie zusammen. Wir brauchen also noch einen kleinen „Trick“ in Form der 1-Bäume.

Frage 4.6: Wie sind 1-Bäume definiert? Wie werden sie benutzt?

Ein 1-Baum ist ein Spannbaum auf den Knoten $2, \ldots, n$ mit zwei zusätzlichen Kanten zum Knoten 1. Ein minimaler 1-Baum kann berechnet werden, indem ein minimaler Spannbaum auf den Knoten $2, \ldots, n$ berechnet wird und die zwei billigsten Kanten zum Knoten 1 hinzugefügt werden. In welchem Zusammenhang stehen TSP-Touren nun mit 1-Bäumen? Jede Tour enthält zwei Kanten die zum Knoten 1 führen. Werden diese entfernt, bleibt ein Spannbaum auf den Knoten $2, \ldots, n$. Also ist eine Tour ein 1-Baum und die Berechnung minimaler 1-Bäume ist ein Relaxation für das TSP. Die Probleme fallen dann zusammen, wenn im minimale 1-Baum jeder Knoten den Grad 2 hat. In diesem Fall entspricht der minimale 1-Baum einer Tour.

Im Folgenden werden wir daher stets versuchen, dass die 1-Bäume Ähnlichkeit mit einer Tour haben. Das Ziel ist es also im Endeffekt, dafür zu sorgen, dass alle Knoten Grad 2 haben.

Frage 4.7: Wie wird die Upper Bound beim symmetrischen TSP berechnet?

Für die Berechnung der Upper Bound bleibt uns leider nichts anderes übrig, als einen Greedy Algorithmus zu verwenden. Greedy Algorithmen liefern beim TSP keine guten Ergebnisse aus diesem Grund ist eine Tiefensuche als Suchstrategie für das Search Modul vorzuziehen.

Frage 4.8: Wie funktioniert das Branching beim symmetrischen TSP? Wie werden Teilprobleme gebildet?

Teilprobleme werden durch Inklusion und/oder Exklusion von Kanten gebildet. Für das k-te Problem P_k existieren die folgenden Kantenmengen:

E_k: Die Menge der verbotenen bzw. durch Exklusion ausgeschlossenen Kanten.

I_k: Die Menge der erzwungenen Kanten.

F_k: Die Menge der freien Kanten, für die noch nichts festgelegt wurde. Diese Menge ergibt sich als $F_k = E - (E_k \cup I_k)$.

Frage 4.9: Wie funktioniert die erste Branching-Regel für die 1-Baum-Relaxation?

Bei der ersten Branching-Regel wird eine Kante ausgewählt, die in einem Fall verboten und im anderen Fall erzwungen wird. Das ursprüngliche Problem wird also in zwei disjunkte Teilprobleme geteilt. Dabei wird der optimale 1-Baum T betrachtet. In ihm suchen wir einen Knoten i der nicht Grad 2 hat und für den es eine freie Kante (i, j) in T gibt. Unter all diesen den Knoten i berührenden freien Kanten (i, l) wählen wir die Kante, die die geringsten Kosten hat. Diese wird nun einmal zu I_k und einmal zu E_k hinzugefügt.

Der Nachteil diesem Vorgehen ist, dass der durch Inklusion entstehende Teilbaum mit dem ursprünglichen Teilbaum identisch ist. Dies liegt daran, dass die entsprechende Kante eh schon im minimalen 1-Baum enthalten war und dementsprechend auch im neuen minimalen 1-Baum enthalten sein wird.

Frage 4.10: Was ist die Aufgabe des Learning Moduls?

Wie etwas weiter oben schon erwähnt wurde, besteht unser Ziel darin, minimale 1-Bäume zu erhalten, die auch Touren sind. In einer Tour hat jeder Knoten Grad 2. Dementsprechend können die beiden folgenden Konsequenzen aus getroffenen Branching-Entscheidung gezogen werden:

- Wurden an einem Knoten i bereits zwei Kanten $\{i, j_1\}$ und $\{i, j_2\}$ erzwungen, können alle anderen Kanten an i verboten werden.
- Sind alle bis auf zwei Kanten an i verboten, so können diese beiden Kanten erzwungen werden.

Die Gründe dürften auf der Hand liegen. Werden mehr als zwei Kanten erzwungen, so kann der resultierende 1-Baum garantiert keine Tour werden, da in einer Tour jeder Knoten nur Grad 2 hat. Andererseits können auch Knoten, bei denen nur noch eine Kante gewählt werden kann, keine Touren werden, d.h. spätestens wenn alle bis auf zwei Knoten verboten sind, müssen wir uns für diese entscheiden.

Es ist nun die Aufgabe des Learning Moduls diese Fälle zu erkennen. Auf diese Art und Weise kann die Tiefe des Branch-and-Bound Baumes erheblich gesenkt und viele unnötige Berechnungen gespart werden. Lediglich der Name „Learning Modul“ ist etwas unglücklich, da das Modul selbst ja nichts Neues lernt.

Frage 4.11: Wie funktioniert die zweite Branching-Regel für die 1-Baum-Relaxation?

Bei der zweiten Branching-Regel wird das Problem P_k betrachtet. Dieses wird nun in q weitere Teilprobleme zerlegt. Dazu wird der optimale 1-Baum für das Problem P_k betrachtet. In diesem werden die freien Kanten F_k nach aufsteigenden Kosten sortiert und anschließend so lange Kanten gewählt, bis für einen Knoten p der folgende Fall eintritt. In I_k gibt es keine zwei Kanten an p. Nach Inklusion der gewählten Kanten ist dies der Fall. Die bis zu diesem Punkt gewählten Kanten seien die Kanten $\{i_1, j_1\}, \ldots, \{i_q, j_q\}$. Es werden nun q neue Teilprobleme $P_{k,1}, \ldots, P_{k,q}$ gebildet, die die folgenden Mengen E und I zugeordnet bekommen:

$$\begin{aligned} I_{k,r} &:= I_k \cup \{\{i_h, j_h\} \mid 1 \leq h \leq r-1\} \text{ für } 1 \leq r \leq q \\ E_{k,r} &:= E_k \cup \{i_r, j_r\} \text{ für } 1 \leq r \leq q-1 \\ E_{k,q} &:= E_k \cup \{\{i, j\} \mid \{i, j\} \notin I_{k,q} \wedge (i = p \vee j = p)\} \end{aligned}$$

Im r-ten Fall wird also die Kante $\{i_r, j_r\}$ verboten und alle Kanten davor, also die Kanten $\{i_1, j_1\}, \ldots, \{i_{r-1}, j_{r-1}\}\}$, erlaubt. Eine Ausnahme wird für $r = q$ gemacht. In diesem Fall werden alle an p anliegenden Kanten, die nicht explizit erzwungen wurden, verboten. Im Prinzip wird dadurch also die Regel des Learning Moduls realisiert, die besagt, dass alle anderen Kanten an einem Knoten verboten werden, wenn zwei Kanten erzwungen werden. Der Vorteil liegt darin, dass in jedem Teilproblem eine Kante des minimalen 1-Baums verboten wird und daher ein anderer 1-Baum berechnet werden muss. Man kann also nicht auf einem schlechten 1-Baum „sitzen bleiben“. Desweiteren sind alle Teilprobleme disjunkt. Der Nachteil besteht darin, dass q meist sehr groß ist.

Frage 4.12: Warum werden Strafkosten eingeführt?

Wie schon erwähnt, ist nicht jeder 1-Baum eine Tour, da nicht jeder Knoten den Grad 2 besitzen muss. Es ist sogar sehr oft so, dass optimale 1-Bäume zentrale Knoten mit einem hohen Grad und dezentrale Knoten mit einem Grad von eins besitzen. Aus diesem Grund sind die Kosten eines 1-Baumes oft sehr weit von den Kosten einer Tour entfernt. Unser Ziel ist es jedoch Touren zu berechnen. Dazu wollen wir nun 1-Bäume, die Touren sind oder Touren zumindest schon recht ähnlich sehen, belohnen und 1-Bäume, die Touren sehr unähnlich sind (z.B. weil sie zentrale Knoten mit hohem Grad besitzen), bestrafen. Dies wird über Strafkosten gemacht. Negative Strafkosten können dabei als Belohnung aufgefasst werden.

Formal wird ein anderes Problem P' mit einer anderen Kostenfunktion $c'(i, j)$ betrachtet. Dabei werden für jeden Knoten i die Strafkosten $\lambda_i \in \mathbb{R}$ festgelegt. Wir setzen:

$$c'(i, j) = c(i, j) + \lambda_i + \lambda_j$$

Das wichtige hierbei ist, dass Touren π stets gleich stark bestraft werden. Dies liegt daran, dass in einer Tour an jedem Knoten genau zwei Kanten anliegen und seine Strafe damit genau zweimal in die Kostenfunktion eingeht. Ist also $\lambda = \lambda_1 + \cdots + \lambda_n$ die Summe aller Strafkosten, so gilt:

$$c'(\pi) = c(\pi) + 2\lambda$$

Die Kosten der Tour mit Strafkosten sind also genau um $2 \cdot (\lambda_1 + \cdots + \lambda_n)$ höher als die Kosten der Tour ohne Strafen. Wenn wir also eine untere Schranke für P' berechnen, erhalten wir über $L = L' - 2\lambda$ eine untere Schranke für P. Dies gilt, weil alle Touren exakt dieselben Strafkosten erhalten. Im Prinzip muss man sich jetzt also nur noch geschickt anstellen und dafür sorgen, dass 1-Bäume, die keine Touren sind, stärker bestraft werden. Im Folgenden ist

T der 1-Baum, der ohne die Strafkosten entsteht, und T' der 1-Baum, der mit den Strafkosten entsteht. Es wird hier nur eine Strategie für die Strafkosten vorgestellt.

Die Idee besteht quasi darin, die Knoten linear in ihrem Grad zu bestrafen. Wenn $d_T(i)$ den Grad des Knotens i im 1-Baum T beschreibt und γ eine Konstante, dann berechnen wir die Strafkosten des Knotens i als:

$$\lambda_i = \gamma \cdot (d_T(i) - 2)$$

Ein Knoten mit Grad 2 erhält also keine Strafe und ein Knoten mit Grad 1 wird sogar belohnt, indem er eine negative „Strafe“ von $-\gamma$ erhält. Knoten mit einem Grad größer als zwei erhalten eine positive Strafe, die ein Vielfaches von γ ist und direkt vom Grad abhängt. Dabei ist die Wahl von γ kritisch, Wird es zu klein gewählt, hat es kaum Einfluss und wenn es zu groß gewählt wird, „kippen“ die Knoten quasi. Das heißt, dass Knoten mit Grad eins so begehrt sind, dass sie im resultierenden Baum zentrale Knoten werden, während zentrale Knoten zu dezentralen Knoten werden.

Frage 4.13: Worin unterscheidet sich das Branch-and-Bound Verfahren für das asymmetrische TSP?

Im Prinzip ändert sich beim asymmetrischen TSP nicht viel. Es muss nur eine andere Relaxation benutzt werden, da der Graph nur als gerichtet interpretiert werden muss. Die Relaxation ist in Form des Zuordnungsproblem gegeben, welches ein spezielles Problem der binären Optimierung ist und mit Hilfe der ungarischen Methode in $O(n^3)$ berechnet werden kann. Damit ist die Relaxation etwas schwerer zu berechnen als im symmetrischen Fall.

Es wird nun das Zuordnungsproblem vorgestellt. Beim Zuordnungsproblem geht es darum, zu jedem Knoten des Graphen genau eine eingehende und eine ausgehende Kante zu wählen, so dass die entstehenden Gesamtkosten minimal sind. Dadurch entstehen unter Umständen mehrere disjunkte Kreise. Warum kann diese Relaxation nun benutzt werden? Beim TSP geht es darum, *einen* Kreis zu finden, der minimale Kosten hat. Da die Gesamtkosten mehrerer disjunkter Kreise jedoch günstiger sein können, stellen diese Kosten also eine untere Schranke für das TSP dar. Wir hoffen also, dass der Fall eines Kreises sehr schnell eintritt. Spätestens nach sehr vielen erzwungenen bzw. verbotenen Kanten wird dieser Fall auftreten müssen. Die jeweilige Auswahl der Kanten wird durch n^2 Variablen $x_{ij} \in \{0, 1\}$ beschrieben, wobei die Variable x_{ij} angibt, ob die Kante (i, j) gewählt wird oder nicht. Wird die Kante (i, j) gewählt, gilt $x_{ij} = 1$, ansonsten $x_{ij} = 0$. Wir beschreiben das Zuordnungsproblem nun noch einmal formal:

$$\sum_{i \neq j} x_{ij} c(i, j) \to \min \text{, wobei}$$

$$\begin{aligned}
\sum_{1 \leq i \leq n} x_{ij} &= 1 \text{ für } 1 \leq j \leq n, \\
\sum_{1 \leq j \leq n} x_{ij} &= 1 \text{ für } 1 \leq i \leq n, \\
x_{ij} &\in \{0, 1\}
\end{aligned}$$

Wie schon erwähnt, lässt sich dieses Problem mit Hilfe der ungarischen Methode in $O(n^3)$ lösen. Es ist jedoch möglich, aus einem gelösten Zuordnungsproblem Teilprobleme zu konstruieren, die sich in Zeit $O(n^2)$ lösen lassen. Dies ist der Fall, wenn eine Kante aus der

errechneten Lösung verboten wird und nur Kanten innerhalb der Lösung erzwungen werden. Andere Kanten, die verboten werden, müssen außerhalb der Lösung liegen. Damit kann man sich also ein Bild davon machen, wie Branching-Regeln aussehen sollten, damit Teilprobleme relativ effizient gelöst werden können.

Frage 4.14: Wie sehen die beiden Branching-Regeln beim asymmetrischen TSP aus?

Wie in der Antwort zur vorherigen Frage schon erwähnt wurde, kann in einigen Fällen in quadratischer Laufzeit die Lösung eines Teilproblems berechnet werden, wenn die Lösung des ursprünglichen Problems bekannt ist. Wir sollten diese günstigen Fälle also beim Branching berücksichtigen.

Wenn wir eine Kante aus einer berechneten Lösung verbieten, dann bedeutet dies in Endeffekt, dass wir einen Kreis aufbrechen. Dies ist für das Branching insofern vorteilhaft, als dass wir im Falle des Verbots einer Kante, die in der Lösung vorkommt, garantiert nicht wieder dieselbe Lösung berechnen. Eine erste einfache Idee besteht also darin, einen Kreis mit q Kanten zu betrachten und q neue Teilprobleme zu bilden, in denen jeweils eine dieser Kanten verboten wird. Damit nicht zu viele neue Teilprobleme entstehen sollte stets der kürzeste Kreis aufgebrochen werden. Diese Lösung hat jedoch den Nachteil, dass die Teilprobleme nicht disjunkt sind. Ein einfaches Beispiel macht dies schnell klar. In einer berechneten Lösung des Zuordnungsproblems komme ein Kreis mit den Kanten $\{1, 2, 3, 4\}$ vor. Jede dieser Kanten wird nun in einem Teilproblem verboten werden. Wenn nun jedoch z.B. in der Lösung, die für die verbotene Kante 1 berechnet wurde, die Kante 3 wieder vorkommt und in der Lösung für die verbotene Kante 3 die Kante 1, dann sind wir also irgendwann in der Situation, dass für zwei Teilprobleme die Kanten 1 und 3 verboten werden. Diese Lösungen werden also mehrfach berechnet.

Das Problem der nicht disjunkten Teilproblem kann dadurch verhindert werden, dass beim Verbot der r-ten Kante die $r - 1$ Kanten davor durch Inklusion erzwungen werden. Im etwas weiter oben erwähnten Beispiel kann es dann also nicht mehr passieren, dass nach dem Verbot der Kante 3 die Kante 1 auch noch verboten wird, da diese in dem Teilproblem, in dem 3 verboten wird, explizit erzwungen wird.

5 Approximationsalgorithmen

5.1 Approximationsalgorithmen für das metrische TSP

Frage 5.1: Wie wird die Güte einer Lösung gemessen?

Wenn es für ein Optimierungsproblem schwer ist, eine exakte Lösung zu berechnen, kann man sich mit Approximationslösungen zufrieden geben. Dies sind Lösungen, die nicht optimal sind, aber unter Umständen schon recht nah an einer optimalen Lösung liegen. In diesem Kapitel werden Algorithmen behandelt, die dabei eine gewisse Güte der Lösung garantieren.

Die Güte einer Lösung wird bei Minimierungsproblemen als der Quotient aus der berechneten Lösung und der optimalen Lösung angegeben. Bei Maximierungsproblemen wird dagegen der Quotient aus der optimalen Lösung und der berechneten Lösung benutzt. Dieses Vorgehen hat den Vorteil, dass berechnete Lösungen, die optimal sind, stets eine Güte von 1 haben, während Lösungen, die nicht optimal sind, stets einen Gütewert größer als 1 haben. Ein weiterer Vorteil des Quotienten ist, dass er gegenüber Änderungen der Maßeinheiten unempfindlich ist. Würde z.B. die Differenz zwischen optimaler und berechneter Lösung benutzt werden, wäre dies nicht der Fall. Für ein Problem sei der Wert der berechneten Lösung 140 und der Wert der optimalen Lösung 100. Die Differenz zwischen den beiden ist 40. Werden die Werte des Problems nun jedoch alle mit dem Faktor 10 multipliziert, wäre der Wert der berechneten Lösung 1400 und der Wert der optimalen Lösung 1000. Nun ist die Differenz jedoch 400. Der Quotient ist gegen solche Änderungen unempfindlich, da z.B. $\frac{140}{100} = \frac{1400}{1000} = 1.4$ ist. Da der Wert einer optimalen Lösung nicht immer bekannt ist, kann die Güte einer Lösung oft nur abgeschätzt werden.

Frage 5.2: Wann ist ein Algorithmus ein Approximationsalgorithmus mit einer bestimmten Güte?

Ein Algorithmus A für ein Optimierungsproblem ist ein Approximationsalgorithmus mit Güte $r_A(n)$, wenn er stets Lösungen liefert, deren Güte durch $r_A(n)$ beschränkt ist.

Frage 5.3: Wie ist das metrische TSP definiert? Welche Approximationsalgorithmen kennst du?

Das metrische TSP ist ein Spezialfall des TSP, in dem die Kostenrelation symmetrisch ist, d.h. es gilt $c(i,j) = c(j,i)$, und in dem die Dreiecksungleichung gilt: $c(i,j) \leq c(i,k) + c(k,j)$. Ein direkter Weg zwischen zwei Knoten i und j ist also immer mindestens so günstig, wie ein Umweg über einen anderen Knoten k. Für das metrische TSP gibt es zwei Approximationsalgorithmen, die in der Idee recht ähnlich sind. Grob betrachtet, berechnen beide einen Spannbaum, aus dem ein Multigraph berechnet wird, in welchem wiederum ein Eulerkreis berechnet wird. Als letzter Schritt wird aus dem Eulerkreis eine Tour berechnet. Der erste Algorithmus erreicht eine Güte von 2 und der zweite Algorithmus, der zusätzlich noch ein Matching berechnet, eine Güte von $\frac{3}{2}$.

Frage 5.4: Was ist ein Eulerkreis?

Ein Eulerkreis ist ein Kreis, in dem Knoten mehrfach vorkommen dürfen, der jedoch jede Kante genau einmal enthalten muss. In einem Multigraphen (zwischen zwei Knoten dürfen mehrere Kanten verlaufen) existiert genau dann ein Eulerkreis, wenn er bis auf isolierte Knoten zusammenhängend ist und alle Knoten einen geraden Grad haben. In diesem Fall lässt sich ein Eulerkreis in linearer Zeit $O(n+m)$ berechnen.

Frage 5.5: Wie funktioniert der Approximationsalgorithmus mit Güte 2 für das metrische TSP?

Der Algorithmus besteht aus vier Schritten, die nun kurz beschrieben werden. Anschließend wird erklärt, warum der Algorithmus funktioniert und die Güte von 2 erreicht.

Algorithmus 5 Der Approximationsalgorithmus für das metrische TSP mit Güte 2

1.) Berechne einen minimalen Spannbaum S_{opt}.
2.) Berechne den Multigraphen G, der jede Kante aus S_{opt} zweimal enthält.
3.) Berechne einen Eulerkreis EK auf G.
4.) Berechne aus EK eine Tour π mit $C(\pi) \leq C(EK)$.

Die Berechnung des Spannbaums kann mit Hilfe des Algorithmus von Prim in Zeit $O(n^2)$ geschehen. In einem Spannbaum ist jeder Knoten des ursprünglichen Graph enthalten und die Kosten des Spannbaums sind nicht höher als die Kosten einer optimalen Tour. Da ein Spannbaum $n-1$ Kanten und n Knoten besitzt, kann der zweite Schritt in $O(n)$ durchgeführt werden. Der zweite Schritt ist nötig, damit in Schritt 3 ein Eulerkreis EK berechnet werden kann. Diese Berechnung benötigt Zeit $O(n+m)$ bzw. in diesem Fall Zeit $O(n)$, da $m = O(n)$ ist. Da die Kanten verdoppelt werden und jede Kante in einem Eulerkreis enthalten ist, betragen die Kosten des Eulerkreis also das doppelte der Kosten des Spannbaums, also $C(EK) = 2 \cdot C(S_{opt})$. Daraus folgt jedoch $C(EK) \leq 2 \cdot C(\pi_{opt})$, da die Kosten des Spannbaums S_{opt} ja höchstens so hoch sind wie die Kosten einer optimalen Tour π_{opt}. Im letzten Schritt wird aus dem Eulerkreis eine Tour π berechnet, deren Kosten höchstens $C(EK)$ und damit höchstens das zweifache einer optimalen Tour betragen. Dies wird gemacht, indem der Eulerkreis EK durchlaufen wird und jeder Knoten, der nicht zum ersten Mal erreicht wird, ausgelassen wird. Eine Ausnahme bildet dabei der Endknoten. Aufgrund der geltenden Dreiecksungleichung sind die Kosten der so entstehenden Tour höchstens so groß wie die Kosten des Eulerkreises. Angenommen, der Knoten k wurde schon besucht, der Knoten j ist der nächste noch nicht besuchte Knoten auf dem Eulerkreis und wir befinden uns im Knoten i. Wenn also die Kanten (i,k) und (k,j) im Eulerkreis EK direkt aufeinander folgen, dann gilt aufgrund der Dreiecksungleichung $c(i,j) \leq c(i,k) + c(k,j)$. Wenn wir den Knoten k auf dem Eulerkreis einfach auslassen, sind die Kosten, direkt vom Knoten i zum Knoten j zu gehen, also nicht höher. Da der vierte Schritt ebenfalls in linearer Zeit durchgeführt werden kann, beträgt die Gesamtlaufzeit des Algorithmus $O(n^2)$.

Frage 5.6: Wie funktioniert der Approximationsalgorithmus mit Güte 3/2 für das metrische TSP?

Der Nachteil des oben beschriebenen Approximationsalgorithmus mit Güte 2 ist, dass für *jeden* Knoten der Grad verdoppelt wird. Der Algorithmus von Christofides verhindert dies und kommt so auf eine Güte von $\frac{3}{2}$. Er funktioniert so ähnlich wie der Algorithmus mit Güte 2, nur dass nicht für alle Knoten der Grad verdoppelt wird. Damit dies funktioniert, muss jedoch im Verlauf des Algorithmus ein Matching mit minimalen Kosten auf einer Knotenmenge (die eventuell V mit $|V| = n$ sein kann) berechnet werden, was zu einem Aufwand von $O(n^3)$ führt.

Schritt 1 geht auch hier in Zeit $O(n^2)$. Da der Spannbaum S_{opt} aus n Knoten und $n-1$ Kanten besteht, kann die Menge M in Zeit $O(n)$ bestimmt werden. In Schritt 3 wird ein Matching auf M mit $\frac{|M|}{2}$ Knoten berechnet. Es muss also gezeigt werden, dass $|M|$ gerade ist, damit $\frac{|M|}{2}$ ganzzahlig ist. Zuerst bilden wir die Summe der Grade der Knoten in M. Da

Algorithmus 6 Der Approximationsalgorithmus für das metrische TSP mit Güte $\frac{3}{2}$

1.) Berechne einen minimalen Spannbaum S_{opt}.
2.) Berechne die Menge M der Knoten, die in S_{opt} einen ungeraden Grad haben.
3.) Berechne auf M ein Matching M_{opt} mit $\frac{|M|}{2}$ Kanten und minimalen Kosten. Dabei besteht ein Matching auf $2r$ Knoten aus r Kanten, die jeden Knoten berühren.
4.) Berechne den Multigraphen G, der aus einer Kopie von S_{opt} und einer Kopie von M_{opt} besteht.
5.) Berechne einen Eulerkreis EK auf G.
6.) Berechne aus EK eine Tour π mit $C(\pi) \leq C(EK)$.

jede Kante zweimal gezählt wird, muss diese Summe gerade sein. Andererseites wissen wir jedoch, dass der Grad aller Knoten in M ungerade ist. Da die Summe einer ungeraden Anzahl von ungeraden Zahlen ebenfalls ungerade ist, muss die Anzahl der Knoten in M also gerade sein, damit die Summe der Grade gerade ist. Die Berechnung eines Matching selbst geht in Zeit $O(n^3)$. Da der Graph G höchstens $\frac{3}{2}n$ Kanten enthalten kann, können die Schritte 4, 5 und 6 in Zeit $O(n)$ durchgeführt werden. Der Graph enthält höchstens $\frac{3}{2}n$ Kanten, da der schlimmste Fall, der auftreten kann, der ist, wenn alle Knoten in S_{opt} einen ungeraden Grad haben. Dies sind dann n Knoten, auf denen ein Matching mit $\frac{n}{2}$ Kanten berechnet wird. Die Kanten dieses Matching werden zu den $n-1$ Kanten des Spannbaum hinzugefügt. Also sind dies höchstens $\frac{3}{2} \cdot n - 1 = O(n)$ Kanten.

Es bleibt noch die Aussage über die Güte zu beweisen. Dazu benutzen wir die folgende Abschätzung:

$$C(\pi) \leq C(EK) = C(S_{opt}) + C(M_{opt}) \leq C(\pi_{opt}) + C(M_{opt})$$

Die Kosten der mit dem Algorithmus berechneten Tour sind also nicht größer als die Kosten des Eulerkreises. Dessen Kosten setzen sich jedoch aus den Kosten des minimalen Spannbaums und des minimalen Matchings zusammen. Die Kosten des Spannbaums sind auch hier wieder nicht größer als die Kosten einer optimalen Tour π_{opt}. Da wir $C(\pi) \leq \frac{3}{2} \cdot C(\pi_{opt})$ zeigen wollen, müssen wir also $C(M_{opt}) \leq \frac{1}{2} \cdot C(\pi_{opt})$ beweisen. Dazu betrachten wir die optimale Tour π_{opt}. In dieser kommen natürlich auch alle Knoten der Menge M vor, für die wir das Matching M_{opt} berechnet haben. Wir wollen nun der Reihe nach alle Knoten aus der optimalen Rundreise π_{opt} rausnehmen, die nicht in der Menge M vorkommen. Wenn wir einen solchen Knoten rausnehmen, werden die Kosten des verbleibenden Kreises aufgrund der Dreiecksungleichung nicht teurer. Der Kreis, der nur noch die Knoten aus M enthält, ist also höchstens so teuer wie die optimale Rundreise π_{opt}, mit der wir ja gestartet sind. In diesem Kreis gibt es dann zwei Matchings, die wir wie folgt erhalten. Wir wählen eine Kante A aus, laufen den Kreis ab und nehmen dabei jede zweite Kante in das Matching mit auf. Es ist offensichtlich, dass wir auf diese Art und Weise nur zwei verschiedene Matchings bekommen können. Die Summe der Kosten dieser beiden Matchings entspricht π_{opt}. Dann kann eines der beiden Matchings höchstens die Kosten $\frac{1}{2} \cdot C(\pi_{opt})$ haben, denn wenn beide echt teurer wären, würden wir höhere Kosten als $C(\pi_{opt})$ für den Kreis bekommen. Da das Matching, dessen Kosten höchstens $\frac{1}{2} \cdot C(\pi_{opt})$ betragen, nicht unbedingt das kostenminimale Matching M_{opt} auf M sein muss, betragen die Kosten von M_{opt} also ebenfalls höchstens $\frac{1}{2} \cdot C(\pi_{opt})$.

5.2 Ein echt polynomielles Approximationsschema für das Rucksackproblem

Frage 5.7: Was ist ein polynomielles Approximationsschema? Wann spricht man von einem echt polynomiellen Approximationsschema?

Es gibt NP-harte Optimierungsprobleme, bei denen es für jedes $\varepsilon > 0$ einen polynomiellen Approximationsalgorithmus mit Güte $1 + \varepsilon$ gibt. Dies bedeutet, dass man sich im Prinzip beliebig nah an eine optimale Lösung herantasten kann. Es wird von einem Approximationsschema oder PTAS (Polynomial Time Approximation Scheme) gesprochen, wenn es einen Algorithmus gibt, der neben der üblichen Eingabe ein ε als Eingabe erhält, und anschließend in polynomieller Zeit eine Lösung mit Güte $1 + \varepsilon$ berechnet.

Dabei kann es auch vorkommen, dass der Algorithmus zwar polynomiell ist, die Laufzeit jedoch von der Größenordnung $n^{\lceil 1/\varepsilon \rceil}$ ist. Ein solcher Algorithmus ist für sehr kleine ε dann nicht mehr praktisch einsetzbar. Ein Beispiel hierfür ist das Approximationsschema für das TSP, bei dem die Orte Punkte in $\mathbb{Z}^n$ sind und $c(i, j)$ der euklidische Abstand der Punkte i und j ist.

Es gibt aber auch Approximationsschemata, bei denen die Laufzeiten nicht ganz so unangenehm ausfallen, wie in dem oben erwähnten Beispiel. Diese Approximationsschemata werden echt polynomiell genannt. Ein Approximationsschema heißt echt polynomiell oder FPTAS (Full Polynomial Time Approximation Scheme) wenn die Rechenzeit bei variablem $\varepsilon > 0$ polynomiell in der Eingabelänge n und in ε^{-1} beschränkt ist. So gibt es z.B. für das Rucksackproblem ein echt polynomielles Approximationsschema, welches im Folgenden vorgestellt wird.

Frage 5.8: Wie funktioniert das echt polynomielle Approximationsschema für das Rucksackproblem?

In Kapitel 3.2 wurde ein pseudopolynomieller Algorithmus für das Rucksackproblem vorgestellt, der effizient arbeitet, wenn die Nutzenwerte der Objekte klein sind. Doch was können wir machen, wenn die Nutzenwerte nicht klein sind? In diesem Fall könnten wir versuchen, die Nutzenwerte gleichmäßig zu verkleinern, so dass eine effiziente Berechnung möglich wird, aber gleichzeitig das Ergebnis nicht allzu sehr verfälscht wird bzw. es nicht zu sehr von der optimalen Lösung abweicht. Der im Folgenden vorgestellte Algorithmus macht genau dies, indem von den Nutzenwerten die letzten t Stellen gestrichen werden. Auf diese Art und Weise wird jede Zahl ungefähr um den Faktor 2^{-t} kleiner. Wie schon erwähnt, darf t dabei nicht zu klein sein, da der Algorithmus ansonsten immer noch nicht effizient arbeitet. Es darf allerdings auch nicht zu groß sein, da ansonsten die angestrebte Güte nicht erreicht werden kann, weil die Eingabe zu sehr verändert wurde.

Es wird im Folgenden angenommen, dass der Gewichtswert eines jeden Objekts kleiner als die Gewichtsschranke ist. Für alle Objekte i in der Eingabe gilt also $g_i \leq G$. Ist dies nicht der Fall können die entsprechenden Objekte sofort aus der Eingabe gestrichen werden, da sie garantiert nicht an einer Lösung beteiligt sein können. Der Algorithmus arbeitet in drei Schritten, wobei eine Eingabe $I = ((g_1, \ldots, g_n, v_1, \ldots, v_n, G), \varepsilon)$ gegeben sei.

Um zu zeigen, dass es sich um ein echt polynomielles Approximationsschema handelt, ist zu beweisen, dass die Laufzeit polynomiell in der Eingabelänge n und in ε^{-1} beschränkt ist. Weiter muss gezeigt werden, dass die erreichte Güte $1 + \varepsilon$ beträgt.

Wir beginnen mit dem Beweis der Laufzeiteigenschaften. Die ersten beiden Schritte können in linearer Zeit $O(n)$ durchgeführt werden. Aus Kapitel 3.2 wissen wir, dass die Laufzeit von

Algorithmus 7 Das Approximationsschema für das Rucksackproblem mit Güte $1+\varepsilon$

1.) Berechne $v_{max} = \max\{v_1, \ldots, v_n\}$ und $t = \left\lfloor \log\left(\frac{\varepsilon \cdot v_{max}}{n \cdot (1+\varepsilon)}\right)\right\rfloor$.
2.) Berechne $v'_i = \lfloor v_i \cdot 2^{-t} \rfloor$ für $i = 1, \ldots, n$.
3.) Berechne mit Hilfe des pseudopolynomiellen Algorithmus aus Kapitel 3.2 die optimale Lösung für die Eingabe $I' = (g_1, \ldots, g_n, v'_1, \ldots, v'_n, G)$.

Schritt 3 durch $O(n^2 \cdot v'_{max})$ beschränkt ist. Wir formen diesen Ausdruck weiter um:

$$\begin{aligned} O(n^2 \cdot v'_{max}) &= O(n^2 \cdot v_{max} 2^{-t}) = O\left(n^2 \cdot v_{max} \cdot \frac{n(1+\varepsilon)}{\varepsilon \cdot v_{max}}\right) \\ &= O\left(n^3 \cdot \frac{1+\varepsilon}{\varepsilon}\right) = O\left(n^3 \cdot \left(\frac{1}{\varepsilon} + \frac{\varepsilon}{\varepsilon}\right)\right) \\ &= O\left(n^3 \cdot (\varepsilon^{-1} + 1)\right) = O(n^3 \cdot \varepsilon^{-1}) \end{aligned}$$

Damit ist die Laufzeit also polynomiell in n und in ε^{-1} beschränkt. Es bleibt also der Beweis der Güte. Dazu soll im Folgenden mit x die Lösung des ursprünglichen Problems I bezeichnet werden und mit x' die Lösung des Approximationsalgorithmus für das Problem I'. Entsprechend ist $V(x)$ der Nutzen der Lösung, wenn x in das Problem I eingesetzt wird und $V'(x)$ der Wert der Lösung, wenn x in das Problem I' eingesetzt wird. Wir beginnen mit einer Umformung, die zu einem Ergebnis führt, welches wir später noch nutzen werden:

$$\begin{aligned} V(x) &\geq V(x') = \sum_{i=1}^{n} v_i x'_i \geq 2^t \sum_{i=1}^{n} v'_i x'_i \geq 2^t \sum_{i=1}^{n} v'_i x_i \\ &= 2^t \sum_{i=1}^{n} \lfloor v_i 2^{-t} \rfloor x_i \geq 2^t \sum_{i=1}^{n} (v_i 2^{-t} - 1) x_i \\ &= \sum_{i=1}^{n} 2^t v_i x_i 2^{-t} - \sum_{i=1}^{n} 2^t x_i \geq \sum_{i=1}^{n} v_i x_i - n2^t \\ &= V(x) - n2^t \end{aligned}$$

Es gilt $V(x) \geq V(x')$, da x die optimale Lösung für I ist und x' dementsprechend höchstens so gut sein kann. Anschließend wird $V(x')$ ausgeschrieben und $v'_i = \lfloor v_i 2^{-t} \rfloor$ zu $v_i \geq v'_i 2^t$ abgeschätzt. Die resultierende Summe entspricht $V'(x')$. Die Lösung x' ist für I' optimal, also kann x, welche die für I optimale Lösung ist, in I' eingesetzt höchstens so gut sein. Nun wird wieder $v'_i = \lfloor v_i 2^{-t} \rfloor$ genutzt und $\lfloor v_i 2^{-t} \rfloor \geq v_i 2^{-t} - 1$ abgeschätzt. Anschließend wird 2^t wieder in die Summe gezogen und die Summe auseinander gezogen. $\sum_{i=1}^{n} 2^t x_i$ kann höchstens $n2^t$ werden, nämlich dann, wenn für alle $x_i = 1$ gilt.

