VORLESUNG EAKT

MIT SCHWERPUNKT

EFFIZIENTE ALGORITHMEN

Sommersemester 2005

Version v. 30.03.2007

Dozent:

Thomas Hofmeister

Universität Dortmund

Lehrstuhl Informatik 2

44221 Dortmund

Vorwort
Das Skript und die Vorlesung werden begleitet von einer Webseite, auf der es ergänzende Materialien, eine Fehlerliste etc. geben wird.
Die URL: `http://Ls2-www.cs.uni-dortmund.de/lehre/sommer2005/EAKT`
Insbesondere wird die Webseite zusätzlich zum Skript in der Semesteranfangsversion auch stets eine Version enthalten, die vorlesungsbegleitend überarbeitet wird. Dort kann ich z.B. Verbesserungsvorschläge einarbeiten, bekannt gewordene Fehler beseitigen, etc. Deshalb freue ich mich wie immer über Hinweise zu Fehlern und/oder Verbesserungsvorschläge, die gerne an meine email-Adresse gesendet werden können. (Siehe die obige Webseite.)

Inhaltsverzeichnis

1 Übersicht

Software kann nicht besser sein als der ihr zugrunde liegende Algorithmus zur Lösung des Problems. Der Entwurf effizienter Algorithmen ist also unbestritten ein zentrales Teilgebiet der Informatik. Methoden zum Entwurf effizienter Algorithmen finden Anwendung in vielen Bereichen, z. B. Datenstrukturen, Logischer Entwurf, Kryptographie (Datenschutz), Künstliche Intelligenz.
Wie entwirft man nun für ein gegebenes Problem einen effizienten Algorithmus? Notwendig sind grundlegende Kenntnisse des Bereichs, aus dem das Problem stammt. Darüber hinaus ist der Entwurf effizienter Algorithmen eine Kunst, die nicht einer erlernbaren Methode folgt. Zum Entwurf eines effizienten Algorithmus gehört also auch eine Portion „Gefühl für das Problem“ und eine Portion „Intuition“. Wozu dient dann eine Vorlesung „Effiziente Algorithmen“?
Erfolgreiche Künstlerinnen und Künstler haben nicht nur die richtige Intuition, sie beherrschen darüber hinaus das benötigte „Handwerk“ meisterhaft. Die Vorlesung kann und soll dieses Handwerkszeug vermitteln und an wichtigen Problemen erproben. Vorausgesetzt werden die Grundvorlesungen, insbesondere die Vorlesung DAP2. Hier wird der Schwerpunkt nicht im Entwurf weiterer Datenstrukturen liegen, sondern in Entwurfsmethoden für Algorithmen. Dazu werden Methoden für die Bewertung der Effizienz eines Algorithmus, insbesondere für die Abschätzung der (sequentiellen) Rechenzeit, vorgestellt. Die Behandlung effizienter Algorithmen für Parallelrechner muss einer Spezialvorlesung vorbehalten bleiben.
Die für diese Vorlesung ausgewählten Probleme haben viele Anwendungen in der Praxis und sind außerdem gut geeignet, um allgemeine Entwurfsmethoden für effiziente Algorithmen zu behandeln. Dabei ist zu beachten, dass die Probleme oft nicht direkt „in der realen Welt vorkommen“; Algorithmen für diese Probleme sind jedoch wichtige Unterprogramme in effizienten Algorithmen für reale Probleme.
Das Skript gliedert sich wie folgt. Wir beginnen mit Algorithmen auf Graphen. Graphen werden auf Zusammenhang, starken Zusammenhang und zweifachen Zusammenhang getestet. Außerdem berechnen wir kürzeste Wege in Graphen. Die meisten effizienten Algorithmen zur Lösung von Graphproblemen benutzen die in diesem Kapitel vorgestellten Algorithmen als Subroutinen.
Im nachfolgenden Kapitel stellen wir Beispiele für die Analyse von Algorithmen vor, darunter eine Datenstruktur für das UNION-FIND-Problem. Wir stellen dort auch das einfachste Problem aus dem Bereich der Mustererkennung, das String Matching, vor. Es wird gezeigt, dass Ideen zum Entwurf nichtdeterministischer endlicher Automaten zu effizienten Algorithmen führen können, obwohl Algorithmen deterministisch arbeiten und obwohl es für das Problem keinen deterministischen endlichen Automaten mit wenigen Zuständen gibt.
Die anschließenden beiden Kapitel beschäftigen sich mit zwei Algorithmentypen, die sehr modern sind, nämlich einerseits Approximationsalgorithmen, andererseits randomisierte Algorithmen.
Bei der Lösung von Optimierungsproblemen ist man häufig mit guten, aber nicht optimalen Lösungen zufrieden. Heuristische Methoden liefern oft in kurzer Rechenzeit ziem-

lich gute Ergebnisse; $(1+\varepsilon)$–Approximationsalgorithmen liefern stets Ergebnisse, deren Wert bei Minimierungsproblemen höchstens das $(1+\varepsilon)$-fache des optimalen Wertes ist. Für das Problem des Handlungsreisenden gibt es in wichtigen Spezialfällen polynomielle 3/2–Approximationsalgorithmen. Für das Rucksackproblem gibt es dagegen für jedes $\varepsilon > 0$ polynomielle $(1+\varepsilon)$-Approximationsalgorithmen und damit ein sogenanntes polynomielles Approximationsschema.
Heutzutage sind viele Algorithmen nicht deterministisch, sondern probabilistisch, weil diese häufig schneller und einfacher zu implementieren sind. Im Kapitel über randomisierte Algorithmen stellen wir einige typische Algorithmen vor.
Anschließend widmen wir uns der Berechnung von maximalen Flüssen in Netzwerken. Viele Probleme lassen sich als Flussproblem umformulieren. Hier werden wir die Entwicklung effizienter Algorithmen „historisch“ nachvollziehen. Wir stellen einige Algorithmen vor, wobei jeder neue Algorithmus auf dem vorherigen Algorithmus aufbaut und diesen mit Hilfe neuer Erkenntnisse verbessert. Diese Erkenntnisse beinhalten teilweise neue Datenstrukturen, und teilweise bestehen sie in einer verbesserten Einsicht in die Struktur des Problems. Während die frühen Algorithmen sehr anschaulich die bereits berechneten Flüsse verbessern, gehen die neueren Algorithmen mit abstrakteren Operationen vor. Dies folgt aus besseren Einsichten in die Problemstruktur.
In Kapitel 7 behandeln wir die aus Sicht der Informatik wichtigen Probleme der Zahlentheorie: Berechnung von Potenzen, Berechnung des größten gemeinsamen Teilers, Berechnung von Inversen modulo m, Berechnung des Jacobi-Symbols und einen Primzahltest. Dies sind genau die Probleme, deren Lösung notwendig und hinreichend zur Implementierung des bekanntesten Public–Key–Kryptosystems, des RSA–Verfahrens, sind. Für alle Probleme mit Ausnahme des Primzahltests werden effiziente deterministische Algorithmen entworfen. Außerdem wird ein effizienter probabilistischer Primzahltest vorgestellt. Dieser Algorithmus kann falsche Ergebnisse liefern, die Wahrscheinlichkeit dafür kann allerdings so klein gemacht werden, dass der Primzahltest für viele Anwendungen geeignet ist.
Die Leserin und der Leser sollten im Auge behalten, dass die Erkenntnisse beim Entwurf spezieller Algorithmen für spezielle Probleme stets auch dazu dienen, allgemeine Erkenntnisse für den Entwurf effizienter Algorithmen zu gewinnen.
Die Vorlesung folgt keinem Lehrbuch. Daher dient die folgende Liste von Lehrbüchern nur als Orientierung, um andere Darstellungen der hier vorgestellten Algorithmen oder weitere effiziente Algorithmen zu finden.

Literatur

- A.V. Aho, J.E. Hopcroft, J.D. Ullman: The design and analysis of computer algorithms, Addison–Wesley, 1974

- A.V. Aho, J.E. Hopcroft, J.D. Ullman: Data structures and algorithms, Addison–Wesley, 1983

- R.K. Ahuja, T.L. Magnanti, J.B. Orlin: Network flows. Theory, Algorithms and Applications, Prentice-Hall, 1993

- J. Bentley: Programming pearls, Addison–Wesley, 1986

- N. Christofides: Graph theory: an algorithmic approach, Academic Press, 1975

- T.H. Cormen, C.E. Leiserson, R.L. Rivest: Introduction to algorithms, MIT Press, 1990

- R.L. Graham, D.E. Knuth, O. Patashnik: Concrete mathematics: A foundation for computer science, Addison–Wesley, 1989

- D.E. Knuth: The art of computer programming, Band 1–3, Addison-Wesley, 1981. Neuauflage 1997/98.

- D.C. Kozen: The design and analysis of algorithms, Springer, 1991

- J.v. Leeuwen (Hrsg.): Handbook of theoretical computer science, Vol.A: Algorithms and complexity, MIT Press, 1990

- S. Martello, P. Toth: Knapsack problems, Wiley, 1990

- K. Mehlhorn: Data structures and algorithms, Band 1-3, Springer, 1984

- R. Motwani, P. Raghavan: Randomized Algorithms, Cambridge University Press, 1995

- T. Ottmann, P. Widmayer: Algorithmen und Datenstrukturen, BI, 1990

- C.H. Papadimitriou, K. Steiglitz: Combinatorial optimization: algorithms and complexity, Prentice-Hall, 1982

- E.M. Reingold, J. Nievergelt, N. Deo: Combinatorial algorithms: theory and practice, Prentice-Hall, 1977

- R. Sedgewick: Algorithms, Addison-Wesley, 1983

- I. Wegener: Effiziente Algorithmen für grundlegende Funktionen, Teubner, 1989

2 Grundlegende Graphalgorithmen

Es gibt kaum ein Gebiet der Informatik, in dem nicht Graphen zur Veranschaulichung benutzt werden. Dies liegt an der „Universalität“ von Graphen, paarweise Beziehungen zwischen Objekten darzustellen. Grundlegende Algorithmen auf ungerichteten Graphen sind somit die Basis vieler effizienter Algorithmen.
Daher wiederholen wir in Kapitel 2.1 Eigenschaften der DFS-Traversierung. Insbesondere können mit dem DFS-Ansatz die Zusammenhangskomponenten in ungerichteten Graphen berechnet werden. Das Analogon für gerichtete Graphen, die so genannten starken Zusammenhangskomponenten, werden in Kapitel 2.2 mit einem „doppelten DFS-Ansatz“ berechnet. Um Ausfälle von Komponenten kompensieren zu können, sollen Netzwerke stärker als „einfach“ zusammenhängend sein. Wir definieren in Kapitel 2.3, wann ungerichtete Graphen „zweifach“ zusammenhängend sind und zeigen, wie mit einem „modifizierten“ DFS-Ansatz die zweifach zusammenhängenden Komponenten eines Graphen effizient berechnet werden können. Schließlich stellen wir in Kapitel 2.5 Algorithmen zur Berechnung kürzester Wege in bewerteten Graphen vor.
Die in Kapitel 2 vorgestellten Algorithmen sind nicht nur grundlegend für das ganze Gebiet „Effiziente Algorithmen“. Sie sind alle einfach zu implementieren und es bereitet keine Schwierigkeiten, ihre Rechenzeiten zu bestimmen. Die Schwierigkeit liegt aber (mit Ausnahme von Kapitel 2.1) darin, zu zeigen, dass die Algorithmen die ihnen gestellte Aufgabe wirklich lösen, also im Korrektheitsbeweis.

2.1 Wiederholung: Depth-First Search

Die Idee der Tiefensuche (Depth-First Search) ist es, von einem beliebigen Knoten aus alle erreichbaren Knoten zu suchen und zu markieren. Von jedem erstmals erreichten Knoten wird direkt eine Tiefensuche gestartet. Wenn kein weiterer Knoten erreicht werden kann und noch nicht alle Knoten markiert sind, wird von einem noch nicht markierten Knoten die Tiefensuche neu gestartet. Die Suche endet, wenn von jedem Knoten aus eine Tiefensuche gestartet wurde. Bei der Tiefensuche vom Knoten v aus werden alle an v anliegenden Kanten (falls der Graph ungerichtet ist) oder alle von v ausgehenden Kanten (falls der Graph gerichtet ist) in beliebiger Reihenfolge untersucht. Charakteristikum der Tiefensuche ist, dass für jede Kante $\{v, w\}$ (oder (v, w)) erst die Tiefensuche von w aus gestartet wird (falls nicht schon früher geschehen), bevor die nächste Kante $\{v, w'\}$ (oder (v, w')) untersucht wird.
Die DFS-Traversierung eines ungerichteten Graphen liefert folgende Informationen. Wenn ein Knoten zum ersten Mal behandelt wird, erhält er seine DFS-Nummer. Die DFS-Nummern werden fortlaufend vergeben und ergeben für viele Algorithmen eine hilfreiche Knotennummerierung. Zu beachten ist, dass es zu einem Graphen $G = (V, E)$ nicht nur eine DFS-Nummerierung gibt. Wir nehmen für die DFS-Traversierung an, dass V als Liste oder Array gegeben ist und für jeden Knoten eine Adjazenzliste gegeben ist. Die DFS-Nummerierung hängt von der gegebenen „Ordnung“ auf V und in den Adjazenzlisten ab. Alle DFS-Nummerierungen liefern jedoch nützliche Informationen.
Für jede ungerichtete Kante $\{v, w\}$ wird eine Richtung ((v, w) oder (w, v)) ausgewählt,

je nachdem, ob die Kante zuerst in der Adjazenzliste von v oder in der Adjazenzliste von w gefunden wurde. Wenn wir die Richtung (v, w) wählen, unterscheiden wir, ob wir die Tiefensuche von w aus bereits begonnen haben (w hat eine DFS-Nummer) oder ob wir w das erste Mal „finden“. Im zweiten Fall wird (v, w) T-Kante (Treekante, Baumkante) und im ersten Fall B-Kante (Backkante, Rückwärtskante). Für einen zusammenhängenden Graphen bilden die T-Kanten einen Baum auf V (daher der Name), im Allgemeinen bilden sie einen Wald, dessen Teilbäume die einfachen Zusammenhangskomponenten darstellen. Während die Einteilung der Kanten in T-Kanten und B-Kanten nicht eindeutig ist, ist die Einteilung von G in Zusammenhangskomponenten eindeutig. Der ungerichtete Graph ist genau dann kreisfrei, wenn die Menge B leer bleibt. Die DFS-Traversierung betrachtet jede Kante zweimal ($v \in Adj(w), w \in Adj(v)$) und benötigt bei der Behandlung einer Kante konstante Zeit. Daher ist die DFS-Traversierung auf ungerichteten Graphen mit n Knoten und m Kanten in linearer Zeit $O(n+m)$ möglich. Bei gerichteten Graphen ist die Symmetrie „$v \in Adj(w) \Leftrightarrow w \in Adj(v)$“ aufgehoben. Wir könnten den Algorithmus zur DFS-Traversierung unverändert übernehmen. Die Menge der T-Kanten stellt weiterhin einen Wald dar. Da wir für gerichtete Graphen gar nicht definiert haben, wann sie zusammenhängend sind, haben die Bäume des Waldes keine Semantik. Darüber hinaus liefert uns die Einteilung in T- und B-Kanten kein notwendiges und hinreichendes Kriterium, um zu entscheiden, ob der Graph kreisfrei ist. Wir nehmen daher eine feinere Kanteneinteilung vor. Die von einem Knoten v aus gestartete Tiefensuche umfasst zeitlich ein Intervall $I(v)$. In Abbildung 2.1.1 haben wir die möglichen Beziehungen zwischen $I(v)$ und $I(w)$ dargestellt.

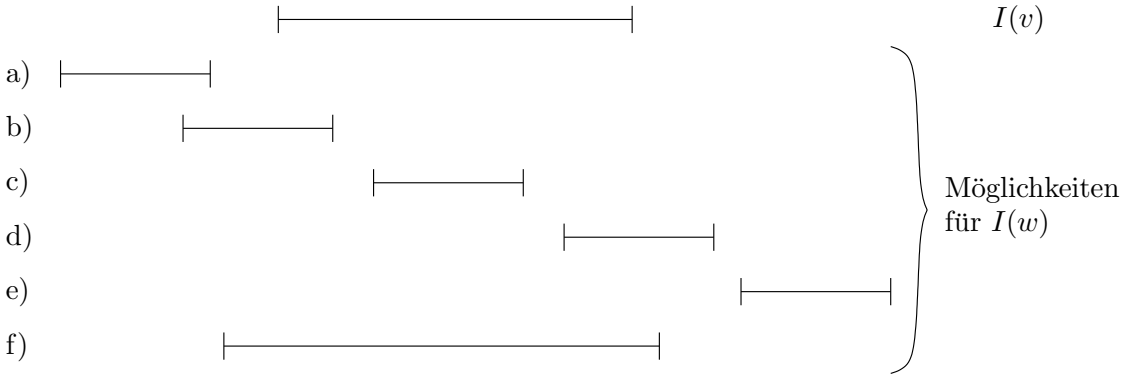


Abbildung 1: Zeitlicher Verlauf der Tiefensuche.

Das Besondere der Tiefensuche ist, dass die Fälle b) und d) unmöglich sind. Wird innerhalb von DFS(v) die Tiefensuche von w aus aufgerufen, wird DFS(w) vor DFS(v) beendet. Dies führt zu folgender Eingruppierung der Kante (v, w) :

- w hat noch keine DFS-Nummer: $(v, w) \rightarrow T$.
- w hat schon eine DFS-Nummer $num(w) > num(v) : (v, w) \rightarrow F$ (Forward-Kante, Vorwärtskante), der Knoten w ist von v aus über T-Kanten erreichbar, da DFS(w) innerhalb von DFS (v) aufgerufen wurde.

– w hat schon eine DFS-Nummer $num(w) < num(v)$, aber DFS(w) ist noch nicht beendet: $(v, w) \rightarrow B$, der Knoten v ist von w aus über T-Kanten erreichbar, da DFS(v) innerhalb von DFS(w) aufgerufen wurde, und die Kante (v, w) schließt einen Kreis.

– w hat schon eine DFS-Nummer $num(w) < num(v)$ und DFS(w) ist bereits beendet: $(v, w) \rightarrow C$ (Cross-Kante, Querkante), der Knoten v ist von w nicht erreichbar. Wenn v und w im selben T-Baum liegen, führt die Kante (v, w) von einem „später“ erzeugten Teilbaum zu einem „früher“ erzeugten Teilbaum. Ansonsten führt die Kante (v, w) von einem „später“ erzeugten Baum zu einem „früher“ erzeugten Baum.

Weiterhin ist die DFS-Traversierung mit Einteilung der Kanten in linearer Zeit $O(n+m)$ möglich. Diese Einteilung ist wiederum nicht eindeutig, aber unabhängig davon ist G genau dann kreisfrei, wenn B leer ist.

2.2 Starke Zusammenhangskomponenten in gerichteten Graphen

In ungerichteten Graphen hängen zwei Knoten v und w zusammen, wenn sie durch einen Weg verbunden sind. Diesen Weg können wir ja in beiden Richtungen benutzen. Daher ist es naheliegend, zwei Knoten v und w in einem gerichteten Graphen als zusammenhängend zu bezeichnen, wenn es sowohl einen Weg von v zu w als auch einen Weg von w zu v gibt. Zur besseren Unterscheidung wird bei gerichteten Graphen der Ausdruck **stark** zusammenhängend verwendet, wenn es zwischen v und w Wege in beiden Richtungen gibt.
Die Relation starker Zusammenhang ($v \leftrightarrow w$) ist offensichtlich eine Äquivalenzrelation:

– $v \leftrightarrow v$ (betrachte den Weg der Länge 0)

– $v \leftrightarrow w \Leftrightarrow w \leftrightarrow v$ (nach Definition)

– $v \leftrightarrow w, w \leftrightarrow z \Rightarrow v \leftrightarrow z$ (konkateniere die entsprechenden Wege).

Damit lässt sich die Knotenmenge V in die zugehörigen Äquivalenzklassen $V_1, \ldots, V_k$ partitionieren. Die Knotenmengen $V_1, \ldots, V_k$ heißen starke Zusammenhangskomponenten des gerichteten Graphen $G = (V, E)$. Manchmal werden auch die Teilgraphen $G_i = (V_i, E_i)$ mit $E_i = E \cap (V_i \times V_i)$ als starke Zusammenhangskomponenten bezeichnet. Es ist zu beachten, dass es Kanten zwischen den verschiedenen Komponenten geben kann (im Gegensatz zu den einfachen Zusammenhangskomponenten bei ungerichteten Graphen).
Von folgendem Algorithmus wollen wir nachweisen, dass er die starken Zusammenhangskomponenten von $G = (V, E)$ berechnet.

2.1 Algorithmus.

1.) Führe eine DFS-Traversierung von G durch, wobei Knotennummern rückwärts von n bis 1 bei Beendigung des DFS-Aufrufs vergeben werden.

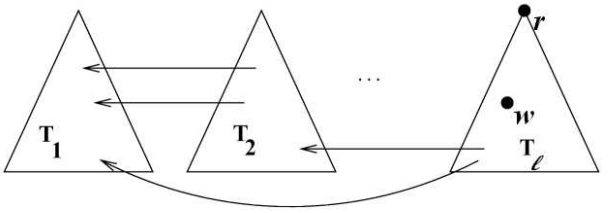
2.) Bilde den rückwärts gerichteten Graphen $G_{rev} = (V, E_{rev})$, wobei $(v, w) \in E_{rev}$ genau dann ist, wenn $(w, v) \in E$ ist.