Es gilt nun also $V(x) \geq V(x')$, woraus wir $V(x) - V(x') \geq 0$ folgern können. Es gilt jedoch auch $V(x') \geq V(x) - n2^t$, was zu $n2^t \geq V(x) - V(x')$ umgeformt werden kann. Insgesamt ergibt dies:

$$0 \leq V(x) - V(x') \leq n2^t \leq \frac{\varepsilon v_{max}}{1+\varepsilon}$$

Die letzte Abschätzung kommt einfach durch das Einsetzen von t zustande. Nun zu dem, was wir eigentlich zeigen wollen. Es handelt sich um ein Maximierungsproblem, also sind wir an

$V(x)/V(x')$ interessiert:

$$\begin{aligned}\frac{V(x)}{V(x')} &= \frac{V(x')+V(x)-V(x')}{V(x')} = 1+\frac{V(x)-V(x')}{V(x')} \leq 1+\frac{n2^t}{V(x)-n2^t} \\ &\leq 1+\frac{\frac{\varepsilon v_{max}}{1+\varepsilon}}{v_{max}-\frac{\varepsilon v_{max}}{1+\varepsilon}} = 1+\frac{\frac{\varepsilon}{1+\varepsilon}}{1-\frac{\varepsilon}{1+\varepsilon}} = 1+\frac{\frac{\varepsilon}{1+\varepsilon}}{\frac{1+\varepsilon}{1+\varepsilon}-\frac{\varepsilon}{1+\varepsilon}} \\ &= 1+\frac{\frac{\varepsilon}{1+\varepsilon}}{\frac{1+\varepsilon-\varepsilon}{1+\varepsilon}} = 1+\frac{\frac{\varepsilon}{1+\varepsilon}}{\frac{1}{1+\varepsilon}} = 1+\frac{\varepsilon\cdot(1+\varepsilon)}{1\cdot(1+\varepsilon)} = 1+\varepsilon\end{aligned}$$

5.3 Methoden zum Entwurf polynomieller Approximationsschemata

Frage 5.9: Welche drei Methoden zum Entwurf polynomieller Approximationsschemata kennst du?

Es können drei Methoden unterschieden werden, die jeweils von einer der drei Komponenten ausgehen:

- Suchraum
- Lösungsraum
- Algorithmen zur exakten Lösung des Optimierungsproblems

Jeder dieser drei Komponenten kann manipuliert werden. Im obigen Beispiel des echt polynomiellen Approximationsschema für das Rucksackproblem wurde der Suchraum verändert. Die Nutzenwerte wurden so verändert, dass die effiziente Berechnung einer Lösung möglich wurde, diese jedoch auch noch gut genug für das ursprüngliche Problem war.

Die zweiten Methode besteht darin, den Lösungsraum so in Bereiche aufzuteilen, dass in jedem Bereich effizient eine Lösung berechnet werden kann und die beste dieser Lösungen auch schon recht nah an die optimale Lösung für das Ursprungsproblem kommt.

Bei der dritten Methode wird der Verlauf des verwendeten Algorithmus so abgeändert, dass er effizient wird und gute Lösungen liefert.

5.4 PTAS durch Glättung des Suchraums

Frage 5.10: Nach welchem Schema gehen Algorithmen, die den Suchraum glätten, vor?

Das benutzte Schema haben wir in etwa schon bei dem echt polynomiellen Approximationsschema für das Rucksackproblem gesehen:

- In Abhängigkeit von $\varepsilon > 0$ wird aus der gegebenen Eingabe I eine geglättete bzw. vereinfachte Eingabe I' berechnet.
- Es wird (effizient) eine optimale Lösung x' für I' berechnet.
- Aus x' wird eine Lösung für I berechnet, deren Güte durch $1+\varepsilon$ beschränkt ist.

Wie man schnell einsieht, ist die richtige Wahl von I' kritisch. Ist die Eingabe I' der Eingabe I für das Ursprungsproblem zu ähnlich, kann eine optimale Lösung nicht effizient gefunden werden. Andererseits darf I' jedoch auch nicht zu verschieden von I sein, da wir ansonsten aus der Lösung x' keine Informationen bezüglich der Lösung von I gewinnen können.

Beim oben beschriebenen FPTAS für das Rucksackproblem war der dritte Schritt insofern trivial, als dass die berechnete Lösung x' sofort für I benutzt werden konnte. Dies wird im gleich folgenden Beispiel für das Lastverteilungsproblem nicht der Fall sein.

Frage 5.11: Welche Methode zur Glättung des Suchraums kennst du?
Es können grob vier Methoden unterschieden werden:

Runden von Zahlen: Diese Methode wurde auch beim FPTAS für das Rucksackproblem verwendet. Allgemein werden Zahlen auf das nächstkleinere Vielfache einer Zweierpotenz 2^m gerundet.

Verschmelzen: Hierbei werden Teile des Problems zu einem neuen Teil verschmolzen. durch dieses Vorgehen werden die einzelnen Teile nicht mehr individuell behandelt, wodurch die Freiheiten bei einer Verteilung der Probleme (oder Ähnlichem) eingeschränkt wird.

Wegschneiden: Besonders irreguläre Teile der Eingabe werden ignoriert.

Anpassen: Objekte, die sich sehr ähnlich sind, werden durch gleiche Objekte mit mittleren Parametern ersetzt. Dazu kann oft der Mittelwert oder der Median benutzt werden.

Die letzten drei Methoden werden im folgenden Beispiel eines PTAS für das Lastverteilungsproblem angewandt.

Frage 5.12: Worin besteht das Lastverteilungsproblem?
Beim Lastverteilungsproblem sind n Aufgaben mit den jeweiligen Bearbeitungszeit $t_1, \ldots, t_n$ gegeben. Die Aufgabe besteht darin, diese Aufgaben so auf zwei Maschinen zu verteilen, dass die maximale Laufzeit der beiden Maschinen minimiert wird. Oder anders gesagt, soll die Differenz zwischen den Laufzeiten der Maschinen minimiert werden. Die Frage, ob eine Differenz von 0 möglich ist, ist das Partitionsproblem, von dem bekannt ist, dass es NP-vollständig ist. Somit ist das Lastverteilungsproblem NP-hart.

Frage 5.13: Wie funktioniert das PTAS für das Lastverteilungsproblem, welches den Suchraum glättet?
Wir beginnen zuerst mit der Abschätzung einer einfachen unteren Schranke für den Wert einer optimalen Lösung. Dazu definierten wir $t_{sum} = t_1 + \cdots + t_n$ und $t_{max} = \max\{t_1, \ldots, t_n\}$. Da die Aufgabe mit der Bearbeitungszeit t_{max} auf jeden Fall bearbeitet werden muss und eine der Maschinen mindestens die halbe Last erhält, setzen wir $L := \max\{t_{sum}/2,\ t_{max}\}$. L ist eine untere Schranke für OPT.

Als nächstes zeigen wir, dass es trivial ist, eine Lösung mit Güte 2 zu erhalten, woraus folgt, dass wir uns auf die $\varepsilon \in]0, 1[$ konzentrieren sollten. Eine Lösung mit Güte 2 erhalten wir nämlich, wenn eine Maschine einfach alle Aufgaben zugeteilt bekommt. Ihre Last entspricht dann t_{sum}. Es gilt jedoch $t_{sum} \leq 2L$, was schnell gezeigt ist. Setzt sich $t_{sum}/2$ bei der Maximierung durch, gilt $L = t_{sum}/2$ und damit $t_{sum} = 2L$. Setzt sich t_{max} durch, muss $t_{max} \geq t_{sum}/2$ gelten und damit $t_{sum} \leq 2t_{max} = 2L$. Da auch $L \leq \mathrm{OPT}$ gilt, folgt also: $t_{sum} \leq 2L \leq 2 \cdot \mathrm{OPT} = (1+1)\mathrm{OPT}$.

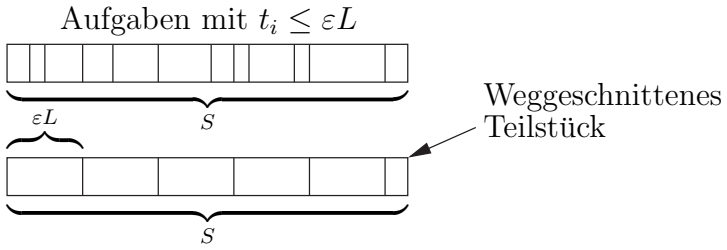


Abbildung 13: Verschmelzung der Aufgaben mit $t_i \leq \varepsilon L$

Die Hauptidee bei dem PTAS ist es nun, die Aufgaben in große und kleine Aufgaben einzuteilen. Dabei setzen wir die Größe der betrachteten Aufgabe in Beziehung zu L, wobei auch die zu erreichende Güte in Form von ε eingehen wird. Eine Aufgabe i zählt im Folgenden zu den großen Aufgaben wenn $t_i > \varepsilon L$ gilt. Im Fall von $t_i \leq \varepsilon L$ zählt i zu den kleinen Aufgaben. Im nächsten Schritt wird die geglättete Eingabe I' berechnet. Dabei werden wir die großen Aufgaben unangetastet lassen und die kleinen Aufgaben verändern. Dies mag auf den ersten Blick unlogisch erscheinen, da man intuitiv erwarten würde, dass die großen, sperrigen Aufgaben die Probleme machen, während man die kleinen Aufgaben doch bestimmt gut irgendwo unterbringen kann. Andererseits wird sich eine Manipulation der kleinen Aufgaben wahrscheinlich auch nicht allzu stark auf die Güte der berechneten Lösung auswirken, da die Fehler hier auch kleiner sein werden. Desweiteren wird sich jedoch auch zeigen, dass von der Manipulation der kleineren Aufgaben am Ende doch wieder alle Aufgaben profitieren.

Wie schon erwähnt, werden alle großen Aufgaben unverändert in I' übernommen. Für die kleinen Aufgaben berechnen wir zuerst einmal die Gesamtbearbeitungszeit S, die sie in I verursachen. Ist M_s also die Menge der kleinen Aufgaben, dann berechnen wir $S = \sum_{i \in M_s} t_i$. Dieser Schritt entspricht einer Verschmelzung der kleinen Aufgaben. Im nächsten Schritt werden wir eine Anpassung vornehmen. Es werden $\lfloor S/(\varepsilon L) \rfloor$ neue Aufgaben mit einer jeweiligen Bearbeitungsdauer von εL erzeugt. Da $S/(\varepsilon L)$ abgerundet wird, geht dabei höchstens εL an Bearbeitungszeit verloren. Hier nutzen wir also die Methode des Wegschneidens. Abbildung 13 verdeutlicht das Vorgehen noch einmal anschaulich. Die Laufzeit dieses Schritts kann mit $O(n)$ abgeschätzt werden. Wir müssen zuerst einmal jede Aufgabe betrachten, um zu wissen, ob sie zu den großen oder kleinen Aufgaben gehört. Anschließend werden die kleinen Aufgaben geändert. Es kann jedoch höchstens n kleine Aufgaben geben und bei der Erstellung der neuen Aufgaben werden diese höchstens weniger.

Wie kann nun eine Lösung für I' berechnet werden? Der Trick besteht darin, sich zuerst einmal die Anzahl der Aufgaben in I' zu überlegen. Wir werden gleich sehen, dass diese nicht mehr von der Anzahl der Aufgaben in I (dort gibt es n Aufgaben) abhängt. Argumentiert wird über die Gesamtbearbeitungszeit aller Aufgaben in I' und die Mindestbearbeitungszeit einer Aufgabe. Für die großen Aufgaben hat sich bei der Gesamtbearbeitungszeit nichts geändert, da diese einfach übernommen wurden. Bei den kleinen Aufgaben kann sich die Gesamtbearbeitungszeit sogar verringert haben, da wir ja unter Umständen ein Reststück abgeschnitten haben. Damit gilt also $t'_{sum} \leq t_{sum}$. Da wir etwas weiter oben schon $t_{sum} \leq 2L$ herausbekommen haben, gilt also auch $t'_{sum} \leq 2L$. Nach Konstruktion von I' hat jede Aufgabe mindestens eine Bearbeitungszeit von εL. Wenn wir also $2L$ durch εL teilen, bekommen wir die maximale Anzahl von Aufgaben:

$$\frac{2L}{\varepsilon L} = \frac{2}{\varepsilon} = 2\varepsilon^{-1}$$

Wie „versprochen“, ist diese Anzahl von n unabhängig. Für ein konstantes ε ist natürlich auch

$2\varepsilon^{-1}$ konstant. Wenn wir nun also alle möglichen Verteilungen auf die beiden Maschinen ausprobieren, müssen wir $2^{2\varepsilon^{-1}}$ Verteilungen betrachten. Bei konstantem ε sind auch dies wieder konstant viele. Um herauszufinden, welche Verteilung optimal ist, müssen wir zuerst die Lasten der beiden Maschinen bei der entsprechenden Verteilung berechnen und anschließend die größere der beiden Lasten gegen die bisher optimale Verteilung vergleichen. Da wir höchstens $2\varepsilon^{-1}$ Aufgaben haben, ist dies in Zeit $O(\varepsilon^{-1})$ möglich. Somit ergibt sich eine Gesamtlaufzeit von $O(n) + O(\varepsilon^{-1}2^{2\varepsilon^{-1}})$.

Bevor darauf eingegangen wird, wie aus der Lösung für I' eine Lösung für I berechnet werden kann, wollen wir erst zeigen, dass der Wert einer Lösung für I' stets eine gewisse Güte in Bezug auf den Wert der Lösung für I erreichen kann. Wir beweisen dies, indem wir von einer optimalen Lösung für I ausgehen, die wir mit OPT bezeichnen, und zeigen, dass wir aus dieser eine gute Lösung für I' generieren können. Diese wird mit OPT' bezeichnet. Wie konstruieren wir nun aus der Lösung für I die Lösung für I'? Die Verteilung der großen Aufgaben übernehmen wir einfach aus I. Dies ist möglich, da wir an diesen ja nichts verändert haben. Jetzt müssen wir nur noch die kleinen Aufgaben verteilen. Davon müssen $\lfloor S/(\varepsilon L)\rfloor$ verteilt werden. Die Verteilung soll nun so aussehen, dass jeweils $\lceil S_i/(\varepsilon L)\rceil$ Aufgaben an Maschine i verteilt werden. Dabei ist S_i die Gesamtlast, die in der optimalen Lösung auf Maschine i durch die kleinen Aufgaben verursacht wird. Zuerst zeigen wir, dass wir so auch tatsächlich alle Aufgaben verteilt bekommen:

$$\left\lceil \frac{S_1}{\varepsilon L} \right\rceil + \left\lceil \frac{S_2}{\varepsilon L} \right\rceil \geq \frac{(S_1 + S_2)}{\varepsilon L} \geq \left\lfloor \frac{S}{\varepsilon L} \right\rfloor$$

Eine solche Verteilung ist also möglich. Nun fragen wir uns als nächstes, wie sehr sich die Last einer Maschine bei dieser Verteilung erhöhen kann.

$$\left\lceil \frac{S_i}{\varepsilon L} \right\rceil \cdot \varepsilon L - S_i \leq \left(\frac{S_i}{\varepsilon L} + 1 \right) \cdot \varepsilon L - S_i = \frac{S_i \cdot \varepsilon L}{\varepsilon L} + \varepsilon L - S_i = \varepsilon L$$

Die Last einer jeden Maschine kann sich also im Vergleich zur optimalen Lösung höchstens um εL erhöhen. Da dies auch für die maximal beladene Maschine gilt, über die sich der Wert der Lösung des Problems definiert, gilt also:

$$\mathrm{OPT}' \leq \mathrm{OPT} + \varepsilon L \leq \mathrm{OPT} + \varepsilon \mathrm{OPT} = (1 + \varepsilon)\mathrm{OPT}$$

Dabei wurde genutzt, dass $L \leq \mathrm{OPT}$ gilt.

Es muss also nur noch geklärt werden, wie aus der Lösung x' zu I' eine Lösung x zu I berechnet wird. Mit L'_i bezeichnen wir im Folgenden die Last der Maschine i bezüglich der Lösung x'. Weiter differenzieren wir zwischen den großen Aufgaben B'_i und den kleinen Aufgaben S'_i. Es gilt $L'_i = B'_i + S'_i$. Für die Summe der kleinen Aufgaben gilt:

$$S'_1 + S'_2 = \varepsilon L \left\lfloor \frac{S}{\varepsilon L} \right\rfloor > \varepsilon L \left(\frac{S}{\varepsilon L} - 1 \right) = S - \varepsilon L$$

Dieses Ergebnis werden wir gleich noch brauchen. Es wird nun in $O(n)$ eine Lösung x für I konstruiert. Die Verteilung der großen Aufgaben übernehmen wir, wie sie in x' erfolgt ist. Für die kleinen Aufgaben benutzen wir ein Greedy Verfahren und reservieren auf Maschine 1 ein Zeitintervall der Länge $S'_1 + \varepsilon L$ und für Maschine 2 ein Intervall der Länge $S'_2 + \varepsilon L$. Da jede kleine Aufgabe höchstens eine Bearbeitungsdauer von εL hat, schaffen wir es in jedem

Fall die ersten Maschine mindestens mit S'_1 zu belasten und die reservierte Zeit dabei auch einzuhalten. Für die restlichen kleinen Aufgaben benötigen wir dann Zeit $S - S'_1$. Wir stellen die Formel $S'_1 + S'_2 > S - \varepsilon L$ um und erhalten $S - S'_1 < S'_2 + \varepsilon L$. Damit schaffen wir es, auch die restlichen Aufgaben in der veranschlagten Zeit abzuarbeiten. Allgemein gilt also für L_i:

$$L_i \leq B'_i + (S'_i + \varepsilon L) = L'_i + \varepsilon L \leq (1+\varepsilon)\text{OPT} + \varepsilon\text{OPT} = (1+2\varepsilon)\text{OPT}$$

Die 2εOPT, die wir im Gegensatz zu optimalen Lösung extra brauchen, können umgangssprachlich mit den folgenden zwei Argumenten erklärt werden. Zum einen brauchen wir zum Verteilen der großen Aufgaben eine Lösung für x'. Wie wir etwas weiter oben gesehen haben, kann deren Güte höchstens εOPT schlechter sein als die der optimalen Lösung. Zum anderen reservieren wir für jede Maschine im Endeffekt $\varepsilon L \leq \varepsilon$OPT Platz extra, um die Aufgaben auf jeden Fall unter zu bekommen.

5.5 PTAS durch Glättung des Lösungsraums

Frage 5.14: Nach welchem Schema gehen Algorithmen, die den Lösungsraum glätten, vor?

Algorithmen, die den Lösungsraum glätten, gehen in drei Schritten vor:

- In Abhängigkeit von der angepeilten Güte wird der Lösungsraum in möglichst disjunkte Bereiche aufgeteilt.
- In jedem Bereich wir eine Lösung berechnet, die möglichst nahe an der optimalen Lösung in diesem Bereich liegen sollte. Bereiche von denen man weiß, dass sie keine optimale Lösung enthalten können, können dabei ignoriert werden. Dies ähnelt ein wenig dem Vorgehen bei Branch-and-Bound Verfahren.
- Aus den berechneten Approximationslösungen wird die beste als Gesamtlösung ausgewählt.

Hierbei muss aufgepasst werden, dass der Lösungsraum nicht zu fein aufgeteilt wird, da ansonsten zu viele Lösungen zu berechnen und zu vergleichen sind. Wird er jedoch zu grob aufgeteilt, werden die zu lösenden Probleme nicht einfacher und man hat ebenfalls nichts gewonnen.

Frage 5.15: Wie funktioniert das PTAS für das Lastverteilungsproblem, welches den Lösungsraum glättet?

Zur Lösung des Lastverteilungsproblems über Glättung des Lösungsraums verwerden wir die gleiche Einteilung in große und kleine Aufgaben, die auch schon in Kapitel 5.4 benutzt wurde. Aufgaben i mit einer Bearbeitungszeit von $t_i > \varepsilon L$ gelten als große Aufgaben. Kleine Aufgaben haben dementsprechend eine Bearbeitungszeit von $t_i \leq \varepsilon L$. Wieviele große Aufgaben gibt es höchstens? Auch hier gilt die untere Schranke von $L = \max\{t_{sum}/2,\ t_{max}\}$ für eine optimale Lösung. Dabei ist t_{sum} die Bearbeitungszeit aller Aufgaben zusammen und t_{max} die Bearbeitungszeit der längsten Aufgabe. Aus der unteren Schranke L folgt $t_{sum} \leq 2L$. Wenn wir nun $2L$ durch εL dividieren, erhalten wir also die maximale Anzahl an großen Aufgaben:

$$\frac{2L}{\varepsilon L} = 2\varepsilon^{-1}$$

Dementsprechend gibt es $2^{2\varepsilon^{-1}}$ viele Möglichkeiten, die großen Aufgaben auf die beiden Maschinen zu verteilen. Für ein konstantes ε ist dies wieder eine Konstante. Wir werden daher im Folgenden jede mögliche Aufteilung der großen Aufgaben als einen Bereich betrachten und teilen den Lösungsraum somit in höchstens $2^{2\varepsilon^{-1}}$ Bereiche auf.

Als nächstes müssen wir nun für jeden Bereich eine Lösung berechnen, die möglichst nah an der optimalen Lösung für diesen Bereich liegt. Betrachten wir also nun einen solchen Bereich. Die großen Aufgaben sind schon verteilt, wir müssen also nur noch die kleinen Aufgaben verteilen. Wir werden gleich sehen, dass eine „falsche“ Verteilung der kleinen Aufgaben auch nur zu kleinen Fehlern führt und wir die Aufgaben daher mit einem Greedy Algorithmus verteilen können. Dabei gehen wir die kleinen Aufgaben der Reihe nach durch und verteilen die nächste kleine Aufgabe immer an die Maschine, die gerade weniger ausgelastet ist. Sind beide Maschinen gleich stark ausgelastet, nehmen wir einfach irgendeine der beiden. Die Laufzeit dieses Vorgehens liegt offensichtlich bei $O(n)$. Wir zeigen als nächstes, dass die so errechneten Lösungen L_m für jeden Bereich R_m durch $(1+\varepsilon)\mathrm{OPT}_m$ beschränkt sind, wobei OPT_m die optimale Lösung des Bereichs R_m ist. Da in einem der Bereiche auch die optimale Lösung OPT liegen muss, haben wir dann gezeigt, dass die zum Schluss ausgewählte Lösung durch $(1+\varepsilon)\mathrm{OPT}$ beschränkt ist. Betrachten wir also einen Bereich B_m. Durch die Aufteilung des Lösungsraums sind die großen Aufgaben schon verteilt. Die Werte dieser Verteilung der großen Aufgaben seien B_1 und B_2. Von diesen beiden Werten nehmen wir zuerst den größeren und bezeichnen diesen mit B. Es gilt dann $B = \max\{B_1, B_2\} \leq \mathrm{OPT}_m$. Wenn der Wert der Lösung L_m nun B entspricht, also $L_m = B$, dann folgt daraus, dass $L_m = \mathrm{OPT}_m$ ist, d.h. für den Bereich R_m wurde eine optimale Lösung berechnet. Denn in diesem Fall wurden auf eine Maschine nur große Aufgaben verteilt und auf die andere Maschine die restlichen großen Aufgaben und alle kleinen. Da sich die Maschine, die nur große Aufgaben bekommen hat, in jedem Fall durchsetzt, muss die Lösung also optimal sein. Falls $L_m \neq B$ gilt, kann nur $L_m > B$ gelten. O.B.d.A. nehmen wir an, dass L_m die Last von Maschine 1 ist. In diesem Fall hat Maschine 1 also noch mindestens eine kleine Aufgabe bekommen. Betrachten wir den Zeitpunkt, zu dem Maschine 1 die letzte kleine Aufgabe bekommen hat. Sie muss also weniger ausgelastet gewesen sein als Maschine 2. Daraus folgt jedoch, dass sie höchstens eine Last von $t_{sum}/2$ gehabt haben kann. Dazu kommt jetzt noch eine Last von höchstens εL hinzu. Dies reicht uns auch schon für eine Abschätzung:

$$L_m \leq t_{sum}/2 + \varepsilon L \leq L + \varepsilon L = (1+\varepsilon)L \leq (1+\varepsilon)\mathrm{OPT} \leq (1+\varepsilon)\mathrm{OPT}_m$$

Die Lösung L_m für den Bereich R_m ist also auf jeden Fall durch $(1+\varepsilon)\mathrm{OPT}_m$ beschränkt.

5.6 FPTAS durch Glättung der Zwischenlösungen eines langsamen Algorithmus

Frage 5.16: Wie funktioniert das FPTAS für das Lastverteilungsproblem, welches die Zwischenlösungen eines langsamen Algorithmus glättet?

Bei diesem Verfahren wird ein langsamer aber exakter Algorithmus modifiziert werden. Der Algorithmus hat eine pseudopolynomielle Laufzeit und berechnet in dieser eine exakte Lösung für das Lastverteilungsproblem. Er wird nun so modifiziert werden, dass die Laufzeit polynomiell wird, das Ergebnis jedoch weiterhin eine Güte von $(1+\varepsilon)$ hat. Wir beginnen mit der Beschreibung des pseudopolynomiellen Algorithmus.

Das Lastverteilungsproblem I besteht aus den Aufgaben (Jobs) $J_1, \ldots, J_n$ mit den Bearbeitungszeiten $t_1, \ldots, t_n$. Wir definieren nun sogenannte Präfixprobleme I_k mit $1 \leq k \leq n$, welche

nur noch die ersten k Jobs $J_1, \ldots, J_k$ mit den dazu gehörigen Bearbeitungszeiten $t_1, \ldots, t_k$ betrachten. Interessiert sind wir an einer Lösung für $I_n = I$. Für jedes Problem I_k gibt es verschiedene zulässige Lösungen, die jeweils den verschiedenen Verteilungen der Aufgaben entsprechen. Diese Lösungen können jeweils durch einen Vektor $(L_1, L_2) \in \mathbb{R}^2$ beschrieben werden, wobei L_1 die Last der ersten Maschine und L_2 die der zweiten Maschine ist. (L_1, L_2) wird auch Charakterisierung genannt. Der Wert einer Lösung für das Lastverteilungsproblem kann aus der Charakterisierung in $O(1)$ berechnet werden und ist $\max\{L_1, L_2\}$. Weiter gilt für die Lasten der Maschinen und damit auch für die zulässigen Charakterisierungen $0 \leq L_1, L_2 \leq t_{sum}$, da eine Maschine höchstens alle Aufgaben bekommen kann.

Um eine optimale Lösung für das Lastverteilungsproblem zu berechnen, kann der Ansatz der dynamischen Programmierung benutzt werden. Dabei wird eine Tabelle mit $(t_{sum} + 1)^2$ vielen Einträgen erstellen, wobei jeder Tabelleneintrag einer Charakterisierung entspricht. Wir wollen die Tabelle nun schrittweise so füllen, dass wir nach dem k-ten Schritt für jede Charakterisierung wissen, ob sie eine zulässige Lösung für das Präfixproblem I_k ist. Wann ist dies der Fall? Wir befinden uns im k-ten Schritt, d.h. wir haben eine Aufgabe mit einer Bearbeitungszeit t_k auf eine der beiden Maschinen zu verteilen. Eine Charakterisierung (x, y) kann nur dann eine zulässige Lösung für I_k sein, wenn $(x - t_k, y)$ oder $(x, y - t_k)$ eine zulässige Lösung ist. Anstatt nun in jedem Schritt rückwärts zu schauen, gehen wir im k-ten Schritt über die Tabelle und setzen, wenn wir eine gültige Charakterisierung (x, y) finden, die Charakterisierungen $(x + t_k, y)$ und $(x, y + t_k)$ ebenfalls auf „gültig“. Bis wir das Problem I_n gelöst haben, sind wir also n-mal über eine Tabelle der Größe $O(t_{sum}^2)$ gelaufen, was zu einer pseudopolynomiellen Laufzeit von $O(nt_{sum}^2)$ führt. Die Laufzeit kann jedoch auch auf $O(nt_{sum})$ gedrückt werden, da die Tabelle Redundanzen enthält. Dies gilt, da in jedem Schritt für jede gültige Charakterisierung (x, y) die Gleichung $x + y = t_1 + \cdots + t_k$ gelten muss. Es würde also eigentlich ausreichen, sich nur die x-Werte zu merken. Da wir jedoch gleich eine modifizierte Tabelle konstruieren, interessiert uns dies nicht. Der Algorithmus ist pseudopolynomiell, da t_{sum} unter Umständen „sehr groß“ sein kann, was wiederum zu einer großen Tabelle führt, die nicht mehr effizient gefüllt werden kann. Die Idee des im Folgenden vorgestellten FPTAS besteht darin, einige nebeneinander liegende Charakterisierungen zusammenzufassen, da sich diese teilweise sehr ähnlich sind. Auf diese Art und Weise wird dafür gesorgt werden, dass nur noch polynomiell viele Tabelleneinträge entstehen. Ferner wird gezeigt werden, dass bei diesem Vorgehen jedoch eine gewisse Güte eingehalten wird.

Im Folgenden werden (gültige) Charakterisierungen auch als (gültige) Vektoren bezeichnet. Jeder Tabelleneintrag bzw. Vektor kann als ein Punkt im Quadrat $[0, t_{sum}] \times [0, t_{sum}] \subseteq \mathbb{R}^2$ aufgefasst werden. Wie schon gesagt, sollen die Zwischenlösungen geglättet werden, indem verschiedene Zwischenlösungen zusammengefasst werden. Dazu legen wir $(L + 1)^2$ Rechtecke über das Gitter. Wie groß L ist, hängt im Endeffekt von der angepeilten Güte und der Anzahl der Aufgaben ab. Um die $(L+1)^2$ Rechtecke zu erzeugen, werden die beiden Intervalle $[0, t_{sum}]$ des Quadrats jeweils in die Teilintervalle $[0, 1 = \Delta^0[, [1 = \Delta^0, \Delta^1[, [\Delta^1, \Delta^2[, \ldots, [\Delta^{L-1}, t_{sum}]$ aufgeteilt. Abbildung 14 zeigt die resultierenden Rechtecke. Da $\Delta \geq 1$ gilt, werden die Rechtecke in jedem Intervall größer. Dabei ist Δ wie folgt definiert:

$$\Delta = 1 + \frac{\varepsilon}{2n}$$

Daraus ergibt sich auch L und damit die Anzahl der entstehenden Rechtecke. Denn es muss

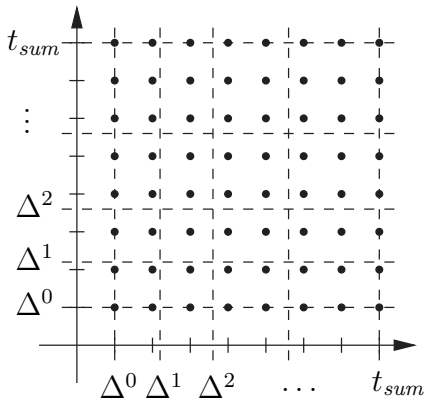


Abbildung 14: Die immer größer werdenden Rechtecke über $[0, t_{sum}]$

ja $\Delta^L \geq t_{sum}$ gelten. Dies wird weiter umgerechnet:

$$L \geq \lceil \log_\Delta t_{sum} \rceil = \left\lceil \frac{\ln t_{sum}}{\ln \Delta} \right\rceil$$

Den Logarithmus von Δ werden wir nicht genau ausrechnen, sondern ihn nur durch seine Taylor-Reihe abschätzen. Wird diese nach dem zweiten Glied abgebrochen, so gilt:

$$\ln z \geq 1 - \frac{1}{z}$$

Wir setzen statt z unser Δ ein und erhalten:

$$\ln \Delta \geq 1 - \frac{1}{1 + \frac{\varepsilon}{2n}} = \frac{1 + \frac{\varepsilon}{2n}}{1 + \frac{\varepsilon}{2n}} - \frac{1}{1 + \frac{\varepsilon}{2n}} = \frac{1 + \frac{\varepsilon}{2n} - 1}{1 + \frac{\varepsilon}{2n}} = \frac{\frac{\varepsilon}{2n}}{1 + \frac{\varepsilon}{2n}}$$

Da wir an $1/\ln \Delta$ interessiert sind, rechnen wir mit dem Kehrwert weiter:

$$\frac{1}{\ln \Delta} \leq \frac{1 + \frac{\varepsilon}{2n}}{\frac{\varepsilon}{2n}} = \frac{\frac{\varepsilon}{2n}}{\frac{\varepsilon}{2n}} + \frac{1}{\frac{\varepsilon}{2n}} = 1 + \frac{2n}{\varepsilon}$$

Wird dieses Ergebnis in die obere Gleichung eingesetzt, ergibt sich also insgesamt:

$$L \geq \left\lceil \ln t_{sum} \cdot \left(1 + \frac{2n}{\varepsilon}\right) \right\rceil$$

Die Glättung der Zwischenergebnisse erfolgt nun dadurch, dass sich für jedes Rechteck nur ein Wert bzw. Vektor gemerkt wird, da die Vektoren in einem Rechteck sich recht ähnlich sind. Zwei Vektoren $(x_1, y_1), (x_2, y_2)$ liegen in einem Rechteck, wenn gilt:

$$\frac{x_1}{\Delta} \leq x_2 \leq x_1 \cdot \Delta \quad \text{und} \quad \frac{y_1}{\Delta} \leq y_2 \leq y_1 \cdot \Delta$$

Warum gilt diese Gleichung? Dazu überlegen wir uns zuerst, was bei einer Division durch Δ mit der Koordinate eines Punkts passiert. Die betrachtete Koordinate liege im Intervall $[\Delta^k, \Delta^{k+1}[$. Um eine Vorstellung zu bekommen, was passiert, schauen wir uns zuerst an, was mit den Begrenzungspunkten Δ^k und Δ^{k+1} des Intervalls passiert. Wird Δ^{k+1} durch

Δ dividiert, landen wir bei Δ^k. Entsprechend landen wir für Δ^k bei Δ^{k-1}. Koordinaten, die im Intervall $[\Delta^k, \Delta^{k+1}[$ liegen, werden also auf Koordinaten des Intervalls $[\Delta^{k-1}, \Delta^k[$ abgebildet, d.h. um ein Intervall (oder im Falle von Punkten um ein Rechteck) nach links bzw. unten geschoben. Ähnliches gilt für eine Multiplikation mit Δ, nur dass die Koordinaten des Intervalls $[\Delta^k, \Delta^{k+1}[$ in diesem Fall in das Intervall $[\Delta^{k+1}, \Delta^{k+2}[$ abgebildet werden. Es bleibt also noch die Frage, was die Ungleichungen bedeuten. Wir betrachten im Folgenden nur die Ungleichungen für x_1 und x_2, da das Argument bei den Ungleichungen für y_1 und y_2 das gleiche ist. Angenommen, zwei Punkte (x_1, y_1) und (x_2, y_2) liegen in einem Rechteck. Dann kann ich den Punkt (x_1, y_1) um ein Rechteck nach links schieben, so dass seine x-Koordinate in jedem Fall kleiner oder gleich x_2 ist. Dies gilt jedoch auch für Punkte, welche sich in einem der Rechtecke links vom betrachteten Rechteck befinden. Für diese Punkte gilt jedoch wiederum nicht, dass ich ihre x-Koordinaten größer als x_2 machen kann, indem ich sie um ein Rechteck nach rechts verschiebe. Dies gilt nur für Punkte, die sich im selben Rechteck wie (x_2, y_2) befinden. Und genau diese Forderung drücken die Ungleichungen aus.

Da nun nur noch $(L+1)^2$ viele Vektoren berechnet werden, hat der Algorithmus eine Laufzeit von $O(nL^2)$ anstatt einer Laufzeit von $O(nt_{sum}^2)$. Ein Einsetzen von L liefert:

$$\begin{aligned} O(nL^2) &= O\left(n \cdot \ln^2 t_{sum} \cdot \left(1 + \frac{2n}{\varepsilon}\right)^2\right) = O\left(n \cdot \ln^2 t_{sum} \cdot \left(1 + \frac{4n}{\varepsilon} + \left(\frac{2n}{\varepsilon}\right)^2\right)\right) \\ &= O\left(n \cdot \ln^2 t_{sum} \cdot \left(1 + \frac{4n}{\varepsilon} + 4n^2\varepsilon^{-2}\right)\right) = O(n^3\varepsilon^{-2} \ln^2 t_{sum}) \end{aligned}$$

Damit ist die Laufzeit polynomiell in der Eingabelänge und in ε^{-1} (sogar ε^{-2}). Es bleibt also nur noch die Güte von $1+\varepsilon$ zu beweisen. Dazu werden wir etwas später beweisen, dass der Approximationsalgorithmus zu jedem für I_k zulässigen Vektor (x, y) einen Vektor (x', y') berechnet, bei dem sich die Komponenten höchstens um einen Faktor Δ^k von den Komponenten des Vektors (x, y) unterscheiden, also: $x' \leq \Delta^k x$ und $y' \leq \Delta^k y$. Wir nehmen nun erst einmal an, dass diese Behauptung stimmt und zeigen, dass dann auch die Behauptung über die Güte korrekt ist. Wenn also (x, y) eine optimale Lösung ist, dann folgt aus obiger Behauptung, dass der Approximationsalgorithmus eine Lösung (x', y') mit Wert $\max\{x', y'\} \leq \max\{x\Delta^n, y\Delta^n\}$ liefert. Wenn zwischen zwei Zahlen das Maximum berechnet wird, wird das Ergebnis nicht verfälscht, wenn beide durch die gleiche Zahl dividiert werden. Damit gilt aber:

$$\max\{x', y'\} \leq \max\{x\Delta^n, y\Delta^n\} = \Delta^n \max\{x, y\} = \Delta^n \mathrm{OPT}$$

Um eine Güte von $1+\varepsilon$ zu beweisen muss nun noch $\Delta^n \leq 1+\varepsilon$ gezeigt werden. Dabei können wir von einem $\varepsilon < 1$ ausgehen, da wir ansonsten alle Aufgaben an die erste Maschine geben und damit auf jeden Fall eine Güte von 2 erreichen. Es gelte also $\varepsilon < 1$:

$$\begin{aligned} \Delta^n = \left(1 + \frac{\varepsilon}{2n}\right)^n &= \left(1 + \frac{\varepsilon/2}{n}\right)^n \\ &\leq e^{\varepsilon/2} = 1 + \frac{(\varepsilon/2)^1}{1!} + \frac{(\varepsilon/2)^2}{2!} + \frac{(\varepsilon/2)^3}{3!} + \dots \\ &\leq e^{\varepsilon/2} = 1 + \left(\frac{\varepsilon}{2}\right)^1 + \left(\frac{\varepsilon}{2}\right)^2 + \left(\frac{\varepsilon}{2}\right)^3 + \dots \\ &\leq 1 + \varepsilon \cdot \left(\frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots\right) \\ &\leq 1 + \varepsilon \end{aligned}$$

Dabei wird ausgenutzt, dass die e-Funktion als der Grenzwert einer Folge aufgefasst werden kann. Genauer:

$$\lim_{n\to\infty}\left(1+\frac{x}{n}\right)^n = e^x$$

Es bleibt also nur noch die Behauptung zu zeigen, dass für einen gültigen Vektor (x, y) für I_k vom Approximationsalgorithmus ein Vektor (x', y') mit $x' \leq \Delta^k x$ und $y' \leq \Delta^k y$ berechnet wird. Dies wird nun per Induktion bewiesen. Für $k = 1$ stimmt die Behauptung, da die Vektoren $(t_1, 0)$ und $(0, t_1)$ berechnet werden. Diese liegen auch in verschiedenen Rechtecken, da $t_1 \geq 1$ gilt. Im Induktionsschritt wird nun angenommen, dass (x, y) eine zulässige Lösung für I_k sei. Dieser Vektor muss aus einem der Vektoren $(x - t_k, y)$ oder $(x, y - t_k)$ entstanden sein. Ohne Beschränkung der Allgemeinheit wird nun angenommen, dass er aus $(x - t_k, y)$ entstanden ist. Da $(x - t_k, y)$ für I_{k-1} zulässig ist, wird für diesen Vektor vom Approximationsalgorithmus nach Induktionsvoraussetzung ein Vektor (x', y') berechnet, für den gilt:

$$x' \leq \Delta^{k-1}(x - t_k) \qquad \text{und} \qquad y' \leq \Delta^{k-1} y$$

Aus diesem Vektor wird nun $(x'+t_k, y')$ berechnet, welcher jedoch unter Umständen zugunsten eines anderen Vektors (x'', y''), der ins gleiche Rechteck fällt, vergessen wird. Da der Vektor im gleichen Rechteck liegt, muss jedoch gelten:

$$\frac{(x' + t_k)}{\Delta} \leq x'' \leq \Delta(x' + t_k)$$

Wir formen die hintere Ungleichung weiter um:

$$x'' \leq \Delta(x' + t_k) \leq \Delta(\Delta^{k-1}(x - t_k) + t_k) = \Delta^k x \underbrace{-\Delta^k t_k + \Delta t_k}_{\leq 0} \leq \Delta^k x$$

Ebenso gilt für y'':

$$y'' \leq \Delta y' \leq \Delta(\Delta^{k-1} y) = \Delta^k y$$

6 Lineare Optimierung, Randomisiertes Runden und Derandomisierung

6.1 Lineare Optimierungsprobleme

Frage 6.1: Was sind lineare Optimierungsprobleme?

Lineare Optimierung bezeichnet die Optimierung linearer Funktionen unter linearen Nebenbedingungen. Es gibt eine Standardform, auf die viele Varianten von linearen Optimierungsproblemen gebracht werden können. Die Standardform lautet:

$$\text{Minimiere } f(x_1, \ldots, x_n) = w_1 x_1 + \ldots, w_n x_n$$

unter den $m + n$ Nebenbedingungen:

$$\begin{aligned} a_{i1}x_1 + \cdots + a_{in}x_n &\geq b_i \quad \text{für } 1 \leq i \leq m \\ \text{und} \qquad x_j &\geq 0 \quad \text{für } 1 \leq j \leq n \end{aligned}$$

Viele Probleme, z.B. aus der BWL, lassen sich als Optimierungsprobleme formulieren.

Die obigen Formeln lassen sich auch geometrisch interpretieren. Gesucht sind n Werte $x_1, \ldots, x_n$, welche die Zielfunktion $f(x_1, \ldots, x_n)$ minimieren und die Nebenbedingungen einhalten. Die möglichen Lösungen lassen sich als Punkte im $\mathbb{R}^n$ interpretieren. Da jedes x_j für $1 \leq j \leq n$ positiv sein soll, kommen dabei also nur positive Koordinaten in Frage. Handelt es sich zum Beispiel um ein Problem, bei dem zwei Werte x_1 und x_2 gesucht werden, dann darf eine Lösung nur im ersten Quadranten liegen. Allgemein beschreibt eine Gleichung der Form

$$z_1 x_1 + \cdots + z_n x_n = c$$

eine Hyperebene im $\mathbb{R}^n$. Eine Forderung der Form

$$a_{i1}x_1 + \cdots + a_{in}x_n \geq b_i$$

beschreibt also für die i-te Bedingung, dass alle gültigen Punkte auf einer Seite der durch $a_{i1}x_1 + \cdots + a_{in}x_n = b_i$ beschriebenen Hyperebene liegen sollen. Durch mehrere solcher Hyperebenen wird im $\mathbb{R}^n$ ein konvexer n-dimensionaler Polyeder beschrieben. Alle Nebenbedingungen zusammen fordern also, dass der gesuchte Punkt, der $f(x_1, \ldots, x_n)$ minimiert, innerhalb dieses Polyeders liegen muss. Die Aufgabe kann nun auch wie folgt umschrieben werden. Punkte, deren Zielfunktionswerte gleich sind, liegen in einer Hyperebene, da durch

$$w_1 x_1 + \ldots, w_n x_n = c$$

eine Hyperebene definiert wird. Für verschiedene c ergeben sich verschiedene Hyperebenen, die jedoch alle parallel zueinander liegen. Wir gehen nun davon aus, dass wir mit $c = -\infty$ anfangen und der durch die Nebenbedingungen definierte Polyeder nicht geschnitten wird. Ansonsten gibt es auch keine eindeutige Lösung für unser lineares Optimierungsproblem bzw. wir bekommen beliebig gute Lösungen, die alle gültig sind. Da wir bei $-\infty$ anfangen und der Poyeder nicht geschnitten wird, sorgen wir nun dafür, dass die Hyperebene immer näher an den Polyeder rückt, indem wir das c wachsen lassen. Die Hyperebene wird also parallel zu sich selbst verschoben. Irgendwann erreichen wir einen Wert für c, so dass die Hyperebene den Polyeder der gültigen Werte zum ersten Mal schneidet. Dies ist dann gleichbedeutend damit, dass es einen Lösungspunkt gibt, der in der Hyperebene der Zielfunktion sowie im Polyeder

der gültigen Lösungen liegt. Dieser Lösungspunkt ist damit der gesuchte Punkt $(x_1, \ldots, x_n)$, der $f(x_1, \ldots, x_n)$ unter Einbehaltung der Nebenbedingungen minimiert. In vielen Fällen wird dieser Schnitt an einem Punkt stattfinden, im $\mathbb{R}^3$ kann die Ebene jedoch auch z.B. eine ganze Ebene schneiden.

Frage 6.2: Wie können Varianten von linearen Optimierungsproblemen auf die Standardform zurück geführt werden?

Zuerst einmal kann man sich überlegen, dass es ja nicht nur Minimierungsprobleme gibt, sondern dass viele bekannte Probleme Maximierungsprobleme sind. Andererseits kann die Maximierung einer Zielfunktion $f(x)$ auch als Minimierung der Zielfunktion $-f(x)$ formuliert werden. In der Standardform kommen auch nur $\geq$-Nebenbedingungen vor. Es ist jedoch kein Problem $\leq$-Ungleichungen mit Hilfe von $\geq$-Ungleichungen zu formulieren. Dazu müssen lediglich beide Seiten mit -1 multipliziert werden. Statt $x \leq 10$ kann also auch $-x \geq -10$ geschrieben werden. Nebenbedingungen der Form $a_{i1}x_1 + \cdots + a_{in}x_n = b_i$ können durch eine $\leq$- und eine $\geq$-Ungleichung simuliert werden, wobei die $\leq$-Ungleichung natürlich wieder mit einer $\geq$-Ungleichung formuliert werden muss.

Frage 6.3: Welche Verfahren zur Lösung von linearen Optimierungsproblemen kennst du?

Im Skript werden drei Verfahren kurz vorgestellt, jedoch nicht ausführlich behandelt:

Simplexalgorithmus: Der Simplexalgorithmus wurde von Dantzig entwickelt und führt auf den Ecken des Polyeders der zulässigen Lösungen eine lokale Suche durch, bis ein lokal optimaler Punkt gefunden wird. Praktisch ist der Algorithmus oft effizient, für (künstliche) Beispiele kann die Laufzeit jedoch exponentiell werden.

Ellipsoidmethode: Bei dieser Methode wird der Polyeder der zulässigen Punkte durch ein Ellipsoid angenähert, welches sukzessive verkleinert werden kann. In der Theorie ist der Algorithmus zwar polynomiell, praktisch ist er jedoch nicht effizient.

Innere-Punkt-Methoden: Diese Methoden verwenden Strafkosten, welche dafür sorgen, dass der Polyeder der zulässigen Lösungen nicht verlassen wird. Varianten dieser Verfahren haben eine polynomielle Worst-Case Rechenzeit und sind auch in der Praxis anwendbar.

Frage 6.4: Welche wichtige Variante linearer Optimierungsprobleme kennst du?

Eine wichtige Variante linearer Optimierungsprobleme sind die ganzzahligen Optimierungsprobleme. Hierbei wird als zusätzliche Nebenbedingung eingefügt, dass die Variablen nur ganzzahlige Werte annehmen dürfen. Bei dieser Einschränkung verkleinert sich der Lösungsraum, das Problem wird jedoch keineswegs einfacher. Es lassen sich Beispiele konstruieren, in denen der Wert einer optimalen nicht ganzzahligen Lösung der Zielfunktion weit weg von der optimalen ganzzahligen Lösung liegt. Praktisch bedeutet dies z.B., dass es nicht ausreicht, eine optimale Lösung ohne die Ganzzahligkeitsbedingung auszurechnen und diese einfach zu runden. Wir haben sogar schon gesehen, dass die Lösung ganzzahliger Optimierungsprobleme ein NP-hartes Problem darstellt. In Kapitel 4.2 haben wir das Rucksackproblem als ein ganzzahliges bzw. binäres Optimierungsproblem formuliert. Wäre es nun möglich, ganzzahlige Optimierungsprobleme in polynomieller Zeit zu lösen, so könnten wir das Rucksackproblem und damit auch alle anderen NP-harten Probleme in polynomieller Zeit lösen.

Frage 6.5: Wie sieht die allgemeine Idee aus, um mit Hilfe der linearen Optimierung NP-harte Probleme zu lösen?

Die allgemeine Idee besteht darin, das NP-harte Problem zuerst einmal als ein ganzzahliges lineares Optimierungsproblem zu formulieren. Anschließend wird die Ganzzahligkeitsbedingung relaxiert und das so entstandene lineare Optimierungsproblem in polynomieller Zeit gelöst. Dabei werden höchstwahrscheinlich nicht ganzzahlige Lösungen entstehen, welche nachbearbeitet werden müssen, so dass diese ganzzahlig wird. Dieser letzte Schritt ist entweder nicht effizient durchführbar oder aber er verschlechtert die Qualität bzw. Güte der Lösung. Wäre dies nicht der Fall, so wäre P=NP bewiesen.

Frage 6.6: Wie ist MAXSAT bzw. MAXkSAT definiert?

Bei MAXSAT ist eine Menge $X = \{x_1, \ldots, x_n\}$ von Variablen sowie eine Menge $C = (C_1, \ldots, C_m)$ von Klauseln über den Variablen von X gegeben. Eine Klausel ist dabei eine Disjunktion („Veroderung“) von Variablen oder negierten Variablen. Die Aufgabe besteht nun darin, eine Belegung der Variablen zu finden, so dass die Anzahl der erfüllten Klauseln maximiert wird. Es leuchtet schnell ein, dass MAXSAT ein NP-hartes Problem ist, da eine Lösung für das NP-vollständige Entscheidungsproblem SAT mit Hilfe von MAXSAT berechnet werden kann. SAT hat genau dann eine Lösung, wenn die Anzahl der erfüllbaren Klauseln in MAXSAT die Anzahl aller Klauseln in C ist.

MAXkSAT ist genau wie MAXSAT definiert, nur das hier die zusätzliche Forderung gegeben ist, dass jede Klausel genau k verschiedenen Literale enthält.

Frage 6.7: Wie kann MAXSAT als ganzzahliges Optimierungsproblem formuliert werden?

Wir formulieren MAXSAT wie folgt als ganzzahliges Optimierungsproblem:

$$\text{Maximiere } \sum_{j=1}^{m} z_j$$

unter den Nebenbedingungen:

$$y_i, z_j \in \{0,1\} \text{ für alle } i = 1, \ldots, n \text{ und } j = 1, \ldots, m$$

und

$$\sum_{i \mid x_i \text{ kommt in Klausel } C_j \text{ vor}} y_i + \sum_{i \mid \overline{x_i} \text{ kommt in Klausel } C_j \text{ vor}} (1 - y_i) \geq z_j$$

für $j = 1, \ldots, m$. Dabei entspricht y_i der Belegung der jeweiligen Variable x_i aus der Eingabe für MAXSAT. Die Variable z_j gibt praktisch an, ob die j-te Klausel erfüllt wurde. Dabei gibt die Summe

$$\sum_{i \mid x_i \text{ kommt in Klausel } C_j \text{ vor}} y_i + \sum_{i \mid \overline{x_i} \text{ kommt in Klausel } C_j \text{ vor}} (1 - y_i)$$

an, wie viele Literale in der j-ten Klausel erfüllt sind. Damit z_j seinen maximalen Wert 1 (da $z_j \in \{0,1\}$ gilt) annehmen kann, muss also mindestens ein Literal erfüllt sein. Damit entsprechen also Lösungen für das MAXSAT-Problem Lösungen für das eben beschriebene Optimierungsproblem und umgekehrt.

MAXkSAT wird genau wie MAXSAT modelliert. Der einzige Unterschied besteht darin, dass durch die zusätzliche Forderung in der Nebenbedingung für jede Klausel nur noch $k+1$ Variablen (k Variablen aus X und eine z-Variable für die jeweilige Klausel) stehen.

6.2 Ein einfacher randomisierter Algorithmus für die Probleme MAXSAT und MAXkSAT

Frage 6.8: Wie funktioniert der einfache randomisierte Algorithmus für MAXSAT und MAXkSAT (Algorithmus A)?

Im einfachsten Algorithmus für das MAXSAT-Problem wird die Belegung der Variablen einfach ausgewürfelt. Dabei wird von einem fairen Würfel ausgegangen, d.h. jede Variable wird mit einer Wahrscheinlichkeit von $\frac{1}{2}$ auf Null bzw. auf Eins gesetzt. Die Rechenzeit ist linear, da für jede Variable einmal gewürfelt werden muss. Ist k nicht allzu klein, werden bei diesem Algorithmus im Durchschnitt schon viele Klauseln erfüllt.

Frage 6.9: Wie groß ist die erwartete Anzahl erfüllter Klauseln bei Algorithmus A für das MAXkSAT-Problem?

Die erwartete Anzahl erfüllter Klauseln bei Algorithmus A beträgt:

$$m \cdot \left(1 - \left(\frac{1}{2}\right)^k\right)$$

Der Beweis ist relativ einfach. Wir definieren, die Zufallsvariable X_i, die den Wert 1 annimmt, wenn die Klausel C_i erfüllt ist und ansonsten den Wert 0 annimmt. Der Erwartungswert für eine Klausel ist nun wie folgt definiert:

$$\begin{aligned} E(X_i) &= \text{Prob}(i\text{-te Klausel nicht erfüllt}) \cdot 0 + \text{Prob}(i\text{-te Klausel erfüllt}) \cdot 1 \\ &= \text{Prob}(i\text{-te Klausel erfüllt}) \end{aligned}$$

Das Schöne ist hierbei, dass die Wahrscheinlichkeit der Nichterfüllung einer Klausel einfach wegfällt, da die Zufallsvariable in diesem Fall den Wert 0 annimmt. Wir berechnen nun die Wahrscheinlichkeit, dass eine Klausel erfüllt wird. Dabei werden wir jedoch den Umweg über den Fall gehen, dass sie nicht erfüllt wird. Jede Klausel enthält k Literale. Damit eine Klausel erfüllt ist, muss mindestens ein Literal zu Eins ausgewertet werden. Die Wahrscheinlichkeit, dass wir ein Literal so wählen, dass es Null wird, beträgt $\frac{1}{2}$. Dementsprechend beträgt die Wahrscheinlichkeit, alle Literale falsch zu wählen, genau $\left(\frac{1}{2}\right)^k$. Wenn wir nicht alle Literale falsch wählen, muss mindestens eines richtig gewählt worden sein. Diese Wahrscheinlichkeit dafür beträgt $1 - \left(\frac{1}{2}\right)^k$ und somit gilt für den Erwartungswert:

$$\forall i \in \{1, \ldots, m\} \ : \ E(X_i) = 1 - \left(\frac{1}{2}\right)^k$$

Da wir an der erwarteten Anzahl erfüllter Klauseln interessiert sind, müssen wir noch etwas weiter rechnen. Zuerst wird eine neue Zufallsvariable X eingeführt, welche die Anzahl erfüllter Klauseln zählt. Es gilt

$$X = X_1 + \cdots + X_m$$

und aufgrund der Linearität des Erwartungswertes gilt weiter:

$$E(X) = E(X_1) + \cdots + E(X_m) = m \cdot \left(1 - \left(\frac{1}{2}\right)^k\right)$$

Für das Problem MAXSAT kann nun allerdings nicht mehr angenommen werden, dass jede Klausel k Literale enthält und es ist sogar möglich, dass alle Klauseln nur $k = 1$ Literale

enthalten. In diesem Fall beträgt die Wahrscheinlichkeit, dass eine Klausel erfüllt wird genau $\frac{1}{2}$ und für die erwartete Anzahl erfüllter Klauseln gilt $E(X) = m \cdot \frac{1}{2}$. Da die maximale Anzahl erfüllbarer Klauseln m beträgt, ist Algorithmus A also ein randomisierter Algorithmus mit einer erwarteten worst case Güte von 2.

6.3 Randomisiertes Runden

Frage 6.10: Was ist die Idee beim randomisierten Runden?

Die Grundidee, wie NP-harte Probleme mit Hilfe der linearen Optimierung gelöst werden können, wurde zu Beginn des Kapitels schon beschrieben. Im Fall des randomisierten Rundens werden wir die Lösung eines relaxierten Problems nun als Indikator nutzen, welche Belegung wir für die Variablen wählen sollten. Dabei wird das Problem wie folgt relaxiert. Anstatt der Bedingung $y_i, z_j \in \{0, 1\}$ lassen wir nun die Bedingung $0 \leq y_i, z_j \leq 1$ zu. Es stellt sich dabei nun jedoch die Frage, wie wir einen Wert aus dem Intervall $[0, 1]$ für eine Variable y_i interpretieren sollen? Wir werden wie folgt vorgehen. Wenn die Variable y_i im relaxierten Problem den Wert $p_i \in [0, 1]$ zugewiesen bekommen hat, dann setzen wir die Variable x_i mit einer Wahrscheinlichkeit von p_i auf den Wert 1 und mit einer Wahrscheinlichkeit von $1 - p_i$ auf den Wert 0. Dieser Algorithmus wird im Folgenden Algorithmus B genannt.

Frage 6.11: Inwiefern ist Algorithmus A ein Spezialfall von Algorithmus B?

Algorithmus A kann als ein Spezialfall von Algorithmus B aufgefasst werden, bei dem die Wahrscheinlichkeit, eine Variable auf den Wert 0 oder 1 zu setzen, nicht erst über die Lösung des relaxierten Problems ermittelt wurde. Stattdessen wird quasi für jede Variable $y_i = \frac{1}{2}$ gesetzt.

Frage 6.12: Mit welcher Wahrscheinlichkeit werden die einzelnen Klauseln in Algorithmus B erfüllt?

Im Folgenden seien y und z jeweils die Werte der Lösung des relaxierten Problems. Wir betrachten die j-te Klausel und gehen davon aus, dass sie die Länge k habe. Dann beträgt die Wahrscheinlichkeit, dass sie durch randomisiertes Runden erfüllt wird, mindestens:

$$\left(1 - \left(1 - \frac{1}{k}\right)^k\right) \cdot z_j$$

Für den Beweis betrachten wir die Klausel $x_1 \vee \cdots \vee x_k$, bei der die Variablen unter Umständen umbenannt wurden. Die Wahrscheinlichkeit, dass diese Klausel nicht erfüllt wird beträgt dann $\prod_{i=1}^{k}(1 - y_i)$. Dementsprechend beträgt die Wahrscheinlichkeit ihrer Erfüllung also $1 - \prod_{i=1}^{k}(1 - y_i)$. Als nächstes möchten wir $\prod_{i=1}^{k}(1 - y_i)$ weiter umformen. Dazu benutzen wir den Satz, dass für positive Zahlen $a_1, \ldots, a_k$ der geometrische Mittelwert nicht größer als der arithmetische Mittelwert ist, also:

$$a_1 \cdot \ldots \cdot a_k \leq \left(\frac{a_1 + \cdots + a_k}{k}\right)^k$$

Damit gilt schonmal:

$$\prod_{i=1}^{k}(1 - y_i) \leq \left(\frac{(1 - y_1) + \cdots + (1 - y_k)}{k}\right)^k$$

Der Zähler auf der rechten Seite stört uns in dieser Form jedoch noch etwas. Wir formen ihn mit Hilfe der Nebenbedingung der j-ten Klausel um. Da $y_1, \ldots, y_n, z_1, \ldots, z_m$ eine Lösung für das relaxierte Optimierungsproblem ist, gilt also:

$$y_1 + \cdots + y_k \geq z_j$$

Wir multiplizieren dies nun mit -1 und erhalten:

$$-y_1 - \cdots - y_k \leq -z_j$$

Nun addieren wir k Einsen auf die linke und rechte Seite. Anschließend klammern wir noch etwas:

$$(1 - y_1) + (1 - y_2) + \cdots + (1 - y_k) \leq k - z_j$$

Also können wir den Zähler mit $k - z_j$ abschätzen:

$$\prod_{i=1}^{k}(1 - y_i) \leq \left(\frac{k - z_j}{k}\right)^k = \left(1 - \frac{z_j}{k}\right)^k$$

Als nächstes wird dieses Ergebnis in die Formel für die Wahrscheinlichkeit der Erfüllung der j-ten Klausel eingesetzt:

$$\prod_{i=1}^{k}(1 - y_i) \leq \left(1 - \frac{z_j}{k}\right)^k, \text{ also } 1 - \prod_{i=1}^{k}(1 - y_i) \geq 1 - \left(1 - \frac{z_j}{k}\right)^k$$

Um die obige Behauptung zu zeigen, benutzen wir einen Satz, der hier nicht weiter bewiesen wird. Es gilt:

$$\forall x \in [0, 1] \ : \ 1 - \left(1 - \frac{x}{k}\right)^k \geq \left(1 - \left(1 - \frac{1}{k}\right)^k\right) \cdot x$$

Also ergibt sich für obiges Zwischenergebnis:

$$1 - \left(1 - \frac{z_j}{k}\right)^k \geq \left(1 - \left(1 - \frac{1}{k}\right)^k\right) \cdot z_j$$

Auch wenn man vom ganzen Abschätzen jetzt genug haben könnte, wollen wir den Ausdruck in der Klammer noch näher untersuchen. Wie schon einige Male zuvor benutzen wir die Abschätzung der e-Funktion als Grenzwert einer Folge:

$$\left(1 - \frac{1}{k}\right)^k = \left(1 + \frac{(-1)}{k}\right)^k \leq e^{-1} = \frac{1}{e}$$

Also gilt:

$$1 - \left(1 - \frac{1}{k}\right)^k \geq 1 - \frac{1}{e} \approx 0.632$$

Frage 6.13: Welche Güte besitzt Algorithmus B?

k	$1-2^{-k}$	$1-(1-\frac{1}{k})^k$
1	0.5	1.0
2	0.75	0.75
3	0.875	0.704
4	0.938	0.684
5	0.969	0.672

Tabelle 1: Vergleich zwischen Algorithmus A und B für verschiedene Klausellängen k

Sei OPT die maximale Anzahl an Klauseln, die gleichzeitig erfüllt werden können. Dann berechnet Algorithmus B eine Lösung, die erwarteterweise mindestens

$$\left(1-\frac{1}{e}\right)\cdot \text{OPT}$$

Klauseln erfüllt.

Die Lösung des relaxierten Problems ist eine obere Schranke für die Lösung des ganzzahligen Problems, daher gilt $\sum_{j=1}^{m} z_j \geq \text{OPT}$. Wir bilden nun die Summe über die Wahrscheinlichkeiten, dass die einzelnen Klauseln erfüllt werden. Diese entspricht auch dem Erwartugswert der Anzahl erfüllter Klauseln:

$$\sum_{j=1}^{m}\left(1-\left(1-\frac{1}{k}\right)^k\right)\cdot z_j \geq \left(1-\frac{1}{e}\right)\cdot\sum_{j=1}^{m} z_j \geq \left(1-\frac{1}{e}\right)\cdot \text{OPT}$$

Frage 6.14: Wie kann eine Lösung für MAXSAT bzw. MAXkSAT mit erwarteter Güte 4/3 berechnet werden?

Wir haben nun für Algorithmus A und Algorithmus B jeweils die Wahrscheinlichkeiten berechnet, dass eine Klausel der Länge k erfüllt wird. In Tabelle 1 werden die beiden Algorithmen für verschiedene k verglichen. Dabei wurde für Algorithmus B das z_j weggelassen. Es verkleinert den Wert höchstens. Trotzdem kann man aus der Tabelle erahnen, dass Algorithmus A für lange Klauseln gut ist, während Algorithmus B eher auf kürzeren Klauseln gut zu funktionieren scheint. Problematisch ist nur, dass in einer Eingabe für MAXSAT lange und kurze Klauseln vermischt vorkommen können. Es zeigt sich jedoch, dass eine erwartete Güte von $\frac{4}{3}$ erreicht werden kann, wenn Algorithmus A und B hintereinander ausgeführt werden und danach die bessere der beiden Lösungen gewählt wird.

Zuerst stellen wir die Formeln für die erwartete Anzahl erfüllter Klauseln für Algorithmus A und B auf. Dabei sei n_A die Anzahl erfüllter Klauseln für Algorithmus A und n_B der entsprechende Wert für Algorithmus B. Da wir MAXSAT betrachten, können die Klauseln wieder unterschiedlich lang sein, was für die j-te Klausel mit dem Wert $l(C_j)$ berücksichtigt wird. Die erwartete Anzahl erfüllter Klauseln ergibt sich als Summe der Erwartungswerte der einzelnen Klauseln:

$$n_A = \sum_{C_j}(1-2^{-l(C_j)})$$

Da $0 \leq z_j \leq 1$ gilt, wird die Summe also höchstens kleiner, wenn wir jeden Summanden mit z_j multiplizieren. Dies machen wir nun, damit wir, wenn wir gleich die Summe von n_A und n_B bilden, z_j ausklammern können:

$$\sum_{C_j}(1-2^{-l(C_j)}) \geq \sum_{C_j}(1-2^{-l(C_j)})\cdot z_j$$

Die erwartete Anzahl erfüllter Klauseln für Algorithmus B ergibt sich ähnlich:

$$n_B \geq \sum_{C_j} \left(1 - \left(1 - \frac{1}{l(C_j)}\right)^{l(C_j)}\right) \cdot z_j$$

Wir wollen nun zeigen, dass entweder Algorithmus A oder Algorithmus B eine Güte von $\frac{4}{3}$ besitzt. Dies ist gleichbedeutend mit $\max\{n_A, n_B\} \geq \frac{3}{4} \cdot \text{OPT}$. Wir zeigen hier sogar mehr, nämlich $\max\{n_A, n_B\} \geq \frac{3}{4} \cdot \sum_j z_j$. Dies ist erlaubt, da $\sum_j z_j \geq \text{OPT}$ ist. Zuerst schätzen wir jedoch $\max\{n_A, n_B\}$ nach unten ab. Es gilt: $\max\{n_A, n_B\} \geq \frac{n_A + n_B}{2}$. Es wird nun also gezeigt:

$$\frac{n_A + n_B}{2} \geq \frac{3}{4} \cdot \sum_j z_j \quad \text{bzw.} \quad n_A + n_B \geq \frac{3}{2} \cdot \sum_j z_j$$

Wir haben n_A und n_B schon abgeschätzt, also gilt für ihre Summe:

$$\begin{aligned} n_A + n_B &\geq \sum_{C_j} \left(1 - 2^{-l(C_j)}\right) \cdot z_j + \sum_{C_j} \left(1 - \left(1 - \frac{1}{l(C_j)}\right)^{l(C_j)}\right) \cdot z_j \\ &= \sum_{C_j} \left[\left(1 - 2^{-l(C_j)}\right) + \left(1 - \left(1 - \frac{1}{l(C_j)}\right)^{l(C_j)}\right)\right] \cdot z_j \end{aligned}$$

Der Term in der eckigen Klammer kann für jedes $l(C_j)$ gegen $\frac{3}{2}$ abgeschätzt werden. Für $l(C_j) = 1$ und $l(C_j) = 2$ ergibt sich genau der Wert $\frac{3}{2}$. Für $l(C_j) \geq 3$ gilt $1 - 2^{l(C_j)} \geq \frac{7}{8}$ und $1 - (1 - \frac{1}{l(C_j)}) \geq 1 - \frac{1}{e}$. Für die Summe ergibt sich mindestens $\frac{7}{8} + 1 - \frac{1}{e} \geq \frac{3}{2}$. Wenn wir die eckige Klammer in jedem Summanden also gegen $\frac{3}{2}$ abschätzen und rausziehen, ergibt sich genau die Behauptung.

6.4 Derandomisierung und Methode der bedingten Wahrscheinlichkeiten

Frage 6.15: Was versteht man unter Derandomisierung?

Unter Derandomisierung wird die Umwandlung eines randomisierten Algorithmus in einen deterministischen Algorithmus verstanden. Randomisierte Algorithmen haben den Vorteil, dass sie oft schneller sind als die deterministischen Algorithmen. Der Nachteil besteht jedoch darin, dass eine gute Lösung nicht in jedem Fall garantiert werden kann. Ein deterministischer Algorithmus, der durch Derandomisierung aus einem randomisierten Algorithmus hervorgegangen ist, garantiert eine Lösung deren Wert mindestens so gut ist, wie der erwartete Wert beim randomisierten Algorithmus. Der Preis, der für diese Sicherheit gezahlt wird, ist, dass der derandomisierte Algorithmus oft eine schlechtere Laufzeit aufweist und komplexer ist.

Frage 6.16: Beschreibe das allgemeine Vorgehen beim Derandomisieren!

Es sei ein randomisierter Algorithmus A_{rand} gegeben, der stets eine gültige Lösung für das betrachtete Optimierungsproblem berechnet. Der Wert der berechneten Lösung sei als eine Zufallsvariable Z gegeben. Dabei hängt der Wert der Zufallsvariablen natürlich von den gezogenen Zufallsbits $y_1, \ldots, y_m$ ab. Im Falle von MAXSAT hängt von den gezogenen Zufallsbits z.B. ab, wie viele Klauseln erfüllt werden. Wir werden nun gleich mit Hilfe der bedingten Wahrscheinlichkeiten einen Algorithmus A_{det} konstruieren, der eine geeignete Belegung der

Zufallsbits berechnet, deren Wert mindestens dem Erwartungswert des randomisierten Algorithmus entspricht. Dabei kann man sich zunächst fragen, ob so eine Lösung bzw. Belegung überhaupt exisitiert. Dies ist jedoch relativ einfach gezeigt. Da der Erwartungswert so etwas wie ein mit den Wahrscheinlichkeiten gewichteter Durchschnitt der Belegungen der Zufallsvariablen ist, muss es eine Belegung geben, die mindestens den Erwartungswert $E(Z)$ als Wert annimmt. Dies liegt daran, dass die Wahrscheinlichkeiten, mit denen gewichtet wird, alle zwischen Null und Eins liegen, den Erwartungswert also höchstens kleiner machen können. Ansonsten hätten wir den Erwartungswert gar nicht berechnen können. Bei einem fairen Würfel, kann es z.B. nicht passieren, dass ein Erwartungswert von Sieben berechnet wird, da die Belegungen der Zufallsvariablen alle zwischen Eins und Sechs liegen und jede nur mit höchstens Eins gewichtet wird, wobei die Summe der Gewichte genau Eins ergibt.