3.) Führe eine DFS-Traversierung auf G_{rev} durch, wobei die Knoten als Liste gemäß der in Schritt 1 berechneten Nummerierung vorliegen. Berechne die Knotenmengen $V_1, \ldots, V_k$ der Bäume, die durch die konstruierten T-Kanten gebildet werden.

Es ist offensichtlich, dass dieser Algorithmus in Zeit $O(n+m)$ läuft. Der Grund, warum die Knotenmengen $V_1, \ldots, V_k$ die starken Zusammenhangskomponenten darstellen, wird im Beweis des Satzes herausgearbeitet.

2.2 Satz. *Algorithmus 2.1 berechnet in Zeit $O(n+m)$ die starken Zusammenhangskomponenten von gerichteten Graphen.*

Beweis: Wir schauen uns zunächst schematisch den DFS-Wald an, der durch den DFS-Algorithmus in Schritt 1 konstruiert wird. Wir folgen bei der Visualisierung der Konvention, dass die DFS-Bäume von links nach rechts aufgebaut werden und dass das gleiche auch innerhalb der einzelnen DFS-Bäume gilt. Der DFS-Wald bestehe aus den Bäumen $T_1, \ldots, T_\ell$, die wir gemäß Konvention von links nach rechts anordnen.



Der DFS-Algorithmus hat die Eigenschaft, dass es keine Kante von einem Baum T_i zu einem Baum T_j gibt, wenn $i < j$ ist. Es folgt direkt, dass keine starke Zusammenhangskomponente Knoten aus zwei verschiedenen DFS-Bäumen enthalten kann. Jede starke Zusammenhangskomponente liegt also innerhalb eines DFS-Baumes.
Die gewählte Nummerierung der Knoten hat die Eigenschaft, dass die Wurzel r des Baums T_ℓ die kleinste Nummer 1 hat.
Der DFS-Durchlauf auf dem Graphen G_{rev} (Schritt 3) wird also zunächst auf dem Knoten r gestartet. Der erste Baum T', der durch diesen DFS-Durchlauf konstruiert wird, hat die Eigenschaft, dass er alle Knoten enthält, die in G_{rev} von r aus erreichbar sind. T' enthält also alle Knoten, von denen aus man in G den Knoten r erreichen kann. Wenn man sich das Bild anschaut, wird klar, dass diese Knotenmenge eine Teilmenge des Baumes T_ℓ ist.
Da man von r aus in G alle Knoten von T_ℓ erreichen kann und alle Knoten des Baums T' einen Weg nach r in G haben, so liegen alle Knoten aus T' in der gleichen starken Zusammenhangskomponente wie r. Man überlegt sich leicht, dass T' *genau* die Knoten enthält, die in der gleichen Zusammenhangskomponente wie r liegen. Diese Knotenmenge nennen wir nun $C(r)$.
Nun haben wir also eingesehen, dass der Algorithmus u.a. die Zusammenhangskomponente des Knotens r berechnet. Für eine induktive Argumentation überlegen wir uns nun, was aus dem Baum T_ℓ wird, wenn wir die Knoten von $C(r)$ aus T_ℓ (gedanklich)

entfernen. Dazu überlegen wir uns folgendes: Wenn w ein Knoten aus $C(r)$ ist, dann ist jeder Knoten w_{up}, der in T_ℓ oberhalb von w liegt, ebenfalls in $C(r)$: Der Weg von r nach w in T_ℓ gefolgt von dem Weg von w nach r (der existiert, weil $w \in C(r)$ ist) enthält w_{up} und ist ein Kreis, also ist auch $w_{up} \in C(r)$.
Wenn wir die Knoten von $C(r)$ aus dem Baum T_ℓ entfernen, so erhalten wir also eine Menge von Teilbäumen $T_1^*, \ldots, T_t^*$ von T_ℓ. (Wobei $t = 0$ möglich ist).
Man kann den beschriebenen Algorithmus nun so ansehen, als arbeite er (implizit) mit den Bäumen $T_1, \ldots, T_{\ell-1}, T_1^*, \ldots, T_t^*$ weiter.
Es zeigt sich, dass man nun einen Induktionsbeweis für die Korrektheit des Algorithmus führen kann. Die Induktion geht dabei über die Anzahl der starken Zusammenhangskomponenten des Graphen G. Wenn diese 1 ist, so gibt der Algorithmus trivialerweise alle Knoten als starke Zusammenhangskomponente aus. Ist sie größer als 1, so hat unsere Überlegung gerade gezeigt, dass wir zunächst diejenige starke Zusammenhangskomponente ausgeben, in der der Knoten r liegt. Der verbleibende Restgraph hat eine starke Zusammenhangskomponente weniger und auf ihm berechnet der Algorithmus nach Induktionsvoraussetzung korrekterweise alle starken Zusammenhangskomponenten. □

Starke Zusammenhangskomponenten spielen nicht nur in typischen Graphenproblemen eine Rolle. Dazu ein Beispiel. Ein effizienter Algorithmus zur Lösung des Entscheidungsproblems 2-SAT arbeitet mit den starken Zusammenhangskomponenten eines geeignet konstruierten Graphen.

2.3 Zweizusammenhangskomponenten

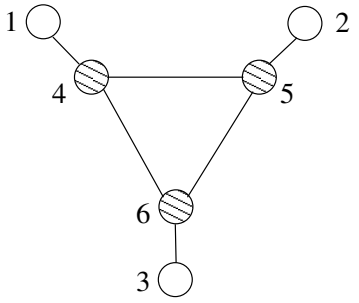
2.3 Definition. *Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph.*

a) Ein Knoten v heißt Schnittpunkt von G, wenn der Graph, der aus G durch Herausnahme des Knotens v und aller Kanten $\{v, w\}$ mit $w \in Adj(v)$ entsteht, nicht zusammenhängend ist.

b) G heißt zweifach zusammenhängend, wenn er keinen Schnittpunkt enthält.

c) Die inklusionsmaximalen zweifach zusammenhängenden Teilgraphen von G heißen Zweizusammenhangskomponenten.

2.4 Beispiel. *Im folgenden Graph sind die Schnittpunkte schraffiert eingezeichnet.*



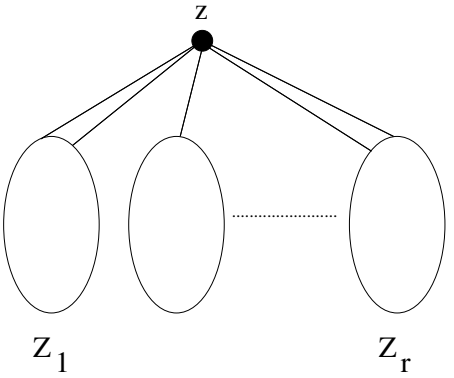
G hat vier Zweizusammenhangskomponenten: Jeweils die induzierten Teilgraphen auf den Knotenmengen $\{1,4\}$, $\{2,5\}$, $\{3,6\}$ und $\{4,5,6\}$.
Ein Kommunikationsnetzwerk kann den Ausfall einer beliebigen Komponente genau dann verkraften, wenn das Netzwerk zweifach zusammenhängend ist.
Eine Reihe von graphentheoretischen Algorithmen benötigt als Eingabe einen zweifach zusammenhängenden Graphen. Häufig lassen sich die Algorithmen auch auf beliebige Graphen verallgemeinern, wenn man sie auf alle Zweizusammenhangskomponenten des Eingabegraphen anwendet. Somit ist es interessant, einen effizienten Algorithmus zu besitzen, der zu einem gegebenen Graphen dessen Zweizusammenhangskomponenten berechnet.
In diesem Kapitel werden wir einen Algorithmus kennenlernen, der das Zweizusammenhangsproblem in Linearzeit löst. Zunächst benötigen wir jedoch etwas mehr „strukturelles“ Wissen über Zweizusammenhangskomponenten.

2.5 Lemma. *Seien $V_1 \neq V_2$ Zweizusammenhangskomponenten von G. Es gilt*

a) $|V_1 \cap V_2| \leq 1$.

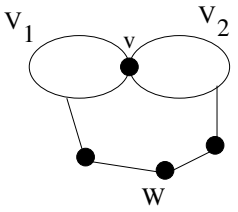
b) $V_1 \cap V_2 = \{v\} \quad \Rightarrow \quad v$ *ist Schnittpunkt in* G.

Zweizusammenhangskomponenten haben also höchstens einen Knoten gemeinsam und wenn, dann ist der gemeinsame Knoten ein Schnittpunkt.
Beweis: Falls V_1 und V_2 keinen Knoten gemeinsam haben, ist nichts zu zeigen. Betrachten wir also den Fall, dass sie mindestens einen Knoten v gemeinsam haben. Der Teilgraph auf V_1 ist ein maximaler zweifach zusammenhängender Teilgraph von G, somit ist der Teilgraph auf $V_1 \cup V_2$, nennen wir ihn G^*, nicht zweifach zusammenhängend. Da V_1 zusammenhängend ist und V_2 auch und beide den Knoten v gemeinsam haben, ist auch G^* zusammenhängend. Da G^* nicht zweifach zusammenhängend ist, enthält G^* einen Knoten z, der Schnittpunkt in G^* ist. Wenn wir z aus G^* entfernen, zerfällt der Restgraph in $r \geq 2$ Zusammenhangskomponenten, nennen wir diese $Z_1, \ldots, Z_r$. Der Sachverhalt sei schematisch skizziert in folgendem Bild, das den Graphen G^* zeigt.



Wir fragen uns, wo die Knoten von V_1 liegen können.
Angenommen, V_1 enthält Knoten aus Z_i und aus Z_j mit $j \neq i$. Dann gibt es zwei Fälle: Entweder ist z nicht in V_1, dann wäre der Teilgraph auf V_1 aber nicht zusammenhängend

gewesen. Oder z ist in V_1 enthalten, dann wäre V_1 aber nicht zweifach zusammenhängend gewesen, weil z auch in dem Teilgraph auf V_1 ein Schnittpunkt wäre. Somit gibt es ein i^*, so dass $V_1 \subseteq Z_{i^*} \cup \{z\}$. Mit der analogen Argumentation für V_2 folgt, dass es auch ein j^* gibt mit $V_2 \subseteq Z_{j^*} \cup \{z\}$. Falls $i^* = j^*$ wäre, dann wäre $V_1 \cup V_2 \subseteq Z_{i^*} \cup \{z\}$, was aber nicht sein kann, da nach Entfernen von z der Graph in mindestens zwei Zusammenhangskomponenten zerfällt.
Also ist $i^* \neq j^*$, und es folgt, dass $V_1 \cap V_2 = \{z\}$ ist. Damit ist Aussage a) gezeigt. Aussage b) ist aber noch nicht gezeigt, denn z ist zwar Schnittpunkt in dem Teilgraphen G^*, aber Aussage b) besagt ja, dass z auch in G ein Schnittpunkt ist.
Also argumentieren wir für den Beweis von b) wie folgt. Angenommen, $V_1 \cap V_2 = \{v\}$. V_1 und V_2 sind nicht einelementig, also ist $V_1 \setminus \{v\}$ und $V_2 \setminus \{v\}$ nicht leer. Nach den Überlegungen von gerade ist v ein Schnittpunkt in dem Teilgraphen G^*, also gibt es zwischen $V_1 \setminus \{v\}$ und $V_2 \setminus \{v\}$ keine Kante.
Wenn v in G kein Schnittpunkt wäre, dann gäbe es nach Entfernen von v in G aber immer noch einen Weg von einem Knoten in $V_1 \setminus \{v\}$ zu einem Knoten in $V_2 \setminus \{v\}$. Da es keine direkte Kante gibt, muss der Weg über Knoten aus $V \setminus (V_1 \cup V_2)$ führen. Schematisch haben wir also folgende Situation vorliegen:



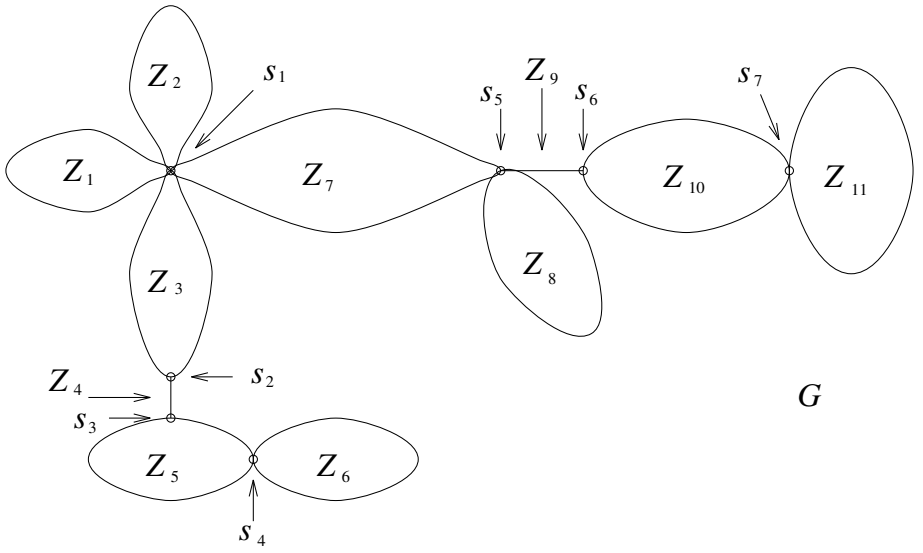
Sei W die Menge der Knoten auf diesem Weg. Nun sieht man relativ leicht ein, dass dann $V_1 \cup V_2 \cup W$ zweifach zusammenhängend ist. Das kann aber nicht sein, da V_1 (und auch V_2) maximal zweifach zusammenhängend ist. □

Jede Kante e ist ein zweifach zusammenhängender Graph. Es gibt also mindestens eine Zweizusammenhangskomponente, die die Kante e enthält. Aus dem gerade bewiesenen Lemma folgt auch, dass eine Kante nicht in zwei verschiedenen Zweizusammenhangskomponenten enthalten sein kann. Jede Kante kann man also genau einer Zweizusammenhangskomponente zuordnen. Es ist also vernünftig, die Zweizusammenhangskomponenten durch Kantenmengen zu beschreiben.
Zweizusammenhangskomponenten „kleben“ an Schnittpunkten zusammen, das zeigt das folgende Lemma:

2.6 Lemma. *Schnittpunkte liegen in mindestens zwei Zweizusammenhangskomponenten.*

Beweis: Sei v ein Schnittpunkt, dessen Herausnahme alle Wege zwischen z_1 und z_2 zerstört. Wir betrachten einen Weg von z_1 nach z_2 über v. Seien y_1 und y_2 die Nachbarn von v auf diesem Weg, d. h. es gibt Kanten $\{y_1, v\}$ und $\{y_2, v\}$. Die Knoten y_1 und y_2 liegen nicht in der gleichen Zweizusammenhangskomponente, da die Herausnahme von v alle Wege zwischen y_1 und y_2 zerstört. Andererseits ist die Knotenmenge $\{y_1, v\}$

zweifach zusammenhängend, und es gibt eine Zweizusammenhangskomponente Z_1, die y_1 und v enthält, und analog eine Zweizusammenhangskomponente Z_2, die y_2 und v enthält. Nach den Vorüberlegungen sind Z_1 und Z_2 verschieden und enthalten beide v. □



2.7 Beispiel.

Der Graph G enthält 11 ZZK und 7 Schnittpunkte. Wie arbeitet nun ein DFS-Algorithmus, der z. B. in Z_7 startet, auf G?
Er arbeitet zunächst in Z_7 und entdeckt vielleicht s_1 vor s_5. Wir werden zeigen, wie wir Schnittpunkte erkennen. Von s_1 kann aber u. U. zunächst in Z_7 weitergearbeitet werden. Er findet dann vielleicht s_5. Wenn er dann eine Kante (s_5, a) mit a in Z_8 betrachtet, wird $DFS(a, s_5)$ aufgerufen. Innerhalb dieses Aufrufs werden alle zu Z_8 gehörigen Kanten aufgerufen. Die Kanten von Z_8 werden also innerhalb eines geschlossenen Zeitintervalls abgearbeitet. Es muss nur erkannt werden, dass es sich dabei um eine ZZK handelt. Wenn dann (s_5, s_6) betrachtet wird, werden Z_{10} und Z_{11} abgearbeitet, bevor ein Backtracking über s_6 hinaus stattfindet. Die Kanten von Z_{11} werden dabei wieder innerhalb eines abgeschlossenen Zeitintervalls während eines Aufrufs $DFS(b, s_7)$ und für einen Knoten b in Z_{11} betrachtet. Falls das Zeitintervall für die Bearbeitung von Z_{11} herausgelöst wird, werden die Kanten von Z_{10} innerhalb eines Zeitintervalls betrachtet.
Unser Ziel muss also darin bestehen, die $DFS(a, s)$-Aufrufe für Schnittpunkte s zu erkennen, für die (s, a) die erste Kante einer ZZK ist. Am Ende von $DFS(a, s)$ sind dann alle Kanten dieser ZZK betrachtet worden. Wenn die Kanten auf einen Stack geschrieben werden und am Ende eines derartigen $DFS(a, s)$-Aufrufs der Stack bis hinunter zu der Kante (s, a) abgearbeitet wird, werden alle ZZK korrekt erkannt. Man beachte dabei, dass ein $DFS(a', s')$-Aufruf innerhalb von $DFS(a, s)$ auch vor dem Ende von $DFS(a, s)$ beendet wird.

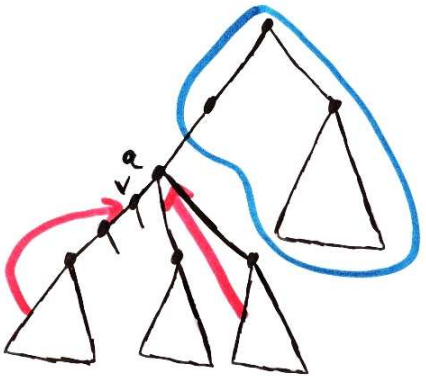
2.8 Definition. *Sei $G = (V, E)$ ein ungerichteter Graph und num : $V \to \{1, \ldots, n\}$*

die von DFS erzeugte Knotennummerierung. Dann ist

$lowpt(v) = \min\{num(x) \mid \exists$ *Pfad* $v \to x$ *aus* T*-Kanten und am Ende evtl. einer* B*-Kante*$\}$

lowpt steht für low point.

Im folgenden Lemma werden wir sehen, dass wir mit der Kanteneinteilung des DFS-Algorithmus in T und B, mit der DFS-Nummerierung und der *lowpt*-Nummerierung Schnittpunkte erkennen können.
Der Beweis des Lemmas wird anschaulich, wenn man sich ein Bild des DFS-Baumes hinzeichnet und sich überlegt, dass für eine T-Kante (a, v) im Baum die Aussage $lowpt(v) \geq num(a)$ besagt, dass es keinen Weg aus dem Teilbaum an v gibt, der zu einem Knoten oberhalb von a führt.



2.9 Lemma. *Sei $G = (V, E)$ ein zusammenhängender, ungerichteter Graph, auf dem DFS die Nummerierung num und die Kanteneinteilung $E = T \cup B$ liefert. Dann sind die beiden folgenden Aussagen äquivalent.*

(1) a ist Schnittpunkt.

(2) Entweder ist a Wurzel im DFS–Baum und hat mindestens zwei Kinder, oder a ist nicht Wurzel und es gibt mindestens ein Kind v von a mit $lowpt(v) \geq num(a)$.

Beweis: G habe mindestens zwei Knoten, ansonsten ist die Aussage trivial.
„(1) $\Longrightarrow$ (2)“: Da G zusammenhängend ist, bildet T einen Baum auf allen Knoten in V. Wenn a Wurzel dieses Baumes ist und nur ein Kind hat, dann ist der Graph auf den Knoten $V \setminus \{a\}$ offensichtlich noch zusammenhängend. Also hat a mindestens zwei Kinder.
Sei nun a keine Wurzel. Wenn a ein Blatt wäre, dann wäre der Graph nach Herausnahme von a immer noch zusammenhängend, also ist a kein Blatt. Seien $T_1, \ldots, T_s$ die Teilbäume mit Wurzeln $v_1, \ldots, v_s$, die in T unter a hängen. Wenn für alle v_i gelten würde, dass $lowpt(v_i) < num(a)$ ist, dann wäre nach Herausnahme von a der Graph noch zusammenhängend. Der Grund ist der folgende: Betrachte einen beliebigen Knoten $w \neq a$. Wenn dieser nicht in einem der Teilbäume T_i enthalten ist, dann gibt es einen

Weg über T-Kanten zur Wurzel von T. Wenn w ein Knoten aus T_i ist, dann kann man zunächst („rückwärts“) über T-Kanten zu v_i laufen. Da $lowpt(v_i) < num(a)$ ist, kann man über T-Kanten und mit einer anschließenden B-Kante zu einem Knoten gelangen, der weder a noch in einem der T_i enthalten ist. Von dort existiert ein Weg zur Wurzel von T. Man gelangt also von jedem Knoten $w \neq a$ zur Wurzel von T, somit wäre der Graph nach Herausnahme von a noch zusammenhängend. Dies ist ein Widerspruch zur Annahme, dass a ein Schnittpunkt ist.
„(2) $\Longrightarrow$ (1)“: Wenn a die Wurzel von T ist, ist die Überlegung wieder einfach, da es nach Entfernen von a keinen Weg mehr zwischen den beiden Kindern von a gibt.
Sei a also keine Wurzel und v_i ein Kind von a, für das $lowpt(v_i) \geq num(a)$ gilt. Nach Entfernen von a gilt für jede übrig bleibende B-Kante mit Startknoten in T_i, dass sie als Zielknoten nur noch Knoten w hat, für die $num(w) > num(a)$ ist. Entlang von T-Kanten erhöht sich die num-Nummer nur. Somit erhalten wir, dass wir, wenn wir in einem Knoten aus T_i starten, nur Knoten erreichen können, deren DFS-Nummer größer als $num(a)$ ist. Somit können wir von einem solchen Knoten aus nicht mehr die Wurzel von T (die DFS-Nummer 1 hat) erreichen. Knoten a ist also Schnittpunkt. □

2.10 Lemma. $lowpt(v) = \min(\{num(v)\} \cup \{num(x) \mid (v,x) \in B\} \cup \{lowpt(x) \mid (v,x) \in T\})$.

Beweis: Die zur Definition von $lowpt(v)$ zugelassenen Wege nennen wir Low-Wege. Sie zerfallen in drei Klassen:

1.) Den Weg der Länge 0.