Allgemein werden bei der Methode der bedingten Wahrscheinlichkeiten nun sukzessive die beste Belegung $d_1, \ldots, d_m$ für die Zufallsbits berechnet. Dabei wird für jedes Bit geschaut, welche Belegung vielversprechender zu sein scheint. Es sei die partielle Belegung $d_1, \ldots, d_i$ mit $i < m$ schon berechnet worden. Für diese berechnen wir dann das sogenannte Gewicht $w(d_1, \ldots, d_i)$, welches so etwas wie ein bedingter Erwartungswert ist. Dies bedeutet, dass wir den Erwartungswert ausrechnen, der sich ergibt, wenn die Bits $d_1, \ldots, d_i$ schon festgelegt sind und die restlichen Bits $d_{i+1}, \ldots, d_m$ noch zufällig gewählt werden können. Formal drücken wir dies wie folgt aus:

$$w(d_1, \ldots, d_i) = E(Z \mid y_1 = d_1,\ y_2 = d_2,\ \ldots,\ y_i = d_i)$$

Wenn wir noch keine der Variablen festgelegt haben, entspricht das Gewicht dem normalen Erwartungswert: $w(\emptyset) = E(Z)$. Haben wir alle Variablen festgelegt, dann haben wir eine eindeutige Lösung berechnet, also $w(d_1, \ldots, d_m) = E(Z \mid y_1 = d_1,\ \ldots,\ y_m = d_m)$. Wie schon erwähnt wurde, wollen wir nun für jedes Bit y_i die Belegung wählen, die eine bessere Lösung erwarten lässt. Wenn also die Bits $d_1 \ldots, d_{i-1}$ schon festgelegt sind, berechnen wir $w(d_1, \ldots, d_{i-1}, 0)$ und $w(d_1, \ldots, d_{i-1}, 1)$. Ist ersterer Wert größer, setzen wir $d_i = 0$ und ansonsten $d_i = 1$. Es ergibt sich also Algorithmus 8.

Algorithmus 8 Algorithmus der Methode der bedingten Wahrscheinlichkeiten

```
for i = 1 to m do
  if w(d_1, ..., d_{i-1}, 0) > w(d_1, ..., d_{i-1}, 1) then
    d_i = 0
  else
    d_i = 1
  end if
end for
Lasse A_rand laufen, aber ersetze die Zufallsbits y_1, ..., y_m durch d_1, ..., d_m.
```

Frage 6.17: Zeige, dass Algorithmus 8 eine Lösung berechnet, die mindestens dem Erwartungswert des randomisierten Algorithmus entspricht!

Zur Berechnung des Gewichts $w(d_1, \ldots, d_i)$ wird der bedingte Erwartungswert berechnet, bei dem davon ausgegangen wird, dass die erste i Bits gesetzt sind und die restlichen zufällig ausgewürfelt werden. Es zeigt sich jedoch, dass dieser bedingte Erwartungswert durch den

Erwartungswert bei dem auch schon das Bit d_{i+1} gesetzt ist, berechnet werden kann. Es gilt:

$$\begin{aligned} w(d_1,\ldots,d_i) &= E(Z \mid y_1 = d_1,\ \ldots,\ y_i = d_i) \\ &= (1-p_{i+1}) \cdot E(Z \mid y_1 = d_1,\ \ldots,\ y_i = d_i,\ y_{i+1} = 0) \\ &\quad + p_{i+1} \cdot E(Z \mid y_1 = d_1,\ \ldots,\ y_i = d_i,\ y_{i+1} = 1) \\ &= (1-p_{i+1}) \cdot w(d_1,\ldots,d_i,0) + p_{i+1} \cdot w(d_1,\ldots,d_i,1) \end{aligned}$$

Dies ist gleichbedeutend damit, dass $w(d_1,\ldots,d_i)$ ein gewichtetes Mittel von $w(d_1,\ldots,d_i,0)$ und $w(d_1,\ldots,d_i,1)$ ist. Da $p_{i+1} \in [0,1]$ gilt, liegt $w(d_1,\ldots,d_i)$ für $p_{i+1} = 0$ bzw. $p_{i+1} = 1$ also genau auf dem Wert $w(d_1,\ldots,d_i,0)$ bzw. $w(d_1,\ldots,d_i,1)$ oder es liegt zwischen diesen Werten. Dann gilt jedoch:

$$w(d_1,\ldots,d_i) \leq \max\{w(d_1,\ldots,d_i,0), w(d_1,\ldots,d_i,1)\}$$

Da wir den Wert von d_{i+1} stets maximal wählen, gilt also insgesamt:

$$E(Z) = w(\emptyset) \leq w(d_1) \leq w(d_1,d_2) \leq \cdots \leq w(d_1,\ldots,d_m)$$

Der Wert der Lösung, die berechnet wird, ist $w(w_1,\ldots,w_m)$, also wird eine Lösung berechnet die mindestens dem Erwartungswert entspricht.

Frage 6.18: Welche Bedingungen müssen gegeben sein, damit mit Hilfe der bedingten Wahrscheinlichkeiten derandomisiert werden kann?

Ein randomisierter Algorithmus muss im Prinzip zwei Eigenschaften haben, damit er mit Hilfe der bedingten Wahrscheinlichkeiten derandomisiert werden kann:

1. Er muss stets gültige Lösungen liefern. Das bedeutet er darf sich nicht irren oder gar keine Lösung liefern.
2. Es muss möglich sein, effizient den erwarteten Wert einer Lösung zu berechnen, wenn einige der Zufallsbits schon festgelegt sind.

Frage 6.19: Wie können Algorithmus A und B derandomisiert werden?

Algorithmus A und B können mit Hilfe der Methode der bedingten Wahrscheinlichkeiten derandomisiert werden. Sie liefern stets zulässige Lösung, da jede Belegung der Variablen eine zulässige Lösung darstellt. Desweiteren können die Gewichte $w(d_1,\ldots,d_i)$ effizient berechnet werden, wie wir gleich sehen werden.

Es sei X_l die Zufallsvariable, die angibt, ob die l-te Klausel erfüllt wurde oder nicht. Der Wert einer Lösung wird in der Zufallsvariable Z gespeichert, die sich als $Z = X_1 + \cdots + X_m$ berechnet. Für den Erwartungswert $E(Z)$ der Zufallsvariable Z gilt auch hier aufgrund der Linearität des Erwartungswerts: $E(Z) = E(X_1) + \cdots + E(X_m)$. Dummerweise wollen wir jedoch nicht nur den Erwartungswert von Z berechnen, sondern auch den bedingten Erwartungswert $E(Z \mid y_1 = d_1,\ \ldots,\ y_i = d_i)$. Wir haben jedoch Glück, denn auch der bedingte Erwartungswert ist linear. Das bedeutet, wir können ihn wie folgt berechnen:

$$\begin{aligned} E(Z \mid y_1 = d_1,\ \ldots,\ y_i = d_i) &= E(X_1 \mid y_1 = d_1,\ \ldots,\ y_i = d_i) \\ &\quad + \cdots + E(X_m \mid y_1 = d_1,\ \ldots,\ y_i = d_i) \end{aligned}$$

Um das Gewicht $w(d_1,\ldots,d_i)$ auszurechnen, reicht es also die Gewichte bzw. bedingten Erwartungswerte der einzelnen Klauseln zu berechnen. Dabei können drei Fälle unterschieden werden:

1. $C_l \equiv 0$. Dies bedeutet, dass alle in C_l vorkommenden Literale schon eine Belegung bekommen haben und zu Null ausgewertet werden. Die Klausel kann in diesem Fall nicht mehr erfüllt werden und dementsprechend gilt für den bedingten Erwartungswert:

$$E(X_i \mid y_1 = d_1, \ldots, \ y_i = d_i) = \text{Prob}(X_i = 1 \mid y_1 = d_1, \ldots, \ y_i = d_i) = 0$$

2. $C_l \equiv 1$. In diesem Fall ist mindestens ein Literal durch die Belegung $d_1, \ldots, d_i$ mit 1 belegt und die Klausel ist in jedem Fall erfüllt. Es gilt:

$$E(X_i \mid y_1 = d_1, \ldots, \ y_i = d_i) = \text{Prob}(X_i = 1 \mid y_1 = d_1, \ldots, \ y_i = d_i) = 1$$

3. Alle durch $y_1 = d_1, \ldots, y_i = d_i$ belegten Literale in C_j sind mit 0 belegt und es gibt noch t unbelegte Literale $x_{i(1)}, \ldots, x_{i(t)}$. Dann gilt für den bedingten Erwartungswert dieser Klausel:

$$\begin{aligned} E(X_l \mid y_1 = d_1, \ldots, y_i = d_i) &= \text{Prob}(X_i = 1 \mid y_1 = d_1, \ldots, y_i = d_i) \\ &= 1 - (1 - p_{i(1)}) \cdot \ldots \cdot (1 - p_{i(t)}) \end{aligned}$$

   Dabei wird nun davon ausgegangen, dass die Literale alle nicht negiert sind. Für ein negiertes Literal $\overline{x}_{i(j)}$ würde statt $(1 - p_{i(j)})$ der Wert $p_{i(j)}$ benutzt werden. Denn in diesem Fall würde das Literal ja nur zum ungünstigen Wert 0 ausgewertet werden, wenn es mit 1 belegt wird, was mit Wahrscheinlichkeit $p_{i(j)}$ passiert.

Da dies alle Fälle sind, welche auftreten können, sind wir also in der Lage, das Gewicht $w(d_1, \ldots, d_i)$ effizient zu berechnen. Damit haben wir alles für eine derandomisierte Variante von Algorithmus A und B zusammen.

7 Randomisierte Suchheuristiken

7.1 Motivation

Frage 7.1: Was für Algorithmen wünschen wir uns für Optimierungsprobleme? Was können wir machen, wenn wir für ein Problem keinen Algorithmus dieser Art kennen?
Im Allgemeinen wünschen wir uns Algorithmen, die mit einer geringen Worst Case Rechenzeit stets eine optimale Lösung liefern. Für einige Probleme kennen wir jedoch keine solchen Algorithmen. In diesem Fall haben wir die folgenden Möglichkeiten:

Branch-and-Bound: Branch-and-Bound Algorithmen haben wir in Kapitel 4 behandelt. Sie liefern immer eine optimale Lösung. Allerdings sind sie bezüglich der Rechenzeit heuristisch. Wir können also nur hoffen, dass sie für Probleme, die für uns interessant sind, eine gute Laufzeit aufweisen.

Approximationsalgorithmen: Approximationsalgorithmen wurden in Kapitel 5 vorgestellt. Sie liefern nicht zwingend eine optimale Lösung, dafür kann jedoch die Güte der Lösung beliebig variiert werden. Aufgrund dieser Einschränkung weisen sie oft eine gute Laufzeit auf. Oft ist die Güte der Lösung auch besser als die durch eine Analyse abgeschätzte Güte der Lösung.

Randomisierung: Randomisierung wird oft benutzt, wenn in einem Algorithmus eine Entscheidung getroffen werden muss, es jedoch nicht effizient berechenbar ist, welche Entscheidung die Beste ist. In diesem Fall wird die Entscheidung einfach zufällig getroffen. Randomisierung kann also helfen, wenn es genügend viele gute Optionen gibt, wir jedoch eine gute Option nicht durch eine gezielte Suche berechnen können.

Frage 7.2: Was sind randomisierte Suchheuristiken? Wie werden sie verwandt?

Frage 7.3: Welche Szenarien für den Einsatz randomisierter Suchheuristiken können unterschieden werden?
Es können zwei Szenarien unterschieden werden:

Gut strukturierte Probleme:

Black-Box-Optimierung:

Frage 7.4: Wie ist der (1+1)EA definiert?
Der (1+1)EA dient zur Maximierung von Funktionen $f : \{0,1\}^n \to \mathbb{R}$. Sie funktioniert wie folgt:

1. Wähle ein $x \in \{0,1\}^n$ gemäß der Gleichverteilung.
2. Wiederhole den folgenden Mutationsschritt bis ein Stopkriterium greift. Bilde ein neues $x' = (x'_1, \ldots, x'_n)$ wobei jedes Bit x'_i unabhängig von den anderen gebildet wird. Es gilt:
$$\text{Prob}(x'_i = 1 - x_i) = \frac{1}{n} \quad \text{und} \quad \text{Prob}(x'_i = x_i) = 1 - \frac{1}{n}$$
Mit einer Wahrscheinlichkeit von $\frac{1}{n}$ wird ein Bit in x also geflippt und mit der Restwahrscheinlichkeit von $1 - \frac{1}{n}$ nicht. Es wird nun x genau dann durch x' ersetzt, wenn $f(x') \geq f(x)$ gilt.

Dabei ist die Wahrscheinlichkeit von $\frac{1}{n}$ eine Standardwahl. Ist die Wahrscheinlichkeit kleiner, passiert oft nicht viel, und wenn sie größer ist, können optimale Suchpunkte oft nicht so genau getroffen werden.

Die erwartete Anzahl geflippter Bits beträgt 1. Es sei X_i die Indikatorvariable, die angibt, ob ein Bit flippt ($X_i = 1$) oder nicht ($X_i = 0$). Dann gilt für ein Bit:

$$E(X_i) = \text{Prob}(X_i = 0) \cdot 0 + \text{Prob}(X_i = 1) \cdot 1 = \frac{1}{n}$$

Die Anzahl geflippter Bits beträgt $X = X_1 + \cdots + X_n$. Aufgrund der Linearität des Erwartungswertes gilt dann:

$$E(X) = E(X_1 + \cdots + X_n) = E(X_1) + \cdots + E(X_n) = n \cdot E(X_i) = n \cdot \frac{1}{n} = 1$$

7.2 Eine allgemeine obere Schranke für die erwartete Optimierungszeit des (1+1)EA

Frage 7.5: Was ist eine f-basierte Partition?
Für Teilmengen $A, B \subseteq \{0,1\}^n$ gilt $A <_f B$ wenn für alle $x \in A$ und $y \in B$ gilt: $f(x) < f(y)$. Damit können wir nun die Elemente des Suchraums wie folgt partitionieren. Eine f-basierte Partition von $\{0,1\}^n$ ist eine Partition $A_1, \ldots, A_m$, für welche die folgende zwei Bedingungen gelten:

1. $A_1 <_f A_2 <_f \cdots <_f A_m$.
2. A_m enthält genau die Suchpunkte mit maximalem f-Wert.

Frage 7.6: Wie groß ist die erwartete Optimierungszeit für eine f-basierte Partition?
Es sei eine f-basierte Partition $A_1, \ldots, A_m$ gegeben. Wir definieren nun zuerst für jedes Element einer Partition die Wahrscheinlichkeit, dass es nach einer Mutation in einer größeren Partition landet. Es ist also $s(a)$ für $a \in A_i$ mit $i < m$ die Wahrscheinlichkeit, dass aus a durch Mutation eine Suchpunkt $b \in A_j$ mit $j > i$ entsteht. Für jede Partition A_i definieren wir dann die Wahrscheinlichkeit $s(i) = \min\{s(a) \mid a \in A_i\}$. Es gibt also ein Element in A_i, welches die kleinste Wahrscheinlichkeit hat, dass es nach einer Mutation in einer höhere Partition gelangt. Diese Wahrscheinlichkeit nehmen wir nun pessimistischerweise als repräsentative Wahrscheinlichkeit für jedes Element der Partition an. Diese pessimistische Wahl machen wir, da wir ja relativ einfach eine obere Schranke für die erwartete Optimierungszeit abschätzen wollen. Für diese gilt dann:

$$E(X_f) = 1 + \sum_{i=1}^{m-1} p(A_i) \cdot E(X_f \mid x_1 \in A_i)$$

Die erwartete Optimierungszeit ist die erwartete Anzahl an Mutationen, die auftreten müssen, bis wir uns in der Partition A_m befinden. Die 1 in der Summe für $E(X_f)$ kommt für die Erzeugung des ersten Elements x zustande. Dann müssen wir noch $m-1$ Fälle unterscheiden. Da das erste Element x zufällig erzeugt wird, können wir in verschiedenen Partitionen starten. Wenn wir sofort in A_m landen müssen wir erwartungmäßig nichts mehr machen. Wenn

wir in einer anderen Partition starten, müssen wir uns erst bis A_m „hocharbeiten“. Daher läuft die Summe nur bis $m-1$. Dabei gibt $P(A_i)$ die Wahrscheinlichkeit an, dass x_1 in der Partition A_i liegt. Und $E(X_f \mid x_1 \in A_i)$ ist die erwartete Anzahl an Schritten, wenn in Partition A_i gestartet wird. Diese erwartete Anzahl können wir auch genauer angeben bzw. abschätzen. Die Wahrscheinlichkeit $s(i)$ gibt an, wie wahrscheinlich es ist, sich in eine höhere Partition zu bewegen. Doch wie bekommen wir aus diesem Wert die erwartete Anzahl an Schritten, bis dies tatsächlich passiert? Die Wahrscheinlichkeitstheorie besagt, dass wenn ein Ereignis mit Wahrscheinlichkeit p eintritt, es erwartungsgemäß nach p^{-1} Versuchen eintritt. Dazu ein Beispiel: Die Wahrscheinlichkeit eine 6 zu Würfeln beträgt $\frac{1}{6}$. Das Ereignis eine 6 zu würfeln hat eine erwartete Wartezeit von $\left(\frac{1}{6}\right)^{-1} = 6$. Entsprechendes gilt nun auch für den Wert $s(i)$. Wenn die Wahrscheinlichkeit $s(i)$ beträgt, sich aus der Partition A_i in eine höhere Partition zu bewegen, dann beträgt die erwartete Zeit, bis dieses Ereignis eintritt $s(i)^{-1}$. Bei der Abschätzung, wie lange wir brauchen, um von A_i nach A_m zu gelangen, werden wir ebenfalls wieder pessimistisch sein und annehmen, dass wir uns dazu durch *alle* Zwischenpartitionen bewegen müssen. Um von A_i zu A_m zu gelangen, brauchen wir also erwartungsgemäß $s(i)^{-1} + s(i+1)^{-1} + \cdots + s(m-1)^{-1}$ Schritte. Dies setzen wir nun in die obige Formel ein und formen diese weiter um:

$$\begin{aligned}
E(X_f) &= 1 + \sum_{i=1}^{m-1} p(A_i) \cdot (s(i)^{-1} + \cdots + s(m-1)^{-1}) \\
&\leq 1 + \sum_{i=1}^{m-1} p(A_i) \cdot (s(1)^{-1} + \cdots + s(m-1)^{-1}) \\
&= 1 + (s(1)^{-1} + \cdots + s(m-1)^{-1}) \cdot \underbrace{\sum_{i=1}^{m-1} p(A_i)}_{\leq 1} \\
&\leq 1 + s(1)^{-1} + \cdots + s(m-1)^{-1}
\end{aligned}$$

Wenn man sich dieses Ergebnis anguckt, ist es im nachhinein etwas ernüchternd. Denn die Formel besagt im Prinzip nichts anderes, als dass wir im schlimmsten Fall annehmen können, uns durch jede Partition zu bewegen, um zu A_m zu gelangen. Trotzdem ist diese Abschätzung in vielen Fällen schon recht gut.

Frage 7.7: Wie ist ONEMAX definiert? Welche erwartete Optimierungszeit hat diese Funktion?

Die Funktion ONEMAX ist definiert als $f(x) = x_1 + \cdots + x_n$ und macht nicht anderes, als die Einsen in x zu zählen. Sie ist maximiert, wenn alle Einsen gesetzt sind. Um die erwartete Optimierungszeit möglichst gut abzuschätzen, müssen wir eine gute f-basierte Partition finden. Wir werden jedoch nicht sehr kreativ sein und einfach alle $x \in \{0,1\}^n$ in die Partition A_i stecken, für die $f(x) = i$ gilt. Eingaben mit gleich vielen Einsen befinden sich also in einer Partition und es entstehen die $n+1$ Partitionen $A_0, \ldots, A_n$. Wie wir in der eben gefundenen oberen Schranke für die erwartete Optimierungszeit gesehen haben, müssen wir die Wahrscheinlichkeiten $s(i)$ kennen bzw. gut abschätzen können.

Wann bewegen wir uns nun in eine höhere Partition? Die Anzahl der Einsen muss dabei um mindestens 1 steigen. Dies kann auf verschiedene Arten und Weisen passieren. Entweder flippt eine 0 auf eine 1 oder mehrere Bits verändern sich, so dass am Ende mehr Einsen als

Nullen entstehen. Wir werden jedoch die meisten dieser Fälle für unsere Abschätzung einfach unterschlagen und nur die Fälle betrachten, bei denen genau eine 0 in eine 1 mutiert. In der Partition A_i gibt es $n - i$ Nullen, die mutieren können. Zuerst betrachten wir nun die Wahrscheinlichkeit, dass sich genau ein Bit ändert. Diese Wahrscheinlichkeit beträgt:

$$\frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^{n-1} \geq \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^{n-1} \cdot \underbrace{\left(1 - \frac{1}{n}\right)}_{\in[0,1[} = \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^{n} \geq \frac{1}{n} \cdot e^{-1} = \frac{1}{en}$$

Dies gilt für alle $n - i$ Bits, die den Wert 0 haben. Also:

$$s(i) \geq \frac{n-i}{en}$$

Wir setzen den Kehrwert dieser Abschätzung von $s(i)$ in unsere obere Schranke für die erwartete Optimierungszeit ein:

$$E(X_f) = 1 + \sum_{i=0}^{n-1} \frac{en}{n-i} = 1 + en \sum_{i=0}^{n-1} \frac{1}{n-i} = 1 + en \sum_{i=1}^{n} \frac{1}{i}$$

Die Summe $\sum_{i=1}^{n} 1/i$ ist eine Harmonische Reihe $H(n)$ und kann nach oben durch $H(n) \leq \ln n + 1$ abgeschätzt werden. Damit bekommen wir eine erwartete Optimierungszeit von:

$$E(X_f) = 1 + en(\ln n + 1)$$

Frage 7.8: Wann heißt eine Funktion unimodal?
Ein Funktion $f : \{0,1\}^n \to \mathbb{R}$ heißt unimodal, wenn es für jeden nicht optimalen Suchpunkt einen benachbarten Suchpunkt mit einem größeren f-Wert gibt. Die Nachbarschaft ist dabei über den Hammingabstand definiert. In diesem Fall sind zwei Suchpunkte benachbart, wenn ihr Hammingabstand 1 beträgt, d.h. sie unterscheiden sich nur in einem Bit. Wir haben schon ein Beispiel für eine unimodale Funktion kennengelernt, denn die Funktion ONEMAX ist unimodal. Sie ist maximiert, wenn der betrachtete Suchpunkt nur aus Einsen besteht. Jeder nicht maximale Suchpunkt enthält daher mindestens eine 0. Wenn diese Null geflippt wird, ist der entstehende Suchpunkt zum ursprünglichen Suchpunkt benachbart und hat einen größeren f-Wert.

Frage 7.9: Wie hoch ist die erwartete Optimierungszeit bei einer unimodalen Funktion?
Im Gegensatz zu ONEMAX wissen wir bei einer allgemeinen unimodalen Funktion nicht, wie viele unterschiedliche Funktionswerte angenommen werden können. Daher bezeichnen wir im Folgenden mit $I(f)$ die Anzahl der unterschiedlichen Funktionswerte für eine gegebene unimodale Funktion f. Wir wissen nun, dass jeder nicht optimale Suchpunkt einen Nachbarn hat, der einen größeren f-Wert hat. Da sich dieser Suchpunkt in der Nachbarschaft befindet, muss also ein Bit flippen, damit wir ihn erreichen. Die Wahrscheinlichkeit, dass genau ein Bit flippt, haben wir etwas weiter oben schon mit $1/en$ abgeschätzt. Wir wissen auch nur, dass es in jeder Partition mindestens einen Suchpunkt gibt, welcher durch Mutation in einen Suchpunkt umgewandelt wird, der in einer höheren Partition liegt. Daher können wir $s(i)$ für jede Partition auch nur mit

$$s(i) \geq \frac{1}{en}$$

abschätzen. Wir haben noch garnicht gesagt, wie wir die Partitionen wählen. Auch hier wählen wir die Partitionen so, dass Suchpunkte, die denselben f-Wert besitzen in einer Partition landen. Entsprechend gibt es $I(f)$ Partitionen $A_1, \ldots, A_{I(f)}$ und wir können die erwartete Optimierungszeit für eine unimodale Funktion wie folgt abschätzen:

$$\begin{aligned} E(X_f) &\leq 1 + s(1)^{-1} + \cdots + s(I(f)-1)^{-1} \\ &= 1 + \sum_{i=1}^{I(f)-1} en = 1 + en \sum_{i=1}^{I(f)-1} 1 = 1 + en(I(f)-1) \end{aligned}$$

Wie wir schon gesehen haben, ist ONEMAX ebenfalls eine unimodale Funktion. Bei ONEMAX haben wir $n+1$ Partitionen $A_0, \ldots, A_n$ benutzt. Mit dem Ergebnis der erwarteten Optimierungszeit für unimodale Funktionen, bekommen wir jedoch lediglich eine erwartete Optimierungszeit von $1 + en^2 = O(n^2)$ heraus. Dies liegt daran, dass wir bei den unimodalen Funktionen angenommen haben, dass in jeder Partition nur ein Bit flippen kann, um in eine höhere Partition zu gelangen. Bei der genaueren Analyse für ONEMAX haben wir jedoch unser Wissen genutzt, dass für die Elemente der Partition A_i genau $n-i$ Bits für uns günstig flippen können. Aus diesem Grund konnten wir die erwartete Optimierungszeit besser abschätzen und kamen aus $1 + en(\ln n + 1)$.

Frage 7.10: Wie groß ist die erwartete Optimierungszeit bei linearen Funktionen?
ONEMAX ist auch eine lineare Funktion. Wir wollen daher untersuchen, ob wir für lineare Funktionen auch eine gute obere Schranke für die erwartete Optimierungszeit angeben können. Eine lineare Funktion ist von der Form:

$$f(x) = w_0 + w_1 x_1 + \cdots + w_n x_n$$

Für ONEMAX gilt also $w_0 = 0$ und $w_1 = \cdots = w_n = 1$. Um die Analyse besser durchführen zu können, wollen wir nun zuerst einige Einschränkungen machen:

1. Der Summand w_0 wird nicht beachtet, da die (1+1)EA durch ihn nicht beeinflusst wird. Dies liegt daran, dass für jeden Suchpunkt w_0 zu seinem f-Wert hinzuaddiert wird. Wenn wir diesen nun in allen Fällen subtrahieren machen wir also nichts falsch.

2. Wir nehmen alle Gewichte als nicht negativ an. Dies dürfen wir, da wir im Falle eines negativen Gewichtes w_i nichts falsch machen, wenn wir w_i durch $-w_i$ ersetzen (es also positiv machen) und das entsprechende x_i durch $(1-x_i)$ ersetzen. Das dies korrekt ist, kann rechnerisch wie umgangssprachlich erklärt werden.

   Zuerst das rechnerische Argument. Wenn wir in f den Summand $-w_i x_i$ durch $+w_i(1-x_1)$ ersetzen, dann können wir dies wie folgt umformen: $w_i(1-x_i) = w_i - w_i x_i$. Wir erhalten also wieder den ursprünglichen Summand $-w_i x_i$ und einen konstanten zusätzlichen Summand w_i. Dass wir diesen zusätzlichen Summand jedoch auch weglassen können, haben wir in Punkt 1 schon gesehen.

   Umgangssprachlich kann wie folgt argumentiert werden. Für den Summand $-w_i x_i$ wirkt sich eine 0 positv aus, da nichts abgezogen wird, und eine 1 negativ, da in diesem Fall $-w_i$ addiert wird. Ersetzen wir dies nun durch $w_i(1-x_i)$, so wirkt sich eine 0 immer noch positiv aus, da in diesem Fall $w_i(1-0) = w_i$ addiert wird, und eine 1 immer noch negativ, da wir in diesem Fall das positive w_i nicht „mitnehmen“. Wir machen also nichts falsch.

3. Zuletzt wollen wir die Gewicht $w_1, \ldots, w_n$ so umstellen und umnummerieren, dass gilt:

$$w_1 \geq \cdots \geq w_n$$

Dies dürfen wir auch machen, da wir uns die Permutation π der ursprünglichen Elemente merken können und aus einem Ergebnisvektor für die umgestellte Funktion durch Anwendung von π^{-1} einen Ergebnisvektor für das ursprüngliche Problem bekommen. wir nutzen dabei auch aus, dass der (1+1)EA alle Bits gleich behandelt, es also nicht auf die Position der Bits ankommt.

Aus dem zweiten Punkt können wir schnell folgern, dass lineare Funktionen auch unimodale Funktionen sind. Eine Abschätzung über das Ergebnis der erwarteten Optimierungszeit für unimodalen Funktionen hilft in diesem Fall jedoch nicht viel, da es nun nicht mehr reicht, alle Suchpunkte mit gleichem f-Wert in eine Partition zu packen. Wenn nämlich zum Beispiel $w_i = 2^{n-i}$ gilt, wird ein Suchpunkt als Binärzahl interpretiert. Davon gibt es 2^n und dementsprechend wäre auch $I(f) = 2^n$. Damit bekämen wir jedoch die sehr unbefriedigende obere Schranke $E(X_f) \leq 1 + en(2^n - 1) = O(n2^n)$.

Wir brauchen also eine bessere f-basierte Partition und dafür werden wir die dritte Einschränkung nutzen. Das Ziel wird es sein, eine f-basiert Partition derart zu wählen, dass wir möglichst wenig Partitionen haben, für jedes Element einer Partition jedoch garantieren können, dass es durch Mutation eines Bits in eine höhere Partition gelangen kann. Wir erzeugen nun $n + 1$ Partitionen A_i mit $i \in \{1, \ldots, n + 1\}$, wobei wir ausnutzen werden, dass die Koeffizienten w_i absteigend sortiert sind. Es gilt für den eigentlich nicht vorhandenen Koeffizienten $w_{n+1} = 1$:

$$A_i = \{a \mid w_1 + \cdots + w_{i-1} \leq f(a) < w_1 + \cdots + w_i\}$$

Ein Suchpunkt a landet also in der Partition A_i wenn sein Funktionswert $f(a)$ zwischen $\sum_{j=1}^{i-1} w_j$ und $\sum_{j=1}^{i} w_j$ liegt. Dabei entspricht $\sum_{j=1}^{i} w_j$ dem Funktionswert des Suchpunkt bei dem nur die ersten i Bits gesetzt sind. Unser Suchpunkt ist durch diesen echt nach oben beschränkt. Also muss es innerhalb der ersten i Bits von a eine Null geben, denn ansonsten würde $a \notin A_i$ folgen, da $w_1 + \cdots + w_i \leq f(a)$ gelten würde, wenn schon mindestens die ersten i Bits gesetzt sind. Es gibt also für jedes $a \in A_i$ ein $j \leq i$ mit $a_j = 0$. Die Differenz der Summen, die $f(a)$ begrenzen, beträgt genau w_i. Wenn $f(a)$ also um mindestens w_i wächst, landen wir also in einer der höheren Partitionen $A_{i+1}, \ldots, A_{n+1}$. Das Bit an Position j mit $a_j = 0$ trägt aufgrund der Sortierung der Koeffizienten einen Wert $w_j \geq w_i$ bei, wenn es auf 1 flippt. Die Wahrscheinlichkeit, dass genau ein Bit flippt, welches wir uns wünschen, haben wir schon mit $1/en$ berechnet. Diese Wahrscheinlichkeit gilt für alle $n + 1$ Partition, wobei wir uns aus der letzten wieder nicht heraus bewegen müssen. Also gilt:

$$E(X_f) \leq 1 + \sum_{i=1}^{n} en = 1 + n \cdot en = 1 + en^2$$

Frage 7.11: Wie groß ist die erwartete Optimierungszeit des (1+1)EA auf Polynomen vom Grad k mit positiven Gewichten?

Wir untersuchen nun Polynome vom Grad k mit positiven Gewichten. Wir gehen dabei davon aus, dass das betrachtete Polynom aus N Summanden der Form $w_i x_{i_1} \ldots x_{i_j}$ mit $j \leq k$

bestehen. An einem Gewicht w kleben also bis zu k Variablen. Um das Gewicht zu aktivieren, müssen alle Variablen auf 1 gesetzt werden.

Auch hier gehen wir wieder davon aus, dass die N Gewichte alle abfallend sortiert sind. Es kann jedoch nicht mehr trivialerweise angenommen werden, dass alle Gewichte positiv sind. Daher wird dies explizit gefordert. Die Partitionierung können wir genau wie bei den linearen Funktionen vornehmen, d.h. es gibt $N+1$ Partitionen. Auch hier gilt wieder, dass ein Gewicht an einer Position $j \leq i$ aktiviert werden muss, damit wir in einer höheren Partition landen. Diesmal ist dies jedoch bedeutend schwerer, da das Gewicht an bis zu k Variablen gebunden ist, die *alle* 1 sein müssen. Wir haben die Anzahl der Partitionen und brauchen jetzt also nur noch die Wahrscheinlichkeit für jede Partition, dass wir von ihr in eine höhere Partition kommen. Dabei werden wir unter den vielen für uns günstigen Fällen nur einen für speziellen Fall betrachten. Wir wünschen uns, dass ein w_j mit $j \leq i$ aktiviert wird. Es kann sein, dass schon einige Variablen an dem Gewicht w_j zu 1 ausgewertet werden, aber es muss mindestens eine Variable geben, für die dies nicht gilt. Allgemein gibt es $m \leq k$ Variablen die zu Null ausgewertet werden. Wenn wir genau diese Variablen mit 1 belegen, wird das Gewicht w_j aktiviert. Dabei kann der Funktionswert auch nur größer werden, da alle Gewichte positiv sind. Wäre dies nicht so, könnte es ja passieren, dass durch das „Einschalten“ dieser m Variablen auch ein anderes stark negatives Gewicht aktiviert wird und wir eher was verlieren. Wir schätzen nun die Wahrscheinlichkeit ab, dass nur genau diese m Bits geflippt werden:

$$\begin{aligned} s(i) &\geq \left(\frac{1}{n}\right)^m \cdot \left(1-\frac{1}{n}\right)^{n-m} \geq \left(\frac{1}{n}\right)^m \cdot \left(1-\frac{1}{n}\right)^n \\ &\geq \left(\frac{1}{n}\right)^k \cdot \left(1-\frac{1}{n}\right)^n \geq \frac{1}{n^k} \cdot e^{-1} = \frac{1}{n^k e} \end{aligned}$$

Also ist der Kehrwert $s(i)^{-1} = n^k e$ und mit unseren $N+1$ Partitionen können wir die erwartete Optimierungszeit wie folgt abschätzen:

$$E(X_f) \geq 1 + \sum_{i=1}^{N} s(i)^{-1} \geq 1 + \sum_{i=1}^{N} n^k e = 1 + N n^k e$$

7.3 Populationen oder Multistarts?

Dieses Kapitel wird nicht besonders ausführlich behandelt werden, da ich mir hier eine Lücke geleistet hab. Das gleiche gilt für das Kapitel „Kreuzungen können die erwartete Optimierungszeit drastisch senken“.

Frage 7.12: Wie wird bei der Strategie der Populationen vorgegangen? Wie bei der Strategie der Multistarts?

Bei der Strategie der Populationen werden mehrere Suchpunkte verwaltet. In jedem Schritt wird einer dieser Suchpunkte zufällig ausgewählt und mutiert. Ist das Ergebnis dieser Mutation mindestens so gut wie der bisher schlechteste Suchpunkt, so wird dieser durch den neuen Suchpunkt ersetzt.

Bei der Strategie der Multistarts wird der Algorithmus mehrmals hintereinander laufen gelassen und das beste Ergebnis genommen.

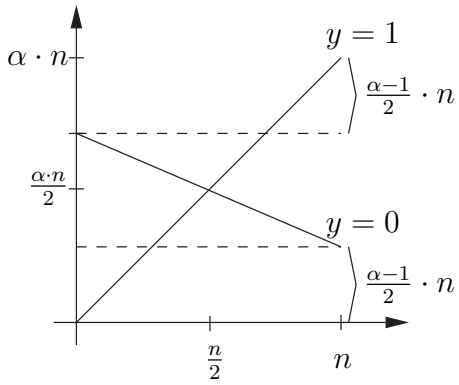


Abbildung 15: Skizze der Funktion CHOICE für $y = 0$ und $y = 1$

Frage 7.13: Wie ist die Funktion CHOICE definiert?
Die Funktion CHOICE ist für $\alpha > 1$ wie folgt definiert, wobei $|x|$ der Funktion ONEMAX entspricht. Es gilt also $|x| = x_1 + \cdots + x_n$.

$$\mathrm{CH}_n(y, x_1, \ldots, x_n) = y \cdot \alpha \cdot |x| + (1 - y) \cdot \left(\frac{(\alpha + 1)}{2} \cdot n - |x| \right)$$

Die Funktion trägt den Namen CHOICE, da über das Bit y jeweils eine der Funktionen $\alpha \cdot |x|$ oder $(\alpha + 1)/2 \cdot n - |x|$ ausgewählt wird. Erstere Funktion ist monoton steigend und hat ihr Maximum für $(1, \ldots, 1)$ bei $\alpha \cdot n$, was auch das globale Optimum darstellt. Die zweite Funktion ist monoton fallend und hat ihr lokales Optimum im Falle von $y = 0$ für die Eingabe $(0, \ldots, 0)$. Die Funktionen haben einen gemeinsamen Funktionswert von $\alpha \cdot \frac{n}{2}$ für $|x| = \frac{n}{2}$:

$$\alpha \cdot |x| = \frac{(\alpha + 1)}{2} \cdot n - |x| \;\Leftrightarrow\; (\alpha + 1) \cdot |x| = \frac{(\alpha + 1)}{2} \cdot n \;\Leftrightarrow\; |x| = \frac{n}{2}$$

Das Aussehen der Funktion ist in Abbildung 15 skizziert. Wie man sieht, nähert sich die Funktion für $y = 0$ ihrem lokalen Optimum, wenn die Anzahl der Nullen wächst, und ihrem globalen Optimum für $y = 1$, wenn die Anzahl der Einsen wächst. Was macht diese Funktion nun problematisch? Wenn man für $y = 0$ oder $y = 1$ schon einen recht guten Wert erreicht hat, wird es immer schwerer, auf die andere Funktion überzuspringen. Für $y = 0$ sind z.B. möglichst viele Nullen gut. Wenn das y nun auf 1 mutiert, sind die vielen Nullen jedoch überhaupt nicht gut und es müssen bei der Mutation zusätzlich noch viele Einsen entstehen, damit der Funktionswert besser als der alte wird. Dies wird jedoch immer unwahrscheinlicher, je näher wir an einem der beiden Optima sind.

Frage 7.14: Wie verhält sich die Funktion CHOICE bei Populationen und wie bei Multistarts?
Bei der Verwendung von Populationen kann es zum sogenannten Hitchhiking-Effekt kommen. Das bedeutet, dass es schon einige recht gute Suchpunkte mit $y = 0$ gibt, welche besser sind als die Suchpunkte für $y = 1$. Bei einer Population kann es nun passieren, dass die Suchpunkte mit $y = 0$ die Suchpunkte mit $y = 1$ nach und nach verdrängen, bis nur noch Suchpunkte mit $y = 0$ übrig sind. Dies kann bei einer Multistartvariante nicht passieren, da hier jeweils unabhängig von den anderen Starts gearbeitet wird.

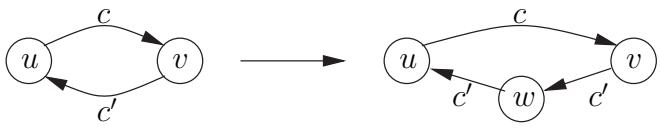


Abbildung 16: Umwandlung einer symmetrischen Kante

8 Maximierung von Flüssen in Netzwerken

8.1 Problemstellung und Beispiele

Frage 8.1: Beschreibe das Grundproblem bei der Maximierung von Flüssen!

Das Grundproblem kann informell in etwa wie folgt umschrieben werden. Gegeben seien zwei Städte A und B. Die Fußballmannschaft der Stadt A hat ein Auswärtsspiel gegen die Mannschaft der Stadt B. Der Besucherstrom soll nun möglichst optimal von Stadt A nach Stadt B gelenkt werden, d.h. die vorhandene Infrstruktur soll möglichst optimal ausgenutzt werden. Das ganze wird nun als ein Netzwerk, ein spezieller Graph, modelliert. Dabei werden Straßen durch Kanten in diesem Graphen dargestellt, die eine gewisse Verkehrskapazität zulassen (z.B. Autos pro Stunde). Stellen an denen sich die Kapazitäten ändern werden als Knoten modelliert. Dies können z.B. Kreuzungen oder Abzweigungen sein.

Frage 8.2: Wie ist ein Netzwerk formal definiert?

Ein Netzwerk ist ein gerichteter, asymmetrischer Graph $G = (V, E)$ mit zwei ausgezeichneten Knoten Q (Quelle) und S (Senke) sowie einer Kapazitätsfunktion $c : E \to \mathbb{N}_0$. Die Quelle Q zeichnet sich dadurch aus, dass keine Kanten in sie hinein führen, und die Senke S dadurch, dass keine keine Kanten aus ihr heraus führen.

Frage 8.3: Wie kann ein nicht asymmetrisches Netzwerk in ein asymmetrisches Netzwerk umgeformt werden?

Wenn ein Netzwerk nicht asymmetrisch ist, gibt es also mindestens eine Kante (u, v) mit $c((u, v)) = c$, für die auch eine Kante (v, u) mit $c((v, u)) = c'$ existiert. Damit dies nicht mehr der Fall ist, werde wir jede solche Kante wie folgt ändern. Es wird ein neuer Knoten w mit den Kanten (v, w) und (w, v) eingefügt und die alte Kante (v, u) gelöscht. Die beiden neuen Kanten bekommen beide die Kosten c' der gelöschten Kante (v, u). Abbildung 16 illustriert dieses Vorgehen.

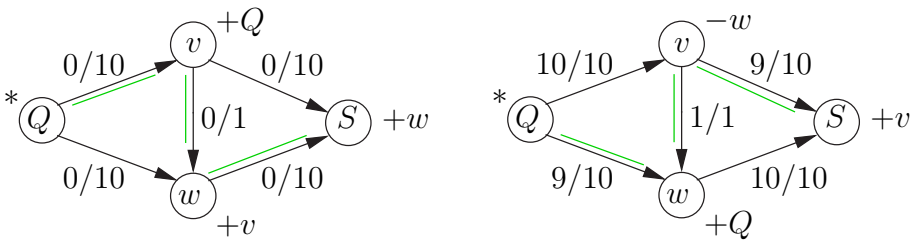
In den folgenden beschriebenen Algorithmen zur Flussberechnung wird sich eine solche Ersetzung in den meisten Fällen auf die Laufzeit auswirken, da die Anzahl der Knoten steigt.

Frage 8.4: Wie ist ein Fluss definiert?

Gegeben sei ein Netzwerk $N = (G = (V, E), Q, S, c)$. Eine Funktion $\varphi : E \to \mathbb{R}_0^+$ heißt Fluss auf N wenn für alle Knoten $v \in V - \{Q, S\}$ die Kirchhoff-Regel erfüllt ist:

$$\sum_{e=(\cdot,v)} \varphi(e) = \sum_{e=(v,\cdot)} \varphi(e)$$

Umgangssprachlich bedeutet die Kirchhoff-Regel also, dass alles, was in einem Knoten v an Fluss ankommt, auch weitergegeben wird.



(a) Flussvergrößernder Weg mit Vorwärtskanten

(b) Flussvergrößernder Weg mit Rückwärtskante

Abbildung 17: Beispiele für FV-Wege mit Vorwärts- und Rückwärtskanten

Frage 8.5: Wann heißt ein Fluss zulässig?
Ein Fluss φ auf einem Netzwerk N heißt zulässig, wenn er für alle Kanten die Kapazitätsschranken einhält, d.h.:

$$\forall e \in E: \ \varphi(e) \leq c(e)$$

Frage 8.6: Wie ist der Wert eines Flusses definiert?
Der Wert eines Flusses ist definiert als die Höhe des Flusses auf allen Kanten, die Q verlassen:

$$w(\varphi) = \sum_{e=(Q,\cdot)} \varphi(e)$$

Frage 8.7: Wann heißt ein Fluss maximal?
Ein Fluss heißt maximal, wenn er zulässig ist und für alle anderen zulässigen Flüsse φ' gilt: $w(\varphi') \leq w(\varphi)$.

Frage 8.8: Welche Probleme kennst du, die mit Hilfe eines Flussalgorithmus gelöst werden können?

8.2 Der Algorithmus von Ford und Fulkerson

Frage 8.9: Wie funktioniert der Algorithmus von Ford und Fulkerson?
Die Hauptidee des Algorithmus von Ford und Fulkerson ist es, so lange einen flussvergrößernden Weg im Netzwerk zu suchen und den Fluss entlang dieses Weges zu vergrößern, bis es keinen flussvergrößernden Weg im Netzwerk mehr gibt. Es wird also von einem Netzwerk ausgegangen, für das ein zulässiger Fluss gegeben ist. Ein flussvergrößernder Weg (im Folgenden FV-Weg) in diesem Netzwerk ist ein Weg w von der Quelle zur Senke, bei dem für jede Kante e auf diesem Weg gilt, dass der Fluss φ auf e die Kapazität nicht voll ausschöpft oder dass etwas von dem Fluss auf e weggenommen werden kann und auf andere Kanten verteilt werden kann, so dass mehr Fluss entsteht.

Der Algorithmus funktioniert nun wie folgt. Auf dem Netzwerk sei ein zulässiger Fluss φ gegeben. Zu Beginn ist dies der Nullfluss $\varphi \equiv 0$. Alle Knoten, bis auf die Quelle Q sind unmarkiert. Die anderen Knoten können mit den beiden folgenden Regeln markiert werden:

1. Der Knoten u sei markiert und der Knoten v sei unmarkiert. Gibt es eine Kante $e = (u, v) \in E$ mit $\varphi(e) < c(e)$, dann wird der Knoten v mit „$+u$“ markiert. Die Kante e wird Vorwärtskante.

2. Der Knoten u sei unmarkiert und der Knoten v sei markiert. Gibt es eine Kante $e = (u, v) \in E$ mit $\varphi(e) > 0$, dann wird der Knoten u mit „$-v$“ markiert. Die Kante e wird Rückwärtskante.

Eine Vorwärtskante bedeutet quasi, dass der Fluss unter Umständen erhöht werden kann, wenn mehr Fluss über diese Kante geschickt wird. Es ist jedoch auch möglich mehr Gesamtfluss zu bekommen, indem Fluss von einer Kante weggenommen wird und auf andere günstigere Kanten, die noch über Kapazität verfügen, verteilt wird. Dieser Fall tritt bei einer Rückwärtskante auf und kann sich anhand von Abbildung 17(b) verdeutlicht werden. Die Laufzeit dieses Algorithmus zur Ermittlung eines FV-Weges beträgt $O(|V| + |E|)$.

Frage 8.10: Wie kann mit dem Algorithmus von Ford und Fulkerson erkannt werden, dass ein Fluss nicht maximal ist? Wie kann ein größerer Fluss berechnet werden?
Wenn im Algorithmus von Ford und Fulkerson die Senke S markiert wird, ist der Fluss φ nicht maximal. Es kann dann in Zeit $O(|V|)$ ein größerer Fluss konstruiert werden. Zuerst zeigen wir, dass es einen Weg von der Quelle Q zur Senke S gibt, der aus Vorwärts- und Rückwärtskanten besteht. Die Senke S wurde markiert und allgemein gilt für einen markierten Knoten w, dass an ihm der Grund für seine Markierung vermerkt wurde. Dies sei entweder „$+u$“ oder „$-v$“. Wir springen nun, bei der Senke beginnend, von jedem Knoten zu dem Knoten, der für seine Markierung verantwortlich war. Es ist nun wichtig zu erkennen, dass jeder Knoten, der Grund für die Markierung eines anderen Knotens ist, vor diesem Knoten markiert worden sein muss. Da wir über die Zeit argumentieren können keine Kreise dabei entstehen. Und da der Knoten Q als erstes markiert wurde, wird dieser schließlich irgendwann erreicht. Es gibt also einen Weg $Q = a_0, a_1, \ldots, a_k = S$ bei dem jeder Knoten a_{i+1} entweder die Markierung „$+a_i$“ oder „$-a_i$“ trägt. Im Falle von „$+a_i$“ ist die Kante $e_i = (a_i, a_{i+1})$ Vorwärtskante, ansonsten ist $e_i = (a_{i+1}, a_i)$ Rückwärtskante.

Wie können wir den Fluss entlang dieses FV-Weges nun vergrößern? Wir berechnen einen Wert δ als das Minimum aller $\varphi(e_i)$ für alle Rückwärtskanten e_i und aller $c(e_i) - \varphi(e_i)$ für alle Vorwärtskanten e_i. Da für Rückwärtskanten $\varphi(e) > 0$ und für Vorwärtskanten $\varphi(e) < c(e)$ und damit $c(e) - \varphi(e) > 0$ gilt, gilt somit auch für $\delta > 0$. Wir ändern den Fluss φ dann wie folgt in einen größeren Fluss φ' ab, wobei $e_0, \ldots, e_{k-1}$ die Kanten des FV-Weges seien:

$$\begin{array}{ll} \varphi'(e) = \varphi(e) & \text{falls } e \notin \{e_0, \ldots, e_{k-1}\} \\ \varphi'(e) = \varphi(e) + \delta & \text{falls } e \in \{e_0, \ldots, e_{k-1}\} \text{ und } e \text{ Vorwärtskante} \\ \varphi'(e) = \varphi(e) - \delta & \text{falls } e \in \{e_0, \ldots, e_{k-1}\} \text{ und } e \text{ Rückwärtskante} \end{array}$$

Kanten, die nicht auf dem FV-Weg vorkommen, bleiben also unverändert. Über Vorwärtskanten wird der Fluss erhöht und über Rückwärtskanten erniedrigt. Aufgrund der Berechnung von δ gilt dabei für jede Kante $0 \leq \varphi'(e) \leq c(e)$. Durch die Veränderung des Flusses könnte an einem der Knoten die Kirchhoff-Regel verletzt worden sein. Da die Quelle und die Senke von der Kirchhoff-Regel ausgeschlossen sind, kann die Regel nur an einem der Knoten $a_1, \ldots, a_{k-1}$ auf dem FV-Weg verletzt worden sein. Der Knoten a_i wird von den Kanten e_{i-1} und e_i berührt. Diese können eine beliebige Kombination von Vorwärts- und Rückwärtskanten sein, also gibt es vier verschiedene Fälle:

Vor/Vor: $\underset{a_{i-1}}{\times} \xrightarrow{+\delta} \underset{a_i}{\times} \xrightarrow{+\delta} \underset{a_{i+1}}{\times}$. Der Fluss über a_i erhöht sich um δ.

Rück/Rück: $\underset{a_{i-1}}{\times} \xleftarrow{-\delta} \underset{a_i}{\times} \xleftarrow{-\delta} \underset{a_{i+1}}{\times}$. Der Fluss über a_i verringert sich um δ.

Vor/Rück: $\underset{a_{i-1}}{\times} \xrightarrow{+\delta} \underset{a_i}{\times} \xleftarrow{-\delta} \underset{a_{i+1}}{\times}$. Der Fluss über a_i bleibt gleich.

Rück/Vor: $\underset{a_{i-1}}{\times} \xleftarrow{-\delta} \underset{a_i}{\times} \xrightarrow{+\delta} \underset{a_{i+1}}{\times}$. Der Fluss über a_i bleibt gleich.

Wie man schnell sieht, bleibt die Kirchhoff-Regel in allen vier Fällen eingehalten. In den beiden ersten Fällen, da zwar mehr bzw. weniger beim Knoten ankommt, jedoch auch mehr bzw. weniger weitergegeben wird. In den beiden letzten Fällen kommt über eine Kante mehr Fluss bei a_i an und über die andere Kante weniger Fluss. Damit kann a_i immer noch genausoviel Fluss wie vorher an andere Knoten weitergeben.

Es ist leicht zu zeigen, dass sich bei dieser Vorgehensweise der Gesamtfluss erhöhen müss. Da Q als erstes markiert wird, kann eine Kante zu einem anderen Knoten, die noch nicht ausgelastet ist, nur eine Vorwärtskante werden. Somit ist die Kante (Q, a_1) auf dem FV-Weg eine Vorwärtskante, auf der der Fluss um δ erhöht wird. Somit gilt $w(\varphi') = w(\varphi) + \delta$. Da höchstens alle Knoten auf diesem Weg liegen können, beträgt die Rechenzeit $O(|V|)$.

Frage 8.11: Wann ist ein Fluss beim Algorithmus von Ford und Fulkerson maximal? Beweise dies!

Wenn die Senke S nicht markiert wird, ist der Fluss φ maximal. Wir unterteilen die Menge der Knoten nun in die Menge der markierten Knoten A und die Menge der Knoten $V - A$, die nicht markiert wurden. Die Senke wurde laut Voraussetzung nicht markiert, daher gilt $S \in V - A$. Da die Quelle immer markiert wird, gilt $Q \in A$. Wir überlegen uns nun, wann die Situation eintritt, dass der eine Knoten einer Kante markiert ist und der andere nicht. Dafür müssen wir lediglich wieder zwischen Vorwärts- und Rückwärtskanten unterscheiden:

Vorwärtskanten: $(x, y) \in E$ mit $x \in A$ und $y \notin A \Rightarrow \varphi((x, y)) = c((x, y))$.

Rückwärtskanten: $(x, y) \in E$ mit $x \notin A$ und $y \in A \Rightarrow \varphi((x, y)) = 0$.

Im ersten Fall wird also ein Knoten markiert, sein Nachbarknoten jedoch nicht, da auf der Kante zwischen den beiden keine freie Kapazität mehr existiert. Im zweiten Fall würde eigentlich eine Rückwärtskante entstehen, jedoch fließt nichts über diese Kante und daher bleibt der Knoten x unmarkiert.

Da mindestens $Q \in A$ und $S \in V - A$ gilt, muss alles, was von Q nach S fließt, auch von A nach $V - A$ fließen. Mit der ersten der beiden Bedingungen oben haben wir jedoch gezeigt, dass die Kanten von A nach $V - A$ ausgelastet sind. Es kann also nicht mehr Fluss von A in Richtung $V - A$ fließen. Es kann jedoch auch nichts mehr zwischen $V - A$ und A zurück fließen. Mit dieser Einteilung in die Mengen A und $V - A$ haben wir also einen Flaschenhals gefunden, der durch den Fluss φ maximal ausgelastet wird. Damit ist es also nicht möglich, noch mehr Fluss durch diesen Flaschenhals zu schicken, womit der momentane Fluss maximal sein muss.

Frage 8.12: Welche Laufzeit hat der Algorithmus von Ford und Fulkerson?

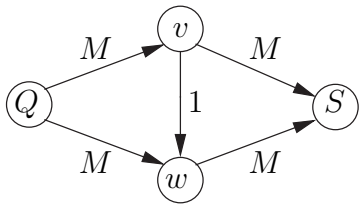


Abbildung 18: Ein Netzwerk bei dem eine exponentielle Laufzeit möglich ist

Bei Algorithmus von Ford und Fulkerson beginnen wir mit dem Nullfluss $\varphi \equiv 0$. Danach erzeugen wir nur ganzzahlige Flüsse, da $c(e)$ für jede Kante e ganzzahlig ist. Da wir auch schon gesehen haben, dass in jedem Schritt der Fluss um $\delta > 0$ erhöht wird, muss also $\delta \geq 1$ mit $\delta \in \mathbb{N}$ gelten. Der Fluss wird also in jedem Schritt um mindestens 1 erhöht. Der Wert eines Flusses ist über die Kanten definiert, die Q verlassen. Die Summe der Kapazitäten dieser Kanten ist auch eine obere Schranke für einen maximalen Fluss. Wir bezeichnen diese Summe der Kapazitäten nun mit $B = \sum_{e=(Q,\cdot)} c(e)$. Da B eine obere Schranke ist und der Fluss in jedem Schritt um mindestens 1 erhöht wird, können wir also höchstens B Schritte machen. Jeder Schritt kostet Zeit $O(|V| + |E|)$ bzw. $O(n + m)$, wenn $|V| = n$ und $|E| = m$ gesetzt wird. Damit können maximale Flüsse in Zeit $O((|V| + |E|) \cdot B)$ berechnet werden.

Wir zeigen nun, dass diese Laufzeit pseudopolynomiell ist. Dazu überlegen wir uns zuerst, dass höchstens $n - 1$ Kanten den Knoten Q verlassen können, wenn es $|V| = n$ Knoten gibt. Unter all diesen Kanten gibt es eine Kante, deren Kapazität maximal ist. Diese Kapazität nennen wir c_{max}. Die Summe aller Kapazitäten ist also durch $B \leq (n-1) \cdot c_{max}$ beschränkt. Wir ersetzen B nun in der oben ermittelten Laufzeit durch diese Abschätzung und erhalten eine Laufzeit von $O((n + |E|) \cdot (n-1) \cdot c_{max}) = O(n(n + |E|)c_{max})$. Damit ist die Laufzeit nun von der Größe einer Zahl in der Eingabe abhängig und somit pseudopolynomiell.

Frage 8.13: Gebe ein Beispiel an, für welches der Algorithmus von Ford und Fulkerson eine exponentielle Laufzeit aufweist!

Abbildung 18 zeigt ein Netzwerk bei dem Algorithmus von Ford und Fulkerson eine exponentielle Laufzeit aufweisen kann. Wenn abwechseln die Wege (Q, u, v, S) und (Q, v, u, S) als FV-Wege gewählt werden, sind $2M$ Flussvergrößerungen nötig, bis ein maximaler Fluss von $2M$ erreicht ist. Die Laufzeit beträgt also $\Theta(M)$, was exponentiell in der Eingabelänge $\Theta(\log M)$ ist.

Frage 8.14: Wie ist ein Restgraph definiert?

Bei den bisherigen Betrachtungen musste zwischen Vorwärts- und Rückwärtskanten unterschieden werden. Damit wir dies nicht mehr machen müssen, definieren wir den sogenannten Restgraph $Rest_\varphi = (V, E_\varphi)$ für ein gegebenes Netzwerk $N = (G = (V, E), Q, S, c)$ und einen Fluss φ wie folgt:

1. Für jede Kante $e \in E$ mit $\varphi(e) < c(e)$ enthält E_φ die Kante e mit einer Kapazität von $r_\varphi(e) = c(e) - \varphi(e)$.
2. Für jede Kante $e \in E$ mit $\varphi(e) > 0$ enthält E_φ die Kante $e' = rev(e)$ mit einer Kapazität von $r_\varphi(e') = \varphi(e)$.

Dabei gilt $rev((u, v)) = (v, u)$. Die Funktion rev dreht die Kante also einfach um. Wie man sieht, können für eine Kante im Netzwerk beide Fälle eintreten, daher können nun auch

zwei Kanten zwischen den Knoten verlaufen. Aus diesem Grund wurde zu Beginn auch die Einschränkung gemacht, dass das Netzwerk asymmetrisch sein soll, denn ansonsten würde ein Multigraph entstehen.

Im Prinzip enthält der Restgraph alle Informationen, die zur Berechnung eines FV-Weges benötigt werden. Vorwärtskanten sind so dargestellt, dass sofort zu sehen ist, wieviel Fluss eine Kante noch zusätzlich transportieren kann und Rückwärtskanten verlaufen nun nicht mehr rückwärts, sondern entlang des Weges, an dem der Fluss geändert werden kann. Damit ist jeder FV-Weg nun als ein gerichteter Weg im Restgraph gegeben.

Mit dem Restgraph ist es sogar möglich, den Fluss zu vergrößern, indem auf dem Restgraph ein Fluss berechnet wird und dieser zum bestehenden Fluss hinzu addiert wird. Dabei muss natürlich Fluss entlang von Kanten, die aus Rückwärtskanten entstanden sind, subtrahiert werden.

8.3 Der Algorithmus von Edmonds und Karp

Frage 8.15: Wie ist der Algorithmus von Edmonds und Karp motiviert? Was verbessert er?

In Abbildung 18 haben wir ein Beispiel gesehen, welches eine exponentielle Laufzeit aufweisen kann. Dies muss jedoch nicht sein. Es ist auch möglich, auf dem gegebenen Netzwerk in zwei Schritten einen maximalen Fluss zu berechnen. Dazu muss z.B. im ersten Schritt der FV-Weg (Q, u, S) und im zweiten Schritt der FV-Weg (Q, v, S) gewählt werden. Edmonds und Karp haben nun gezeigt, dass es möglich ist, systematisch gute FV-Wege zu finden, so dass die Worst-Case Laufzeit stets polynomiell beschränkt ist. Dabei werden stets die kürzesten FV-Wege gewählt, um den Fluss zu erhöhen. Es lässt sich zeigen, dass bei diesem Vorgehen höchstens $|E|(n+1)$ Flussvergrößerungen nötig sind.

Frage 8.16: Was ist unter der Vorgehensweise FVMIN zu verstehen?

Bei der Vorgehensweise FVMIN wird stets der kürzeste FV-Weg gewählt und entlang diesem der Fluss erhöht. Es wird dabei gehofft, dass damit schnell alle kürzesten FV-Wege zerstört und nicht wieder erzeugt werden.

Frage 8.17: Was ist ein Flaschenhals?

Ein Flaschenhals ist eine Kante auf einem FV-Weg, die den Wert δ der Flussvergrößerung definiert. Dies ist also eine der Kanten, an denen der minimale Wert $\varphi(e)$ bzw. $c(e) - \varphi(e)$ anliegt.

Frage 8.18: Beschreibe kurz die Notation, die im Folgenden benutzt wird!

Im Folgenden wird mit $q^{(k)}(v)$ der kürzeste Weg von Q zu einem Knoten v im Restgraphen, der nach k Flussvergrößerungen entstanden ist, bezeichnet. Dieser Restgraph wird mit $Rest_{\varphi_k}$ bezeichnet, wobei φ_k der Fluss nach k Schritten ist. Mit φ_0 bezeichnen wir den Nullfluss $\varphi \equiv 0$.

Frage 8.19: Zeige, dass die Wege von Q zu einem Knoten v bei der Anwendung von FVMIN nicht verkürzen können!

Gegeben sei also ein Knoten v mit dessen Entfernung von Q nach k Schritten $q^{(k)}(v)$ betrage. Zu zeigen ist also:

$$q^{(k+1)}(v) \geq q^{(k)}(v)$$

Wir betrachten den Restgraph $Rest_{\varphi_k}$ in dem gemäß FVMIN ein FV-Weg ausgewählt wurde. Dieser besteht aus den Knoten $Q = v_0, v_1, \ldots, v_r = S$ und den entsprechenden Kanten $e_1, \ldots, e_r$. Im folgenden Schritt ist es wichtig, sich klar zu machen, dass in einem Restgraph *alle* Wege von Q nach S, die keinen Knoten doppelt benutzen, FV-Wege sind. Da der kürzeste FV-Weg im Restgraph gewählt wurde, gilt $q^{(k)}(v_i) = i$ für jeden Knoten v_i auf diesem Weg. Größer als i kann die Entfernung nämlich nicht werden, da wir schon einen Weg mit der Entfernung i für jeden Knoten v_i gefunden haben. Kleiner als i kann der Wert auch nicht werden, da der betrachtete kürzeste FV-Weg dann von vornherein nicht der kürzeste FV-Weg gewesen wäre. Als nächstes überlegen wir uns nun, wie sich der Restgraph verändern kann, wenn der Fluss auf den Kanten e_i des FV-Weges verändert wird. Steht die Kante e_i für eine Vorwärtskante, so kann diese wegfallen, wenn der Fluss über diese Kante maximal ist (da in diesem Fall $c(e_i) - \varphi(e_i) = 0$ gilt). Es kann auch eine neue Kante $rev(e_i)$ entstehen, wenn vorher noch nichts über diese Kante geflossen ist. Steht die betrachtete Kante e_i für eine Rückwärtskante, so kann diese ebenfalls wegfallen. Nämlich dann, wenn sie Flaschenhals ist und der Fluss so geändert wird, dass nichts mehr über sie fließt. Und auch hier kann eine neue Kante $rev(e_i)$ entstehen. Dies passiert, wenn die Kapazität der Kante voll ausgenutzt war und nun Fluss weggenommen wird. In beiden Fallen kann also eine Kante $v_{i-1} \to v_i$ wegfallen bzw. eine neue Kante $v_i \to v_{i-1}$ entstehen. Durch denn Wegfall von Kanten können die kürzesten Wege nicht kürzer werden. Es lässt sich auch zeigen, dass sich beim Hinzufügen von Kanten, wie es hier geschieht, nichts an den Entfernungen ändert.

Wir können uns also abschließend vorstellen, dass zuerst alle Kanten $rev(e_i)$ eingefügt werden, was an den Distanzen nichts ändert, und anschließend einige Kanten e_i entfernt werden, wodurch die Distanzen höchstens größer werden können. Damit ist dann die Behauptung gezeigt.

Frage 8.20: Zeige, dass das Einfügen einer Kante von einem Knoten mit einer größeren Entfernung zu Q zu einem Knoten mit einer kleineren Entfernung nichts an den Entfernungen aller Knoten ändert!

Wie wir im Beweis zuvor gesehen haben, werden auf dem FV-Weg unter Umständen Kanten $v_i \to v_{i-1}$ eingefügt, wobei für die Entfernungen $q^{(k)}(v_i) = i$ gilt. Um zu zeigen, dass sich an den Entfernungen aller Knoten nichts ändert, werden wir sogar noch etwas mehr zeigen. Allgemein gilt, dass das Einfügen einer Kante (v, w) nichts an den Distanzen ändert, wenn $d(v) \geq d(w) - 1$ gilt. Der Beweis arbeitet in drei Schritten. Zuerst wird gezeigt, dass sich für v nichts ändert. Daraus wird folgen, dass sich für w nichts ändert, woraus wiederum folgen wird, dass sich für die restlichen Knoten ebenfalls nichts ändert. Es sei nun G' der Graph, der aus G entsteht, wenn die Kante (v, w) eingefügt wird. Mit d' bezeichnen wir die Distanzen im Graph G'.

Schritt 1: Für v ändert sich durch das Einfügen von (v, w) nichts. Dies liegt daran, dass kein kürzester Weg zu v in G' die Kante (v, w) benutzen wird, da wir in diesem Fall schon beim Knoten v angelangt wären und der Weg nur länger als nötig werden würde. Es gilt also $d'(v) = d(v)$, was wir in Schritt 2 benötigen werden.

Schritt 2: Wir zeigen, dass sich auch für w nichts an den Distanzen ändert. Wenn wir die Kante (v, w) für einen kürzesten Weg zu w in G' benutzen, sollten wir vorher den kürzesten Weg in G' bis zu v benutzen. Dieser hat die Länge $d'(v)$. Von v müssen wir noch einen Schritt über (v, w) bis w machen. Also hätte ein solcher Weg die Länge

$d'(v) + 1$. Mit der Erkenntnis aus Schritt 1 folgt dann jedoch:

$$d'(w) = d'(v) + 1 = d(v) + 1 \geq d(w) - 1 + 1 = d(w)$$

Insgesamt gilt also $d'(w) \geq d(w)$, was bedeutet, dass die Distanzen bei der Benutzung von (v, w) höchstens länger werden können.

Schritt 3: Wir zeigen nun, dass sich die Distanzen für die restlichen Knoten ebenfalls nicht ändern. Dazu betrachten wir nun einen Knoten x, für den $x \neq v$ und $x \neq w$ gelte. Für diesen Knoten gibt es in G einen kürzesten Weg, der die Kante (v, w) nicht benutzt, da wir uns im Graph G befinden. Angenommen in G' würde nun die Kante (v, w) benutzt werden und wir erhalten einen kürzesten Weg der Länge l. Dieser Weg enthält dann also einen Teilweg bis w. Wir haben jedoch schon in Schritt 2 gezeigt, dass ein Weg bis w nicht kürzer werden kann, wenn wir (v, w) benutzen. Also können wir auch den kürzesten Weg bis w in G benutzen und erhalten ebenfalls einen Weg, dessen Länge jedoch höchstens l ist.

Frage 8.21: Zeige, dass die Laufzeit polynomiell beschränkt ist, wenn stets kürzeste FV-Wege benutzt werden!

Wir werden nun zeigen, dass bei der Benutzung von FVMIN spätestens nach $|E|(n + 1)$ Flussvergrößerungen ein maximaler Fluss berechnet ist. Dabei werden wir uns überlegen, wie oft eine Kante Flaschenhals sein kann. Dazu werden wir nun zuerst zeigen, dass die Summe der Distanzen zweier Knoten um mindestens vier wächst, wenn die Kante, die zwischen diesen Knoten verläuft zweimal Flaschenhals ist. Gegeben sei also eine Kante (v, w), die bei der $(k + 1)$-ten und der $(m + 1)$-ten Flussvergrößerung Flaschenhals ist. Wir wollen zeigen:

$$q^{(m)}(v) + q^{(m)}(w) \geq q^{(k)}(v) + q^{(k)}(w) + 4$$

Damit die Kante (v, w), nachdem sie schon einmal Flaschenhals war, wieder ein Flaschenhals werden kann, muss zwischendurch im $(l + 1)$-ten Schritt (mit $k < l < m$) ein Rückfluss stattfinden, d.h. es muss einen kürzesten FV-Weg gegeben haben, auf dem die Kante (w, v) vorkommt. Zu diesem Zeitpunkt gilt für die Distanzen der Knoten v und w zum Knoten Q: $q^{(k)}(v) = q^{(k)}(w) + 1$. Der Knoten v ist also einen Schritt weiter von Q entfernt als w. Da die Distanzen zum Knoten Q für jeden Knoten, der auf einem FV-Weg vorkommt nur größer werden können, schätzen wir nun wie folgt ab:

$$q^{(m)}(v) \geq q^{(l)}(v) = q^{(l)}(w) + 1 \geq q^{(k)}(w) + 1 = q^{(k)}(v) + 1 + 1 = q^{(k)}(v) + 2$$

Eine ähnliche Abschätzung können wir für $q^{(m)}(w)$ vornehmen:

$$q^{(m)}(w) = q^{(m)}(v) + 1 \geq q^{(l)}(v) + 1 = q^{(l)}(w) + 1 + 1 \geq q^{(k)}(w) + 2$$

Insgesamt gilt also, dass die Distanzen der Knoten v und w jeweils um mindestens 2 wachsen, wenn die Kante (v, w) zweimal Flaschenhals ist. Daraus folgt die Behauptung.