2.) Wege, die aus einer B-Kante bestehen.

3.) Wege, die mit einer T-Kante (v,w) beginnen und mit einem Low-Weg von w aus fortgesetzt werden.

Also beachtet die obige Formel für $lowpt(v)$ alle Low-Wege. □

Lemma 2.10 legt einen rekursiven Algorithmus zur Berechnung von $lowpt(v)$ nahe. Der folgende Algorithmus berechnet die $lowpt$-Nummerierung und gleichzeitig auch Zweizusammenhangskomponenten. Es muss dabei sichergestellt werden, dass $lowpt(v)$ erst dann benutzt wird, wenn es seinen richtigen Wert erhalten hat.

2.11 Algorithmus. *Input: $G = (V, E)$, ein ungerichteter, zusammenhängender Graph. Output: Die Zweizusammenhangskomponenten von G als Kantenmengen.*

1.) Initialisierung: $i := 0$; $S :=$ empty stack; für alle $x \in V$: $num(x) := 0$.

2.) BICON$(1, 0)$. (Graph ist zusammenhängend, also starten wir z.B. an Knoten 1.)

BICON(v, u)

1.) $i := i + 1$; $num(v) := i$; $lowpt(v) := i$.

2.) Für $w \in Adj(v)$ *mit* $w \neq u$:

3.) *if* $num(w) = 0$
then $(v, w) \to S$; BICON(w, v);

4.) $lowpt(v) := \min\{(lowpt(v), lowpt(w))\}$.

5.) *if* $lowpt(w) \geq num(v)$
then entferne Kanten vom Stack bis einschließlich (v, w). *Die Kanten bilden eine ZZK und werden in die Ausgabe geschrieben.*

6.) *else if* $(num(w) < num(v))$
then $(v, w) \to S$; $lowpt(v) := \min\{(lowpt(v), num(w))\}$

2.12 Satz. *Algorithmus 2.11 produziert in $O(n + e)$ Rechenschritten die Zweizusammenhangskomponenten von G.*

Beweis: Der Rahmen ist ein DFS-Algorithmus. Zunächst zeigen wir, dass am Ende von BICON$(v, \cdot)$ der Wert $lowpt(v)$ korrekt berechnet ist: In der Initialisierung in Zeile 1 von BICON wird $num(v)$ berücksichtigt. Für jede B-Kante (v, w) wird in Zeile 6 $num(w)$ berücksichtigt. Für jede T-Kante (v, w) wird in Zeile 4 $lowpt(w)$ berücksichtigt. Da BICON (w, v) vor BICON(v, u) beendet ist, ist zu diesem Zeitpunkt $lowpt(w)$ korrekt berechnet. Also wird nach Lemma 2.10 auch $lowpt(v)$ korrekt berechnet.
Betrachten wir nun das Verhalten des Algorithmus auf einem zusammenhängenden Graphen und weisen wir nach, dass er alle Zweizusammenhangskomponenten korrekt berechnet.
Wir führen eine Induktion über die Anzahl der Zweizusammenhangskomponenten. Wenn der Graph genau eine Zweizusammenhangskomponente hat, also keinen Schnittpunkt enthält, dann sieht man leicht ein, dass Algorithmus BICON alle Kanten am Ende des Algorithmus ausgibt. Denn: die DFS-Nummer der Wurzel ist 1 und die $lowpt$-Nummer des einzigen Kindes der Wurzel ist auch mindestens 1. Das führt zur Ausgabe des Stacks in Schritt 5. Außer am Ende wird keine Ausgabe produziert, da für alle Knoten, die nicht Wurzel sind, eine Ausgabe nur produziert wird, wenn der Knoten Schnittpunkt in G ist. Wenn der Graph aus $k > 1$ Zweizusammenhangskomponenten besteht, dann enthält er mindestens einen Schnittpunkt. Sei v der Schnittpunkt mit der größten DFS-Nummer. Unterhalb von v in T gibt es also keinen weiteren Schnittpunkt von G. v ist natürlich

wieder kein Blatt, es gibt also in T mindestens eine Kante (v, w) mit $lowpt(w) \geq num(v)$. (Dies gilt auch, wenn v die Wurzel von T ist.)
Beim Aufruf von $BICON(w, v)$ wird keine Ausgabe produziert, und alle Kanten, die dort gefunden werden, werden auf den Stack gelegt. Schließlich wird erkannt, dass $lowpt(w) \geq num(v)$ ist und alle Kanten, die nach der Kante $\{v, w\}$ auf den Stack gelegt wurden (inklusive der Kante selbst) werden ausgegeben. Es ist klar, dass die ausgegebene Kantenmenge eine Zweizusammenhangskomponente bildet. Nun kann man sich von Folgendem überzeugen: Im weiteren Verlauf verhält sich der Algorithmus genau so (bzgl. der Ausgabe der Kantenmengen), als wäre zu Beginn der Knoten w und dessen gesamter Unterbaum nicht vorhanden gewesen. Der restliche Graph aber enthält genau eine Zweizusammenhangskomponente weniger, so dass der Algorithmus nach Induktionsvoraussetzung auf ihm die richtigen Zweizusammenhangskomponenten ausgibt. Die Rechenzeit wird offensichtlich vom DFS-Algorithmus bestimmt und beträgt $O(n+e)$. □

2.4 Transitiver Abschluss

2.13 Definition. *Der transitive Abschluss eines Graphen $G = (V, E)$ ist durch die Matrix $A^* = (a^*_{ij})$ gegeben, wobei $a^*_{ij} = 1$ genau dann gilt, wenn es einen Weg von i nach j gibt, und $a^*_{ij} = 0$ sonst.*

2.14 Bemerkung. *Der transitive Abschluss kann für ungerichtete Graphen in Zeit $O(n^2)$ berechnet werden.*

Beweis: In $O(n+e)$ Schritten werden die Zusammenhangskomponenten berechnet. Für jede Komponente Z wird $a^*_{ij} = 1$ für alle $i, j \in Z$ gesetzt. □

Für gerichtete Graphen ist mehr zu tun. Ein k-Weg sei ein Weg, dessen Länge höchstens k ist. Wenn es einen Weg von i nach j gibt, dann gibt es auch einen $(n-1)$-Weg von i nach j. Es sei nun $a^{(k)}_{ij} = 1$ genau dann, wenn es einen k-Weg von i nach j gibt, und $a^{(k)}_{ij} = 0$ sonst. Wir erzeugen in $O(n^2)$ Schritten die Adjazenzmatrix A von G. Es ist $a^{(1)}_{ij} = a_{ij}$ für $i \neq j$ und $a^{(1)}_{ii} = 1$. Jeder $2k$-Weg zerfällt in zwei k-Wege. Also ist

$$a^{(2k)}_{ij} = \bigvee_{1 \leq m \leq n} a^{(k)}_{im} \wedge a^{(k)}_{mj},$$

und $A^{(2k)}$ ist das Boolesche Matrizenprodukt von $A^{(k)}$ und $A^{(k)}$. Wir berechnen durch fortgesetztes Quadrieren für $N = 2^{\lceil \log (n-1) \rceil}$ $A^{(2)}, A^{(4)}, A^{(8)}, \ldots, A^{(N)}$. Da $N \geq n-1$, ist $A^{(N)} = A^*$.

2.15 Satz. *Sei $M(n)$ die Komplexität, um das Boolesche Matrizenprodukt zweier $n \times n$-Matrizen zu berechnen. Dann kann der transitive Abschluss eines gerichteten Graphen in $O(M(n) \cdot \log n)$ Schritten berechnet werden.*

Alle Algorithmen zur Matrizenmultiplikation können auch auf die Multiplikation Boolescher Matrizen angewendet werden. Es wird $\sum_{1\le k\le n} a_{ik}b_{kj}$ berechnet. Es gilt

$$\bigvee_{1\le k\le n} a_{ik}b_{kj} = 0 \Longleftrightarrow \sum_{1\le k\le n} a_{ik}b_{kj} = 0.$$

Also genügen bei Benutzung der Schulmethode für die Matrizenmultiplikation (diese hat $M(n) = O(n^3)$) zur Berechnung des transitiven Abschlusses $O(n^3 \log n)$ Schritte.

2.16 Satz. *Der transitive Abschluss eines gerichteten Graphen kann mit $2n^3$ Bitoperationen berechnet werden.*

Beweis: Wir benutzen die Methode der Dynamischen Programmierung. Es sei $b_{ij}^{(\ell)} = 1$ genau dann, wenn es einen Weg von i nach j gibt, der nur Knoten aus $\{1,\dots,\ell\}$ als Zwischenpunkte benutzt und $b_{ij}^{(\ell)} = 0$ sonst. Dann ist $a_{ij}^* = b_{ij}^{(n)}$ und $b_{ij}^{(0)} = a_{ij}$ für $i \neq j$ und $b_{ii}^{(0)} = 1$. Es gibt nun einen Weg von i nach j mit Zwischenpunkten in $\{1,\dots,\ell+1\}$ genau dann, wenn es einen $i-j$-Weg bereits mit Zwischenpunkten in $\{1,\dots,\ell\}$ gibt oder sowohl einen $i-(\ell+1)$-Weg und einen $(\ell+1)-j$-Weg mit Zwischenpunkten in $\{1,\dots,\ell\}$ gibt. Also ist

$$b_{ij}^{(\ell+1)} = b_{ij}^{(\ell)} \vee \left(b_{i,\ell+1}^{(\ell)} \wedge b_{\ell+1,j}^{(\ell)}\right).$$

$B^{(\ell+1)}$ kann mit $2n^2$ Bitoperationen aus $B^{(\ell)}$ berechnet werden. □

2.5 Kürzeste Wege

Es sei nun eine Gewichtsmatrix $W = (w_{ij})$ mit $w_{ij} \in \mathbb{R}$ gegeben, wobei $w_{ii} = 0$ für alle i ist. Wir interpretieren w_{ij} als die Länge der Kante von i nach j. Wir möchten kürzeste Wege zwischen Knotenpaaren berechnen und unterscheiden 3 Probleme.

Problem 1: Berechne für ein Knotenpaar (i,j) die Länge eines kürzesten Weges von i nach j.

Problem 2: Berechne für einen Knoten i die Länge kürzester Wege von i nach j für $1 \le j \le n$.

Problem 3: Berechne die Länge kürzester Wege von i nach j für alle Knotenpaare (i,j).

2.17 Satz. *Falls $w_{ij} \ge 0$ für alle (i,j), können Problem 1 und Problem 2 mit $O(n^2)$ Rechenschritten gelöst werden.*

Beweis: Problem 1 ist ein Unterproblem von Problem 2. Wir betrachten nun also Problem 2. Es soll $label(j)$ für $1 \le j \le n$ am Ende die Länge ℓ_{ij} kürzester Pfade von i nach j beinhalten. Wir benutzen wieder Dynamische Programmierung und lassen als

Zwischenknoten nur solche Knoten k zu, für die wir die Länge des kürzesten Weges von i nach k bereits kennen. Da wir stets kürzeste Verbindungen auswählen, kann der Algorithmus auch als Greedy Algorithmus interpretiert werden.
Wir starten mit $label(i) = 0$ und $label(j) = w_{ij}$ für $i \neq j$. Erlaubter Zwischenpunkt ist Knoten i. Wir erkennen nun, dass für den Knoten $j^* \neq i$ mit minimalem w_{ij}-Wert ein kürzester Weg Länge w_{ij^*} hat. Hier wird benutzt, dass alle w-Werte nicht negativ sind. Jede erste Kante eines den Knoten i verlassenden Weges hat mindestens Länge w_{ij^*}.
Allgemein wird in A die Menge der Knoten j verwaltet, für die wir sicher sind, dass $label(j) = \ell_{ij}$ ist. Die Situation stellt sich allgemein folgendermaßen dar. Es gibt einen Knoten j^*, der im letzten Schritt zu A hinzugefügt wurde, die neue Menge nennen wir A_{neu}. Für alle Knoten k enthält $label(k)$ die Länge kürzester Wege von i nach k, wenn nur Zwischenpunkte aus A zugelassen sind. Wir wollen nun $label(k)$ für alle Knoten k, die nicht in A_{neu} sind, aktualisieren.
Betrachten wir also einen kürzesten Weg von i zum Knoten k, der nur Knoten aus A_{neu} benutzt. Sei t der Vorgängerknoten von k auf diesem Weg. Wenn $t \neq j^*$ ist, dann hätte man auch den kürzesten Weg von i zu t ohne Benutzung von j^* nehmen können, da es ja einen kürzesten Weg von i zu t gibt, der nur Knoten aus A benutzt. Dieser Weg war aber schon vorher in $label(k)$ berücksichtigt. Also müssen wir nur noch den Fall $t = j^*$ betrachten. Somit erhalten wir $label(k) := \min\{label(k), label(j^*) + w_{j^*k}\}$.
Für den Knoten $k^* \in \{1, \ldots, n\} - A$, für den $label(k)$ am kleinsten ist, wurde nun ein kürzester $i-k^*$-Weg gefunden. Jeder andere die Menge A verlassende Weg ist mindestens so lang. Also kann im nächsten Schritt k^* zu A hinzugefügt werden. Der Algorithmus braucht maximal $n - 1$ Runden, und jede Runde kann in $O(n)$ Schritten durchgeführt werden. □

Dieser Algorithmus ist eine Variante des in der Vorlesung DAP2 behandelten Algorithmus von Dijkstra. Diese Variante ist für vollbesetzte Graphen günstiger. Der Algorithmus von Dijkstra benötigt eine Rechenzeit von $O(n + e \log n)$, da eine Priority Queue verwaltet werden muss.
Wir erlauben nun auch negative Werte für w_{ij}. Wenn allerdings Kreise negative Länge haben, wird die Frage nach kürzesten Wegen sinnlos. Durch mehrfaches Durchlaufen von Kreisen negativer Länge kann jeder Weg beliebig verkürzt werden.

2.18 Satz. *Wenn alle Kreise nichtnegative Länge haben, kann Problem 3 in $O(n^3)$ Schritten gelöst werden.*

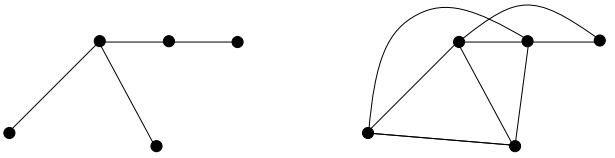
Beweis: Der Beweis verläuft analog zum Beweis von Satz 2.16. Es sei $w_{ij}^{(\ell)}$ die Länge des kürzesten Weges von i nach j mit Zwischenpunkten in $\{1, \ldots, \ell\}$. Nach Voraussetzung müssen nur kreisfreie Wege betrachtet werden. Es ist $w_{ij}^{(0)} = w_{ij}$ und $w_{ij}^{(\ell+1)} = \min\{w_{ij}^{(\ell)}, w_{i,\ell+1}^{(\ell)} + w_{\ell+1,j}^{(\ell)}\}$. $W^{(\ell+1)}$ kann aus $W^{(\ell)}$ in $O(n^2)$ Schritten berechnet werden. $W^{(n)}$ enthält das Ergebnis. □

Der Algorithmus aus Satz 2.17 ist asymptotisch optimal. (Dies folgt im Wesentlichen aus der Eigenschaft, dass man sich alle Matrixeinträge ansehen muss.) In der allgemeinen

Situation mit Gewichten $w_{ij} \in \mathbb{R}$ sind unter der Annahme, dass alle Kreise nichtnegative Länge haben, noch keine $o(n^3)$-Algorithmen für die Probleme 1 und 2 bekannt. Dies zeigt wieder einmal, dass kleine Änderungen an Problemen für den Entwurf effizienter Algorithmen große Folgen haben können. Wenn wir also zu Beginn „aus Versehen“ mit der für uns zu allgemeinen Voraussetzung $w_{ij} \in \mathbb{R}$ arbeiten und Problem 1 oder Problem 2 lösen wollen, ist es wesentlich, dass wir merken, woran unsere Versuche, effiziente, d. h. in diesem Fall $o(n^3)$-Algorithmen zu entwerfen, scheitern. Nur dann kommen wir hoffentlich auf die Idee, uns auf den Fall $w_{ij} \geq 0$ zu beschränken, und nur dann kommen wir hoffentlich zu den obigen $O(n^2)$-Algorithmen.
Zum Schluss betrachten wir noch einen weiteren Algorithmus für Problem 3, allerdings diesmal in der „ungewichteten“ Version, d.h., alle Kanten im Graph haben Gewicht 1. Problem 3 ist übrigens in der Literatur auch unter dem Namen „All Pairs Distances“ bzw. „All Pairs Shortest Paths“ bekannt. Eingabe ist ein ungerichteter Graph, es soll zu allen Knotenpaaren $v \neq w$ die Länge des kürzesten Weges von v nach w berechnet werden. Die Länge eines kürzesten Weges von v nach w wird auch als die Distanz von v nach w bezeichnet.
Wir zeigen hier, dass man dieses Problem in Laufzeit $O(M(n) \log n)$ lösen kann, wobei $M(n)$ wieder die Laufzeit ist, die man in einem Unterprogramm benötigt, um zwei Matrizen mit n Zeilen und n Spalten miteinander zu multiplizieren. Anstelle der Schulmethode, die Laufzeit $\Theta(n^3)$ hat, kann man zum Beispiel auch Strassens Multiplikationsalgorithmus benutzen, dann wäre $M(n) = O(n^{2,7\cdots})$.
Das Interessante hier ist, dass man die Berechnung der kürzesten Wege auf das Durchführen von ein paar Matrixmultiplikationen zurückführen kann.

2.19 Definition. *Sei A die Adjazenzmatrix von G. Mit A^* bezeichnen wir (in diesem Abschnitt) die Adjazenzmatrix zum Graphen G^*, der für alle Knoten $v \neq w$ eine Kante $\{v, w\}$ enthält, wenn v und w in G Abstand kleiner gleich 2 zueinander haben.*

Mit D bezeichnen wir die Matrix der Distanzen in G, d.h., $D_{i,j}$ gibt die Distanz von Knoten i zu Knoten j in G an. Entsprechend bezeichne D^* die Distanzmatrix für den Graphen G^*. D ist also die Matrix, die wir in Problem 3 berechnen möchten.



G $\qquad$ G*

2.20 Beispiel.

Man kann sich als Übungsaufgabe überlegen, dass man die Matrix A^* wie folgt aus A berechnen kann: Berechne $A + A^2$, trage dann im Ergebnis auf der Hauptdiagonalen Nullen ein und ersetze alle Zahlen in der Ergebnismatrix, die größer als Eins sind, durch eine Eins. Man kann also A^* in Laufzeit $O(M(n))$ berechnen, denn die Laufzeit $\Theta(n^2)$ für die Addition der zwei Matrizen bzw. das Abändern der Einträge verschwindet in dem Groß-O, da auf jeden Fall $M(n) = \Omega(n^2)$ ist.

Um unseren Algorithmus später als korrekt nachzuweisen, benötigen wir folgenden Begriff: Der Durchmesser eines Graphen bezeichnet das Maximum der Distanzen über alle Knotenpaare, also die größte Zahl in der Matrix D.

2.21 Lemma. *Der Durchmesser eines Graphen ist kleiner oder gleich 2 genau dann, wenn G^* der vollständige Graph ist. Wenn G^* der vollständige Graph ist, dann gilt $D = 2A^* - A$.*

Beweis: In der Hinrichtung ist die erste Aussage klar, da je zwei Knoten durch eine Kante in G^* verbunden werden. Für die Rückrichtung beobachten wir, dass eine Kante in G^* besagt, dass die entsprechende Kante auch schon in G vorhanden ist oder dass die beteiligten Knoten in G Abstand kleiner gleich 2 voneinander haben. Nach Definition des Durchmessers hat G also Durchmesser höchstens 2.
Für die andere Behauptung beobachten wir, dass $2A^*$ eine Matrix ist, die auf der Hauptdiagonalen Nullen und überall sonst Zweien stehen hat. Man überlegt sich wie gerade, dass die Distanz zwischen zwei Knoten i und j zwei ist, wenn es keine Kante zwischen den beiden gibt, und 1 sonst. Dies gibt genau die Formel $2A^* - A$ wieder. □

Unsere Absicht ist es, einen rekursiven Algorithmus zu entwerfen. Genauer: Wir wollen A^* berechnen, dann rekursiv zu A^* die Distanzmatrix D^* berechnen, um daraus dann Erkenntnisse über D zu bekommen. Wir müssen also zunächst untersuchen, wie D und D^* zueinander stehen. Es gilt folgendes Lemma:

2.22 Lemma. *Seien $i \neq j$ Knoten aus G. Es gilt:*

a) $D_{i,j}$ *gerade* $\Rightarrow \quad D_{i,j} = 2D^*_{i,j}$

b) $D_{i,j}$ *ungerade* $\Rightarrow \quad D_{i,j} = 2D^*_{i,j} - 1$

c) $D_{i,j}/2 \leq D^*_{i,j} \leq (D_{i,j} + 1)/2$

Beweis: Wir beweisen nur a), Teil b) lässt sich vollkommen analog nachweisen. Betrachte den kürzesten Weg von i nach j in G, dieser sei $i = v_1, \ldots, v_r = j$. Da die Länge des Wegs gerade ist, ist r ungerade. In G^* gibt es die Kanten von v_1 nach v_3, etc., also gibt es in G^* den Weg $i = v_1, v_3, \ldots, v_r$, der nur noch die v_t benutzt, für die t ungerade ist. Somit ist in G^* die Distanz zwischen i und j höchstens $D_{i,j}/2$, also $D^*_{i,j} \leq D_{i,j}/2$. Umgekehrt kann man, wenn man einen Weg von i nach j in G^* gegeben hat, jede Kante dieses Weges durch maximal zwei Kanten in G ersetzen, somit folgt $D_{i,j} \leq 2 \cdot D^*_{i,j}$, insgesamt also Behauptung a). Behauptung c) folgt aus Behauptungen a) und b). □

Die Konsequenz aus Lemma 2.22 ist, dass wir D kennen, wenn wir die Paritäten der $D_{i,j}$ kennen, (also, ob es ungerade oder gerade ist), und wenn wir D^* kennen. Auf den ersten Blick haben wir nichts gewonnen, denn wie sollen wir an die Paritäten der Distanzen herankommen, wenn wir die Distanzen nicht schon kennen, die wir erst berechnen wollen? Hier gibt es aber einen überraschenden Trick zum Berechnen der Paritäten von D, wenn man D^* kennt. Dazu kommen wir nun.