Im nächsten Schritt überlegen wir uns nun, wie oft eine Kante (v, w) ein Flaschenhals sein kann. Ist die Kante Flaschenhals auf einem kürzesten FV-Weg, gilt nämlich dass v höchstens die Distanz $n - 2$ zu Q haben kann und w höchstens die Distanz $n - 1$. Wir schätzen die

maximalen Distanzen im Folgenden jedoch mit $n-1$ bzw. n ab. Wenn eine Kante (v, w) also bei der $(m+1)$-ten Flussvergrößerung zum i-ten Mal Flaschenhals ist, dann gilt:

$$q^{(m)}(v) + q^{(m)}(w) \geq 1 + 4(i-1)$$

Wir haben etwas weiter oben auch schon beobachtet, wie groß $q^{(m)}(v)$ und $q^{(m)}(w)$ maximal werden können. Es gilt also:

$$\begin{aligned} 1 + 4(i-1) &\leq n - 1 + n \\ 4(i-1) &\leq 2n - 2 \\ i - 1 &\leq \frac{n-1}{2} \\ i &\leq \frac{n-1}{2} + 1 \\ i &\leq \frac{n+1}{2} \end{aligned}$$

Jede Kante im Restgraph kann also höchstens $\frac{n+1}{2}$-mal ein Flaschenhals sein. Im ursprünglichen Graph haben wir $|E|$ Kanten. Wenn auf diesem Graph gearbeitet wird, kann im resultierenden Restgraph für jede Kante (v, w) unter Umständen eine Kante (w, v) entstehen. Also hat der Restgraph höchstens $2|E|$ Kanten, wovon jede höchstens $\frac{n+1}{2}$-mal ein Flaschenhals sein kann. Da jede Flussvergrößerung einen Flaschenhals hat, ergibt dies höchstens

$$2|E| \cdot \frac{n+1}{2} = |E|(n+1)$$

Flussvergrößerungen, womit die Behauptung gezeigt ist und die Laufzeit polynomiell ist.

Frage 8.22: Wie kann ein kürzester FV-Weg berechnet werden? Welche Laufzeit hat der Algorithmus?

Um einen kürzesten Weg zu konstruieren, kann ein BFS-Ansatz, also eine Breitensuche, benutzt werden. Es ist wichtig, sich klar zu machen, dass auf dem Restgraph gearbeitet wird, in dem jeder Weg von Q zu S, der keinen Knoten mehrfach benutzt, ein FV-Weg ist. In Algorithmus 9 wird sich für jeden Knoten mit Hilfe der Markierung gemerkt, ob er schon auf einem potentiellen kürzesten FV-Weg vorkommt oder ob er zum ersten Mal erreicht wird. Zusätzlich wird sich in $d(v)$ gemerkt, wie weit v vom der Quelle Q entfernt ist.

Algorithmus 9 BFS-Markierungsalgorithmus auf dem Restgraphen

Markiere die Quelle Q und füge den Knoten Q in eine leere Queue ein. Setze $d(Q) = 0$.
while queue not empty **do**
  Entferne das oberste Element aus der Queue, nenne es v.
  for jede Kante (v, w) im Restgraphen, bei der w nicht markiert ist **do**
    Markiere w und füge w in die Queue ein.
    Setze $d(w) = d(v) + 1$. Vermerke an w, dass w von v aus erreicht wurde.
  end for
end while

Die Laufzeit des Algorithmus beträgt $O(n + |E|)$, da er eine Breitensuche implementiert, welche jeden Knoten und jede Kante einmal anschaut. Wir haben oben schon herausgefunden, wie oft wir höchstens einen kürzesten FV-Weg berechnen müssen: $|E|(n+1)$-mal, also

$O(|E|n)$. Die Breitensuche wird im Prinzip durch die Anzahl der Kanten dominiert, da ein Graph mit n Knoten maximal $n(n-1) = n^2 - n = O(n^2)$ viele Kanten haben kann. Die Breitensuche benötigt also Laufzeit $O(|E|)$ und insgesamt ergibt sich: $O(|E|n \cdot |E|) = O(n|E|^2)$. Wir schätzen $O(|E|)$ durch $O(n^2)$ ab und erhalten als Endergebnis $O(n^5)$.

8.4 Der Algorithmus von Dinic

Frage 8.23: Welche Verbesserung bringt der Algorithmus von Dinic?

Der erste Algorithmus zur Berechnung von maximalen Flüssen, der in diesem Kapitel betrachtet wurde, hat im worst-case eine pseudopolynomielle Laufzeit. Dies liegt daran, dass er relativ unstrukturiert vorgeht. Mit etwas mehr Struktur ist es jedoch leicht möglich, die Laufzeit polynomiell werden zu lassen. Edmonds und Karp haben gezeigt, dass dies möglich ist, wenn nicht irgendein FV-Weg benutzt wird, sondern der kürzeste FV-Weg. Diese Vorgehen führte zu einer Laufzeit von $O(n^5)$. Dinic hat nun gezeigt, dass diese weiter auf $O(n^4)$ gedrückt werden kann. Der Algorithmus von Edmonds und Karp sucht nämlich in jedem Schritt nur *einen* kürzesten FV-Weg und verbessert den Fluss entlang diesem. Danach wird im geänderten Restgraph erneut der BFS-Algorithmus ausgeführt, welcher einen anderen FV-Weg findet, der unter Umständen jedoch genauso lang ist wie der zuvor gefundene Weg. Dies bedeutet also, dass es in einem Netzwerk nicht nur einen kürzesten FV-Weg geben muss. Der Algorithmus von Dinic versucht nun, in einem Schritt alle kürzesten FV-Wege zu betrachten und den Fluss entlang dieser zu verbessern. Die Frage ist nun, wie man alle kürzesten Wege in einem Netzwerk finden kann. Wie wir gleich sehen werden, muss das Netzwerk dazu lediglich in ein Niveau-Netzwerk umgewandelt werden.

Frage 8.24: Wie ist ein Niveau-Netzwerk definiert? Wie ist es motiviert?

Niveau-Netzwerke werden für einen gegebenen gültigen Fluss φ auf dem Restgraphen $Rest_\varphi$ berechnet. Wie wir in Kapitel 8.3 schon gesehen haben, ist es mit Hilfe eines BFS-Ansatzes möglich, die Länge eines kürzesten Weges $d(v)$ von der Quelle Q zu einem Knoten v zu bestimmen. Ein kürzester FV-Weg ist ein Weg von der Quelle Q zur Senke S und hat dementsprechend die Länge $d(S)$. Dies bedeutet schonmal, dass wir Knoten $v \neq S$, die mindestens $d(S)$ von der Quelle entfernt sind, gar nicht mehr betrachten müssen, da sie nicht an einem kürzesten FV-Weg beteiligt sein können. Wir haben in den Kapiteln zuvor auch schon beobachtet, dass die Differenz der Entfernungen von zwei im FV-Weg benachbarten Knoten genau 1 betragen muss. Formal bedeutet dies, dass für einen FV-Weg $Q = v_0, \ldots, v_n = S$ für zwei Knoten v_i und v_{i+1} mit $0 \leq i \leq n-1$ jeweils $d(v_{i+1}) = d(v_i) + 1$ gilt. Wenn dies für zwei Knoten v_i und v_{i+1} nicht gelten würde, könnten zwei Fälle unterschieden werden:

1. Wenn $d(v_{i+1}) \leq d(v_i)$ gilt, könnte ein kürzerer FV-Weg konstruiert werden, indem direkt von der Quelle zu v_{i+1} gelaufen wird.

2. Gilt $d(v_{i+1}) \geq d(v_i) + 2$ müssten die Entfernungen falsch berechnet worden sein. Denn durch Benutzung der Kante (v_i, v_{i+1}) würden wir für den Knoten v_{i+1} die Entfernung $d(v_i) + 1$ bekommen.

In Niveau-Netzwerken werden wir nun genau diese „Abstandseigenschaft“ für kürzeste Wege berücksichtigen. Wir teilen daher zuerst einmal die Knoten entsprechend ihrer Distanzen bzw.

ihrer „Niveaus“ zur Quelle Q ein. Wir benutzen dazu die Mengen $V_0, \ldots, V_{d(S)}$, die wie folgt definiert sind:

$$V_i = \{v \in V \mid d(v) = i\} \text{ für } 0 \leq i \leq d(S) - 1$$

Die Menge $V_{d(S)}$ besteht nur aus der Senke, da wir ja alle Knoten $v \neq S$, deren Distanz mindestens $d(S)$ beträgt, ausschließen wollen. Also gilt $V_{d(S)} = \{S\}$. Die Knotenmenge V_φ des Niveau-Netzwerks ist die Vereinigung aller V_i, wobei V_0 nur die Quelle Q enthält.

Wir haben nun etwas weiter oben schon gesehen, dass die Distanz zu Q auf einem kürzesten FV-Weg von Knoten zu Knoten jeweils um 1 wächst. Dies bedeutet jedoch nichts anderes, als dass für kürzeste FV-Wege nur Kanten in Frage kommen, die zwischen benachbarten Niveaus verlaufen. Also definieren wir die Kantenmenge E_φ des Niveau-Netzwerks als Vereinigung der Mengen $E_1, \ldots, E_{d(S)}$, wobei E_i wie folgt definiert ist:

$$E_i = \{(v, w) \in V_{i-1} \times V_i \mid (v, w) \in Rest_\varphi\}$$

Da wir das Niveau-Netzwerk benutzen werden, um Flüsse auf diesem zu berechnen, benötigen wir natürlich auch noch die Kapazitätsinformation aus dem Restgraph. Diese wird einfach übernommen, d.h. auch im Niveau-Netzwerk wird mit $r_\varphi(e)$ die Kapazität der Kante e zum Fluss φ bezeichnet.

Frage 8.25: Wie kann ein Niveau-Netzwerk zu einem Restgraph berechnet werden? Welche Laufzeit hat dieser Algorithmus?

Um ein Niveau-Netzwerk zu einem Restgraph zu berechnen, muss lediglich der BFS-Algorithmus von Seite 91 modifiziert werden. Wir überlegen uns nun zuerst, was wir alles für das Niveau-Netzwerk berechnen müssen. Wir müssen die Knoten entsprechend ihrer berechneten Distanzwerte in die verschiedenen Mengen V_i einordnen und wollen nur Kanten im Niveau-Netzwerk haben, die zwischen benachbarten Niveaus verlaufen.

Die Distanzberechnung wird genau wie in Algorithmus 9 vorgenommen. Wird dabei für einen Knoten v die Distanz $d(v) = i$ berechnet, so fügen wir den Knoten in $O(1)$ zur Menge V_i hinzu. Jeder Knoten, der zum ersten Mal erreicht wird, wird markiert.

Wir müssen uns nur noch überlegen, wie wir die für das Niveau-Netzwerk relevanten Kanten ermitteln. Wenn wir den Knoten w vom Knoten v aus erreichen, betrachten wir also die Kante (v, w). Handelt es sich beim Knoten w um einen neu entdeckten, unmarkierten Knoten, wird die Kante (v, w) in das Niveau-Netzwerk übernommen. Denn in diesem Fall gilt immer $d(w) = d(v) + 1$, da die Distanzwerte eines neu entdeckten Knotens eben auf diese Art und Weise im Algorithmus vergeben werden. Treffen wir auf einen markierten Knoten w, so müssen wir schauen, ob für den Knoten v, von dem aus wir ihn erreicht haben gilt: $d(v) = d(w) - 1$ (bzw. $d(w) = d(v) + 1$). Ist dies der Fall, wird die Kante in die Menge $E_{d(w)}$ eingeordnet und in $O(1)$ in die Adjazenzliste des Knotens v im Niveau-Netzwerk gepackt.

Wie man schnell sieht, verändert sich bei diesem Vorgehen die Laufzeit des BFS-Algorithmus nicht, so dass wir auch hier eine Laufzeit von $O(|V| + |E|)$ bzw. $O(n + |E|)$ bekommen.

Frage 8.26: Wann heißt eine Kante saturiert? Was ist ein Sperrfluss?

Eine Kante e heißt bezüglich eines Flusses ψ saturiert, wenn $\psi(e) = r_\varphi(e)$ gilt. Wir betrachten also einen Fluss ψ im Niveau-Netzwerk und bezüglich diesem wird die Kante e voll auslastet.

Ein Fluss ψ heißt dabei Sperrfluss, wenn auf allen Q-S-Wegen im einem Niveau-Netzwerk eine saturierte Kante liegt.

Frage 8.27: Wie lautet die grobe Beweisidee beim Algorithmus von Dinic?

Algorithmus 10 Konstruktion eines Sperrflusses ψ für ein Niveau-Netzwerk

1) $\psi(e) = 0$ für alle Kanten e im Niveau-Netzwerk.
2) Der aktuelle Weg ist der Weg der Länge 0 von Q nach Q, d.h. $v_i = Q$ für $i = 0$.
3) Falls $v_i = S$ ist, gehe zu Schritt 4. Ansonsten schaue, ob es in der Adjazenzliste von v_i einen Knoten $w \in V_{i+1}$ gibt.

Ja: Verlänger den Pfad um diese Kante (v_i, w), d.h. setze $i = i + 1$ und $v_i = w$. Gehe zu Schritt 3.

Nein: Es gibt im Niveau-Netzwerk keinen Weg von v_i zu S. Entferne v_i und alle zu ihm adjazenten Kanten aus dem Niveau-Netzwerk. Ist $i = 0$ stoppe, ansonsten gehe einen Schritt zurück, indem $i = i - 1$ gesetzt und zu Schritt 3 gegangen wird.

4) Es wurde ein Weg W von Q nach S gefunden. Berechne $\delta = \min\{r_\varphi(e) - \psi(e) \mid e \in W\}$ und ersetze auf allen Kanten $e \in W$ den Wert des Sperrflusses ψ durch $\psi(e) = \psi(e) + \delta$. Entferne alle saturierten Kanten und gehe zu Schritt 2.

Beim Algorithmus von Dinic wird in jedem Schritt das zum Restgraph gehörige Niveau-Netzwerk berechnet und auf diesem effizient ein Sperrfluss ermittelt. Anschließend wird der Sperrfluss auf den schon vorhandenen Fluss addiert (bzw. für Rückwärtskanten abgezogen) und das Verfahren wiederholt, bis die Senke im Restgraph nicht mehr erreicht werden kann. Es wird sich zeigen, dass nach Addition des Sperrflusses die Länge der kürzesten Wege im Restgraph um mindestens 1 gewachsen ist. Da ein kürzester Weg in einem Graphen mit n Knoten jedoch höchstens die Länge $n - 1$ haben kann, ist dies eine obere Schranke für die Anzahl der Schritte, die gemacht werden.

Frage 8.28: Gebe ein Beispiel an, welches zeigt, dass ein Sperrfluss nicht notwendigerweise ein maximaler Fluss für das Niveau-Netzwerk sein muss!

Frage 8.29: Gebe einen Algorithmus an, um einen Sperrfluss zu berechnen! Welche Laufzeit hat er?

Es wird nun zuerst ein Algorithmus vorgestellt, der die Methode des Backtracking benutzt und eine Laufzeit von $O(n|E|) = O(n^3)$ aufweist. Im anschließenden Kapitel wird ein effizienterer Algorithmus mit einer Laufzeit von $O(n^2)$ vorgestellt.

Gegeben ist das Niveau-Netzwerk und ein Fluss ψ, welcher zu Beginn der Nullfluss ($\psi \equiv 0$) ist. Zu jedem Zeitpunkt betrachtet der Algorithmus einen Weg $Q = v_0 \to v_1 \to \cdots \to v_i$ und versucht, wenn $v_i \neq S$ gilt, diesen zu einem Weg zur Senke auszubauen. Wird dabei für einen Knoten festgestellt, dass er nicht zu einem Weg zur Senke ausgebaut werden kann, so wird dieser Knoten und alle zu ihm adjazenten Kanten entfernt. Wird ein Q-S-Weg gefunden, so wird entlang diesem der Fluss entsprechend angepasst, wodurch mindestens eine Kante saturiert wird. Da nicht noch mehr Fluss über diese Kante geleitet werden kann, braucht diese in zukünftigen Schritten nicht mehr betrachtet werden und wird ebenfalls entfernt.

Algorithmus 10 beschreibt das Vorgehen im Detail. Die beiden ersten Schritte dienen im Prinzip nur zur Initialisierung und werden auch nur einmal ausgeführt. In Schritt 3 findet das eigentliche Backtracking statt und Schritt 4 beschreibt, wie vorgegangen wird, wenn ein Q-S-Weg im Niveau-Netzwerk gefunden wurde.

Algorithmus 11 Berechnung eines maximalen Flusses mit dem Algorithmus von Dinic

1) Setze $\varphi(e) = 0$ für alle $e \in E$.
2) Berechne das zum Fluss φ gehörige Niveaunetzwerk. Wird dabei festgestellt, dass die Senke nicht mehr erreicht werden kann, ist der Fluss φ maximal und der Algorithmus terminiert.
3) Berechne einen Sperrfluss ψ für das Niveau-Netzwerk.
4) Addieren den Sperrfluss ψ zum Fluss φ unter Berücksichtigung der Richtung der Kanten. Starte mit dem neuen Fluss φ in Schritt 2.

Die Laufzeit des Algorithmus beträgt $O(n|E|)$, wobei über das Wegfallen von Kanten argumentiert wird. In jedem Schritt versucht der Algorithmus einen Weg von Q zu S zu bilden. Ein solcher Weg kann aus höchstens $n-1$ Kanten bestehen. Spätestens nach der Betrachtung von $n-1$ Kanten, haben wir also entweder einen Weg gefunden oder merken, dass wir keinen Weg finden. In beiden Fällen fällt mindestens eine Kante weg und ein neuer Versuch wird gestartet. Da das Niveau-Netzwerk aus höchstens $|E|$ Kanten besteht, werden also höchstens $|E|$ Versuche unternommen, was eine Gesamtlaufzeit von $O(n|E|)$ ergibt. Das Wegfallen mindestens einer Kante in jedem Schritt garantiert auch, dass der Algorithmus terminiert.

Frage 8.30: Beschreibe den Algorithmus von Dinic algorithmisch! Welche Laufzeit hat der Algorithmus?

Der Algorithmus von Dinic kann in vier Phasen unterteilt werden, die Algorithmus 11 beschreibt. Zur Berechnung der Laufzeit analysieren wir die Laufzeit der einzelnen Schritte und überlegen uns anschließend, wie oft die Schritte durchlaufen werden können. Schritt 1 benötigt Zeit $O(|E|) = O(n^2)$. Schritt 2 kann, wie wir schon gesehen haben, in Zeit $O(n + |E|) = O(n + n^2) = O(n^2)$ ausgeführt werden. Die Berechnung des Sperrflusses in Schritt 3 erfordert Zeit $O(n|E|) = O(n^3)$ und Schritt 4 braucht $O(|E|) = O(n^2)$. Die Berechnung des Sperrflusses dominiert mit $O(n^3)$ also die Laufzeit.

Wir werden nun zeigen, dass diese Schritte nur höchstens n-mal durchlaufen werden können, woraus sich dann eine Gesamtlaufzeit von $O(n^3) \cdot O(n) = O(n^4)$ ergibt. Dazu reicht es zu zeigen, dass die Länge der kürzesten Wege nach der Addition des Sperrflusses um mindestens 1 wächst. Wir betrachten dazu den Restgraphen und wie er sich in einer Phase von Algorithmus 11 verändert. Zum Restgraph $Rest_\varphi$ mit Fluss φ wird ein Sperrfluss ψ für das zu $Rest_\varphi$ gehörige Niveau-Netzwerk berechnet und zu φ addiert, wodurch ein neuer Fluss φ' entsteht. Dadurch entsteht auch ein neuer Restgraph $Rest_{\varphi'}$. Wir überlegen uns nun, inwiefern sich $Rest_{\varphi'}$ von $Rest_\varphi$ unterscheidet und was dies für die Länge von kürzesten Wegen in $Rest_{\varphi'}$ bedeutet. Die Länge der kürzesten Wege in $Rest_\varphi$ sei dabei l. Zuerst überlegen wir uns, was für die Kanten im Restgraph $Rest_\varphi$ gilt. Kanten, die von einem Knoten x starten, der die Entfernung x von der Quelle hat, führen zu Knoten, die höchstens die Entfernung $x+1$ zur Quelle haben (ansonsten kann irgendwas mit den Entfernungen nicht stimmen). Im Prinzip bedeutet dieses Aussage, dass auf einem Weg der Länge $l = d(S)$ mindestens l Knoten bzw. $l-1$ Kanten benutzt werden müssen. Wenn nur diese „alten“ Kanten aus $Rest_\varphi$ benutzt werden, beträgt die Länge der kürzesten Weg also l. Im neuen Restgraph $Rest_{\varphi'}$ kommen jedoch auch einige neue Kanten hinzu bzw. einige Kanten fallen weg. Wir zeigen nun, dass uns die neuen Kanten „nichts bringen“ und dass wegfallende Kanten die Situation auch nur verschlechtern:

Neue Kanten: Neue Kanten entstehen, wenn über eine noch nicht benutzte Kante ein Fluss

geleitet wird oder wenn von einer voll ausgelasteten Kante Fluss weggenommen wird. Wir überlegen uns nun, zwischen welchen Knoten diese Kanten entstehen können. Da wir auf einem Niveau-Netzwerk arbeiten, bei dem die Kanten von Knoten mit Niveau j zu Knoten mit Niveau $j+1$ verlaufen, können die neuen, umgedrehten Kanten also nur von einem Knoten mit Niveau $j+1$ zu einem Knoten mit Niveau j verlaufen. Dabei haben wir nun zwei Optionen. Entweder wir benutzen keine der neuen Kanten oder mindestens eine neue Kante. Die Nutzung einer neuen Kante bringt uns jedoch nichts. Denn wir waren ja schon an einem Knoten, der $j+1$ von der Quelle entfernt ist und gehen über diese Kante zu einem Knoten zurück, der nur j von der Quelle entfernt ist. Damit wir uns von diesem Knoten wieder in Richtung Senke bewegen, müssen wir also eine Kante finden, die uns wieder zu einem Knoten mit Niveau $j+1$ führt. Damit haben wir mindestens zwei Schritte mehr gemacht, als wenn wir nicht zurück gegangen wären. Wären wir nicht zurück gegangen, hätte der Weg mindestens Länge l gehabt, also hat er bei der Benutzung von mindestens einer neuen Kante eine Länge von $l+2$. Aber auch wenn wir die neuen Kanten nicht benutzen, werden die kürzesten Wege länger. Dafür sorgen die wegfallenden Kanten.

Wegfallende Kanten: Im neuen Restgraph $Rest_{\varphi'}$ fallen die Kanten weg, die im Niveau-Netzwerk zu $Rest_{\varphi}$ durch die Berechnung des Sperrflusses saturiert wurden. Durch das Wegnehmen von Kanten werden die kürzesten Wege in einem Graphen garantiert nicht kürzer. Andererseits werden durch das Wegnehmen der saturierten Kanten auch *alle* kürzesten Q-S-Wege zerstört, da auf jedem dieser Wege mindestens eine saturierte Kante liegt. Wenn die Wege also nicht kürzer werden und jeder Weg der Länge l zerstört wird, dann können durch das Wegnehmen der Kanten nur längere Wege entstehen. Diese haben also mindestens die Länge $l+1$.

Ob wir die neuen Kanten nun benutzen oder nicht, die kürzesten Wege werden nach jeder Addition des Sperrflusses um mindestens 1 länger. Wir haben auch schon zuvor gesehen, dass ein kürzester Weg von der Quelle zur Senke nicht beliebig lang werden kann, sondern bei n Knoten höchstens die Länge $n-1$ haben kann. Damit ist die Anzahl der Durchläufe also durch $O(n)$ beschränkt.

8.5 Ein verbesserter Algorithmus zur Sperrflussberechnung

Frage 8.31: Wie ist eine Verbesserung der Sperrflussberechnung motiviert?

Wir haben im vorherigen Kapitel gesehen, dass die Laufzeit des Algorithmus von Dinic durch die Laufzeit der Sperrflussberechnung von $O(n|E|) = O(n^3)$ dominiert wurde. Alle anderen Schritte benötigten höchsten Zeit $O(n^2)$. Von daher ist es sinnvoll, sich zu überlegen, ob die Sperrflussberechnung nicht ebenfalls in $O(n^2)$ durchführbar ist. In diesem Fall ergäbe sich durch die maximal n Interationen des Algorithmus eine Gesamtlaufzeit von $O(n^3)$.

Frage 8.32: Wie funktioniert der Algorithmus von Malhotra, Kumar und Maheshvari?

Der Algorithmus von Malhotra, Kumar und Maheshvari reduziert die Laufzeit der Sperrflussberechnung auf $O(n^2)$. Im Prinzip schafft er dies, da er nicht bescheiden versucht, erst einen Q-S-Weg zu finden und auf diesem mindestens eine Kante zu sperren, sondern in jedem Schritt das Ziel hat, einen ganze Knoten mitsamt seiner ein- oder ausgehenden Kanten zu sperren.

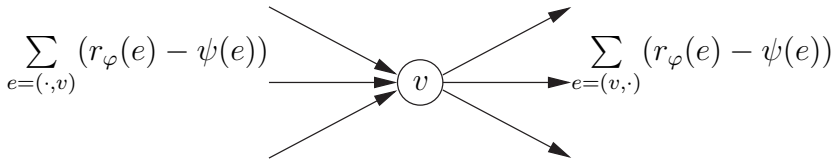


Abbildung 19: Potenzial $pot(v)$ eines Knotens v

Gearbeitet wird auch hier auf einem Niveau-Netzwerk mit den Niveaus $V_0 = \{Q\}, \ldots, V_r = \{S\}$. Wieder versuchen wir für dieses Netzwerk einen Sperrfluss zu berechnen und beginnen dabei mit dem Nullfluss $\psi \equiv 0$. Zur Erhöhung des Sperrflusses gehen wir nun jedoch anders vor. Wir versuchen diesmal nicht, den Fluss auf einem Weg von der Quelle zur Senke zu erhöhen, sondern fangen bei einem Knoten des Netzwerk an und schicken von diesem aus Fluss in Richtung Senke und Quelle. Auf den ersten Blick mag dieses Vorgehen paradox erscheinen, da wir ja Fluss von der Quelle zur Senke schicken möchten. Der Knoten von dem aus Fluss in Richtung Quelle und Senke geschickt wird, wird anhand seines Potenzials ausgesucht. Das Potenzial ist für einen Knoten v wie folgt definiert:

$$pot(v) = \min \left\{ \sum_{e=(\cdot,v)} (r_\varphi(e) - \psi(e)), \sum_{e=(v,\cdot)} (r_\varphi(e) - \psi(e)) \right\}$$

Abbildung 19 verdeutlicht die Formel noch etwas. Im Prinzip wird nur berechnet, wieviel Fluss noch zusätzlich über die eingehenden Kanten durch den Knoten geleitet werden kann bzw. wieviel Fluss durch die ausgehenden Kanten noch weitergegeben werden kann. Von diesem beiden Werten wird das Minimum genommen, da die Kapazität des Knotens durch den kleineren der beiden Werte beschränkt ist.

Der Algorithmus berechnet nun für alle Knoten ihr Potenzial und sucht dabei einen Knoten v mit minimalem Potenzialwert. Wir möchten es nun schaffen, dass von der Quelle zur Senke exakt dieser Wert an zusätzlichem Fluss fließt, wobei dieser zusätzliche Fluss komplett durch den Knoten v fließen soll. Wenn wir dies schaffen, müssen entweder alle eingehenden oder alle ausgehenden Kanten des Knotens v saturiert sein, da das Potenzial des Knotens dann völlig erschöpft ist. Im nächsten Schritt brauchen wir den Knoten und die zu ihm adjazenten Kanten dann nicht mehr weiter betrachten, da zusätzlicher Fluss diesen Knoten entweder gar nicht mehr erreichen würde oder von ihm aus nicht mehr weitergeleitet werden könnte. Aus diesem Grund kann der Knoten samt seiner adjazenten Kanten entfernt werden. Wenn wir in jedem Schritt einen Knoten und seine Kanten entfernen, betrachten wir also nach $O(n)$ Schritten irgendwann die Quelle oder die Senke und saturieren entweder alle ausgehenden Kanten (im Falle der Quelle) oder eingehenden Kanten (im Falle der Senke). Danach kann es keinen Weg mehr zwischen Quelle und Senke geben und wir haben einen Sperrfluss berechnet.

Wie können wir nun erreichen, dass der zusätzliche Fluss komplett durch den Knoten v mit minimalem Potenzial fließt? Es wird sich zeigen, dass wir dazu am besten bei eben jenem Knoten beginnen sollten. Wir wissen, dass der zusätzliche Fluss komplett durch v fließen soll. Die Menge an zusätzlichem Fluss nennen wir im Folgenden Überschuss. Dieser Überschuss muss irgendwie bei v ankommen und irgendwie von v weitergeleitet werden. Die Kapazität dazu hat der Knoten v auf jeden Fall, da sich die Menge des Überschuss ja gerade durch sein Potenzial ergibt. Wenn sich das Potenzial also durch die Kapazität der eingehenden Kanten ergeben hat, wissen wir, dass exakt die Menge an Überschuss zusätzlich beim Knoten ankom-

men kann, ansonsten kann exakt diese Menge weitergegeben werden. Wir werden nun zeigen, dass es möglich ist, den Überschuss so durch das Netzwerk zu leiten, dass er irgendwann bei der Senke ankommt. Es wird nur diese Richtung gezeigt, da die Argumentation, dass der Überschuss tatsächlich von der Quelle bis zu diesem Knoten gelangen kann, analog läuft. Insgesamt ist dann gezeigt, dass ein solcher zusätzlicher Fluss bzw. eine Erhöhung des Sperrflusses um diesen Wert möglich ist. Wir betrachten ein Niveau-Netzwerk, d.h. wenn Fluss vom Knoten v weitergegeben wird, kann dieser nur in Richtung Senke weitergegeben werden und muss dort auch nach endlich vielen Schritten irgendwann ankommen. Es bleibt also nur noch zu zeigen, dass wir den Überschuss auch tatsächlich vollständig weiterleiten können. Dazu zeigen wir, dass jeder Knoten im Netzwerk im schlimmsten Fall den vollständigen Überschuss an Knoten des nächsthöheren Niveaus weitergeben kann und dass diese den Überschuss auch komplett aufnehmen können. Die Argumentation läuft in beiden Fällen ähnlich. Würde es einen Knoten w geben, der nicht in der Lage wäre, den kompletten Überschuss weiterzuleiten, dann würde dies bedeuten, dass das Potenzial des Knotens w echt kleiner ist als der Überschuss und damit echt kleiner als das Potenzial des Knotens v. Dies würde jedoch bedeuten, dass wir von vornherein nicht Knoten v als den Knoten mit dem kleinsten Potenzial gewählt hätten. Mit dem gleichen Argument kann gezeigt werden, dass jeder andere Knoten des Netzwerks in der Lage sein muss, den Überschuss komplett aufzunehmen.

Frage 8.33: Beschreibe den Algorithmus von Malhotra, Kumar und Maheshvari algorithmisch! Beweise, dass seine Laufzeit quadratisch ist!

Nachdem der Algorithmus zuvor umgangssprachlich erklärt wurde, wird er hier nun algorithmisch beschrieben. Wir beschreiben den Hauptalgorithmus namens Forward-Backward-Propagation und anschließend den Unteralgorithmus Forward(v), welcher von diesem benutzt wird. Die Prozedur Backward(v) dürfte der Prozedur Forward(v) sehr ähnlich sehen, da bei ihr wahrscheinlich einfach die Kanten umgedreht betrachtet werden.

Algorithmus 12 Forward-Backward-Propagation

```
Berechne für alle Knoten v den Wert pot(v).
while Netzwerk enthält noch Q und S do
  Berechne einen Knoten v_min mit minimalem Potenzialwert.
  if pot(v_min) > 0 then
    Forward(v_min); Backward(v_min);
  end if
  Entferne v_min und die zu ihm adjazenten Kanten aus dem Netzwerk. Aktualisiere dabei
  die pot(·)-Werte für seine Vorgänger und Nachfolger.
end while
```

Algorithmus 12 beschreibt den Hauptalgorithmus Forward-Backward-Propagation, der genau das macht, was etwas zuvor schon umgangssprachlich erklärt wurde. Zuerst wird für alle Knoten ihr Potenzial berechnet. So lange der Fluss noch erhöht werden kann, wird dann unter allen Knoten der mit dem kleinsten Potenzial gewählt und von ihm aus der entsprechende Fluss in Richtung der Quelle und der Senke propagiert. Allerdings nur, wenn das Potenzial des Knotens größer als 0 ist, der Knoten also tatsächlich etwas weitergeben kann. Das Potenzial eines Knotens kann z.B. 0 sein, wenn er keine ein- oder ausgehenden Kanten mehr hat (die Quelle und Senke werden extra behandelt). Ist dies geschehen wird der Knoten entfernt, da entweder alle seine eingehenden oder alle seine ausgehenden Kanten saturiert sind. Da bei

Algorithmus 13 Forward(v)

Sei Überschuss[v]=$pot(v)$ und für alle Knoten $w \neq v$ sei Überschuss[w] = 0.
Initialisiere eine Queue Q mit dem Knoten v.
while Queue $Q \neq \emptyset$ **do**
  Entferne das oberste Element aus der Queue Q und nenne es u.
  Sei e die erste von u ausgehende Kante im Niveau-Netzwerk.
  while Überschuss[u] > 0 **do**
    Sei $e = (u, w)$. Berechne $\delta = \min\{r_\varphi(e) - \psi(e),\ \text{Überschuss}[u]\}$.
    Erhöhe den Fluss auf e um δ und aktualisiere $pot(u)$ und $pot(w)$.
    if Überschuss[w] $= 0 \ \wedge\ \delta > 0 \ \wedge\ w \neq S$ **then**
      Füge w in die Queue Q ein.
    end if
    Überschuss[u] = Überschuss[u] $- \delta$; Überschuss[w] = Überschuss[w] $+ \delta$.
    if $r_\varphi(e) - \psi(e) = 0$ **then**
      Entferne die Kante e aus dem Niveau-Netzwerk.
    end if
    Es sei e nun die nächste von u ausgehende Kante.
  end while
end while

der Entfernung des Knotens auch Kanten wegfallen, müssen wir die Potenzialwerte für alle Vorgänger und Nachfolger aktualisieren.

Algorithmus 13 beschreibt den Unteralgorithmus Forward(v), der den Fluss durch das Netzwerk in Richtung Senke propagiert. Zu Beginn liegt der ganze Überschuss bei Knoten v. Aus diesem Grund ist er zu Beginn auch der einzige Knoten in die Queue Q, welche die Knoten verwaltet, von denen noch Überschuss in Richtung Senke propagiert werden muss. Nun wird, so lange die Queue nicht leer ist, immer das oberste Element aus der Queue genommen. Die Verwendung einer Queue sichert, dass das Niveau-Netzwerk Niveau für Niveau durchlaufen wird. So lange wir nun noch Überschuss für den betrachteten Knoten weiterzugeben haben, betrachten wir der Reihe nach die Kanten an dem Knoten. Mit der Berechnung von δ schauen wir, ob wir den Überschuss komplett über die aktuell betrachtete Kante (u, w) weitergeben können oder nur einen Teil davon. Entsprechend erhöhen wir den Fluss über die Kante und aktualisieren auch schon die Potenzialwerte der Knoten, die durch diese Kante verbunden werden. Nun schauen wir als nächstes, ob wir den Knoten w auch in die Queue einfügen müssen. Dies müssen wir nur, wenn er zum ersten Mal betrachtet wird (dann ist sein Überschuss noch 0), wenn überhaupt Überschuss über die Kante zu w geleitet wird und wenn er nicht die Senke ist. Anschließend aktualisieren wir die Überschusswerte für u und w. Wenn die Kante (u, w) durch den Fluss saturiert wird, entfernen wir sie. Danach wird mit der nächsten Kante an u fortgefahren.