Seien $i \neq j$ zwei fest gewählte Knoten. Für alle Nachbarn k von i gilt, dass

$$D_{i,j} - 1 \leq D_{k,j} \leq D_{i,j} + 1 \tag{1}$$

ist. Dies ist einfach einzusehen, da jeder Weg von i nach j durch Benutzen der Kante zwischen k und i zu einem Weg von k nach j gemacht werden kann. Umgekehrt ebenso. Außerdem gibt es einen Nachbarn k von i, für den $D_{k,j} = D_{i,j} - 1$ gilt, z.B. kann man als k denjenigen Nachbarn von i wählen, der auf einem Weg der Länge $D_{i,j}$ von i nach j liegt.
Zusammen mit Lemma 2.22 erhalten wir folgende Aussage:

2.23 Lemma. *Seien $i \neq j$ Knoten aus G.*

- *$D_{i,j}$ gerade $\Rightarrow D^*_{k,j} \geq D^*_{i,j}$ für alle Nachbarn k von i.*
- *$D_{i,j}$ ungerade $\Rightarrow D^*_{k,j} \leq D^*_{i,j}$ für alle Nachbarn k von i, und es gibt einen Nachbarn k von i, für den $D^*_{k,j} < D^*_{i,j}$ ist.*

Beweis: Falls $D_{i,j} = 2x$ ist für eine ganze Zahl x, dann ist nach Lemma 2.22, Teil a), $D^*_{i,j} = x$. Da i und k Nachbarn sind, gilt $D_{k,j} \in \{2x-1, 2x, 2x+1\}$, also $D^*_{k,j} \in \{x, x, x+1\} = \{x, x+1\}$. Also ist $D^*_{i,j} = x \leq D^*_{k,j}$.
Falls $D_{i,j} = 2x+1$ ist, dann ist $D^*_{i,j} = x+1$. Da i und k Nachbarn sind, gilt $D_{k,j} \in \{2x, 2x+1, 2x+2\}$, also $D^*_{k,j} \in \{x, x+1, x+1\} = \{x, x+1\}$. Also ist $D^*_{i,j} = x+1 \geq D^*_{k,j}$. Für den Nachbarn k^* von i, der auf dem kürzesten Weg von i nach j liegt, ist $D_{k^*,j} = 2x$, also $D^*_{k^*,j} = x < D^*_{i,j}$. □

Aus Lemma 2.23 folgt direkt, wie man die Paritäten berechnen kann (hier bezeichne $\Gamma_G(i)$ die Menge der Nachbarn von i in G):

2.24 Satz. *$D_{i,j}$ ist gerade $\iff \sum_{k \in \Gamma_G(i)} D^*_{k,j} \geq |\Gamma_G(i)| \cdot D^*_{i,j}$*

Beweis: Triviale Konsequenz aus Lemma 2.23, man muss nur die Summe über alle Nachbarn k betrachten. □

Den Ausdruck $\sum_{k \in \Gamma_G(i)} D^*_{k,j}$ kann man für alle i und j gleichzeitig berechnen, durch eine Matrixmultiplikation. Es ist nämlich, wenn wir die Matrix S durch $S = A \cdot D^*$ definieren,

$$\sum_{k \in \Gamma_G(i)} D^*_{k,j} = \sum_{k=1}^{n} A_{i,k} \cdot D^*_{k,j} = S_{i,j}.$$

Wir erhalten den folgenden Algorithmus:

Algorithmus APD (All Pairs Distances)
1.) Berechne $Z = A^2$ und $A^* = A + Z$. Setze die Diagonale von A^* auf Null und Elemente, die größer als 1 sind, auf 1.
2.) Falls $A^*_{i,j} = 1$ für alle $i \neq j$ ist, **return** $2A^* - A$. (G^* ist der vollständige Graph.)

3.) Berechne $D^* := APD(A^*)$ rekursiv.
4.) Berechne $S = A \cdot D^*$.
5.) Konstruiere D wie folgt: $D_{i,j} = \begin{cases} 2 \cdot D^*_{i,j} & \text{falls } S_{i,j} \geq D^*_{i,j} \cdot Z_{i,i}. \\ 2 \cdot D^*_{i,j} - 1 & \text{falls } S_{i,j} < D^*_{i,j} \cdot Z_{i,i}. \end{cases}$
6.) **return** D.

Der Algorithmus vollzieht genau das nach, was wir uns vorher überlegt haben, es bleibt nur noch zu bemerken, dass $Z_{i,i}$ den Grad von Knoten i angibt, dass also $Z_{i,i} = |\Gamma_G(i)|$ gilt.
Die Korrektheit haben wir also bewiesen, bleibt noch die Laufzeit zu zeigen.

2.25 Satz. *Algorithmus APD berechnet die Distanzmatrix eines zusammenhängenden, ungerichteten Graphen, in dem jede Kante Gewicht 1 hat, in Laufzeit $O(M(n) \cdot \log n)$.*

Beweis: Da der Graph zusammenhängend ist, ist sein Durchmesser höchstens $n - 1$. Für den Durchmesser von G^* gilt $dm(G^*) \leq \lceil dm(G)/2 \rceil$ und somit für die Laufzeit und eine Konstante $c > 0$:

$$T(n, dm) \leq T(n, \lceil dm/2 \rceil) + c \cdot M(n),$$

sowie $T(n, 2) = O(n^2) = O(M(n))$. Daraus folgt direkt die angegebene Laufzeit. □

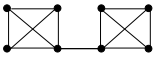
2.6 Das MINCUT–Problem

Das MINCUT–Problem spielt im VLSI–Design und überhaupt beim Einsatz als Subroutine in anderen graphentheoretischen Algorithmen eine wichtige Rolle.
(Man denke sich die Kanten als Drähte zwischen elektronischen Bauteilen. Ein Buch, das die Wichtigkeit solcher graphentheoretischer Algorithmen für den Schaltkreisentwurf verdeutlicht, ist z.B. das Buch „Combinatorial Algorithms for Integrated Circuit Layout“ von Thomas Lengauer.)

2.26 Definition. (MINCUT-Problem)
Gegeben ein ungerichteter Graph $G = (V, E)$. Zerlege die Knotenmenge disjunkt in zwei nichtleere Mengen, also $V = V_1 \cup V_2$ mit $V_1 \cap V_2 = \emptyset$, so dass die Anzahl der Kanten zwischen den beiden Mengen minimal ist.

Wir werden mit mincut(G) die Zahl der Kanten im kleinsten Schnitt bezeichnen. Manchmal werden wir auch den Schnitt mit der Menge der Kanten identifizieren, die zwischen V_1 und V_2 verlaufen.
V_1 und V_2 kann man auch als die „verschiedenen Seiten“ der Zerlegung auffassen.
Es gibt dieses Problem auch in einer gewichteten Version. Dann sind allen Kanten e Gewichte $w(e)$ zugeordnet, und es soll die Knotenmenge so zerlegt werden, dass die Summe der Gewichte der Kanten zwischen den Knotenmengen minimal wird. Die Gewichte dürfen nur nicht–negativ sein, denn wenn man negative Gewichte zuließe, erhielte

man als Spezialfall das Problem MAXCUT, das aber ein altbekanntes NP–hartes Problem ist. Für MINCUT kennt man jedoch Polynomialzeitalgorithmen, einen davon, den effizientesten bekannten deterministischen Algorithmus, werden wir vorstellen.
Wenn man dem (ungewichteten) Problem zum ersten Mal begegnet, könnte man zur Vermutung gelangen, dass in der besten Zerlegung V_1 aus einem Knoten besteht und V_2 aus den restlichen Knoten. Dies ist aber ein Irrtum, wie sich der Leser und die Leserin an folgendem Beispiel klarmachen können. Wie sieht dort eine optimale Zerlegung aus?



Wir betrachten das MINCUT–Problem, weil es hier in den letzten Jahren einige neue und interessante Ergebnisse gegeben hat.

2.7 Ein Algorithmus für MINCUT

Wir behandeln das *gewichtete* MINCUT–Problem. Das Gewicht einer Kante e sei durch eine natürliche Zahl $w(e) \geq 0$ gegeben.
Beim MINCUT–Problem interessiert man sich für die Zerlegung der Knotenmenge V in zwei Teile, so dass die Anzahl bzw. das Gewicht der Kanten zwischen den beiden Teilen möglichst klein wird, also für den minimalen Schnitt.
Seien nun Q und S zwei Knoten aus dem Graphen. Bei dem Q–S–MINCUT–Problem fordert man zusätzlich, dass die beiden Knoten Q und S nicht in der gleichen Menge der Zerlegung liegen. Man interessiert sich also für den minimalen „Q–S–Schnitt“. Seinen Wert (also das Gewicht der Kanten im minimalen Q–S–Schnitt) bezeichnen wir mit $\text{mincut}_{Q,S}(G)$.
Für unseren Algorithmus benötigen wir eine Operation „Kontraktion“, die zwei gegebene Knoten v und w „zusammenzieht“, also „kontrahiert“.

Operation Kontraktion:

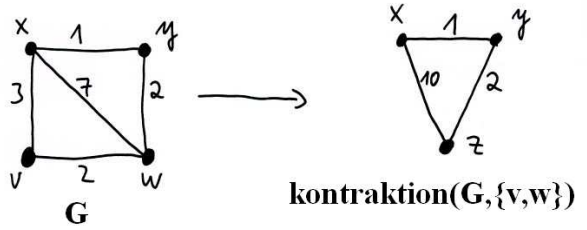
Eingabe: Ein gewichteter Graph $G = (V, E)$ und zwei Knoten $v \neq w$.
Ausgabe: Ein gewichteter Graph $G' = (V', E')$ mit $|V'| = |V| - 1$.

Der Graph G' entsteht aus G wie folgt - dabei sei Γ die Menge aller Knoten aus $V \setminus \{v, w\}$, die zu mindestens einem Knoten aus $\{v, w\}$ adjazent sind.
Kopiere zunächst den Graphen G. Entferne, falls vorhanden, die Kante zwischen v und w. Füge einen Knoten z hinzu mit Kanten $\{z, c\}$ für alle $c \in \Gamma$. Das Gewicht einer hinzugefügten Kante $\{z, c\}$ ergibt sich wie folgt:

$$w(\{z,c\}) := \begin{cases} w(\{v,c\}), & \text{falls } c \text{ Nachbar von } v, \text{ aber nicht von } w \text{ ist} \\ w(\{w,c\}), & \text{falls } c \text{ Nachbar von } w, \text{ aber nicht von } v \text{ ist} \\ w(\{v,c\}) + w(\{w,c\}), & \text{falls } c \text{ Nachbar von } v \text{ und } w \text{ ist.} \end{cases}$$

Entferne schließlich die Knoten v und w.

Die Knoten im neu entstehenden Graphen G' werden (zur besseren Unterscheidung von den Knoten in G) auch „Metaknoten“ genannt.
Den Ergebnisgraph G' nennen wir auch kontraktion$(G, \{v, w\})$.
Bei der Implementierung der Kontraktion sollte man sich am Knoten z merken, dass er aus der Kontraktion von v und w entstanden ist.
Wenn $e = \{v, w\}$ eine Kante ist, so ist logischerweise auch kontraktion(G, e) definiert.
Beispiel:



Wenn wir mehrere Kontraktionen auf einen Graphen G anwenden, dann entsteht ein Graph, dessen Metaknoten jeweils für eine (möglicherweise auch einelementige) Knotenteilmenge des ursprünglichen Graphen stehen.
Die Zeit für die Durchführung der Kontraktion ist offenbar $O(n)$ für $|V| = n$, da jeder Knoten in G maximal $O(n)$ Nachbarn hat. Wie gesagt: Hierbei ist es für spätere Zwecke von Vorteil, sich zu merken, welche Knoten zu einem Metaknoten gehören.

2.27 Satz. *Gegeben zwei Knoten Q und S aus dem ungerichteten Graphen G. Es gilt:*

$$\text{mincut}(G) = \min\{ \quad \text{mincut}_{Q,S}(G) \quad , \quad \text{mincut}(\textit{kontraktion}(G, \{Q, S\})) \quad \}.$$

Beweis: Ein Schnitt hat entweder Q und S auf verschiedenen Seiten, dann ist er auch ein Q–S–Schnitt. Oder er hat Q und S auf der gleichen Seite, dann ist er auch ein Schnitt mit gleichem Wert in kontraktion$(G, \{Q, S\})$. □

So wenig bemerkenswert diese Aussage auch scheint, sie hat die folgende Konsequenz: Wenn es uns gelingt, in jedem Graphen effizient für irgendein Q und irgendein S einen minimalen Q–S–Schnitt zu berechnen, dann können wir auch einen minimalen Schnitt effizient berechnen – entsprechend der Aussage in Satz 2.27. Den entsprechenden Algorithmus werden wir später noch genauer aufschreiben, auf jeden Fall haben wir schon die Motivation, warum es wichtig sein kann, „irgendeinen“ minimalen Q–S–Schnitt zu berechnen.
Wir zeigen, dass der folgende Algorithmus ein Q und ein S berechnet, zu denen er auch einen minimalen Q–S–Schnitt ausgibt. Wir verwenden folgende abkürzende Schreibweise: Für eine Teilmenge A der Knoten bezeichne $w(v, A)$ die Summe aller Kantengewichte von Kanten, die es zwischen v und Knoten aus A gibt.

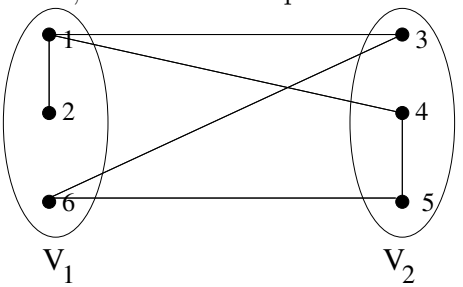
Algorithmus „Irgendein minimaler Q–S–Schnitt“
Eingabe: Graph G.
Ausgabe: Q, S. (Minimaler Q–S–Schnitt ist dann implizit gegeben, siehe Satz 2.28)
Initialisiere die Menge A mit einem beliebigen Knoten $a \in V$.

while $A \neq V$ **do begin**
Füge zu A denjenigen Knoten v aus $V \setminus A$ hinzu, für den $w(v, A)$ maximal ist.
end

Sei Q der vorletzte zu A hinzugefügte Knoten und S der letzte.

2.28 Satz. *Die Zerlegung $V_S = \{S\}$ und $V_Q = V \setminus \{S\}$ für die vom Algorithmus „Irgendein minimaler Q–S–Schnitt“ berechneten Q und S ist ein minimaler Q–S–Schnitt.*

Beweis: Die Knoten aus V werden nach und nach zu A hinzugefügt. Um uns das notationelle Leben einfach zu machen, nehmen wir $V = \{1, \ldots, n\}$ an und nummerieren die Knoten so um, dass sie genau in der Reihenfolge 1 bis n zu A hinzugefügt werden. Somit berechnet der Algorithmus $Q = n-1$ und $S = n$.
Betrachte in G einen beliebigen Q–S–Schnitt, der die Kantenmenge C hat. Für den Beweis benötigen wir auch noch die Notation C_i, die alle Kanten bezeichnet, die in C sind, aber nur zwischen Knoten aus der Menge $\{1, \ldots, i\}$ verlaufen. $w(C_i)$ bezeichnet dann die Summe aller Kantengewichte aus C_i.
Nenne einen Knoten i (mit $i > 1$) „aktiv“, wenn i und sein Vorgängerknoten $i-1$ auf verschiedenen Seiten der Zerlegung C liegen.
Bevor wir im Beweis fortfahren, ein kleines Beispiel zur Illustration:



Die Kantenmenge C enthält alle Kanten zwischen V_1 und V_2. Wie sehen zu diesem Graphen und der gewählten Zerlegung die Mengen C_1 bis C_6 aus? Es ist

$$C_1 = \emptyset, C_2 = \emptyset, C_3 = \{\{1,3\}\}, C_4 = \{\{1,3\},\{1,4\}\},$$
$$C_5 = C_4, C_6 = C.$$

Die aktiven Knoten im Beispiel sind die Knoten 3 und 6.
Wir zeigen folgende Zwischenbehauptung induktiv:

Für jeden aktiven Knoten i gilt, dass $w(i, \{1, \ldots, i-1\}) \leq w(C_i)$ ist.

Natürlich gilt die Aussage für den ersten aktiven Knoten. (Dort gilt sie sogar mit Gleichheit.) Sei die Aussage also wahr bis zum aktiven Knoten v und nehmen wir an, dass der

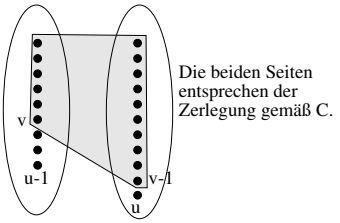
nächste aktive Knoten u ist, für den wir die Aussage zeigen wollen. Es gilt

$$w(u, \{1, \ldots, u{-}1\}) = w(u, \{1, \ldots, v{-}1\}) + w(u, \{v, \ldots, u{-}1\}) =: \alpha.$$

Schätzen wir den ersten Summanden ab: Es ist $w(u, \{1, \ldots, v{-}1\}) \leq w(v, \{1, \ldots, v{-}1\})$, weil ansonsten Knoten u vor Knoten v zur Menge A hinzugefügt worden wäre. Außerdem gilt nach Induktionsvoraussetzung: $w(v, \{1, \ldots, v{-}1\}) \leq w(C_v)$, also können wir den ersten Term in α durch $w(C_v)$ abschätzen:

$$\alpha \leq w(C_v) + w(u, \{v, \ldots, u{-}1\}).$$

Dies ist jedoch kleiner oder gleich $w(C_u)$, wie man sich an folgendem Bild klarmache, in dem die Knotennummern von oben nach unten größer werden und in dem die graue Fläche alle Kanten aus C_v enthält. Damit ist die Zwischenbehauptung gezeigt.



Nun beobachten wir, dass der Knoten $S = n$ immer ein aktiver Knoten bezüglich C ist, da Knoten $Q = n{-}1$ auf der anderen Seite liegt. Damit können wir die Zwischenbehauptung auf Knoten n anwenden und erhalten, dass

$$w(n, \{1, \ldots, n{-}1\}) \leq w(C_n) = w(C)$$

ist, was aber genau besagt, dass das Gewicht des Schnitts C mindestens so groß ist wie das Gewicht des Schnitts, der zu der Zerlegung $\{n\} \cup \{1, \ldots, n{-}1\}$ gehört. Wenn man nun C als einen minimalen Q-S-Schnitt wählt, wird die Korrektheit des Satzes deutlich. □

Satz 2.27 legt folgenden Algorithmus zur Berechnung eines minimalen Schnitts nahe:

Algorithmus MINCUT
mincutmerk := ∞; anzahl := n;
while anzahl ≥ 2 **do**
begin
  Verwende Algorithmus „Irgendein minimaler Q–S–Schnitt“, erhalte Q, S,
  sowie einen Schnitt vom Gewicht W.
  if $W <$ mincutmerk **then begin** speichere entsprechende Zerlegung ab;
    mincutmerk := W;
  end;
  Kontrahiere Q und S; anzahl = anzahl-1;
end
mincutmerk ist der Wert des minimalen Schnitts, gib die zugehörige (gemerkte) beste Zerlegung aus.

2.29 Satz. *Algorithmus MINCUT berechnet einen minimalen Schnitt und kann so implementiert werden, dass die Laufzeit $O(e \cdot n + n^2 \log n)$ beträgt.*

Beweis: Durch Induktion über die Knotenanzahl. Wenn der (Multi–)Graph aus genau 2 Knoten besteht, funktioniert der Algorithmus offensichtlich richtig. Für einen Graphen auf $n+1$ Knoten berechnet Algorithmus „Irgendein Q–S–Schnitt“ ein Q und ein S, für das er $\text{mincut}_{Q,S}(G)$ richtig berechnet. Nach Induktionsvoraussetzung berechnet Algorithmus MINCUT auch $\text{mincut}(\text{kontraktion}(G, \{Q, S\}))$ richtig. Da wir von beiden das Minimum nehmen, berechnen wir nach Satz 2.27 einen minimalen Schnitt korrekt. Zur Laufzeit bleibt zu sagen, dass es ausreicht, zu zeigen, dass die Laufzeit von Algorithmus „Irgendein minimaler Q–S–Schnitt“ durch $O(e + n \log n)$ beschränkt ist.
In diesem Algorithmus ist die entscheidende Operation diejenige, einen Knoten zu ermitteln, für den $w(v, A)$ maximal ist. Hier könnte man mit einem Heap arbeiten, der als Priority–Queue benutzt wird. Dann könnte man den Knoten v, für den $w(v, A)$ maximal ist, an der Wurzel ablesen. Durch das Hinzufügen der Knoten zur Menge A sind insgesamt $O(e)$ `increase-key`-Operationen erforderlich. (Die Gewichte $w(x, A)$ erhöhen sich, da A größer wird.)[1]
Die Initialisierung des Heaps kostet Zeit $O(n)$, eine `increase-key`-Operation kostet Zeit $O(\log n)$ und die Abfrage des Maximums via `extract-max` kostet Zeit $O(\log n)$. Wir wären also bei Laufzeit $O(e \log n + n \log n)$ für diese Implementierung des Algorithmus „Irgendein Q–S–Schnitt“.
Um die im Satz genannte Laufzeit zu erreichen, verwendet man Fibonacci–Heaps. Diese gestatten $O(e)$ `increase-key`-Operationen in Zeit $O(e)$ und die Abfrage des Maximums in (amortisierter) Laufzeit $O(\log n)$, also reicht für n Maximumsabfragen die Laufzeit $O(n \log n)$. □

[1] Nun haben wir zwar Datenstrukturen kennengelernt, die die Operation `decrease-key` unterstützt haben, aber es sollte klar sein, dass es absolut kein Problem ist, die Datenstrukturen so zu verändern, dass sie mit gleicher Effizienz die `increase-key`–Operation unterstützen. (Statt MIN–Heaps wählt man dann MAX–Heaps).

3 Einige Beispiele zur Analyse von Algorithmen

Die Analyse der Rechenzeit der in Kapitel 2 vorgestellten Algorithmen war einfach. Es wurden alle Kanten des betrachteten Graphen einmal betrachtet und die Bearbeitung einer Kante hatte meistens konstante Kosten, beim Algorithmus von Dijkstra wurde eine konstante Anzahl von Operationen auf einer priority queue angestoßen. Wenn wir eine Schleife eines Algorithmus analysieren, lässt sich die Rechenzeit stets durch das Produkt aus maximaler Anzahl von Schleifendurchläufen und maximaler Zeit für einen Schleifendurchlauf abschätzen. Oft ist diese Abschätzung zumindest asymptotisch optimal. Das Ergebnis kann aber auch ganz schlecht sein. Es gibt Algorithmen, bei denen teure Schleifendurchläufe implizieren, dass (einige) andere Schleifendurchläufe billig sein müssen. In derartigen Fällen ist es viel schwieriger, zu guten Abschätzungen der Rechenzeit zu kommen. Wir wollen in Kapitel 3.1 die beiden wichtigsten Methoden zur Analyse derartiger Algorithmen an einer bekannten Implementierung von UNION-FIND Datenstrukturen vorstellen. Es folgen dann weitere Algorithmen für zentrale Probleme, die alle einfach zu implementieren sind und für die Korrektheitsbeweise auf der Hand liegen, allerdings erfordert die Analyse einige neue Ideen. In Kapitel 3.3 analysieren wir die Pfadkomprimierung (path compression) bei baumorientierten UNION-FIND Datenstrukturen. In Kapitel 3.4 betrachten wir das einfachste und grundlegendste Problem der Mustererkennung, bei dem ein Textmuster (pattern) in einem Text (string) gesucht werden soll.

3.1 Amortisierte Rechenzeit, Buchhaltermethode und Potenzialfunktionen

Um den Begriff der amortisierten Rechenzeit einzuführen, stellen wir uns eine Datenstruktur auf n Daten vor, die eine Operation U unterstützt. Die worst case Rechenzeit für die einmalige Durchführung von U sei $t(n)$. Unter der amortisierten Rechenzeit $T(n, m)$ für m Aufrufe von U verstehen wir die Rechenzeit für eine ununterbrochene Folge von m U-Befehlen. Natürlich ist $T(n, m) \leq m \cdot t(n)$. Die Betrachtung der amortisierten Rechenzeit wird interessant, wenn wir in bestimmten Anwendungen stets eine größere Anzahl von Aufrufen von U haben und $T(n, m)$ viel kleiner als $m \cdot t(n)$ ist.
Als Beispiel betrachten wir die Analyse von UNION-FIND-Datenstrukturen.

3.2 UNION-FIND-Datenstrukturen

UNION-FIND-Datenstrukturen sollten aus der Vorlesung DAP2 bekannt sein.
Kurz zur Erinnerung: UNION-FIND-Datenstrukturen sollen die Operationen UNION und FIND auf Mengen unterstützen. Zu Beginn liegen n Elemente $1, \ldots, n$ in n einelementigen Mengen vor. Es soll eine Datenstruktur konstruiert werden, die Folgen von UNION- und FIND-Befehlen effizient verarbeiten kann.
FIND(x): Gib den Namen der Menge an, in der sich das Element x momentan befindet.
C:=UNION(A, B): Vereinige die Mengen A und B, vernichte die beiden Mengen A und B und gib den Namen der Vereinigungsmenge zurück. (Die Operation UNION darf einen

Namen für die Vereinigungsmenge wählen.)
Eine Eigenschaft, die erfüllt bleibt, ist, dass zu jedem Zeitpunkt jedes Element in genau einer Menge ist.
In der Vorlesung DAP2 hatten wir zwei verschiedene Ansätze zur Implementierung von UNION–FIND–Datenstrukturen vorgestellt. Wir betrachten zunächst die UNION-FIND Datenstruktur, in der wir ein Array mit den Objekten $1, \ldots, n$ verwalten, in dem für jedes Objekt der Name der Menge steht, in der sich das Objekt momentan befindet. Außerdem verwalten wir ein Array für die Mengen (deren Namen ebenfalls aus $\{1, \ldots, n\}$ stammen), wobei wir für jede Menge die Größe und eine Liste aller Objekte abspeichern. Da FIND-Befehle die Datenstruktur nicht verändern, betrachten wir nur UNION-Befehle U, bei denen zwei Mengen vereinigt werden. Dies geschieht durch Auflösung der kleineren Menge und Hinzufügung der darin enthaltenen Elemente zu der größeren Menge, deren Name auch für die neue Menge übernommen wird. Außerdem werden für die Objekte der kleineren Menge die Arrayeinträge aktualisiert. Die Kosten dieser Operation sind proportional zur Größe der kleineren Menge. Wir vergessen den konstanten Faktor und nehmen als Maß die Größe der kleineren Menge. Da UNION-Befehle nur für aktuelle, also nicht leere Mengen aufgerufen werden, kann eine Befehlsfolge maximal $n-1$ Befehle haben.
Offensichtlich kostet die Vereinigung zweier Mengen maximal $n/2$. Diese Kosten entstehen zum Beispiel bei der Vereinigung von $\{1, \ldots, n/2\}$ und $\{n/2+1, \ldots, n\}$. Andererseits verursacht der erste Aufruf stets nur Kosten 1. Schon nach $n/2$ Operationen kann ein Aufruf Kosten $n/4$ verursachen. Wir erhalten also die obere Schranke $(n-1)n/2$ und wir wissen auch, dass jeder Aufruf in der zweiten Hälfte der Operationen Kosten von mindestens $n/4$ verursachen kann. Hier müssen wir sorgfältig sein: jede Operation, aber nicht alle gleichzeitig! In der Tat können wir nachweisen, dass die amortisierten Kosten $\Theta(n \log n)$ sind.
Eine schlechte Operationenfolge ist für $n = 2^k$ durch einen vollständig balancierten binären Baum charakterisiert, in dem die inneren Knoten die Vereinigungen symbolisieren. Wir haben dann $n/2$ Operationen der Kosten 1, $n/4$ Operationen der Kosten 2, allgemein $n/2^m$ Operationen der Kosten 2^{m-1}, $1 \leq m \leq k$. Die Gesamtkosten betragen also

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

Allerdings ist nicht klar, dass dies wirklich eine schlechteste Operationenfolge ist.
Bei der Buchhaltermethode versuchen wir die Rechenschritte auf andere Kostenträger „umzubuchen“. Am Ende erhalten wir als obere Schranke die Summe der bei den verschiedenen Kostenträgern angefallenen Kosten. Diese können wir als Produkt aus Anzahl der Kostenträger (an Stelle der Schleifendurchläufe oder Aufrufe der Operation) und der größtmöglichen Belastung eines Kostenträgers abschätzen. Im günstigsten Fall haben wir die Kosten so umgeschichtet, dass alle Kostenträger (etwa) gleich belastet sind und erhalten eine gute obere Schranke.
In der Vorlesung DAP2 wurde eine Analye mit Hilfe der Buchhaltermethode vorgestellt. Wir wollen daher die Analyse hier nicht wiederholen, sondern eine andere Sichtweise

einnehmen, nämlich die Sichtweise einer Analyse mit Hilfe einer Potenzialfunktion.
Bei der Methode der Potenzialfunktionen messen wir den Fortschritt des Algorithmus an einer vorgegebenen Funktion. In unserem Beispiel sei für die Situation nach m UNION-Befehlen das Potenzial definiert als

$$\Phi = \sum_{1 \leq i \leq n} \varphi(i),$$

wobei $\varphi(i)$ der Logarithmus von $|M|$ ist, wenn Element i aktuell in der Menge M liegt. Das Potenzial startet mit dem Wert 0, da jedes Element in einer einelementigen Menge liegt. Bei jeder UNION(A,B)-Operation erhöht sich für jedes Element i in der kleineren der beiden Mengen der Wert $\varphi(i)$ mindestens um 1, da das Element nach der UNION-Operation in einer mindestens doppelt so großen Menge liegt. (Für den Logarithmus bedeutet das, dass er um 1 gewachsen ist.) Für die anderen Elemente wird $\varphi(i)$ auf keinen Fall kleiner.
Da die Laufzeit der UNION-Operation im Wesentlichen gleich der Anzahl der Objekte in der kleineren Menge ist, so ist die vom Algorithmus aufgewendete Laufzeit immer kleiner als der aktuelle Potenzialwert.
Die Summe aller φ-Werte ist aber immer durch $n \log n$ beschränkt, da jede Menge höchstens n Elemente enthält und somit $\varphi(i) \leq \log n$ für alle Elemente i ist. Also gilt, dass auch die Laufzeit des Algorithmus insgesamt durch $n \log n$ beschränkt ist.
Die Buchhaltermethode und die Methode der Potenzialfunktionen können nicht schematisch eingesetzt werden. Es bedarf weiterhin der richtigen Intuition, um diese Methoden so einzusetzen, dass sie zu guten Ergebnissen führen.
Wir wenden uns nun einer Analyse der zweiten UNION-FIND-Datenstruktur zu.

3.3 Eine UNION–FIND–Datenstruktur mit schnellen UNION–Befehlen

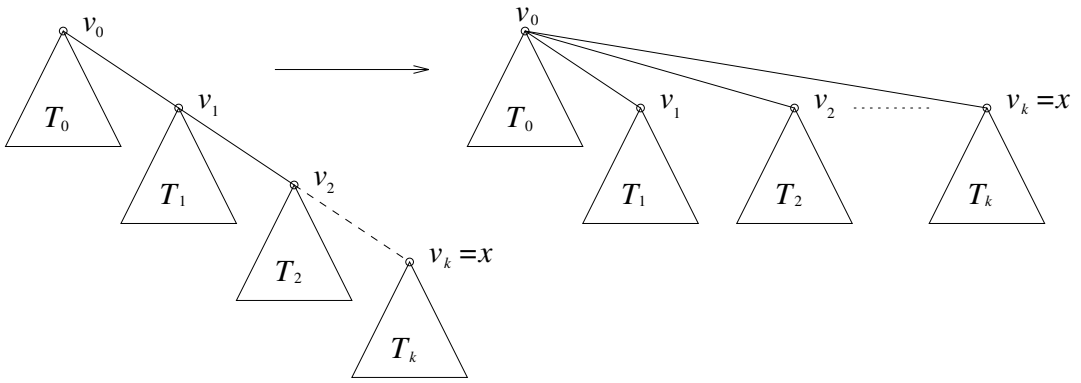
Wir stellen nun Mengen durch Bäume dar, bei denen die Kanten in Richtung auf die Wurzel zeigen. An der Wurzel steht der Name der zugehörigen Menge und die Mächtigkeit der Menge. Für einen FIND-Befehl muss man vom Element x aus bis an die Wurzel des zugehörigen Baumes laufen. Die Rechenzeit ist proportional zur Weglänge. Für einen UNION-Befehl verschmelzen wir die zugehörigen Bäume, indem wir die Wurzel des ersten Baumes auf die Wurzel des zweiten Baumes zeigen lassen und dort den Namen der neuen Menge und ihre Mächtigkeit notieren.
Große Bäume tendieren dazu, längere Wege zu haben als kleine Bäume. Daher wählen wir stets den Baum mit der geringeren Knotenzahl als denjenigen Baum, den wir in den anderen Baum einhängen.
Wir wollen eine Implementierungsmöglichkeit andeuten: Wir verwenden zwei Arrays, `vater`$[i]$ sowie `size`$[i]$ für $i = 1, \ldots, n$. `vater` soll die Nummer des Vaterknotens angeben. Falls kein Vater vorhanden ist (wir also an der Wurzel sind), dann soll `vater`$[i] = i$ sein.

Initialisierung: `vater`$[i] = i$ und size$[i]$=1 für alle $i = 1, \ldots, n$.

FIND(x): **while** `vater`$[x] \neq x$ **do** x :=`vater`$[x]$;
return x;

UNION(a, b): **if** size$(a) >$ size(b) **then** vertausche a und b.
`vater`$[a] = b$.
size(b) = size(b)+size(a). **return** b.

Jeder UNION-Befehl ist in Zeit $O(1)$ durchführbar. Für FIND(x) ist die Länge des Weges von x zu „seiner“ Wurzel ausschlaggebend. Dieser Weg kann kurz sein, dann ist FIND effizient durchführbar. Wenn FIND(x) teuer ist, erhalten wir nebenbei viel Information über die zugehörige Menge.
Wenn FIND(x) die Knoten $x = v_k, \ldots, v_0$ besucht, so hat dieser Aufruf die Kosten $O(k{+}1)$. (Den Term $+1$ müssen wir hier deswegen nehmen, weil $k{=}0$ sein kann!)
Wenn wir hinterher die Zeiger von $v_k, \ldots, v_1$ direkt auf v_0 zeigen lassen, enthält der Baum, d. h. die Menge, die gleichen Elemente wie zuvor. Für viele Knoten, d. h. Elemente, ist jedoch der Weg zur Wurzel kürzer geworden. Für diese Elemente sind die Kosten zukünftiger FIND-Befehle gesunken.



3.1 Beispiel.

Mit diesem Trick, „Pfadkompression“ (auf Englisch: Path Compression) genannt, bauen wir für die Zukunft vor. Wir wollen zeigen, dass sich diese Vorsorge lohnt.
Vorher sehen wir uns eine Möglichkeit an, die Pfadkompression in den obigen Code zu integrieren: Die Implementierung von FIND wird wie folgt geändert:

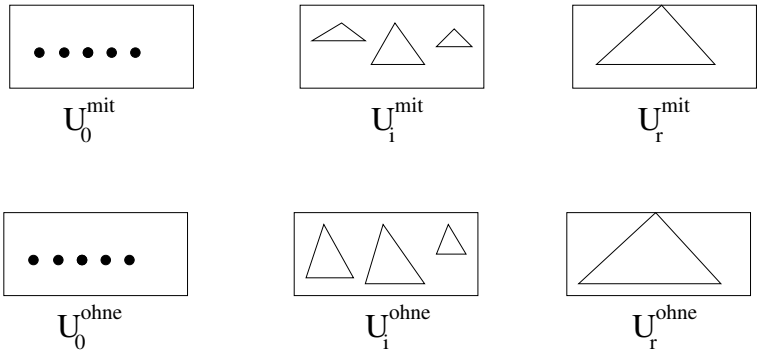
FIND(x): wurzel:= x
while `vater`[wurzel] $\neq$ wurzel **do** wurzel:=`vater`[wurzel];
while $x \neq$ wurzel **do** v :=`vater`$[x]$;
`vater`$[x]$:= wurzel
$x := v$.
end;
return wurzel.

3.2 Lemma. *Wenn der eine Menge darstellende Baum Tiefe h hat, d. h. die Länge des*

längsten Weges von einem Blatt zur Wurzel beträgt h, dann enthält der Baum mindestens 2^h Knoten. (Dies gilt auch ohne die Durchführung von Pfadkompression.)

Beweis: Da FIND-Befehle auch bei Verwendung von Pfadkompression die Anzahl der Knoten in einem Baum nicht verändern, müssen wir uns nur Bäume ansehen, die durch UNION-Befehle entstanden sind.
Wir führen nun eine Induktion über h. Für $h = 0$ hat der Baum einen Knoten. Sei nun T ein Baum, der unter allen Bäumen mit Tiefe h die kleinste Knotenzahl hat. T entstand durch Vereinigung der Bäume T_1 und T_2, wobei o.B.d.A. $\text{size}(T_1) \leq \text{size}(T_2)$ ist. Da T_1 an T_2 angehängt wird, ist $\text{tiefe}(T_1) \leq h - 1$. Falls $\text{tiefe}(T_1) < h - 1$, ist, da $\text{tiefe}(T) = h$, $\text{tiefe}(T_2) = h$. Dies ist ein Widerspruch zur Konstruktion von T, da $\text{size}(T_2) < \text{size}(T)$ ist. Also gilt $\text{tiefe}(T_1) = h - 1$. Nach Induktionsvoraussetzung folgt $\text{size}(T_1) \geq 2^{h-1}$ und $\text{size}(T_2) \geq \text{size}(T_1) \geq 2^{h-1}$. Also ist $\text{size}(T) = \text{size}(T_1) + \text{size}(T_2) \geq 2^h$. □

Da wir n Elemente verwalten, sind die Kosten der FIND-Befehle auch ohne Pfadkompression durch $\log n$ beschränkt. Allerdings sind Kosten von $\Theta(\log n)$ auch möglich (Übungsaufgabe). Wir wollen nun erarbeiten, welche Ersparnis Pfadkompression bringt.
Wir analysieren den Verlauf der Datenstruktur bei Anwendung einer UNION–FIND–Befehlsfolge, die $n - 1$ UNION–Befehle enthält. Wir gehen davon aus, dass die Befehlsfolge r Befehle enthält und schauen uns an, wie sich die Datenstruktur entwickeln würde: Einmal mit Pfadkompression, und einmal ohne Pfadkompression.
Und zwar sei $U_0^{mit} = U_0^{ohne}$ die Datenstruktur aus den n einelementigen Bäumen. Allgemein sei U_i^{mit} die Datenstruktur, die sich aus U_0^{mit} nach Abarbeitung von i Befehlen der Befehlsfolge ergibt, wenn die Pfadkompression verwendet wird. Analog für U_i^{ohne}, wenn Pfadkompression nicht verwendet wird. Wir stellen uns das an folgendem Bild vor.

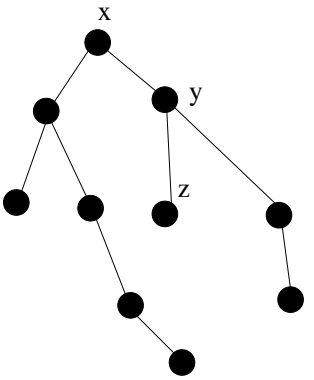


Da die UNION–FIND–Befehlsfolge $n - 1$ UNION–Befehle enthält, bestehen die Datenstrukturen U_r^{ohne} und U_r^{mit} aus jeweils einem einzelnen Baum.
Wir beobachten folgendes:

- FIND–Befehle ändern bei Pfadkompression zwar etwas am Aussehen des Baumes, durch den der FIND–Befehl läuft, aber
  a) die Knotenzahl des Baumes bleibt unverändert.
  b) die Wurzel des Baumes bleibt unverändert.

– UNION–Befehle hängen Bäume ineinander. Die Entscheidung, welche Wurzel in die andere eingehängt wird, hängt nur von der Größe, d.h. Knotenzahl der beiden beteiligten Bäume ab.

Aus diesen beiden Beobachtungen ergibt sich: Wenn es in U_i^{ohne} einen Baum mit Wurzel x gibt, der die Menge S darstellt, so gibt es auch in U_i^{mit} einen Baum mit Wurzel x, der die Menge S darstellt. (Auch, wenn er ansonsten ein anderes Aussehen haben kann.) Der Leser und die Leserin können sich das mit den beiden obigen Beobachtungen unter Verwendung einer einfachen Induktion klarmachen.
Zur Analyse der Laufzeit schaut man sich den Baum in der Datenstruktur U_r^{ohne} genauer an und definiert für jedes Element 1 bis n den Rang des Elements über diesen Baum.
Rang(x) sei die Länge des längsten Wegs von einem Blatt aus zu x im Baum U_r^{ohne}. (Achtung: Die Zeiger im Baum zeigen von unten nach oben.)
Beispiel:



Im Beispielbild ist Rang(x) = 4, Rang(y) = 2, Rang(z) = 0.
Der Rang eines Elements in den Datenstrukturen ist also stets fixiert und hängt nur vom Aussehen des „letzten Baumes“ ab, d.h. des Baumes in der Datenstruktur U_r^{ohne}.
Auch sollte klar sein, dass im Baum der Datenstruktur U_r^{ohne} die Ränge „von unten nach oben wachsen“, d.h. es gilt die „Rangwachstumseigenschaft“:
Für alle v, w gilt: v Vater von $w \Rightarrow$ Rang(v) > Rang(w).
Wir wollen zunächst zeigen, dass die Rangwachstumseigenschaft in allen Bäumen gilt, die in irgendeiner der Datenstrukturen U_i^{mit} vorkommen. (Achtung: Auch wenn wir andere Bäume betrachten, der Rang ist immer über den Baum in der Datenstruktur U_r^{ohne} definiert ! Das kann man nicht oft genug betonen!)