Nun kommen wir zur Abschätzung der Laufzeit. Der Initialisierungsschritt in Algorithmus 12 (Forward-Backward-Propagation) benötigt Zeit $O(n)$. Die anschließende Schleife wird $O(n)$-mal durchlaufen, da in jedem Schritt mindestens ein Knoten entfernt wird. Die Berechnung eines Knotens mit minimalem Potenzialwert erfolgt in Zeit $O(n)$. Danach werden unter Umständen die Unteralgorithmen Forward(v) und Backward(v) aufgerufen, deren Laufzeit wir gleich analysieren. Das Entfernen von Kanten geht in Zeit $O(1)$, wir müssen uns also nur noch überlegen, wieviele Kanten höchstens entfernt werden können. Während des Aufrufs von

Algorithmus 13 werden ebenfalls Kanten entfernt. Doch eins nach dem anderen. Die äußerste Schleife von Algorithmus 13 läuft, so lange die Queue Q nicht leer ist. Da jeder Knoten höchstens einmal in die Queue eingefügt wird, wird die äußere Schleife also $O(n)$-mal aufgerufen. In der inneren Schleife werden für jeden betrachteten Knoten seine Kanten betrachtet und Flusswerte sowie Potenziale aktualisiert. Diese Schritte gehen in $O(1)$. Die Hauptarbeit liegt hier also in der Betrachtung bzw. Entfernung von Kanten. Jede Kante die saturiert ist, wird entfernt. Da für einen Knoten nur eine folgende Kante angeschaut wird, wenn die vorherige saturiert wurde, kann es also für jeden Knoten nur höchstens eine Kante geben, die betrachtet wird, jedoch nicht entfernt wird, da sie nicht saturiert ist. Für jeden betrachteten Knoten der Queue kann es also höchstens eine Kante geben, die Arbeit verursacht, jedoch nicht entfernt wird und damit später noch einmal Arbeit verursachen kann. Damit kann die Laufzeit von Algorithmus 13 also mit $O(n + \text{Anzahl entfernter Kanten})$ abgeschätzt werden. Die Betrachtungen für Backward(v) laufen analog. Kanten werden bei der Berechnung eines Sperrflusses abwechseln von Algorithmus 12 und Algorithmus 13 entfernt und kommen nicht wieder. Wir betrachten die Arbeit, die maximal durch die Kanten entstehen kann, daher nun „in einem Rutsch“. Wie viele Kanten können höchstens entfernt werden? Bei dem abwechselnden Entfernen können höchstens alle Kanten entfernt werden, also $O(|E|) = O(n^2)$. Damit haben wir die Arbeit, die durch Kanten verursacht werden kann, also schon abgeschätzt. Die Arbeit in Forward(v) beträgt ohne Betrachtung der Arbeit, die durch Kanten verursacht wird, somit nur noch $O(n)$, wobei Forward(v) und Backward(v) höchstens $O(n)$-mal aufgerufen werden. Somit kann die Gesamtlaufzeit mit $O(n^2) + O(|E|) = 2 \cdot O(n^2) = O(n^2)$ abgeschätzt werden.

9 Algorithmen für Probleme der elementaren Zahlentheorie

9.1 Kryptographische Systeme

Frage 9.1: Aus welchen Komponenten besteht ein kryptographisches System?
Ein kryptographisches System besteht aus den folgenden Komponenten:

- Der endlichen Schlüsselmenge K (z.B. $\{0,1\}^{1024}$)
- Der Nachrichtenmenge M (z.B. $\{0,1\}^*$)
- Der Menge der Kryptogramme Y (z.B. $\{0,1\}^*$)
- Einer Chiffrierfunktion $E : K \times M \to Y$
- Einer Dechiffrierfunktion $D : K \times Y \to M$, so dass $D(k, E(k, m)) = m$ für alle $k \in K$ und $m \in M$.

Um miteinander zu kommunizieren, vereinbaren zwei Benutzer A und B des kryptographischen Systems also einen Schlüssel $k \in K$. Um eine Nachricht $m \in M$ gesichert zu versenden, wird diese mit Hilfe der Chiffrierfunktion E in ein Kryptogramm $y \in Y$ umgewandelt, also: $y = E(k, m)$. Dieses wird nun über einen unter Umständen unsicheren Kanal versendet. Wenn nun z.B. Benutzer B dieses Kryptogramm erhält, benutzt er die Dechiffrierfunktion D und den zuvor vereinbarten Schlüssel k, um die Nachricht m zurück zu gewinnen: $m = D(k, y)$.

Der Kanal, über den das Kryptogramm geschickt wird, ist unter Umständen unsicher, d.h. es kann passieren, dass ein unbefugter Dritter (Gegener) das Kryptogramm abfängt. Das kryptographische System muss daher so beschaffen sein, dass der Gegner, aus dem Kryptogramm allein nicht die Nachricht m bzw. den Schlüssel k errechnen kann. Die Nachricht kann in einem kryptographischen System aufgrund von Redundanzen in der natürlichen Sprache errechnet werden und der Schlüssel k aufgrund der Tatsache, dass die Schlüsselmenge endlich ist. Das System muss also so beschaffen sein, dass die Errechnung einer der beiden Komponenten selbst auf den schnellsten Rechnern nicht effizient möglich ist, also z.B. 1000 Jahre CPU-Zeit in Anspruch nimmt.

Frage 9.2: Welche zwei Arten von kryptographischen Systemen können grob unterschieden werden?
Es kann grob zwischen kryptographischen Systemen mit geheimen Schlüsseln und mit öffentlichen Schlüsseln unterschieden werden:

Geheime Schlüssel: Bei diesen Verfahren wird ein geheimer Schlüssel verwendet, der nur den beiden Benutzern bekannt ist. Solche Verfahren sind z.B. gut im militärischen Bereich einsetzbar. Die Übermittlung des geheimen Schlüssels kann dann z.B. durch einen vertrauenswürdigen Kurier geschehen.

Öffentliche Schlüssel: Diese Verfahren benutzen öffentlich bekannte Schlüssel. Auf den ersten Blick erscheint dies paradox, denn wenn die Schlüssel bekannt sind, kann ein Gegner eine abgefangene Nachricht ebenfalls mit den Schlüsseln dekodieren. Der Trick besteht nun darin, dass jeder Benutzer B zum öffentlich bekannten Schlüssel k_B eine zusätzliche Information $I(k_B)$ erzeugt. Die Verschlüsselung einer Nachricht geschieht nun mit dem öffentlich bekannten Schlüssel k_B, d.h. es wird ein Kryptogramm $y = E(k_B, m)$

erzeugt. Das System ist nun so beschaffen, dass die Dekodierung von y mit Hilfe der Zusatzinformation $I(k_B)$ sowie k_B effizient möglich ist, nicht jedoch, wenn nur k_B zur Verfügung stehen.

Frage 9.3: Beschreibe das RSA-Kryptosystem!

Das RSA-Kryptosystem wurde von Rivest, Shamir und Adleman entwickelt und ist ein kryptographisches System mit öffentlichen Schlüsseln („public key cryptosystem“).

Jeder Benutzer erzeugt zwei sehr große Primzahlen $p, q \geq 2^l$, die also je $l+1$ Bits lang sind. Die Erzeugung dieser Primzahlen geschieht derart, dass zufällige Zahlen erzeugt werden und darauf getestet werden, ob sie Primzahlen sind. Dazu kann z.B. der probabilistische Primzahltest von Solovay und Strassen benutzt werden, der in Kapitel 9.6 vorgestellt wird. Aus diesen beiden Primzahlen wird nun $n = pq$ sowie der Wert der Eulerfunktion $\varphi(n)$ errechnet. Die Eulerfunktion $\varphi(n)$ gibt die Anzahl der Elemente $a \in \{1, \ldots, n-1\}$ an, die mit n keinen gemeinsamen Teiler haben. Für das Produkt zweier Primzahlen kann diese Zahl schnell berechnet werden. Wir betrachten die Zahl $n = pq$. Da p und q Primzahlen sind, können wir n auch nur in diese beiden Primzahlen faktorisieren. Dementsprechend kann n auch nur die Teiler p und q mit den Zahlen in $\{1, \ldots, n-1\}$ bzw. $\{1, \ldots, pq-1\}$ gemeinsam haben. Wieviele Zahlen gibt es nun, die den Teiler p mit n gemeinsam haben? Dies sind $(q-1)$ Zahlen, nämlich $1 \cdot p,\ 2 \cdot p, \ldots,\ (q-1) \cdot p$. Entsprechend gibt es $(p-1)$ Zahlen, die mit n den Teiler q gemeinsam haben. Diese beiden Zahlen müssen wir also von $pq-1$ subtrahieren, um die Anzahl der Zahlen zu erhalten, die teilerfremd zu n sind:

$$pq - 1 - (p-1) - (q-1) = pq - 1 - p + 1 - q + 1 = pq - p - q + 1 = (p-1)(q-1)$$

Für zwei Primzahlen p und q gilt also stets $\varphi(pq) = (p-1)(q-1)$. Nachdem $\varphi(n)$ berechnet wurde, wird eine weitere zufällige Zahl $e \in \{2, \ldots, n-1\}$ gewählt, die teilerfremd zu $\varphi(n)$ sein muss. Diese Zahl e ist der Chiffrierexponent und wird, wie wir gleich sehen werden, benutzt, um eine Nachricht zu chiffrieren. Zum Dechiffrieren des Kryptogramms werden wir das multiplikativ inverse Element zu e benötigen, daher wird auch noch $d \equiv e^{-1} \bmod \varphi(n)$ berechnet. Damit haben wir nun alles zusammen, um den öffentlichen und den geheimen Schlüssel anzugeben. Der öffentliche Schlüssel ist das Paar (n, e) und der geheime Schlüssel das Paar $(\varphi(n), d)$.

Wie werden diese Schlüssel nun verwendet? Um eine Nachricht zu chiffrieren, kann nur der öffentliche Schlüssel benutzt werden, d.h. wir haben die Information (n, e) zur Verfügung. Gegeben sei also die Nachricht m. Zuerst muss diese, in Abhängigkeit von der Größe $l+1$ der Primzahlen, in Blöcke m_i der Länge $2l$ zerlegt werden. Jeder dieser Blöcke wird dann als eine Binärzahl aufgefasst, für die gilt: $0 \leq m_i < 2^{2l}$. Die Chiffrierfunktion ist definiert als:

$$E((n, e), m_i) \equiv m_i^e \bmod n$$

Die Dechiffrierfunktion ist ähnlich definiert:

$$D((n, d), y) \equiv y^d \bmod n$$

Wir wir in Kapitel 9.2 sehen werden, lassen sich diese beiden Funktionen effizient realisieren. Es bleibt noch zu zeigen, dass mit Hilfe der Dechiffrierfunktion auch tatsächlich die ursprüngliche Nachricht zurück gewonnen werden kann. Es gilt:

$$D((n, d), E((n, e), m_i)) \equiv m_i^{ed} \bmod n$$

Wir haben e und d so gewählt, dass $ed \equiv 1 \bmod \varphi(n)$ gilt. Aufgrund einiger Ergebnisse der elementaren Zahlentheorie folgt dann, dass $m_i^{ed} \equiv m_i \bmod n$ ist. Dabei spielt unter anderem der Euler'sche Satz eine Rolle, worauf nun jedoch nicht genauer eingegangen wird. Wir müssen jedoch zeigen, dass die Operation $m_i \bmod n$ unsere Nachricht nicht mehr verändert. Die Nachricht m_i wurde als Zahl zwischen 0 und 2^{2l} interpretiert. Da $n \geq 2^{2l}$ ist, wird auch tatsächlich eindeutig m_i berechnet.

Es wird nun kurz darauf eingegangen, warum das RSA-Kryptosystem wahrscheinlich sicher ist. Dabei wird davon ausgegangen, dass dem Gegner nur der öffentliche Schlüssel zur Verfügung steht, da die Dechiffrierung eines Kryptogramms mit dem geheimen Schlüssel offensichtlich trivial ist. Um ein Kryptogramm zu entschlüsseln, benötigt der Gegner im wesentlichen die geheime Information d. Die Frage ist nun, ob es effizient möglich ist, diese aus dem öffentlichen Schlüssel zu berechnen. Der öffentliche Schlüssel besteht aus dem Paar e und n, d.h. der Gegner weiß, dass d das multiplikativ Inverse zu e ist, wenn modulo $\varphi(n)$ gerechnet wird. Um $\varphi(n)$, ebenfalls ein Teil des geheimen Schlüssels, zu berechnen, müsste der Gegner jedoch die zwei Primzahlen p und q kennen, die multipliziert n ergeben. Da n ein Teil des öffentlichen Schlüssel ist könnte der Gegner also versuchen, n zu faktorisieren, um p und q zu bekommen. Es wird jedoch davon ausgegangen, dass die Faktorisierung einer Zahl, zumal einer so großen wie n, sehr schwierig ist und sehr viel Resourcen (z.B. Rechenzeit) beanspruchen würde.

Frage 9.4: Welche Algorithmen werden zur Realisierung des RSA-Kryptosystems benötigt?

Das RSA-Kryptosystem benötigt mehrere effiziente Algorithmen, damit es funktioniert. Zu diesen Algorithmen, die im Folgenden vorgestellt werden, gehören unter anderem:

- Die Berechnung von großen Potenzen.
- Die Berechnung des größten gemeinsamen Teilers (ggT) um festzustellen, ob eine Zahl teilerfremd zu einer anderen ist.
- Die Berechnung des multiplikativ Inversen.
- Primzahltest.

9.2 Potenzen, multiplikative Inverse und größte gemeinsame Teiler

Frage 9.5: Welche Operationen werden im Folgenden als Elementaroperationen betrachtet?

Die arithmetischen Operationen Addition, Subtraktion, Multiplikation und ganzzahlige Division werden im folgenden als Elementaroperationen betrachtet. Dadurch ist auch die Berechnung von $a \bmod m$ in konstanter Zeit möglich, woraus folgt, dass die Operationen Addition, Subtraktion und Multiplikation im $\mathbb{Z}_m$ ebenfalls als Elementaroperationen angesehen werden können.

Frage 9.6: Gebe einen effizienten Algorithmus zum Potenzieren an!

Im RSA-System werden Nachrichten verschlüsselt und entschlüsselt, indem sie teilweise mit großen Zahlen potenziert werden. Gesucht ist nun also ein effizientes Verfahren, mit dem

$x^n \bmod m$ berechnet werden kann. Ein naives Verfahren würde x einfach $(n-1)$ mal multiplizieren und anschließend eine modulare Reduktion durchführen, d.h.

$$(x \cdot \ldots \cdot x) \bmod m$$

berechnen. Dieses Verfahren hat jedoch mehrere Nachteile. Zum einen würden relativ viele Multiplikationen benötigt werden und zum anderen würden die Zwischenergebnisse sehr groß werden. Dies liegt daran, dass bei der Addition, Subtraktion und Multiplikation von zwei k Stellen langen Binärzahlen höchstens eine $2k$ lange Zahl herauskommen kann. Wenn x und n z.B. $2^9 = 512$ Bit lange Zahlen sind, verdoppelt sich die Länge der Zwischenergebnisse sehr schnell und das Endergebnis kann bis zu $2^9 \cdot 2^{512} = 2^{521}$ Binärstellen lang werden. Zum Vergleich: Ein Gigabyte entspricht 2^{33} Bit, d.h. selbst heutige Rechner könnten solche Zwischenergebnisse nicht abspeichern.

Wir müssen also ein in jeder Hinsicht effizienteres Verfahren finden. Die Problematik der großen Zwischenergebnisse bekommen wir in den Griff, indem wir ausnutzen, dass Zwischenergebnisse bei der Moduloarithmetik ebenfalls modulo m gerechnet werden können. Die große Anzahl der Multiplikationen reduzieren wir, indem wir $x^n \bmod m$ rekursiv wie in Algorithmus 14 berechnen, wobei alle Zwischenergebnisse modulo m gerechnet werden.

Algorithmus 14 Effizienter Algorithmus zur Berechnung von $x^n \bmod m$

Es ist $x^1 = x$.

1. Falls n gerade ist, berechne rekursiv $y = x^{\frac{n}{2}} \bmod m$. Dann ist $x^n = y^2 \bmod m$.
2. Falls n ungerade ist, berechne rekursiv $y = x^{n-1} \bmod m$, Dann ist $y^n = x \cdot y \bmod m$.

Dadurch, dass in jedem Schritt modulo m gerechnet wird, werden die Zwischenergebnisse stets kleiner als m gehalten. Auch die Anzahl der Multiplikationen reduziert sich bei diesem Vorgehen. Es kann gezeigt werden, dass zur Berechnung von $x^n \bmod m$ höchstens $2\lceil \log(n+1) \rceil - 2$ Multiplikationen benötigt werden. Wir führen den Beweis per Induktion über n. Die Binärdarstellung von $n \geq 1$ sei $(n_{k-1}, \ldots, n_0)$ für $k = \lceil \log(n+1) \rceil$. Mit $A(n)$ bezeichnen wir im folgenden die Anzahl der Einsen in der Binärdarstellung von n. Da n mit $\lceil \log(n+1) \rceil$ Bits dargestellt wird, beträgt die Anzahl der Einsen in n also höchstens $\lceil \log(n+1) \rceil$. Wir zeigen nun, dass $x^n \bmod m$ mit $k + A(n) - 2 \leq 2\lceil \log(n+1) \rceil - 2$ Multiplikationen berechnet werden kann.

Für $n = 1$ stimmt die Annahme, da die Binärdarstellung von $n = 1$ aus einer 1 besteht und eine Stelle lang ist. Es werden damit $1 + 1 - 2 = 0$ Multiplikationen benötigt, was auch korrekt ist, da x^1 die Abbruchbedingung für die Rekursion darstellt. Für den Induktionsschritt müssen wir nun zwischen den beiden Fällen im Algorithmus unterscheiden.

n gerade: Wenn n gerade ist, ist die letzte Stelle der Binärdarstellung von n eine Null, d.h. $n_0 = 0$. Wir berechnen nun rekursiv $x^{n/2}$. Zuerst überlegen wir uns, wie die Binärdarstellung von $n/2$ aussieht. Wird eine Zahl durch zwei geteilt, entspricht dies in der Binärdarstellung einem Shift der Bits „nach rechts“. Die Zahl $n/2$ besteht also aus den Bits $(n_{k-1}, \ldots, n_1)$ und ist $k-1$ Bits lang. Die Anzahl der Einsen in der Binärdarstellung von $n/2$ hat sich gegenüber n jedoch nicht geändert, da mit $n_0 = 0$ nur eine Null weggefallen ist. Nach Induktionsvoraussetzung wird $x^{n/2}$ mit $(k-1) + A(n) - 2$ Multiplikationen berechnet. Zur Berechnung von x^n müssen wir $x^{n/2}$ mit sich selbst multiplizieren, was einer weiteren Multiplikation entspricht. Damit ergibt sich insgesamt die

folgende Anzahl an Multiplikationen:

$$(k-1) + A(n) - 2 + 1 = k + A(n) - 2$$

n ungerade: Ist n ungerade, berechnen wir rekursiv x^{n-1}. Wie unterscheidet sich $n-1$ von n? Da n ungerade ist, gilt für das letzte Bit $n_0 = 1$. Eine Subtraktion von 1 setzt dieses Bit auf Null, womit die Anzahl der Einsen in $n-1$ gegenüber n um eins abnimmt. Beide Zahlen sind jedoch weiterhin k Bits lang. Nach Induktionsvoraussetzung wird x^{n-1} demnach mit $k + (A(n) - 1) - 2$ Multiplikationen berechnet. Zur Berechnung von x^n kommt wieder eine Multiplikation hinzu, da sich x^n als $x \cdot x^{n-1}$ berechnet. Insgesamt ergibt sich auch hier:

$$k + (A(n) - 1) - 2 + 1 = k + A(n) - 2$$

Frage 9.7: Wie kann der größte gemeinsame Teiler (ggT) zweier Zahlen berechnet werden?

Die Schulmethode zur Berechnung des größten gemeinsamen Teilers zweier Zahlen a und b benutzt die Primfaktorzerlegung der beiden Zahlen. Wir sind allerdings nicht in der Lage, eine Zahl effizient in ihre Primfaktoren zu zerlegen. Wenn wir dies könnten, wäre das RSA-Kryptosystem ja auch nicht sicher. Glücklicherweise kann der ggT zweier Zahlen auch auf andere Weise berechnet werden. Der euklidische Algorithmus nutzt dabei hauptsächlich die folgende Eigenschaft des ggTs zweier Zahlen a und b aus:

$$\mathrm{ggT}(a, b) = \mathrm{ggT}(b, a \bmod b) \text{ wenn } b \neq 0$$

Weiter wird $\mathrm{ggT}(a, 0) = a$ definiert. Bevor der euklidische Algorithmus selbst angegeben wird, wird die obige Behauptung bewiesen. Wir nehmen im Folgenden an, dass für die beteiligten Zahlen $a \geq b$ gilt. Warum wir dies annehmen können, wir später ersichtlich, wenn der euklidische Algorithmus erläutert wird. Das Ergebnis der Berechnung von $a \bmod b$ ist die Zahl, die sich ergibt, wenn wir b so oft wie möglich von a abziehen, also $a - \lfloor \frac{a}{b} \rfloor \cdot b$. Um zu zeigen, dass diese Zahl und b denselben ggT wie a und b haben, reicht es zu zeigen, dass b und $a - b$ denselben ggT wie a und b haben. Wir können uns dann vorstellen, dass bei der Operation $a \bmod b$ sukzessive b von a abgezogen wird und sich der ggT dabei nicht ändert. Wir zeigen nun also:

$$\mathrm{ggT}(a, b) = \mathrm{ggT}(b, a - b)$$

Dabei sind zwei Richtungen zu zeigen. Allgemein ist es so, dass eine Zahl c, die zwei Zahlen i und j mit $i \geq j$ teilt, auch $i + j$ bzw. $i - j$ teilt. Dies kann schnell gezeigt werden. Die Zahl c teilt i und j, d.h. das Ergebnis beider Divisionen ist ganzzahlig. Da wir $(i + j)/c$ auch als $i/c + j/c$ schreiben können und die Summe zweier ganzer Zahl ebenfalls eine ganze Zahl ist, ist c auch ein Teiler von $i + j$. Entsprechend wir die Behauptung für $i - j$ gezeigt. Somit teilt also jeder gemeinsame Teiler von a und b auch $a - b$ und b. Und umgekehrt teilt jeder gemeinsame Teiler von $a - b$ und b auch die Zahlen $(a - b) + b = a$ und b. Damit haben wir alles zusammen, um den Algorithmus zu beschreiben.

Die Eingabe für Algorithmus 15 sind laut Skript zwei ganze Zahlen a und b mit $a \geq b \geq 1$. Allerdings würde die Forderung $a, b \geq 1$ ausreichen. Gilt beim ersten Aufruf von $\mathrm{ggT}(a, b)$ für die Zahlen $a < b$, so wird im ersten rekursiven Aufruf die Position von a und b vertauscht, so

Algorithmus 15 Der euklidische Algorithmus zur Berechnung von $\mathrm{ggT}(a, b)$

```
c = a mod b (also 0 ≤ c ≤ b − 1)
if c = 0 then
  return b;
else
  return ggT(b, c);
end if
```

dass im nächsten Schritt quasi $a' = b > b' = a$ gilt. Dies liegt daran, dass für $a < b$ der Wert $c = a \bmod b = a$ errechnet wird und im Anschluss rekursiv $\mathrm{ggT}(b, a)$ berechnet wird.

Die Laufzeit des euklidischen Algorithmus ist $O(\log(a + b))$. Dazu zeigen wir, dass die Anzahl der wesentlichen Operationen, in diesem Fall die Modulo-Operationen, nicht größer als $\lfloor \log_{\frac{3}{2}}(a+b) \rfloor$ ist. Dazu zeigen wir zuerst, dass in jedem Schritt die Summe der beteiligten Zahlen höchstens zwei Drittel so groß ist wie die Summe der im Schritt zuvor beteiligten Zahlen:

$$b + c \leq \frac{2}{3}(a + b)$$

Dies ist äquivalent zu $a+b \geq 3/2(b+c)$, was wir nun zeigen, indem wir Überlegungen bezüglich a und b anstellen. Es gilt $c = a - k \cdot b$ für ein $k \geq 1$. Dies stellen wir um zu $a = c + k \cdot b$. Der kleinstmögliche Wert für k ist 1, also ist a mindestens so groß wie $1 \cdot b + c$. Demnach gilt $a \geq b + c$. Der Schritt für b ist etwas einfacher. Wir schreiben b als $b = \frac{2b}{2} = \frac{b+b}{2}$. Da $b \geq c$ gilt, können wir b wie folgt abschätzen, wenn wir ein b im Zähler durch c ersetzen: $b \geq \frac{b+c}{2}$. Damit folgt insgesamt:

$$a + b \geq (b + c) + \frac{b + c}{2} = \frac{3}{2}(b + c)$$

Dieses Ergebnis werden wir gleich in einem Induktionsbeweis benötigen. Wir bezeichnen nun mit $L(a, b)$ die Anzahl der Modulo-Operationen für die Eingabe a und b. Es gilt dann für $L(1, 1) = 1$ und allgemein für $L(a, b) = L(b, c) + 1$ für ein $a \geq 2$. Die Induktion läuft nun über a und b (laut Skript über $a + b$?). Zu zeigen ist:

$$L(a, b) \leq \log_{\frac{3}{2}}(a + b)$$

Der Induktionsanfang ist:

$$L(1, 1) = 1 \leq \log_{\frac{3}{2}}(1 + 1) = \log_{\frac{3}{2}} 2$$

Der Induktionschritt läuft wie folgt:

$$\begin{aligned} L(a, b) &= L(b, c) + 1 \leq \log_{\frac{3}{2}}(b + c) + 1 \leq \log_{\frac{3}{2}}\left(\frac{2}{3}(a + b)\right) + 1 \\ &= \log_{\frac{3}{2}}\left(\frac{2}{3}(a + b)\right) + \log_{\frac{3}{2}}\left(\frac{3}{2}\right) = \log_{\frac{3}{2}}\left(\frac{2}{3}(a + b) \cdot \frac{3}{2}\right) \\ &= \log_{\frac{3}{2}}(a + b) \end{aligned}$$

Frage 9.8: Wie können multiplikative Inverse berechnet werden?

Algorithmus 16 Erweiterter euklidischer Algorithmus INV(a, b)

```
if b = 1 then
  return 1;
end if
c = a mod b;
c* = INV(b, c);
return ((1 − c*a)/b) mod a;
```

Zur Berechnung des multiplikativ Inversen kann der erweiterter euklidische Algorithmus (Algorithmus 16) verwendet werden. Er sieht dem etwas weiter oben beschriebenen Algorithmus recht ähnlich und berechnet zu einem Element $b \bmod a$ mit $\mathrm{ggT}(a, b) = 1$ (d.h. b hat ein multiplikativ inverses Element) das Element $b^{-1} \bmod a$.

Der Beweis, dass Algorithmus 16 tatsächlich das multiplikativ Inverse berechnet, erfolgt per Induktion über b. Für $b = 1$ wird tatsächlich 1 als korrektes multiplikatives Inverses zurückgeliefert.

Zu zeigen ist nun also, dass für $b > 1$ ebenfalls das korrekte Ergebnis berechnet wird. Dazu müssen wir im wesentlichen zeigen, dass der Aufruf von INV(b, c) gültig ist, da wir hierfür gleich als Induktionsvoraussetzung annehmen, dass das multiplikativ Inverse von $c \bmod b$ korrekt berechnet wird. Damit der Aufruf gültig ist, müssen zwei Bedingungen gelten. Der ggT von b und c muss ebenfalls 1 sein, da c ansonsten kein multiplikativ Inverses besitzt, und desweiteren muss $c > 0$ gelten. Letzteres gilt, da c als $c = a \bmod b$ berechnet wird. Wäre $c = 0$, so würde dies bedeuten, dass a ein Vielfaches von b wäre. Da $b > 1$ angenommen wird, müsste dann jedoch auch $a > 1$ gelten und der größte gemeinsame Teiler von a und b wäre also ebenfalls größer als 1. Dies kann jedoch nicht sein, da zu Beginn für die Eingabe a und b gilt, dass $\mathrm{ggT}(a, b) = 1$ ist. Das rekursiv betrachtete Paar ist das Paar b und $c = a \bmod b$. Im Beweis zum euklidischen Algorithmus haben wir jedoch schon gesehen, dass $\mathrm{ggT}(a, b) = \mathrm{ggT}(b, a \bmod b)$ ist. Somit muss auch der ggT von b und $c = a \bmod b$ gleich 1 sein. Wir nehmen nun also an, dass der Aufruf INV(b, c) mit c^* korrekt das multiplikativ Inverse $c^{-1} \bmod b$ zurückliefert.

Das Ziel im nächsten Schritt ist es nun, zu zeigen, dass wenn c^* korrekt als $c^{-1} \bmod b$ berechnet wird, dann auch mit $\frac{1-c^*a}{b} \bmod a$ das Inverse von $b \bmod a$ berechnet wird. Zuerst überlegen wir uns einige Eigenschaften von $\frac{1-c^*a}{b}$. Wir beginnen mit $c = a \bmod b$ und formen dies etwas um. Der Ausdruck $c = a \bmod b$ ist gleichbedeutend mit $c - a \equiv 0 \bmod b$. Also gilt auch $c^*(c - a) \equiv 0 \bmod b$. Wir multiplizieren c^* in die Klammer und erhalten $c^*c - c^*a \equiv 0 \bmod b$. Da c^* nach Induktionsvoraussetzung das Inverse zu c ist, kann dies zu $1 - c^*a \equiv 0 \bmod b$ umgeformt werden, was bedeutet, dass $1 - c^*a$ ein Vielfaches von b ist. Wenn wir $1 - c^*a$ durch b dividieren, erhalten wir also eine ganze Zahl Z. Eine erneute Multiplikation von Z mit b ergibt wieder $1 - c^*a$: $Z \cdot b = 1 - c^*a$. Dieser Ausdruck bedeutet jedoch, dass $Z \cdot b$ bei der Division durch a den Rest 1 lässt, eben weil es sich als ein Vielfaches von a $(-c^*a)$ plus 1 schreiben lässt. Schreiben wir dies mathematisch hin, sieht man schnell, dass wir damit das Behauptete gezeigt haben:

$$Z \cdot b \equiv 1 \bmod a$$

$Z = \frac{1-c^*a}{b} \bmod a$ ist also das multiplikativ Inverse zu $b \bmod a$.

Die Laufzeit ergibt sich genau wie beim euklidischen Algorithmus, da auch hier aus einem Paar a und b in jedem Schritt ein Paar b und $a \bmod b$ gemacht wird.

9.3 Der chinesische Restklassensatz

Frage 9.9: Wofür kann der chinesische Restklassensatz benutzt werden?
Im RSA-System wird mit Zahlen in $\mathbb{Z}_m = \{0, \ldots, m-1\}$ gerechnet, wobei das m sehr groß sein kann. Der chinesische Restklassensatz besagt nun, dass wir unter gewissen Voraussetzungen Rechnungen in $\mathbb{Z}_m$ auf Rechnungen in $\mathbb{Z}_{m(i)}$ für kleine $m(i)$ zurückführen können. Aus den Ergebnissen, die wir in $\mathbb{Z}_{m(i)}$ errechnet haben, lassen sich dann effizient die Ergebnisse aus $\mathbb{Z}_m$ berechnen.

Frage 9.10: Was besagt der chinesische Restklassensatz? Beweise ihn!
Es seien $m_1, \ldots, m_k$ natürliche Zahlen, die relativ prim zueinander sind, d.h. $\mathrm{ggT}(m_i, m_j) = 1$ für $i \neq j$. Ebenso seien k weitere natürliche Zahlen $a_1, \ldots, a_k$ gegeben. Dann hat das folgende Gleichungssystem *genau eine* Lösung $x \in \{0, \ldots, m_1 \cdot \ldots \cdot m_k - 1\}$:

$$\begin{array}{rcl} x & \equiv & a_1 \bmod m_1 \\ x & \equiv & a_2 \bmod m_2 \\ \vdots & \vdots & \vdots \\ x & \equiv & a_k \bmod m_k \end{array}$$

Im Prinzip bedeutet dieses Gleichungssystem, dass für fest gewählte $m_1, \ldots, m_k$ jede Zahl aus der Menge $\{0, \ldots, m_1 \cdot \ldots \cdot m_k - 1\}$ eindeutig durch ein Tupel $(a_1, \ldots, a_k)$ repräsentiert ist und umgekehrt. Für uns ist nun die Tatsache interessant, dass gewisse Rechnungen auch mit den Tupeln durchgeführt werden können. Ist x z.B. durch $(a_1, \ldots, a_k)$ repräsentiert und y durch $(b_1, \ldots, b_k)$, so kann anstatt $x \cdot y$ auch $(a_1 \cdot b_1, \ldots, a_k \cdot b_k)$ (alles jeweils $\bmod m_i$) berechnet werden.

Um zu zeigen, dass die Zuordnung zwischen x und $(a_1, \ldots, a_k)$ sowie anders herum eindeutig (also bijektiv) ist, müssen wir zwei Dinge zeigen:

1. Für jedes Tupel $(a_1, \ldots, a_k)$ existiert genau eine Lösung x.
2. Für jedes x existiert nur ein Tupel $(a_1, \ldots, a_k)$.

Letzteres ist schnell gezeigt. Es sei das Tupel $(a_1, \ldots, a_k)$ eine Repräsentation für x. Die i-te Stelle dieses Tupels entspricht der i-ten Zeile des obigen Gleichungssystems. Es gilt also $x \equiv a_i \bmod m_i$, d.h. x hinterlässt bei der Division durch m_i den Rest a_i. Wenn wir nun also versuchen würden, auch nur eines dieser a_i durch ein $a_i' \neq a_i$ (mit $0 \leq a_i, a_i' < m_i$) zu ersetzen, so wäre die i-te Zeile nicht mehr erfüllt, da x bei der Division durch m_i nur einen Rest hinterlassen kann. Also kann ein x nur durch genau ein Tupel $(a_1, \ldots, a_k)$ repräsentiert sein.