3.3 Lemma. *Sei i beliebig. Wenn in der Datenstruktur U_i^{mit} das Element v Vater von Element w ist, dann gilt Rang(v) > Rang(w).*

Beweis: Zu Beginn, für $i = 0$, liegen nur n einelementige Mengen vor, dort ist kein Knoten Vater eines anderen, also ist die Aussage sicherlich erfüllt. Für den Induktionsschritt nehmen wir an, dass die „Rangwachstumseigenschaft“ in den Datenstrukturen $U_0^{mit}, \ldots, U_{i-1}^{mit}$ gilt.
Wenn U_i^{mit} aus U_{i-1}^{mit} durch einen FIND-Befehl entsteht, so gilt immer noch die Rangwachstumseigenschaft, denn es werden ja nur Knoten weiter nach oben (d.h., an die

Wurzel) gehängt, und dort ist der Rang noch höher (denn in U_{i-1}^{mit} galt die Rangwachstumseigenschaft ja.)
Also betrachten wir den Fall, dass U_i^{mit} aus U_{i-1}^{mit} durch einen UNION–Befehl entstanden ist. Nehmen wir an, dass der Baum mit der Wurzel t_2 in den Baum mit der Wurzel t_1 eingehängt wird. Die einzige neue Vater–Sohn–Beziehung im entstehenden Baum ist die zwischen den beiden Elementen t_1 und t_2. Wir müssen nun also den Rang dieser beiden Elemente miteinander vergleichen. Unsere Überlegungen vorhin haben gezeigt, dass in der Folge U_j^{ohne} die gleichen Wurzeln ineinandergehängt werden, also entsteht auch U_i^{ohne} aus U_{i-1}^{ohne} durch Einhängen eines Baumes mit Wurzel t_2 in einen Baum mit Wurzel t_1. In der Datenstrukturfolge ohne Pfadkompression jedoch bleiben Vater–Sohn–Beziehungen erhalten, das heißt, wenn einmal Knoten v Vater von Knoten w ist, so bleibt v Vater von w. Das heißt aber auch, dass im Baum U_r^{ohne} das Element t_1 immer noch Vater von Element t_2 ist, also ist nach Definition des Ranges $\text{Rang}(t_1) \geq \text{Rang}(t_2)$ und wir sehen, dass auch im Baum U_i^{mit} die Rangwachstumseigenschaft erfüllt ist. □

Nun schauen wir uns weitere Eigenschaften an:

3.4 Lemma. *Es gibt höchstens $n/2^t$ Elemente, die den Rang t haben.*

Beweis: Wir betrachten den Baum U_r^{ohne}. Nach Lemma 3.2 hat der Teilbaum mit Wurzel x, wenn $\text{Rang}(x) = t$ ist, mindestens 2^t Knoten. Kein Element mit Rang t kann Vorgänger oder Nachfolger eines anderen Elements mit Rang t sein. Gäbe es mehr als $n/2^t$ Elemente mit Rang t, so müsste es also mehr als n Elemente geben. □

Um die Laufzeit angeben zu können, benötigen wir zwei Funktionen:

3.5 Definition. *Es sei $F(0) := 1$ und $F(i) := 2^{F(i-1)}$ sowie*

$$\log^* n := \min\{k \mid F(k) \geq n\}.$$

$\log^* n$ ist gewissermaßen die „Umkehrfunktion“ zu F.

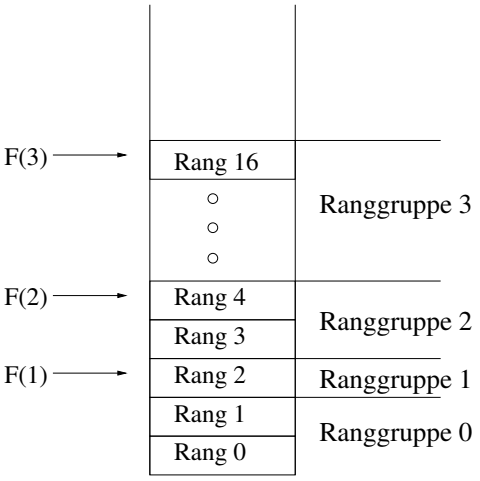
F wächst „wahnsinnig schnell“, $\log^*$ wächst „wahnsinnig langsam“,

i	0	1	2	3	4	5
$F(i)$	1	2	4	16	65536	2^{65536}

i	0	1	2	3	4	$\cdots$	100000
$\log^* i$	0	0	1	2	2	$\cdots$	5

$F(k)$ ist ein Turm aus k Zweien. Es ist $\log^* n \leq 5$ für alle $n \leq 2^{65536}$. Man kann $\log^* n$ auch anders interpretieren: $\log^* n$ gibt an, „wie oft man n logarithmieren muss, um eine Zahl $a \leq 1$ zu erhalten.“
Wir können nicht verhindern, dass einzelne FIND–Befehle Kosten von $\Theta(\log n)$ verursachen. Wir wollen aber zeigen, dass in einer UNION–FIND–Befehlsfolge mit $n-1$ UNION–Befehlen und m FIND–Befehlen die Kosten der FIND–Befehle insgesamt nur $O((m+n) \cdot \log^* n)$ betragen.

Dazu wollen wir die Buchhaltermethode verwenden. Kosten, die anfallen, werden auf verschiedenen Kostenkonten verbucht. Schließlich analysieren wir, was auf diesen einzelnen Konten höchstens verbucht werden kann.
Wir haben den Elementen Ränge zugeordnet. Verschiedene Ränge werden nun zu Ranggruppen zusammengefasst, beginnend mit Rang 0.
Wir teilen die Ränge so ein, dass $F(i)$ der höchste Rang in Ranggruppe i ist. Da $F(0) = 1, F(1) = 2, F(2) = 4, F(3) = 16$ ist, erhalten wir zum Beispiel die Einteilung der Ränge wie in folgendem Bild:



Ranggruppe $i \geq 1$ umfasst also alle Ränge von $F(i-1)+1$ bis $F(i)$. Umgekehrt überlegt man sich leicht, dass ein Element mit Rang r in Ranggruppe $\log^* r$ liegt. Diese Wahl der Ranggruppen wird sich bei der folgenden Rechnung als „gut“ erweisen.
Sei nun FIND(x) ein Befehl, dessen Weg zur Wurzel der Weg $x = v_k \to v_{k-1} \to \cdots \to v_0$ ist.
Welche Kostenkonten verwenden wir? Für jeden FIND–Befehl steht ein Konto zur Verfügung und für jedes Element $1, \ldots, n$ steht ein Konto zur Verfügung. Für den Befehl FIND(x) haben wir $k+1$ Kosteneinheiten zu verteilen. Eine Einheit dafür, dass der Befehl aufgerufen wurde sowie eine Einheit für jede betrachtete Kante. Wir verteilen nun diese $k+1$ Kosten wie folgt auf die Konten: Eine Einheit kommt in das Konto für den FIND–Befehl dafür, dass der Befehl aufgerufen wurde. Eine Einheit für eine Kante $v_1 \to v_0$ kommt auf das Konto für den FIND–Befehl. Für die anderen Kanten $v_i \to v_{i-1}$ hängt die Wahl des Kontos von den Ranggruppen der beteiligten Elemente ab:

- v_i in einer anderen Ranggruppe als v_{i-1}:
  Verbuche Kosten 1 auf dem Konto des FIND–Befehls.
- v_i in der gleichen Ranggruppe wie v_{i-1}:
  Verbuche Kosten 1 auf dem Konto des Elements v_i.

Es ist klar, dass wir die Gesamtkosten erhalten, wenn wir die Kosten in allen Konten aufaddieren.
Schauen wir uns zunächst das Konto eines FIND–Befehls an: Dort können nur $\log^* n + 2$ viele Kosteneinheiten verbucht werden: maximal zwei für die Kante $v_1 \to v_0$ und für den

Aufruf. Da die höchste auftretende Ranggruppe $\log^* n$ ist, können wir beim Hochlaufen eines Pfades nur $\log^* n$ mal die Ranggruppe wechseln.
In den Konten aller m FIND–Befehle zusammen genommen können also höchstens $m(\log^* n + 2) = O(m \cdot \log^* n)$ Kosteneinheiten verbucht werden.
Nun kommen wir zu den Konten für die Elemente 1 bis n. Wenn ein Element bei unserer Kostenaufteilung eine Kosteneinheit erhalten hat, dann bekommt es durch die Pfadkompression einen neuen Vater, nämlich v_0. Der Rang dieses Vaters ist wegen der Rangwachstumseigenschaft also um mindestens 1 größer als der Rang des vorigen Vaters. Wir betrachten ein Element in Ranggruppe $g \geq 1$. Sein Rang liegt dann also in $\{F(g-1)+1, \ldots, F(g)\}$. Spätestens nachdem dieses Element $F(g) - F(g-1)$ Kosteneinheiten zugeteilt bekommen hat, liegt der Vater in einer höheren Ranggruppe. Danach wird diesem Element keine weitere Kosteneinheit zugeordnet. Jedes Element in Ranggruppe $g \geq 1$ erhält also höchstens $F(g) - F(g-1) \leq F(g)$ Kosteneinheiten zugeteilt. Ein Element in Ranggruppe 0 erhält höchstens eine Zuweisung, es ist $1 = F(0)$.
Es sei $N(g)$ die Zahl der Elemente in Ranggruppe g, also die Zahl der Elemente mit Rang in $\{F(g-1)+1, \ldots, F(g)\}$. Nach Lemma 3.4 ist

$$\begin{aligned} N(g) &\leq \sum_{r=F(g-1)+1}^{F(g)} \frac{n}{2^r} \leq \sum_{r=F(g-1)+1}^{\infty} \frac{n}{2^r} \\ &= \frac{n}{2^{F(g-1)+1}} \cdot (1 + \frac{1}{2} + \frac{1}{4} + \cdots) = \frac{n}{2^{F(g-1)}} = n/F(g). \end{aligned}$$

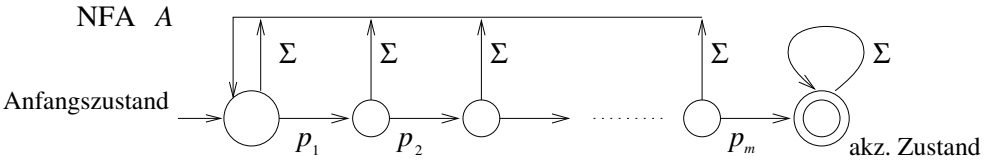
Die letzte Gleichung gilt, da $2^{F(g-1)} = F(g)$ ist.
Die Gesamtzahl der Kosteneinheiten, die Elemente aus Ranggruppe g erhalten, ist durch $N(g)F(g) \leq n$ beschränkt. Da nur $\log^* n + 1$ Ranggruppen betrachtet werden, ist die Gesamtzahl der Kosteneinheiten für die Elemente höchstens $n \cdot (\log^* n + 1)$. Zusammenfassend erhalten wir folgenden Satz.

3.6 Satz. *Für die UNION–FIND–Datenstruktur mit Bäumen und Pfadkompression gilt: Die $n-1$ UNION–Befehle und m FIND–Befehle können in Zeit $O((n+m) \cdot \log^* n)$ durchgeführt werden.*

3.4 String Matching

Wir wollen hier das einfachste Problem aus dem Gebiet der Mustererkennung diskutieren. Die Aufgabe besteht darin, einen String $S = (s_1, \ldots, s_n)$ darauf zu untersuchen, ob er ein Muster (Pattern) $P = (p_1, \ldots, p_m)$ als Teilstring enthält. Das Problem hat eine einfache Lösung. Teste für $i \in \{0, \ldots, n-m\}$, ob $p_j = s_{i+j}$ für alle $j \in \{1, \ldots, m\}$ gilt. Also genügen $O(nm)$ Rechenschritte. Diese worst–case–Rechenzeit wird z. B. für $S = (a, a, \ldots, a)$ und $P = (a, a, \ldots, a, b)$ erreicht. Bevor wir einen besseren, weil schnelleren, Algorithmus beschreiben, betrachten wir das Problem und den Algorithmus ganz kurz aus der Sichtweise der endlichen Automaten. Hier besteht das String Matching Problem für das Grundalphabet Σ in der Erkennung der regulären Sprache

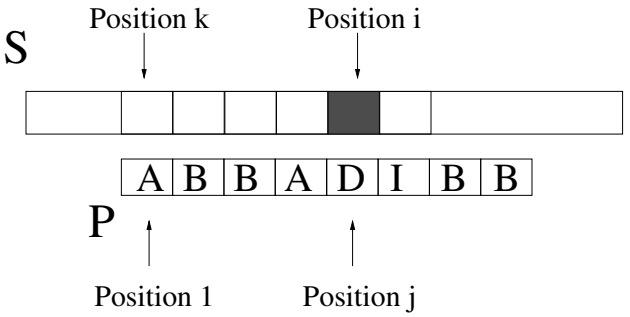
$SM = \Sigma^* p_1 \dots p_m \Sigma^*$. Der folgende nichtdeterministische endliche Automat A (NFA) erkennt genau die Sprache SM.



Aus der Vorlesung über endliche Automaten wissen wir, dass es einen deterministischen endlichen Automaten A' (DFA) gibt, der A simuliert. Prinzipiell gilt zwar, dass der kleinste DFA für eine Sprache L exponentiell viele Zustände mehr haben kann als ein NFA für L, der Leser und die Leserin mögen sich aber nach Lektüre dieses Kapitels überlegen, dass ein DFA für die Sprache SM existiert, der $m + 1$ Zustände hat. (Wer sich mit der Theorie auskennt, kann mit Hilfe der Nerode-Relation auch nachweisen, dass mindestens $m + 1$ Zustände gebraucht werden, da von den $m + 1$ Wörtern $v_i = p_1 \cdots p_i$, $0 \leq i \leq m$, keine zwei Nerode-äquivalent sind.) Wir erhalten damit einen deterministischen Algorithmus, der in optimaler Zeit, nämlich in n Schritten, entscheidet, ob $S \in SM$ ist. Allerdings kann es sehr aufwendig sein, den DFA A' zu konstruieren. Zum Beispiel muss man ja bei einem DFA zu jedem Zustand und jedem Buchstaben spezifizieren, welches der Nachfolgezustand ist. Für ein großes Alphabet Σ kann dann schon das Abspeichern der entsprechenden Tabelle unpraktikabel sein. Der in diesem Kapitel beschriebene Algorithmus unterscheidet sich bei genauem Hinsehen vom Vorgehen eines endlichen Automaten nur dadurch, dass der Buchstabe, an dem man sich in dem zu durchsuchenden Text S befindet, mehrfach hintereinander gelesen werden darf. Ein Automat hingegen darf einen Buchstaben nur einmal lesen und muss dann durch einen Zustandswechsel reagieren. Damit wollen wir den kleinen Exkurs in die endlichen Automaten beenden.

Auf jeden Fall sollte man die Preprocessingzeit für den Aufbau einer Datenstruktur für ein Pattern P von der Rechenzeit für einen speziellen String S unterscheiden. Knuth, Morris und Pratt haben gezeigt, wie in Zeit $O(m)$ eine Datenstruktur aufgebaut werden kann, mit der in Zeit $O(n)$ entschieden werden kann, ob S das Pattern P enthält. Ähnlich wie für den NFA A wollen wir ein Diagramm mit $m + 1$ Knoten aufbauen. Die Knoten sollen mit $0, \dots, m$ bezeichnet werden. Die Datenstruktur soll es ermöglichen, dass wir beim Lesen von S kontrollieren können, „wie weit der String das Muster enthält“. Wenn das Muster gefunden wurde, kann die Suche abgebrochen werden. Ansonsten können wir von S nur einen Suffix gebrauchen, der mit einem Präfix von P übereinstimmt.

Angenommen, wir suchen in S nach dem Pattern P und geraten in folgende Situation:



Dies soll bedeuten: Wir haben das Pattern an der Position k im String S angelegt und eine Übereinstimmung in den Buchstaben „ABBA“ gefunden, allerdings sind wir an der Stelle mit dem Buchstaben „D“ auf einen Mismatch (einen nicht passenden Buchstaben) gestoßen. Müssen wir nun im nächsten Schritt das Pattern an der Stelle k+1 anlegen und nachsehen, ob wir dort ein Vorkommen von P finden? Nein, das müssen wir nicht, denn da wir an den Positionen „ABBA“ eine Übereinstimmung hatten, brauchen wir P nicht an der Stelle k+1 anzulegen, denn dort würde sofort ein Mismatch entstehen, weil wir ein A(in P) unter ein B (in S) gelegt hätten. Ausprobieren zeigt, dass wir das Pattern erst wieder unter das letzte „A“ von „ABBA“ anlegen müssen, um dort nach einem Vorkommen zu suchen.
Wichtig an dieser Überlegung ist, dass wir die Information, wie weit wir das Pattern nach rechts „durchrutschen“ lassen können, nur aus dem Pattern selbst und der Position, an der ein Mismatch aufgetreten ist, berechnen können. Im Beispiel brauchen wir nur zu wissen, dass ein Mismatch an Position $j = 5$ aufgetreten ist, dann können wir das Pattern soweit durchrutschen lassen, dass anschließend die Position 2 unter dem schraffierten Feld zu stehen kommt.
Der Pattern Matching Algorithmus funktioniert nun in zwei Phasen. In der ersten Phase berechnet man für alle j aus dem Pattern die Information F, wie weit man das Pattern durchrutschen lassen kann, wenn an der Stelle j ein Mismatch auftritt. $F(j)$ (auch „Fehlerkantenfunktion“ genannt) gibt dabei die Nummer des Feldes an, das anschließend unter der i-ten Position in S zu liegen kommt. Im obigen Beispiel hätten wir $F(5) = 2$, da beim Mismatch an der 5-ten Position das Pattern soweit durchrutschen kann, dass anschließend die Position Nummer 2 unter der i–ten Position in S zu liegen käme.
Übrigens ist es damit auch naheliegend, $F(1) = 0$ zu definieren, denn beim Mismatch gleich an der ersten Stelle kann man das Pattern eine Stelle weiterrutschen lassen, so dass die „imaginäre“ nullte Stelle unter der Position i zu liegen kommt.
Wie wir F effizient berechnen, werden wir gleich sehen. Auch werden wir sehen, dass $F(j) < j$ für alle j ist.
Nehmen wir einmal an, dass wir F schon berechnet haben. Dann sieht der Pattern Matching Algorithmus wie folgt aus:

3.7 Algorithmus. *String Matching*
Input: $n; m; P = (p_1, \ldots, p_m) \in \Sigma^m$; $S = (s_1, \ldots, s_n) \in \Sigma^n$; $F(1), \ldots, F(m)$.
Output: Erfolg oder kein Erfolg, abhängig davon, ob P in S enthalten ist.
$j := 0$;
for $i := 1$ **to** n **do**
begin $j := j + 1$;
    while $j \neq 0$ *und* $p_j \neq s_i$ *setze* $j := F(j)$.
    Falls $j = m$, *Output „Erfolg“ und STOP.*
end;
Output „kein Erfolg“.

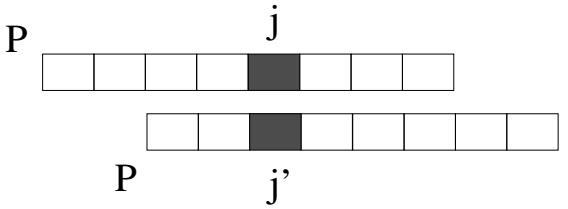
3.8 Satz. *Wenn die Fehlerkanten $F(1), \ldots, F(m)$ gegeben sind, dann löst Algorithmus 3.7 das String Matching Problem in $O(n)$ Rechenschritten.*

Beweis: Die Korrektheit folgt aus unseren Vorüberlegungen. Für die Rechenzeit betrachten wir die „Potenzialfunktion“ $\Phi := 2i - j$. Zu Beginn des Algorithmus ist $\Phi = 0$ (wir denken uns am Anfang $i := 0$). Wir beobachten, dass Φ nach einer konstanten Anzahl an Schritten mindestens um 1 größer wird: Durch die Zuweisung $j := F(j)$ wird Φ um mindestens 1 größer, und wenn $j := j + 1$ ausgeführt wird, ist Φ insgesamt auch um 1 größer geworden, weil vorher i um 1 erhöht worden ist. Da Φ aber durch $2n$ beschränkt ist, ergibt sich die Laufzeit $O(n)$. □

Kommen wir nun zur effizienten Berechnung der Fehlerkanten $F(j)$: Wir definieren:

3.9 Definition. *Das Paar (j, j') mit $j > j' \geq 1$ heißt Präfixmatch, falls folgende Situation vorliegt: $p_1 \cdots p_{j'-1} = p_{j-j'+1} \cdots p_{j-1}$.*

An einem Bild verstehen wir die Definition besser:



D.h., wir können das Pattern so unter sich selbst legen, dass die j'–te Stelle unter der j–ten Stelle zu liegen kommt und wir haben in dem Wort vor der j'–ten Stelle eine Übereinstimmung.
Nach unseren obigen Überlegungen sollte klar sein, dass alle Präfixmatches (j, j') Kandidaten sind, wie wir bei einem Mismatch an der j–ten Stelle das Wort wieder anlegen können. Damit wir kein mögliches Vorkommen von P übersehen, dürfen wir P natürlich nur soweit wie möglich durchrutschen lassen, genauer:

$$F(j) = \max\{j' \mid (j, j') \text{ ist Präfixmatch}\}.$$

Da für $j = 1$ die Menge, über die das Maximum gebildet wird, leer ist, definiert man $F(1) = 0$ extra.

Betrachten wir als Beispiel das Wort `ABACABABA`.
Es ist

j	1	2	3	4	5	6	7	8	9
$F(j)$	0	1	1	2	1	2	3	4	3

Schauen wir uns beispielhaft an, wie man aus der obigen Definition $F(9)$ berechnen könnte. Welche Paare $(9, j')$ sind Präfixmatche? Dies sind, wie man durch Ausprobieren herausbekommen kann, die Paare $(9, 3)$ sowie $(9, 1)$, also $F(9) = 3$.
Welche Paare $(8, j')$ sind Präfixmatche? $(8, 4)$, $(8, 2)$ und $(8, 1)$. Also $F(8) = 4$.
Wir wollen die Fehlerkanten $F(j)$ aufsteigend für alle j berechnen. $F(1) = 0$ und $F(2) = 1$ sind dabei obligatorisch.
Wann ist $(j, j' + 1)$ ein Präfixmatch? Wenn wir uns hier wieder ein Bild hinmalen (das sei an dieser Stelle dem Leser und der Leserin überlassen), dann stellen wir fest, dass $(j, j' + 1)$ genau dann ein Präfixmatch ist, wenn $(j - 1, j')$ ein Präfixmatch ist und $p_{j'} = p_{j-1}$ ist.
Um $F(j)$ zu ermitteln, müssen wir also nur alle Präfixmatche durchlaufen, die von der Form $(j - 1, \cdot)$ sind, also $j - 1$ als erste Komponente besitzen. Diese durchlaufen wir so, dass die zweite Komponente in $(j - 1, \cdot)$ immer kleiner wird. Wir testen für jeden Präfixmatch $(j - 1, j')$, ob $p_{j'} = p_{j-1}$ ist und falls ja, ist für den ersten so gefundenen Präfixmatch $F(j) = j' + 1$.

3.10 Algorithmus. *Fehlerkantenberechnung*
Input: m; $P = (p_1, \ldots, p_m) \in \Sigma^m$.
Output: $F(1), \ldots, F(m)$, *die Fehlerkanten.*
$F(1) := 0$; $F(2) := 1$;
$j' := 0; j := 2;$ *# eigentlich überflüssig, dient nur der Analyse*
for $j := 3$ **to** m **do**
begin $j' := F(j-1)$.
  while $j' \neq 0$ *und* $p_{j'} \neq p_{j-1}$, *setze* $j' := F(j')$.
  $F(j) := j' + 1$;
end;

3.11 Satz. *Algorithmus 3.10 berechnet die Fehlerkanten in $O(m)$ Rechenschritten.*

Beweis: Die Korrektheit folgt aus unseren Vorüberlegungen. Die **while**–Schleife durchläuft alle Präfixmatche mit $j - 1$ als erster Komponente.
Für die Rechenzeit beobachten wir, dass man in der for-Schleife den Befehl $j' := F(j-1)$ auch durch den Befehl $j' := j' + 1$ ersetzen könnte, da am Ende der for-Schleife $F(j) = j' + 1$ gesetzt wird und beim nächsten Durchlauf j um Eins erhöht ist. Die Struktur des Algorithmus ist also die gleiche wie beim Algorithmus 3.7 und mit der Potenzialfunktion $2j - j'$, die diesmal durch $2m$ beschränkt ist, kann man die Laufzeit $O(m)$ nachweisen. □

Die Berechnung der Fehlerkanten baut eine Datenstruktur F auf, die für das String Matching vieler Strings benutzt werden kann. Wir unterscheiden bei derartigen Problemen

die Preprocessingzeit für den Aufbau einer Datenstruktur und die Query Zeit für die Verarbeitung einer Eingabe, hier eines Strings.

3.12 Satz. *Das String Matching Problem kann mit einer Preprocessing Zeit von $O(m)$ und einer Query Zeit von $O(n)$ gelöst werden.*

4 Approximationsalgorithmen

4.1 Einleitung

Wenn es zu schwierig ist, Optimierungsprobleme exakt zu lösen, sollten wir uns mit Approximationslösungen zufrieden geben. Hier wollen wir Approximationsalgorithmen untersuchen, die eine bestimmte Güte der Lösung garantieren. Wenn wir beim TSP eine Tour mit Kosten 140 berechnen, während die optimalen Kosten 100 betragen, wie messen wir dann die Güte unserer Lösung? Es ist nicht vernünftig, die Differenz $140 - 100 = 40$ zu wählen. Wenn wir nämlich alle Kostenwerte um den Faktor 10 erhöhen (eine andere Maßeinheit wählen), haben wir es praktisch mit demselben Problem zu tun. Die Güte derselben Lösung wäre aber auf $1400 - 1000 = 400$ gestiegen. Daher wählen wir den Quotienten $140/100 = 1{,}4$, der nach Änderung der Maßeinheit unverändert bleibt: $1400/1000 = 1{,}4$.

4.1 Definition. *Bei Minimierungsproblemen messen wir die Güte einer Lösung durch den Quotienten aus dem Wert der berechneten Lösung und dem Wert einer optimalen Lösung. Bei Maximierungsproblemen messen wir die Güte durch den Quotienten aus dem Wert einer optimalen Lösung und dem Wert der berechneten Lösung.*

Durch diese Konvention ist die Güte eine Zahl, die mindestens 1 beträgt, wobei Güte 1 optimale Lösungen charakterisiert. Man muss sich daran gewöhnen, dass schlechtere Lösungen größere Gütewerte haben. Die Berechnung der Güte einer Lösung ist problematisch, da wir nicht optimale Lösungen ja nur betrachten, weil wir keine optimale Lösung kennen. Dennoch geht der Wert einer optimalen Lösung in die Güte der berechneten Lösung ein. Daher können wir die Güte oft nur abschätzen (wie übrigens auch die Rechenzeit).

4.2 Definition. *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.*

Approximationsalgorithmen werden um so besser, je näher die Güte an 1 herankommt. Es gibt NP-harte Optimierungsprobleme, bei denen es für jedes $\varepsilon > 0$ einen polynomiellen Approximationsalgorithmus mit Güte $1 + \varepsilon$ gibt. Wir sprechen von einem Approximationsschema, wenn es einen Algorithmus gibt, der neben der üblichen Eingabe eine Zahl $\varepsilon > 0$ als Eingabe erhält und dann in polynomieller Zeit eine Lösung mit Güte $1 + \varepsilon$ berechnet.

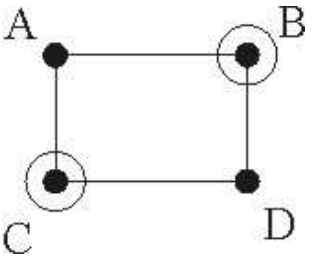
4.3 Definition. *Ein polynomielles Approximationsschema (PTAS, polynomial time approximation scheme) ist für konstantes $\varepsilon > 0$ ein polynomieller Approximationsalgorithmus mit Güte $1 + \varepsilon$. Es heißt echt polynomiell (FPTAS, fully PTAS), wenn die Rechenzeit bei variablem $\varepsilon > 0$ polynomiell in der Eingabelänge n und in ε^{-1} beschränkt ist.*

Polynomielle Approximationsschemata erlauben die Berechnung von Lösungen mit guter Güte, aber die Laufzeit kann von der Größenordnung $n^{\lceil 1/\varepsilon \rceil}$ sein. So ein Algorithmus ist

für jedes $\varepsilon > 0$ polynomiell, aber praktisch nicht für sehr kleine ε einsetzbar. Dies gilt für ein polynomielles Approximationsschema für das euklidische TSP, bei dem die Orte Punkte in $\mathbb{Z}^n$ sind und $c(i, j)$ der euklidische Abstand der Punkte i und j ist. Bei echt polynomiellen Approximationsschemata sind derartige Rechenzeiten ausgeschlossen.

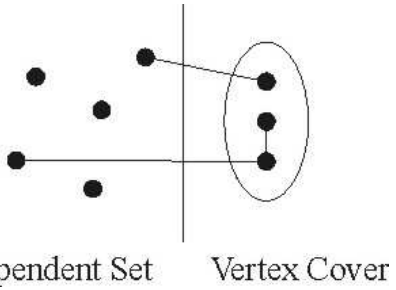
4.2 Ein einfacher Approximationsalgorithmus für Vertex Cover

Das Vertex-Cover-Problem:
Ein Vertex Cover ist eine Menge $V' \subseteq V$ mit der Eigenschaft, dass $\forall e \in E : e \cap V' \neq \emptyset$.



Ein Vertex Cover in dem im Bild angegebenen Graph ist z.B. $\{C, B\}$.
Eingabe: Ungerichteter Graph $G = (V, E)$.
Aufgabe: Gesucht ist ein Vertex Cover mit möglichst wenigen Knoten.

Dieses Problem ist NP-hart, da Independent Set bekanntermaßen NP-hart ist und auf Vertex Cover polynomiell reduziert werden kann: V' ist ein Vertex Cover genau dann, wenn $V \setminus V'$ eine unabhängige Menge ist.



4.4 Behauptung. *Für das Vertex-Cover-Problem gibt es einen Approximationsalgorithmus, der polynomielle Laufzeit und Güte 2 hat.*

Beweis: Durch Angabe eines Greedy-Algorithmus:

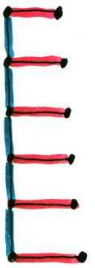
$M := \emptyset$
while es gibt eine Kante, die mit keiner Kante aus M einen Knoten gemeinsam hat.
    Wähle eine solche Kante e.
    $M := M \cup \{e\}$.
end;
Wähle als V' alle Knoten, die auf den Kanten aus M liegen.

Wir zeigen zunächst, dass die berechnete Menge ein Vertex Cover ist: Da die while–Schleife terminierte, ist jede Kante zu einer Kante aus M adjazent, also ist jede Kante zu einem Knoten aus V' inzident. Also ist V' ein Vertex Cover.
Jedes Vertex Cover, insbesondere ein optimales, muss für jede Kante aus M mindestens einen Endpunkt der Kante enthalten. Also gilt für das von A berechnete Vertex Cover V':

$$|M| \leq |\mathrm{VC}_{\mathrm{opt}}| \leq |V'| = 2|M|, \text{ somit } \frac{|V'|}{|\mathrm{VC}_{\mathrm{opt}}|} \leq \frac{2|M|}{|M|} = 2.$$

Daher berechnet der Greedy-Algorithmus nur Vertex Cover, deren Güte höchstens 2 ist. □

Es gibt Beispiele, bei denen der Algorithmus auch tatsächlich um den Faktor 2 daneben liegt, zum Beispiel bei folgendem „Kammgraph“:



Wenn der Greedy-Algorithmus bei dem gegebenen Graphen alle roten (horizontalen) Kanten wählt, dann berechnet er ein Vertex Cover aus 12 Knoten.
Hätte er hingegen die erste, dritte und fünfte grüne (vertikale) Kante von oben gewählt, dann hätte er ein Vertex Cover aus 6 Knoten berechnet. 6 ist das Optimum, da jede der 6 roten Kanten mindestens einen ihrer Knoten in einem Vertex Cover liegen haben muss.

Wir haben nun einen Approximationsalgorithmus mit Güte 2 für das Vertex-Cover-Problem. Haben wir damit etwa (siehe die Polynomialzeitreduktion oben) auch einen Approximationsalgorithmus mit Güte 2 für das Independent-Set-Problem?
Man könnte ja zum Beispiel eine unabhängige Menge dadurch berechnen, dass man ein Vertex Cover V' berechnet und das Komplement $V \setminus V'$ als unabhängige Menge ausgibt. An dem Beispiel sieht man aber, dass dieses Vorgehen schiefgeht: Obwohl das Vertex Cover, das alle 12 Knoten enthält, nur um den Faktor 2 neben dem Optimum 6 liegt, so ist die zugehörige unabhängige Menge die leere Menge. Eine maximale unabhängige Menge hätte aber Kardinalität 6 (wenn man z.B. alle Knoten an der „Kammspitze“ wählt.) Damit wäre die Güte der für Independent Set berechneten Lösung „$6/0 = \infty$“.

4.3 Das Set-Cover-Problem

Gegeben: Eine Grundmenge $X = \{1, \ldots, n\}$ sowie eine Kollektion von m Mengen $S_1, \ldots, S_m$, wobei $S_i \subseteq \{1, \ldots, n\}$. Für jede Menge S_i ist eine natürliche Zahl $cost(S_i)$ als Kosten vorgegeben.

Gesucht: Eine möglichst billige Auswahl der S_i, deren Vereinigung die Grundmenge $\{1, \ldots, n\}$ ergibt.
Motivation: Die Zahlen 1 bis n könnten stellvertretend für „Fähigkeiten“ stehen, die Mitarbeiter in einer Firma besitzen. Für Mitarbeiterin i wäre dann die Menge S_i die Menge ihrer Fähigkeiten.
Gesucht ist ein Team von Mitarbeitern, in dem jede benötigte Fähigkeit von mindestens einer Person abgedeckt ist und die Kosten des Teams sollen minimal sein.

Implizite Voraussetzung $\bigcup_{i=1}^{m} S_i = \{1, \ldots, n\}$, die gegebenen Mengen S_i müssen die gesamte Menge ergeben, da ansonsten kein Set Cover existieren würde.
Bsp.:
$\{1, 3, 5\}$, Kosten: 22
$\{1, 2, 5\}$, Kosten: 16
$\{2, 4, 6\}$, Kosten: 5
$\{5\}$, Kosten: 1

Wir wollen die Menge $M = \{1, 2, 3, 4, 5, 6\}$ darstellen. Dies wird erreicht zum Beispiel durch Auswahl der Mengen $\{1, 3, 5\}$ und $\{2, 4, 6\}$. Diese Auswahl hat Kosten 27, was gleichzeitig eine billigste Überdeckung darstellt, wie man sich leicht überlegt.

Behauptung: SET-COVER ist eine Verallgemeinerung von VERTEX COVER.
Die Eingabe für Vertex Cover kann man auch als Inzidenzmatrix darstellen:

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 \\ \vdots & \vdots & 0 & \vdots & 0 \\ & & 1 & & 0 \\ & & 0 & & 1 \\ & & 1 & & 0 \end{pmatrix}$$

In der Inzidenzmatrix stehen die Zeilen für Kanten, die Spalten für Knoten. Pro Zeile gibt es genau zwei Einsen, und zwar an den beiden zur Kante inzidenten Knoten/Spalten. Ein Vertex Cover entspricht nun einer Auswahl der Spalten, so dass deren „ODER“ die 1-Spalte ergibt, denn dann sind alle Kanten abgedeckt.
Zum Lösen des Problems Vertex Cover ist eine kleinste Teilmenge der Spalten gesucht, deren ODER die 1-Spalte ergibt.
Auch die Eingabe für Set Cover kann man als Matrix darstellen.

Die Zeilen stehen für die Elemente $\{1, \ldots, n\}$, die Spalten für die Mengen $S_1, \ldots, S_m$. Gesucht ist eine Auswahl von Spalten, deren ODER den 1-Vektor ergibt.
$\rightarrow$ Set Cover ist eine Verallgemeinerung von Vertex Cover.
Für Set Cover kann man eine Approximationsgüte von $1+\ln n$ erreichen, wie wir gleich beweisen werden. Eine wesentlich bessere Güte kann man vermutlich nicht erreichen, denn es existiert ein Resultat, das wir hier nicht beweisen, aber wenigstens zitieren wollen.
Sei DTIME($t(n)$) die Menge der Probleme, die man auf einer deterministischen Turingmaschine in Zeit $O(t(n))$ lösen kann. In dieser Notation könnte man etwa die Menge P der in Polynomialzeit lösbaren Probleme als DTIME($n^{O(1)}$) schreiben.
Die meisten Wissenschaftler/innen glauben nicht nur, dass NP $\neq P$, also NP $\not\subseteq$ P ist, sondern auch, dass NP $\not\subseteq$ DTIME($n^{O(\log\log n)}$) ist. Bekannt ist nun folgendes Resultat: Wenn NP $\not\subseteq$ DTIME($n^{O(\log\log n)}$) ist, dann gibt es kein $c < 1$ mit der Eigenschaft, dass Set Cover in Polynomialzeit mit Güte $c \cdot \ln n$ approximiert werden kann.
(Siehe den Artikel: U. Feige: *A Threshold of* $\ln n$ *for Approximating Set Cover.* Journal of the ACM 45(4): 634-652 (1998).)

Greedy-Algorithmus für das Set-Cover-Problem:

C := $\emptyset$;
while $C \neq \{1, \ldots, n\}$ **do**
begin
  Finde Menge $S \in \{S_1, \ldots, S_m\}$ mit „relativen Kosten“ $\alpha(S) = \frac{\text{cost}(S)}{|S \setminus C|}$ minimal.
  Wähle S.
  $C := C \cup S$.
end;

Definition: Die relativen Kosten einer Menge S sind bestimmt durch:

$$\alpha(S) := \frac{cost(S)}{|S \setminus C|}.$$

Dabei ist $|S \setminus C|$ die Anzahl an neuen, durch S überdeckten Elementen.
Für das Beispiel von oben ergibt sich dann folgendes Vorgehen:

Menge	Kosten	relative Kosten in Durchgang 1	relative Kosten in Durchgang 2	relative Kosten in Durchgang 3
$\{1,3,5\}$	22	$\frac{22}{3}$	$\frac{22}{2}$	$\frac{22}{2}$
$\{1,2,5\}$	16	$\frac{16}{3}$	$\frac{16}{2}$	$\frac{16}{1}$
$\{2,4,6\}$	5	$\frac{5}{3}$	$\frac{5}{3}$	∞
$\{5\}$	1	$\frac{1}{1}$	∞	∞

Der Algorithmus operiert auf der Beispielmenge wie folgt:

1. wähle $\{5\}$; relative Kosten aktualisieren.

2. wähle $\{2, 4, 6\}$; relative Kosten aktualisieren.

3. wähle $\{1, 3, 5\}$; FERTIG

Die Kosten der vom Greedy-Algorithmus getroffenen Auswahl sind 28.

Beobachtung Da nach Voraussetzung $\bigcup_{i=1}^{m} S_i = \{1, \ldots, n\}$ ist, berechnet der Greedy-Algorithmus auf jeden Fall ein Set Cover.

4.5 Satz. *Der Greedy-Algorithmus berechnet ein Set Cover, dessen Kosten höchstens* $(1 + \ln n) \cdot$ *OPT sind, wenn OPT die Kosten des optimalen Set Covers sind.*

Beweis: Sei rest_i die Anzahl Elemente, die nach i Runden des Algorithmus noch zu überdecken sind. Nach t Runden sei Schluss. Wir haben also $\text{rest}_0 = n, \ldots, \text{rest}_t = 0$.
In jeder Runde kann die Restmenge $\{1, \ldots, n\} \setminus C$ mit Kosten höchstens OPT überdeckt werden. (Dies ist klar, da wir ja sogar die ganze Menge mit Kosten OPT abdecken können.)
Angenommen, in Runde $i + 1$ kann die Überdeckung durch r Mengen mit Kosten $c_1, \ldots, c_r$ stattfinden, die jeweils $t_1, \ldots, t_r$ noch nicht abgedeckte Elemente überdecken. Dann ist $c_1 + \cdots + c_r \leq \text{OPT}$ und $t_1 + \cdots + t_r \geq \text{rest}_i$.
Aus dem Schubfachprinzip (Satz 8.2 auf Seite 137 im Anhang) folgt also, dass es in Runde $i + 1$ eine Menge geben muss, deren relative Kosten c_j/t_j durch OPT/rest_i beschränkt sind. Da der Greedy-Algorithmus eine Menge S mit minimalen relativen Kosten $\alpha(S)$ auswählt, ist $\alpha(S)$ durch OPT/rest_i beschränkt. Die Kosten der Menge S sind gleich

$$\begin{aligned} |S \setminus C| \cdot \alpha(S) &\leq |S \setminus C| \cdot \frac{\text{OPT}}{\text{rest}_i} \\ &\leq (\text{rest}_i - \text{rest}_{i+1}) \cdot \frac{\text{OPT}}{\text{rest}_i}. \end{aligned}$$

Der Grund für die letzte Abschätzung ist, dass $(\text{rest}_i - \text{rest}_{i+1})$ gerade die Anzahl Elemente zählt, die von der gewählten Menge S neu abgedeckt worden sind.
Nun folgt ein kleiner Rechentrick: Wir können $(\text{rest}_i - \text{rest}_{i+1}) \cdot \frac{1}{\text{rest}_i}$ lesen als $(\text{rest}_i - \text{rest}_{i+1})$-fache Summe des Terms $\frac{1}{\text{rest}_i}$:

$$\begin{aligned} (\text{rest}_i - \text{rest}_{i+1}) \cdot \frac{1}{\text{rest}_i} &= \frac{1}{\text{rest}_i} + \frac{1}{\text{rest}_i} + \cdots + \frac{1}{\text{rest}_i} \\ &\leq \frac{1}{\text{rest}_{i+1} + 1} + \frac{1}{\text{rest}_{i+1} + 2} + \cdots + \frac{1}{\text{rest}_i} \end{aligned}$$

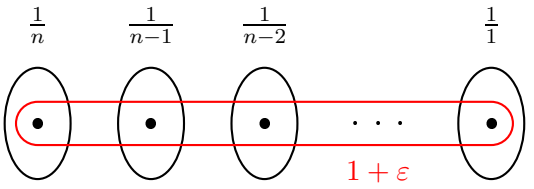
Die letzte Abschätzung gilt, weil wir in der letzten Summe auch eine Summe aus $(\text{rest}_i - \text{rest}_{i+1})$ Termen haben und jeder davon größer oder gleich $\frac{1}{\text{rest}_i}$ ist.
Wenn wir nun die Kosten aller Mengen für alle Runden aufsummieren, so sehen wir, dass dies gleich

$$\begin{aligned}\text{OPT} \quad \cdot \quad & \Big(\frac{1}{\text{rest}_1+1}+\cdots+\frac{1}{\text{rest}_0}+\\ & +\frac{1}{\text{rest}_2+1}+\cdots+\frac{1}{\text{rest}_1}+\\ & +\ldots+\\ & +1+\ldots+\frac{1}{\text{rest}_{t-1}}\Big)\end{aligned}$$

ist. Wegen $\text{rest}_0 = n$ ist dies gleich $\text{OPT} \cdot (1+(1/2)+(1/3)+\cdots+(1/n)) = \text{OPT} \cdot H(n)$, wenn wir mit $H(n)$ die n-te Harmonische Zahl bezeichnen. Bekannt ist: $H(n) \leq \ln n + 1$. □

Um zu zeigen, dass die Analyse scharf ist, benötigen wir ein Beispiel, bei dem der Algorithmus tatsächlich so schlecht rechnet. Sei $n \geq 2$ und $\varepsilon > 0$ eine kleine Zahl. (Zum Beispiel $\varepsilon < 0.5$).
Wir legen als Eingabe für das Set-Cover-Problem $n+1$ Mengen $S_1, \ldots, S_{n+1}$ fest, und zwar wie folgt:
Für $1 \leq i \leq n$ sei $S_i := \{i\}$ mit Kosten $1/(n-i+1)$.
S_1 hat also Kosten $1/n$, S_2 hat Kosten $1/(n-1)$ etc.
Schließlich sei $S_{n+1} = \{1, \ldots, n\}$ mit Kosten $1+\varepsilon$.
Aufgrund der relativen Kosten wählt der Greedy-Algorithmus der Reihe nach $S_1, \ldots, S_n$. Dabei entstehen Gesamtkosten von $\sum_{i=1}^{n} \frac{1}{i} = H(n) \geq 1.5$. Die optimale Lösung hingegen wäre die Wahl der Menge S_{n+1}, die Kosten $1+\varepsilon < 1.5$ hat.