Als nächstes müssen wir nun Punkt 1 zeigen. Wir überlegen uns zuerst, wieviele Gleichungssysteme es gibt. Der Wert a_1 kann m_1 verschiedene Werte annehmen, der Wert a_2 kann m_2 verschiedene Werte annehmen, usw. Demnach gibt es $m_1 \cdot \ldots \cdot m_k$ verschiedene Gleichungssysteme. Wenn wir nun zeigen, dass jedes dieser Gleichungssysteme *mindestens* eine gültige Lösung besitzt, dann gilt aufgrund von Punkt 2 und $|\{0, \ldots, m_1 \cdot \ldots \cdot m_k - 1\}| = m_1 \cdot \ldots \cdot m_k$, dass jedes Tupel genau eine Lösung besitzt. Wir führen den Beweis, indem wir ein Verfahren angeben, welches zu einem Tupel $(a_1, \ldots, a_k)$ einen gültigen Wert x berechnet. Der Beweis der Existenz einer Lösung wird per Induktion über k geführt. Für $k = 1$ ist $a_1 \bmod m_1$ eine

Lösung. Es sei also für $k-1$ eine gültige Lösung L gegeben für die $0 \leq L \leq m_1 \cdot \ldots \cdot m_{k-1} - 1$ gilt und die das folgende Gleichungssystem erfüllt:

$$\begin{array}{rcl} x & \equiv & a_1 \bmod m_1 \\ \vdots & \vdots & \vdots \\ x & \equiv & a_{k-1} \bmod m_{k-1} \end{array}$$

Gesucht ist nun also eine Zahl L^*, die obiges Gleichungssystem erfüllt und für die zusätzlich $L^* \equiv a_k \bmod m_k$ gilt. Zuerst überlegen wir uns, wie in Abhängigkeit von L weitere Zahlen aussehen, die die ersten $k-1$ Gleichungen erfüllen. Diese müssen von der Form $L + t \cdot (m_1 \cdot \ldots \cdot m_{k-1})$ sein. Welches t erfüllt nun $L + t \cdot (m_1 \cdot \ldots \cdot m_{k-1}) \equiv a_k \bmod m_k$? Wir wählen t wie folgt:

$$\begin{array}{rcl} L + t \cdot (m_1 \cdot \ldots \cdot m_{k-1}) & \equiv & a_k \bmod m_k \\ t \cdot (m_1 \cdot \ldots \cdot m_{k-1}) & \equiv & a_k - L \bmod m_k \\ t & \equiv & (a_k - L) \cdot (m_1 \cdot \ldots \cdot m_{k-1})^{-1} \bmod m_k \end{array}$$

Das multiplikativ Inverse zu $(m_1 \cdot \ldots \cdot m_{k-1})$ existiert, da alle m_i und m_j für $i \neq j$ relativ prim zueinander sind und $(m_1 \cdot \ldots \cdot m_{k-1})^{-1}$ nichts anderes ist als $m_1^{-1} \cdot \ldots \cdot m_{k-1}^{-1}$. Bei dieser Wahl von t gilt dann:

$$\begin{array}{rcl} L + (a_k - L) \cdot (m_1 \cdot \ldots \cdot m_{k-1})^{-1} \cdot (m_1 \cdot \ldots \cdot m_{k-1}) & \equiv & a_k \bmod m_k \\ L + (a_k - L) & \equiv & a_k \bmod m_k \\ a_k & \equiv & a_k \bmod m_k \end{array}$$

Wenn wir t so wählen, wird also auch noch die letzte Forderung erfüllt. Zuletzt bleibt zu zeigen, dass L^* auch die Forderung $0 \leq L^* \leq m_1 \cdot \ldots \cdot m_k - 1$ erfüllt. Da t Modulo m_k gerechnet wird, kann es also höchstens $m_k - 1$ groß werden. Der Wert L kann höchstens den Wert $m_1 \cdot \ldots \cdot m_{k-1} - 1$ annehmen. Also gilt:

$$\begin{array}{rcl} 0 \leq L^* & \leq & L + (m_k - 1) \cdot (m_1 \cdot \ldots \cdot m_{k-1}) \\ & \leq & (m_1 \cdot \ldots \cdot m_{k-1} - 1) + (m_k - 1) \cdot (m_1 \cdot \ldots \cdot m_{k-1}) \\ & = & (m_1 \cdot \ldots \cdot m_{k-1}) + (m_k - 1) \cdot (m_1 \cdot \ldots \cdot m_{k-1}) - 1 \\ & = & m_k \cdot (m_1 \cdot \ldots \cdot m_{k-1}) - 1 \\ & = & m_1 \cdot \ldots \cdot m_{k-1} \cdot m_k - 1 \end{array}$$

9.4 Grundlagen aus der Zahlentheorie

Frage 9.11: Was ist $\mathbb{Z}_n$ und was ist $\mathbb{Z}_n^*$?
$\mathbb{Z}_n$ ist die Menge der Zahlen $\{0, \ldots, n-1\}$ und $\mathbb{Z}_n^*$ ist die Menge der Zahlen aus $\{1, \ldots, n-1\}$, die zu n teilerfremd sind und damit ein Inverses modulo n besitzen.

Frage 9.12: Wann heißen zwei Zahlen teilerfremd?
Zwei Zahlen a und b heißen teilerfremd, wenn es keine Zahl außer der 1 gibt, die beide Zahlen teilt, also $\mathrm{ggT}(a, b) = 1$. Es ist also jede Zahl teilerfremd zu einer Primzahl p (außer natürlich die Zahl p selbst).

Frage 9.13: Wie ist die Ordnung einer Zahl aus $\mathbb{Z}_n^*$ definiert?
Die Ordnung $ord_n(a)$ einer Zahl $a \in \mathbb{Z}_n^*$ ist definiert als das kleinste $i \geq 1$, so dass $a^i \equiv 1 \bmod n$ gilt. Die Ordnung gibt also an, wie oft eine Zahl mit sich selbst multipliziert werden muss (nämlich $(ord_n(a) - 1)$-mal), damit man das neutrale Element erhält.

Frage 9.14: Wann ist eine Zahl ein quadratischer Rest modulo n?
Eine Zahl $a \in \mathbb{Z}_n^*$ heißt quadratischer Rest modulo n, wenn es ein $x \in \mathbb{Z}_n^*$ gibt, so dass die Gleichung $x^2 \equiv a \bmod n$ erfüllt ist. Die Menge $QR(n)$ ist die Menge aller Elemente, die quadratischer Rest modulo n sind. Ist obige Gleichung erfüllt, gilt also $a \in QR(n)$. Wenn eine Zahl in der Menge $QR(n)$ enthalten ist, bedeutet dies also, dass es ein Element in $\mathbb{Z}_n^*$ gibt, welches mit sich selbst multipliziert a ergibt.

Frage 9.15: Wann heißt ein Element erzeugendes Element von $\mathbb{Z}_n^*$?
Ein Element $g \in \mathbb{Z}_n^*$ heißt erzeugendes Element von $\mathbb{Z}_n^*$, wenn $\mathbb{Z}_n^* = \{g^i \mid i \geq 0\}$ ist. Es kann also die ganze Menge $\mathbb{Z}_n^*$ erzeugt werden, indem g sukzessive mit sich selbst multipliziert wird. Bildlich gesprochen beginnen wir mit g und multiplizieren es mit sich selbst. Wir erhalten ein Ergebnis, welches wieder mit g multipliziert wird. Wenn wir diesen Vorgang unendlich fortsetzen, wird dabei jedes Elemente von $\mathbb{Z}_n^*$ mal als Ergebnis auftauchen. Da $\mathbb{Z}_n^*$ endlich ist, werden die Elemente in regelmäßiger Reihenfolge auftauchen, da wir irgendwann beim neutralen Element landen (dies muss ja auch erzeugt werden, da es in $\mathbb{Z}_n^*$ vorkommt), welches wieder mit g multipliziert wird, so dass wir wieder am Anfang sind.

Frage 9.16: Wie ist der Index einer Zahl aus $\mathbb{Z}_n^*$ definiert?
Der Index einer Zahl $a \in \mathbb{Z}_n^*$ ist stets bezüglich eines erzeugenden Elements definiert, da es mehrere erzeugende Elemente geben kann. Es sei also ein $a \in \mathbb{Z}_n^*$ und ein erzeugendes Element g gegeben. Dann ist der Index $index_g(a)$ definiert als das kleinste $i \geq 0$, so dass $g^i \equiv a \bmod n$ ist. Der Index gibt also an, wie oft ich das entsprechende erzeugende Element höchstens multiplizieren muss, damit ich das Element a erzeuge.

Frage 9.17: Wann heißt eine Zahl quadratfrei?
Eine Zahl n heißt quadratfrei, wenn in der Primfaktorzerlegung von n keine Primzahl mit einem Exponenten größer als 1 vorkommt. Die Zahl $21 = 3 \cdot 7$ ist also z.B. quadratfrei, während $50 = 2 \cdot 25 = 2 \cdot 5^2$ es nicht ist.

Frage 9.18: Was gibt die Carmichael-Funktion an? Welche Aussagen sind in der Zahlentheorie für die Carmichael-Funktion bekannt?
Die Carmichael-Funktion $\lambda(n)$ gibt das kleinste $i \geq 1$ an, so dass für alle Elemente $a \in \mathbb{Z}_n^*$ gilt, dass $a^i \equiv 1 \bmod n$ ist. Die folgenden Aussagen sind in der Zahlentheorie bekannt:

1. Es sei p eine ungerade Primzahl (also wohl $p \geq 3$) und $e \geq 1$. Dann gilt:
$$\lambda(p^e) = p^{e-1} \cdot (p-1)$$
2. $\lambda(2) = 1,\ \lambda(4) = 2$ und $\lambda(2^e) = 2^{e-2}$ für $e \geq 3$.
3. $\lambda(p_1^{\alpha_1} \cdot \ldots \cdot p_r^{\alpha_r}) = \mathrm{kgV}_{1 \leq i \leq r} \lambda(p_i^{\alpha_i})$, wobei alle p_i voneinander verschiedene Primzahlen seien, also $p_i \neq p_j$ für $i \neq j$.

Frage 9.19: Wieviele Elemente in $\mathbb{Z}_p^*$ sind quadratischer Rest, wenn p eine Primzahl ist?

Für eine Primzahl $p \geq 3$ sind genau die Hälfte aller Elemente in $\mathbb{Z}_p^*$ quadratischer Rest. Es gilt also:

$$|QR(p)| = \frac{1}{2}|\mathbb{Z}_p^*|$$

Frage 9.20: Was muss für den Exponenten e einer Zahl $a \in \mathbb{Z}_n^*$ gelten, wenn $a^e \equiv 1$ mod n ist?

Es sei ein $a \in \mathbb{Z}_n^*$ gegeben. Es gilt $a^e \equiv 1 \bmod n$, genau dann, wenn e ein Vielfaches der Ordnung $ord_n(a)$ von a ist:

$$a^e \equiv 1 \bmod n \;\Leftrightarrow\; e \text{ ist ein Vielfaches von } ord_n(a)$$

Die Richtung „⇐“ ist schnell gezeigt. Wir nehmen an, dass $e = k \cdot ord_n(a)$ gilt. Dann folgt:

$$a^{k \cdot ord_n(a)} \equiv (a^{ord_n(a)})^k \equiv 1^k \equiv 1 \bmod n$$

Nun zur Richtung „⇒“. Wir zerlegen e als $e = k \cdot ord_n(a) + j$ mit $0 \leq j < ord_n(a)$ und zeigen, dass wenn $a^{k \cdot ord_n(a)+j} \equiv 1 \bmod n$ gilt, dann $j = 0$ sein muss. Wir formen zuerst $a^e \equiv 1 \bmod n$ etwas um:

$$\begin{aligned}
a^e &\equiv 1 \bmod n \\
a^{k \cdot ord_n(a)+j} &\equiv 1 \bmod n \\
a^{k \cdot ord_n(a)} \cdot a^j &\equiv 1 \bmod n \\
(a^{ord_n(a)})^k \cdot a^j &\equiv 1 \bmod n \\
1 \cdot a^j &\equiv 1 \bmod n \\
a^j &\equiv 1 \bmod n
\end{aligned}$$

Da wir j als $0 \leq j < ord_n(a)$ gewählt haben, kann es aufgrund der Definition der Ordnung jedoch nur 0 sein, da die Ordnung das kleinste $i \geq 1$ ist, für die $a^i \equiv 1 \bmod n$ gilt.

Frage 9.21: Was besagt der kleine Satz von Fermat? Wie kann er verwendet werden?

Es sei eine Primzahl p gegeben. Dann gilt für alle $a \in \mathbb{Z}_p^*$ dass $a^{p-1} \equiv 1 \bmod p$ ist. Der kleine Satz ist ein notwendiges jedoch nicht hinreichende Kriterium für Primzahlen, d.h. er gilt für *alle* Primzahlen, jedoch gibt es auch Zahlen, die keine Primzahlen sind, für die er ebenfalls gilt.

Ein Beispiel. Wir wollen testen, ob 51 eine Primzahl ist. Damit sie überhaupt eine Primzahl sein kann, muss für alle $a \in \mathbb{Z}_{51}^*$ gelten: $a^{50} \equiv 1 \bmod 51$. Mit der Zahl 2 finden wir jedoch recht schnell eine Zahl, für die dies nicht gilt, da $2^{50} \equiv 4 \bmod 51$. Die Zahl $2 \in \mathbb{Z}_{51}^*$ ist also ein Zeuge dafür, dass 51 keine Primzahl ist. Unglücklicherweise gibt es jedoch Zahlen, die keine Primzahlen sind und für die es keine solche Zeugen gibt. Dies sind die sogenannten Carmichael-Zahlen.

Frage 9.22: Was ist eine Carmichael-Zahl? Wie können Carmichael-Zahlen charakterisiert werden bzw. welche Eigenschaften haben sie?

Eine Zahl $n \geq 2$ heißt Carmichael-Zahl, wenn n keine Primzahl ist und für alle $a \in \mathbb{Z}_n^*$ gilt $a^{n-1} \equiv 1 \bmod n$. Carmichael-Zahlen können wie folgt charakterisiert werden:

$$\text{Eine zusammengesetzte Zahl } n \text{ ist eine Carmichael-Zahl} \;\Leftrightarrow\; n-1 \text{ ist ein Vielfaches von } \lambda(n)$$

Die Richtung „⇐“ ist wieder leicht zu zeigen. Es gelte also $n-1 = k \cdot \lambda(n)$. Dann gilt für jedes $a \in \mathbb{Z}_n^*$:

$$a^{n-1} \equiv a^{k \cdot \lambda(n)} \equiv (a^{\lambda(n)})^k \equiv 1^k \equiv 1 \bmod n$$

Für die Richtung „⇒“ können wir also annehmen, dass n eine Carmichael-Zahl ist, d.h. dass für alle $a \in \mathbb{Z}_n^*$ gilt: $a^{n-1} \equiv 1 \bmod n$. Wir haben schon gezeigt, dass $n-1$ dann ein Vielfaches der Ordnung von a sein muss. Da $a^{n-1} \equiv 1 \bmod n$ für alle $a \in \mathbb{Z}_n^*$ gilt, muss $n-1$ also ein Vielfaches aller Ordnungen $ord_n(a)$ sein und damit ein Vielfaches des kleinsten gemeinsamen Vielfachen aller Ordnungen.

Desweiteren kann gezeigt werden, dass wenn n eine Carmichael-Zahl ist, n ungerade und quadratfrei ist.

9.5 Das Jacobi-Symbol

Frage 9.23: Wie ist das Jacobi-Symbol definiert?

Das Jacobi-Symbol ist wie folgt definiert:

1. Für eine Primzahl p und eine Zahl $a \in \mathbb{Z}_p^*$ ist das Jacobi-Symbol $\left(\frac{a}{p}\right)$ wie folgt definiert:

$$\left(\frac{a}{p}\right) = \begin{cases} +1, & \text{falls } a \in QR(p) \\ -1 & \text{falls } a \notin QR(p) \end{cases}$$

2. Für zusammensetzte, ungerade Zahlen $q \geq 3$ mit der Primfaktorzerlegung $q = q_1 \cdots q_k$ ist das Jacobi-Symbol $\left(\frac{a}{q}\right)$ von a bezüglich q (mit $\text{ggT}(a,q) = 1$) definiert als das Produkt der Jacobi-Symbole $\left(\frac{a}{q_i}\right)$ mit $1 \leq i \leq k$.

Dabei können im zweiten Fall auch Primfaktoren mehrfach vorkommen, d.h. es ist möglich, dass $q_i = q_j$ für $i \neq j$.

9.6 Der probabilistische Primzahltest von Solovay und Strassen

Frage 9.24: Beschreibe den probabilistischen Primzahltest von Solovay und Strassen!

Algorithmus 17 beschreibt den probabilistischen Primzahltest von Solovay und Strassen. Der Algorithmus erwartet als Eingabe die zu testende Zahl $n \geq 3$ und die Anzahl der Versuche k, die gemacht werden sollen, um n als zusammengesetzte Zahl zu entlarven. Wir werden nun auf die Schritte (1)-(4) im einzelnen eingehen:

(1) Als erstes wird geschaut, ob n eine gerade Zahl ist. Da jede Primzahl $p > 2$ ungerade ist und nur Eingaben $n \geq 3$ erwartet werden, kann n also keine Primzahl sein, wenn n gerade ist.

(2) Nachdem eine Zufallszahl a erzeugt wurde, wird der ggT von a und n berechnet. Ist der ggT von a und n ungleich 1, so kann n keine Primzahl sein, da es eine Zahl gibt, die sowohl a als auch n teilt. Andererseits heißt ein ggT von 1 noch lange nicht, dass n eine Primzahl ist, da z.B. der ggT von $6 = 2 \cdot 3$ und $35 = 5 \cdot 7$ auch 1 ist. Wenn in diesem Schritt nicht gestoppt wird, wissen wir zumindest, dass $a \in \mathbb{Z}_n^*$ gilt, was wir in den nächsten beiden Schritten voraussetzen müssen.

Algorithmus 17 Der probabilistische Primzahltest von Solovay und Strassen

if n ist gerade **then**
(1) Gebe „n ist zusammengesetzt“ aus und stoppe.
end if
for $i = 1$ **to** k **do**
Wähle eine Zufallszahl $a \in \{1, \ldots, n-1\}$ gemäß Gleichverteilung.
(2) Berechne $\mathrm{ggT}(a, n)$. Falls $\mathrm{ggT}(a, n) \neq 1$, gebe „$n$ ist zusammengesetzt“ aus und stoppe.
(3) Berechne $b \equiv a^{\frac{n-1}{2}} \bmod n$. Falls $b \notin \{-1, 1\}$, gebe „n ist zusammengesetzt“ aus und stoppe.
(4) Berechne das Jacobi-Symbol $c = \left(\frac{a}{n}\right)$. Falls $b \not\equiv c \bmod n$, gebe „n ist zusammengesetzt“ aus und stoppe.
end for
Gebe „n ist vermutlich eine Primzahl“ aus und stoppe.

(3) Bisher hat kein Test ergeben, dass n eine zusammengesetzte Zahl ist. Wir müssen also weiter testen und verwenden dazu eine verschärfte Variante des kleinen Satzes von Fermat. Zur Erinnerung: Der kleine Satz von Fermat besagt, dass für eine Primzahl p für alle $a \in \mathbb{Z}_p^*$ gilt: $a^{p-1} \equiv 1 \bmod p$. Die 1 hat zwei Wurzeln, nämlich -1 und 1. Mit $a^{\frac{n-1}{2}} (= \sqrt{a^{n-1}})$ testen wir also quasi implizit, ob der kleine Satz von Fermat gilt, indem wir schauen, ob eine der beiden Wurzeln herauskommt. Ist dies nicht der Fall, kann n keine Primzahl sein. Ansonsten wird noch ein weiterer Test gemacht.

(4) Dieser Schritt ist quasi unsere letzte Chance, für eine zusammengesetzte Zahl n zu erkennen, dass sie zusammengesetzt ist. Wir nutzen eine Eigenschaft des Jacobi-Symbols für eine Zahl a und eine Primzahl p aus, die im Folgenden auch noch bewiesen wird. Für jede Primzahl $p > 2$ gilt für alle $a \in \mathbb{Z}_p^*$:

$$a^{\frac{p-1}{2}} \equiv \left(\frac{a}{p}\right) \bmod p$$

Zwar gibt es auch zusammengesetzte Zahlen für die dies gilt, jedoch werden wir zeigen, dass diese Eigenschaften wirklich nur für *alle* $n \in \mathbb{Z}_n^*$ gilt, wenn n eine Primzahl ist. Desweiteren kann die Anzahl der Elemente in der Menge $\mathbb{Z}_n^*$, für die dies gilt, wenn n keine Primzahl ist, abgeschätzt werden, wodurch sich die Irrtumswahrscheinlichkeit ergeben wird.

Wir werden nun erst die verwendeten mathematischen Eigenschaften beweisen und danach die Laufzeit und die Irrtums- bzw. Erfolgswahrscheinlichkeit abschätzen.

Frage 9.25: Zeige, dass die Verschärfung des kleinen Satzes von Fermat gilt!
Wir wollen also zeigen, dass für jede Primzahl $p > 2$ für jedes $a \in \mathbb{Z}_p^*$ gilt:

$$a^{\frac{p-1}{2}} \equiv \left(\frac{a}{p}\right) \bmod p$$

Zuerst einmal gilt nach dem kleinen Satz von Fermat $a^{p-1} \equiv 1 \bmod p$ für alle $a \in \mathbb{Z}_p^*$, wenn p eine Primzahl ist. Der Ausdruck $a^{(p-1)/2}$ ist also quasi die Wurzel, wovon es bezüglich der 1 nur

zwei gibt: -1 und 1. Das heißt, für eine Primzahl p kann $a^{(p-1)/2}$ auch nur den Wert -1 oder 1 ergeben. Das Jacobi-Symbol auf der rechten Seite gibt den Wert 1, wenn a ein quadratischer Rest modulo p ist, also $a \in QR(p)$. Es reicht also zu zeigen, dass $a^{(p-1)/2} \equiv 1 \bmod p$ ist, genau dann, wenn a ein quadratischer Rest bezüglich p ist. Daraus folgt dann automatisch, dass $a^{(p-1)/2} \equiv -1 \bmod p$ gilt, genau dann, wenn a kein quadratischer Rest bezüglich p ist. Wir zeigen nun also:

$$a \text{ ist quadratischer Rest modulo } p \;\Leftrightarrow\; a^{\frac{p-1}{2}} \equiv 1 \bmod p$$

Wir beginnen mit der Richtung „⇒“. Es sei a also ein quadratischer Rest bezüglich p, d.h. es gibt ein x mit $a \equiv x^2 \bmod p$. Dann gilt jedoch:

$$a^{\frac{p-1}{2}} \equiv (x^2)^{\frac{p-1}{2}} \equiv x^{p-1} \equiv 1 \bmod p$$

Nun zur Richtung „⇐“. Wir gehen also davon aus, dass $a^{\frac{p-1}{2}} \equiv 1 \bmod p$ gilt.

$$1 \bmod p \equiv a^{\frac{p-1}{2}} \equiv \left(g^{index_g(a)}\right)^{\frac{p-1}{2}} \equiv g^{index_g(a)\cdot\frac{p-1}{2}}$$

Wenn $g^{index_g(a)\cdot(p-1)/2} \equiv 1 \bmod p$ gilt, muss $index_g(a) \cdot (p-1)/2$ also ein Vielfaches der Ordnung von g sein. Es gilt $ord_p(g) = p - 1$. Also gilt insgesamt:

$$\begin{aligned} index_g(a) \cdot \frac{p-1}{2} &= k \cdot (p-1) \\ \Leftrightarrow index_g(a) &= 2 \cdot k \end{aligned}$$

Es gilt also, dass a durch g erzeugt wird, wenn für ein k gilt:

$$g^{2k} \equiv \left(g^k\right)^2 \equiv a \bmod p$$

Damit ist a also ein quadratischer Rest modulo p, d.h. $a \in QR(p)$.

Frage 9.26: Welche Laufzeit hat der Algorithmus von Solovay und Strassen?

Für ein konstantes k hat der Algorithmus von Solovay und Strassen eine Laufzeit von $O(k \log n)$. Die Laufzeit wird dabei durch die k Schleifendurchläufe dominiert, wobei jeder Schleifendurchlauf höchstens Zeit $O(\log n)$ benötigt. Dies liegt daran, dass $a < n$ gewählt wird, also die Berechnung des ggTs z.B. durch $O(\log(a+n)) = O(\log(n+n)) = O(\log(2n)) = O(\log n + \log 2) = O(\log n)$ abgeschätzt werden kann. Ebenso kann die Laufzeit zur Berechnung des Jacobi-Symbols von a und n abgeschätzt werden. Der Wert $a^{(n-1)/2}$ kann mit Hilfe des Algorithmus zum effizienten Potenzieren berechnet werden. Auch hier ergibt sich eine Laufzeit von $O(\log n)$, da die Laufzeit des Algorithmus durch den Exponenten bestimmt wird und gilt $O(\log((n-1)/2)) = O(\log n)$.

Frage 9.27: Wie groß ist die Erfolgswahrscheinlichkeit des Algorithmus von Solovay und Strassen?

Zuerst einmal zeigen wir, dass der Algorithmus für Primzahlen p nichts falsch macht. Dabei kann man sich den Algorithmus wie ein Sieb vorstellen, welches in jedem Schritt versucht, zusammengesetzte Zahlen auszufiltern. Für Zahlen, die durch das Sieb fallen, wird zum Schluss die Ausgabe „n ist vermutlich eine Primzahl gemacht“. Alle Primzahlen $p > 2$ sind ungerade

und werden daher Schritt (1) passieren. Danach wird der ggT von einer Zufallszahl a und von p berechnet. Da eine Primzahl keinen Teiler $2 \leq t \leq p-1$ hat, wird eine Primzahl auch diesen Test passieren. Ebenso haben wir auch schon gezeigt, dass eine Primzahl Schritt (3) und (4) bestehen wird.

Doch wie groß ist nun die Wahrscheinlichkeit, dass eine zusammengesetzte Zahl n nicht erkannt wird? Mindestens die Hälfte aller zusammengesetzten Zahlen wird in Schritt (1) erkannt. Wir überlegen uns nun, wo unser Sieb in Schritt (2)-(4) am „engmaschigsten“ ist, d.h. wo die meisten Zahlen gefiltert werden. Bedingung (3) und (4) können nur gelten, wenn (2) erfüllt ist, wobei Schritt (4) eine Verfeinerung von Schritt (3) ist. Also ist Schritt (4) die gesuchte Stelle. Um zu wissen, wie groß unsere Erfolgswahrscheinlichkeit ist, müssen wir uns also überlegen, wie groß die Wahrscheinlichkeit ist, dass zusammengesetzte Zahlen auch im letzten Schritt nicht erkannt werden. Dazu führen wir nun die Menge $E(n)$ ein, welche wie folgt definiert ist:

$$E(n) = \{a \in \mathbb{Z}_n^* \mid a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right) \mod n\}$$

Die Menge $E(n)$ kann also als die Menge der Zahlen betrachtet werden, die die Testzahl n auch im vierten Schritt nicht als zusammengesetzt entlarven. Um die Größe der Menge $E(n)$ bezüglich der Menge $\mathbb{Z}_n^*$ abzuschätzen, benötigen wir zuerst eine weitere Aussage:

$$E(n) = \mathbb{Z}_n^* \;\Leftrightarrow\; n \text{ ist prim.}$$

Die Richtung „$\Leftarrow$“ folgt aus der folgenden Aussage für Primzahlen p, die zuvor schon bewiesen wurde:

$$\forall a \in \mathbb{Z}_p^* : a^{\frac{p-1}{2}} \equiv \left(\frac{a}{p}\right) \mod p$$

Wir gehen nun davon aus, dass n eine zusammengesetzte Zahl ist, woraus folgt, dass $E(n) \neq \mathbb{Z}_n^*$ gelten muss. Also hat die Menge $E(n)$ in diesem Fall weniger Elemente als die Menge $\mathbb{Z}_n^*$. Wir werden nun zeigen, dass die Menge $E(n)$ höchstens halb soviele Elemente wie $\mathbb{Z}_n^*$ enthalten kann, dass also $|E(n)| \leq |\mathbb{Z}_n^*|/2$ gilt. Dies zeigen wir, indem wir beweisen, dass die Menge $E(n)$ eine echte Untergruppe der Gruppe $(\mathbb{Z}_n^*, \cdot)$ ist. Weiter benutzen wir den Satz von Lagrange, der besagt, dass die Ordnung einer Untergruppe U ein Teiler der Ordnung der Gruppe G ist. Der größtmögliche Teiler ist $|U| = |G|$. Dies ist in unserem Fall jedoch nicht möglich, da $E(n) \neq \mathbb{Z}_n^*$ gilt. Der nächstgrößte mögliche Teiler kann also höchstens $|\mathbb{Z}_n^*|/2$ sein, was wir ja auch zeigen wollen. Es bleibt also nur zu zeigen, dass $E(n)$ eine Untergruppe bezüglich der Multiplikation modulo n ist. Dazu reicht es zu zeigen, dass für $a, b \in U$ auch $ab \in U$ ist. Es gelte also $a, b \in U$ bzw. in unserem Fall $a, b \in E(n)$:

$$(ab)^{\frac{n-1}{2}} = a^{\frac{n-1}{2}} \cdot b^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right)\left(\frac{b}{n}\right) \mod n = \left(\frac{ab}{n}\right) \mod n$$

Der letzte Schritt folgt dabei aus einer Rechenregel für Jacobi-Symbole, die besagt:

$$\left(\frac{ab}{n}\right) = \left(\frac{a}{n}\right)\left(\frac{b}{n}\right)$$

Wir wissen nun also, dass die Menge der für den Test ungünstigen Zufallszahlen a höchstens halb so groß ist, wie die Menge $\mathbb{Z}_n^*$. Wenn aus $\mathbb{Z}_n^*$ also zufällig eine Zahl a gezogen wird, dann ist die Wahrscheinlichkeit eine ungünstige zu ziehen, also höchstens $\frac{1}{2}$. Damit eine Zahl in k Schritten nicht als zusammengesetzt entlarvt wird, müssen wir als k-mal ungünstig wählen,

was mit einer Wahrscheinlichkeit von $\left(\frac{1}{2}\right)^k$ geschieht. Die Misserfolgwahrscheinlichkeit beträgt also höchstens $\left(\frac{1}{2}\right)^k$, woraus folgt, dass die Erfolgswahrscheinlichkeit mindestens $1 - \left(\frac{1}{2}\right)^k$ ist.

Abbildungsverzeichnis

1 Beispiele für Tree-, Forward-, Back- und Cross-Kanten 5
2 Beispiel für starke Zusammenhangskomponenten 7
3 Der Grund, warum $v_1 \notin V_1 \cup \cdots \cup V_{i-1}$ gilt 8
4 Die Situation mit dem Knoten v^* 10
5 Warum zwei ZZKs V_1 und V_2 maximal einen Schnittpunkt gemeinsam haben können 11
6 Beispiel für Schnittpunkte und ZZKs 12
7 Schematische Darstellung für die Richtung „$2 \Rightarrow 1$“ 13
8 Fall 2 der Richtung „$1 \Rightarrow 2$“ 15
9 Prinzip der Buchhaltermethode 20
10 Der NFA A zur Erkennung des Musters $P = (p_1, \ldots, p_m)$ 25
11 Die Idee des Algorithmus von Hirschberg 34
12 Beispiel für eine Branch-and-Bound Zerlegung 41
13 Verschmelzung der Aufgaben mit $t_i \leq \varepsilon L$ 56
14 Die immer größer werdenden Rechtecke über $[0, t_{sum}]$ 61
15 Skizze der Funktion CHOICE für $y = 0$ und $y = 1$ 82
16 Umwandlung einer symmetrischen Kante 83
17 Beispiele für FV-Wege mit Vorwärts- und Rückwärtskanten 84
18 Ein Netzwerk bei dem eine exponentielle Laufzeit möglich ist 87
19 Potenzial $pot(v)$ eines Knotens v 97

Tabellenverzeichnis

1 Vergleich zwischen Algorithmus A und B für verschiedene Klausellängen k . . 70

Liste der Algorithmen

1 Der rekursive Algorithmus zur Berechnung von $\mathbf{ZZK}(v, u)$ 16
2 Der Query-Algorithmus für das KMP-Verfahren 27
3 Der Algorithmus zur Berechnung der Fehlerkanten 28
4 Der Algorithmus des Upper Bound Modul 43
5 Der Approximationsalgorithmus für das metrische TSP mit Güte 2 50
6 Der Approximationsalgorithmus für das metrische TSP mit Güte $\frac{3}{2}$ 51
7 Das Approximationsschema für das Rucksackproblem mit Güte $1 + \varepsilon$ 53
8 Algorithmus der Methode der bedingten Wahrscheinlichkeiten 72
9 BFS-Markierungsalgorithmus auf dem Restgraphen 91
10 Konstruktion eines Sperrflusses ψ für ein Niveau-Netzwerk 94
11 Berechnung eines maximalen Flusses mit dem Algorithmus von Dinic 95
12 Forward-Backward-Propagation 98
13 Forward(v) 99
14 Effizienter Algorithmus zur Berechnung von $x^n \bmod m$ 104
15 Der euklidische Algorithmus zur Berechnung von $\mathrm{ggT}(a, b)$ 106
16 Erweiterter euklidischer Algorithmus $\mathrm{INV}(a, b)$ 107
17 Der probabilistische Primzahltest von Solovay und Strassen 113

Literatur

[Sieling 2002] Sieling, Detlef (2002). *Skript Effiziente Algorithmen*.

Index

(1+1)EA, 75
1-Baum, 44

Algorithmus
    von Christofides, 50
    von Hirschberg, 34
    von Needleman und Wunsch, 33
Alignment
    globales, 33
    lokales, 35
Antisymmetrie, 6
Approximationsschema
    echt polynomielles, 52
    polynomielles, 52
Äquivalenzrelation, 6

Back-Kante, 5
Backward-Algorithmus, 39
BFS-Markierungsalgorithmus, 91
Black-Box-Optimierung, 75
Branching Modul, 40, 41

Chiffrierfunktion, 101
Cross-Kante, 5

Dechiffrierfunktion, 101
Decodierungsproblem, 37
dynamische Programmierung, 29

Eulerkreis, 49

f-basierte Partition, 76
Fibonacci-Zahlen, 29
Flaschenhals, 86
Flschenhals, 88
Fluss, 83
    maximaler, 84
    Wert eines, 84
    zulässiger, 84
flussvergrößernder Weg, 84
Forward-Algorithmus, 38
Forward-Kante, 5
FPTAS, 52
FVMIN, 88

Güte, 49

Harmonische Reihe, 78
Hidden Markoff Modell, 36

Intervalltyp, Probleme des, 30

Kapazitätsfunktion, 83
Kirchhoff-Regel, 83
KMP-Algorithmus, 25
Kreis, 4
Kryptogramm, 101

Learning Modul, 45
lineare Funktion, 79
Lower Bound Modul, 40
Lückenstrafe, 33

Markoffkette, 36
Matching, 50
Multigraph, 49

Nachrichtenmenge, 101
Netzwerk, 83
Niveau-Netzwerk, 92
Nullfluss, 84

ONEMAX, 77
Optimierungsproblem
    ganzzahlig, 64
    ganzzahliges, 65

partielle Ordnung, 5
pseudopolynomielle Laufzeit, 31
PTAS, 52

Quelle, 83

Reflexivität, 5
Relaxation, 40
Restgraph, 87
Rucksackproblem
    Branch-and-Bound, 42
    dyn. Programmierung, 30

saturierte Kante, 93
Schlüsselmenge, 101
Schnittpunkt, 10
Score-Funktion, 33

Search Modul, 40, 42
Senke, 83
Sperrfluss, 93
Strafkosten, 46
Suffix, guter, 26
Symmetrie, 7

Teilgraph, 9
Tiefensuche, 4
    Laufzeit, 5
topologische Nummerierung, 5
Transitivität, 6
Traveling Salesman Problem, *siehe* TSP
Tree-Kante, 5
TSP
    asymmetrisches, 47
    Branch-and-Bound, 43
    metrisches, 49
    symmetrisches, 44

unimodale Funktion, 78
Upper Bound Modul, 40

Viterbi-Algorithmus, 37

Zusammenhangskomponente
    Berechnung starker, 7
    starke, 6