Wenn man ε sehr klein wählt, so sieht man, dass also die Kosten $H(n)$ des berechneten Set Covers fast um den Faktor $H(n)$ schlechter sein können als die Kosten $1+\varepsilon$ des Optimums.



4.4 Approximationsalgorithmen für das metrische TSP

Wir betrachten hier zwei polynomielle Approximationsalgorithmen für das metrische TSP, dem Spezialfall des TSP, in dem die Kostenrelation symmetrisch ($c(i,j) = c(j,i)$) ist und die Dreiecksungleichung ($c(i,j) \leq c(i,k) + c(k,j)$) erfüllt. Hierbei setzen wir folgende Ergebnisse über Eulerkreise in Multigraphen (Kanten dürfen mehrfach vorhanden sein) voraus. Ein Eulerkreis ist ein Kreis, auf dem Knoten wiederholt vorkommen dürfen, der aber jede Kante genau einmal enthalten muss. Ein Multigraph enthält genau dann

einen Eulerkreis, wenn er bis auf isolierte Knoten zusammenhängend ist und jeder Knoten geraden Grad hat. In diesem Fall lässt sich ein Eulerkreis in linearer Zeit $O(n+m)$ berechnen. Wenn wir einen Eulerkreis EK auf $G=(V,E)$ haben, wobei Knoten doppelt benutzt werden dürfen, erhalten wir daraus in linearer Zeit eine Tour π, deren Kosten nicht größer als die Kosten von EK sind. Dazu durchlaufen wir EK. Jeden Knoten, den wir nicht zum ersten Mal erreichen (außer am Ende), lassen wir aus. Wegen der Dreiecksungleichung sind die direkten Wege, die wir nun gehen, nicht teurer als die Umwege auf EK. Der erste Approximationsalgorithmus benutzt die bekannte Tatsache, dass ein optimaler Spannbaum nicht teurer als eine optimale Tour ist. Woran liegt das? Wenn man eine Tour nimmt und daraus eine Kante entfernt, so erhält man einen Spannbaum, der billiger ist als die Tour. Der billigste Spannbaum ist höchstens noch billiger.
Zudem sind Spannbäume zusammenhängend und verbinden die gesamte Knotenmenge. Eine Verdoppelung aller Kanten ergibt für jeden Knoten einen geraden Grad. Der auf dieser Kantenmenge existierende Eulerkreis enthält jede Kante des Spannbaums zweimal. Die Kosten sind also durch die doppelten Kosten einer optimalen Tour nach oben beschränkt. Aus dem Eulerkreis erhalten wir eine Tour mit nicht größeren Kosten, also mit einer Güte, die nicht größer als 2 ist.

4.6 Algorithmus.

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 $C(\pi) \leq C(EK)$.

4.7 Satz. *Algorithmus 4.6 hat Laufzeit $O(n^2)$ und ist ein Approximationsalgorithmus für das metrische TSP mit Güte 2.*

Beweis: Die Aussage über die Güte haben wir bereits oben gezeigt. Optimale Spannbäume lassen sich in Zeit $O(n^2)$ berechnen (Algorithmus von Prim). Da Spannbäume genau $n-1$ Kanten haben, lassen sich die anderen Schritte in Zeit $O(n)$ realisieren. □

Der Nachteil dieses Verfahrens ist, dass wir den Grad von allen Knoten verdoppeln, um zu garantieren, dass er gerade wird. Der Algorithmus von Christofides vermeidet dies.

4.8 Algorithmus. *(von Christofides)*

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 $|M|/2$ 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 $C(\pi) \leq C(EK)$.

4.9 Satz. *Der Algorithmus von Christofides hat Laufzeit $O(n^3)$ und ist ein Approximationsalgorithmus für das metrische TSP mit Güte $3/2$.*

Beweis: Schritt 1 kostet $O(n^2)$ Rechenschritte. Wenn wir alle Grade in einem Graphen aufsummieren, ist das Ergebnis immer gerade, da jede Kante zweimal gezählt wird. Daher ist $|M|$ gerade und $|M|/2$ eine ganze Zahl. Die Berechnung von M ist in linearer Zeit möglich. Kostenminimale Matchings können in Zeit $O(n^3)$ berechnet werden, was wir hier nicht beweisen wollen. Da G höchstens $\frac{3}{2}n$ Kanten enthält, lassen sich die Schritte 4, 5 und 6 in linearer Zeit durchführen. Die Aussage über die Güte wird folgendermaßen bewiesen. Es ist

$$C(\pi) \leq C(EK) = C(S_{\text{opt}}) + C(M_{\text{opt}}) \leq C_{\text{opt}} + C(M_{\text{opt}}).$$

Es genügt also zu zeigen, dass $C(M_{\text{opt}}) \leq C_{\text{opt}}/2$ ist. Es sei $i(1), \ldots, i(2k)$ die Reihenfolge, in der die Knoten aus M auf einer optimalen Tour π_{opt} vorkommen. Da wir ein metrisches TSP betrachten, sind die Kosten des Kreises $(i(1), \ldots, i(2k), i(1))$ nicht größer als C_{opt}, die Kosten von π_{opt}. Der betrachtete Kreis enthält aber zwei disjunkte Matchings auf den Knoten in M, einerseits $(i(1), i(2)), \ldots, (i(2k-1), i(2k))$ und andererseits $(i(2), i(3)), \ldots, (i(2k-2), i(2k-1)), (i(2k), i(1))$. Die Kosten des billigeren dieser beiden Matchings betragen also höchstens $C_{\text{opt}}/2$. Dies gilt dann natürlich erst recht für M_{opt}. □

Beispiele zeigen, dass wir weder in Satz 4.7 noch in Satz 4.9 die Güte durch eine kleinere Konstante abschätzen können. Wenn NP $\neq$ P ist, gibt es für das allgemeine TSP keinen polynomiellen Approximationsalgorithmus, dessen Güte auch nur durch ein Polynom in n beschränkt ist.

4.5 Ein echt polynomielles Approximationsschema für das Rucksackproblem

Das Rucksackproblem ist sicherlich schon aus einer früheren Vorlesung wie zum Beispiel DAP2 bekannt. Eine Instanz des Rucksackproblems ist dabei gegeben durch Gewichtslimit G, Objektgewichte $g_1, \ldots, g_n$ und Nutzenwerte $v_1, \ldots, v_n$.
Wir stellen ein echt polynomielles Approximationsschema für das Rucksackproblem vor. Dazu erinnern wir uns zunächst an einen pseudopolynomiellen Algorithmus für das Rucksackproblem, den wir in der Vorlesung DAP2 kennengelernt haben. Dieser hat Laufzeit $O(n \cdot G)$ und ist somit effizient, wenn das Gewichtslimit des Rucksacks klein ist. Wir skizzieren nun, wie man auch einen effizienten Algorithmus bekommen kann, wenn alle Nutzenwerte klein sind.

4.10 Lemma. *Das Rucksackproblem hat einen pseudopolynomiellen Algorithmus mit Laufzeit $O(n^2 \cdot v_{\max})$, wobei $v_{\max} = \max_i v_i$ der größte auftretende Nutzenwert eines Objekts ist.*

Beweis: Für $i \in \{0, \ldots, n\}$ und $V \in \{0, \ldots, n \cdot v_{\max}\}$ sei $A_{i,V}$ das kleinstmögliche Gewicht, mit dem man den Nutzen V exakt erreichen kann, wenn man nur Objekte aus der Menge $\{1, \ldots, i\}$ verwenden darf. Wir setzen $A_{i,V} = \infty$, falls der Nutzen V nicht durch eine Teilmenge von $\{1, \ldots, i\}$ erreicht werden kann. Korrekte Initialwerte sind sicherlich $A_{0,V} = 0$, wenn $V = 0$ ist und $A_{0,V} = \infty$ sonst. Ansonsten gilt

$$A_{i+1,V} \;=\; \min(A_{i,V},\, A_{i,V-v_{i+1}} + g_{i+1}).$$

(Die erste Zahl nach der Minimumsbildung steht für den Fall, dass Objekt $i+1$ nicht eingepackt wird, die zweite Zahl für den Fall, dass das Objekt eingepackt wird.)
Wenn dabei $V - v_{i+1}$ negativ ist, so ist $A_{i,V-v_{i+1}} + g_{i+1} = \infty$, da man negative Nutzen nicht erreichen kann.
Wir verwenden den Ansatz der dynamischen Programmierung und berechnen Zeile für Zeile alle Einträge in der Tabelle bzw. Matrix $(A_{i,V})_{i \in \{1,\ldots,n\}, V \in \{0,\ldots,n \cdot v_{\max}\}}$. Die Laufzeit entspricht der Größe der Tabelle und ist somit $O(n^2 \cdot v_{\max})$.
Der gesuchte, maximal erreichbare Nutzenwert ist

$$\max\{V \mid A_{n,V} \leq G\},$$

wenn G das zulässige Gewicht des Rucksacks ist. Die zugehörige Rucksackbepackung lässt sich durch Abspeichern geeigneter Zusatzinformationen zu den einzelnen Tabelleneinträgen rekonstruieren, so dass die optimale Rucksackbepackung in Zeit $O(n^2 \cdot v_{\max})$ berechnet werden kann. □

Wenn die Nutzenwerte nicht klein sind, dann ist die Grundidee, dass wir sie alle durch eine feste Zahl k dividieren und auf die nächstkleinere ganze Zahl abrunden:

> Wähle $k := \frac{\varepsilon \cdot v_{\max}}{(1+\varepsilon) \cdot n}$.
>
> Setze $v'_i := \lfloor \frac{v_i}{k} \rfloor$ als neue Nutzenwerte fest.
>
> Berechne für die neuen Nutzenwerte eine optimale Lösung mit dem dynamischen Programmierungsalgorithmus und gib die entsprechende Menge aus.

Bemerkung: Da $v_{\max} \geq 1$ und $\varepsilon > 0$ ist, ist $k > 0$ und wir teilen nie durch Null.
Im folgenden bezeichnen wir für eine Teilmenge $A \subseteq \{1, \ldots, n\}$ mit

$$\begin{aligned} \text{Nutzen}\ (A) &:= \sum_{i \in A} v_i \\ \text{Nutzen}'(A) &:= \sum_{i \in A} v'_i. \end{aligned}$$

4.11 Satz. *Der gerade beschriebene Algorithmus hat eine Güte von* $(1+\varepsilon)$ *und eine Laufzeit von* $O(n^3\varepsilon^{-1})$*, ist also ein FPTAS.*

Beweis: Die Laufzeit ergibt sich leicht durch einfaches Einsetzen:

$$O(n^2 \cdot \lfloor \frac{v_{\max}}{k} \rfloor) = O(n^3\varepsilon^{-1}).$$

Dass die berechnete Füllung des Rucksacks zulässig ist, ist ebenfalls offensichtlich, da die Gewichte bei der Transformation nicht verändert werden. Wir müssen also nur noch die Einhaltung der Güte beweisen. Da

$$\frac{v_i}{k} - 1 \leq v_i' \leq \frac{v_i}{k}$$

gilt, folgt daraus auch für alle $A \subseteq \{1, \ldots, n\}$, dass

$$\frac{\text{Nutzen}(A)}{k} - |A| \leq \text{Nutzen}'(A) \leq \frac{\text{Nutzen}(A)}{k} \qquad (*)$$

ist. In der ersten Abschätzung kann man wegen $|A| \leq n$ auch noch $|A|$ durch n ersetzen. Sei nun M_{opt} eine optimale Menge für das ursprüngliche Problem. Bei der dynamischen Programmierung berechnet der Algorithmus eine Menge S mit optimalem Nutzen$'$-Wert, also ist

$$\text{Nutzen}'(S) \geq \text{Nutzen}'(M_{\text{opt}}). \quad (**)$$

Also können wir den Nutzen der berechneten Lösung in Bezug auf unsere ursprünglichen Nutzenwerte folgendermaßen abschätzen:

$$\begin{array}{rcll}
\text{Nutzen}(S) & \geq & k \cdot \text{Nutzen}'(S) & \text{(wg. (*))} \\
& \geq & k \cdot \text{Nutzen}'(M_{\text{opt}}) & \text{(wg. (**))} \\
& \geq & \text{Nutzen}(M_{\text{opt}}) - k \cdot n & \text{(wg. (*))} \\
& = & \text{OPT} - \frac{\varepsilon}{1+\varepsilon} \cdot v_{\max} & \text{(Definition von } k\text{)} \\
& \geq & \text{OPT} \cdot (1 - \frac{\varepsilon}{1+\varepsilon}) & \text{(da } OPT \geq v_{\max}\text{)} \\
& = & \text{OPT} \cdot \frac{1}{1+\varepsilon} &
\end{array}$$

Dabei ist $\text{OPT} \geq v_{\max}$, weil alle $g_i \leq G$ sind. □

4.6 Makespan-Scheduling auf identischen Maschinen

Wir untersuchen ein fundamentales Problem aus der Schedulingtheorie.

MAKESPAN SCHEDULING:

Gegeben sei eine Menge von Jobs $\{1, \ldots, n\}$ mit Größen $p_1, \ldots, p_n \in \mathbb{N}$ und eine Zahl $m \in \mathbb{N}$, die die Anzahl der zur Verfügung stehenden Maschinen angibt.

Ein Schedule ist eine Abbildung $f : \{1, \ldots, n\} \to \{1, \ldots, m\}$, die die n Jobs auf die m Maschinen verteilt.
Wenn wir einen Schedule f haben, dann ist die Füllhöhe t_i der Maschine i gegeben durch

$$t_i := \sum_{j \in \{1,\ldots,n\}: f(j)=i} p_i.$$

Also ist t_i der Zeitpunkt, zu dem Maschine i mit ihrer Arbeit fertig wird. Das Problem Minimum Makespan Scheduling fragt nach einem Schedule, der den so genannten „Makespan“ $\max_{i=1,\ldots,m} t_i$ minimiert.

Zu einem Schedule gehört normalerweise auch eine Beschreibung, in welcher Reihenfolge die Jobs auf den einzelnen Maschinen abgearbeitet werden. Diese Reihenfolge spielt jedoch bei der Makespan-Minimierung offensichtlich keine Rolle.

Algorithmus Least-Loaded :

Für $i = 1$ bis n: Weise Job i derjenigen Maschine zu, die bisher die geringste Last hat.

Wie gut ist diese Heuristik?

Ein Beispiel:

- Sei $n = m^2 + 1$.
- Jobs 1 bis m^2 haben Größe 1.
- Job $m^2 + 1$ hat Größe m.
- Least-Loaded berechnet einen Schedule mit Makespan $2m$.
- Der optimale Makespan ist $m + 1$.

Damit ist die Approximationsgüte bestenfalls 2. Der folgende Satz zeigt, dass dieses Beispiel tatsächlich den schlimmsten Fall beschreibt.

4.12 Satz. *Der Algorithmus Least-Loaded garantiert eine 2-Approximation.*

Beweis: Es gelten die folgenden zwei trivialen unteren Schranken für ein optimales Schedule:

$$opt \quad \geq \quad \max \left\{ \max_{i \in \{1,\ldots,n\}} (p_i) \ , \ \frac{1}{m} \sum_{i \in \{1,\ldots,n\}} p_i \right\} \ .$$

Wir gehen davon aus, dass jede Maschine ihre Jobs nacheinander in der Reihenfolge ihrer Zuweisung abarbeitet. Sei i' der Index desjenigen Jobs, der als letztes fertig wird. Sei $j' = f(i')$, d.h. Maschine j' wird als letztes fertig und bestimmt damit den Makespan. Zu dem Zeitpunkt, als Job i' Maschine j' zugewiesen wurde, hatte diese Maschine die geringste Last. Die Last von Maschine j' zu diesem Zeitpunkt war also höchstens $\frac{1}{m}\sum_{i<i'} p_i$. Damit ist die Last von Maschine j' höchstens

$$\left(\frac{1}{m}\sum_{i<i'} p_i\right) + p_{i'} \quad \leq \quad 2opt \ .$$

□

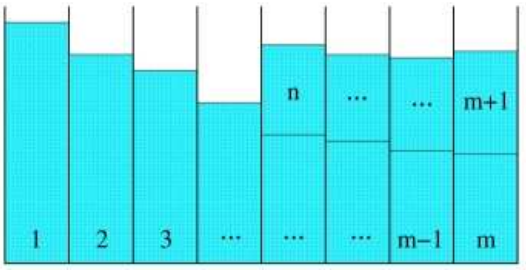
Algorithmus Longest-Processing-Time (LPT):

1. Sortiere die Jobs so, dass $p_1 \geq p_2 \geq \cdots \geq p_n$ gilt.
2. Für $i = 1$ bis n: Weise Job i derjenigen Maschine zu, die bisher die geringste Last hat.

Graham hat 1969 gezeigt, dass LPT eine Güte von höchstens 4/3 hat. Auch diese Schranke ist scharf.

4.13 Satz. *LPT garantiert eine* 4/3*-Approximation.*

Beweis: Eine Eingabe $p_1 \geq p_2 \geq \cdots \geq p_n$ heiße Gegenbeispiel, wenn der von LPT auf dieser Eingabe berechnete Makespan $\tau > \frac{4}{3}opt$ ist.
Angenommen, es gäbe mindestens ein Gegenbeispiel. Dann wählen wir unter allen solchen eines mit minimalem n und analysieren es nun genauer.
Sei i' die Nummer desjenigen Jobs, der als letztes fertig wird. Es gilt $i' = n$, sonst wäre ja $p_1, \ldots, p_{i'}$, $i' < n$, ein kleineres Gegenbeispiel.
In dem Moment, wo Job n platziert wird, befindet er sich auf der aktuell am wenigsten belasteten Maschine. Diese Maschine hat in diesem Augenblick höchstens Last $\frac{1}{m}\sum_{i=1}^{n-1} p_i \leq opt$. Damit $\tau > \frac{4}{3}opt$ gilt, muss also $p_n > \frac{1}{3}opt$ gelten.
Aus $p_n > \frac{1}{3}opt$ folgt, dass jeder Job größer als $\frac{1}{3}opt$ ist, weil $p_1 \geq p_2 \geq \cdots \geq p_n$.
Falls jeder Job größer als $\frac{1}{3}opt$ ist, so kann ein optimaler Schedule nicht mehr als zwei Jobs an eine Maschine zuweisen. Wir zeigen nun, dass jeder Schedule mit höchstens zwei Jobs pro Maschine in den folgenden schematisch dargestellten Schedule überführt werden kann, ohne den Makespan zu erhöhen.



In Worten: Die Jobs 1 bis $2m-n$ liegen alleine auf den Maschinen 1 bis $2m-n$ und die anderen Jobs befinden sich paarweise auf den restlichen Maschinen, und zwar so, dass:

Job $2m-n+1$ mit Job n gepaart ist	(Maschine $2m-n+1$)
Job $2m-n+2$ mit Job $n-1$ gepaart ist	(Maschine $2m-n+2$)
...	...
Job m mit Job $m+1$ gepaart ist	(Maschine m)

Dass ein solcher Umbau möglich ist, zeigen wir nun. Zur Vereinfachung nehmen wir zunächst an, dass wir es mit genau $2m$ Jobs zu tun haben. Dies erreichen wir dadurch, dass wir (eventuell) Dummyjobs $n+1$ bis $2m$ hinzufügen, die die Größe 0 haben und nun untersuchen, wie man einen Schedule umbauen kann, der jeweils exakt zwei Jobs pro Maschine enthält.
Man kann sich einen solchen Schedule als Menge von Kreuzen im Zweidimensionalen vorstellen. Wir setzen für jede Maschine ein Kreuz, und zwar an die Stelle (i,j), wenn die beiden Jobs $i<j$ die beiden der Maschine zugewiesenen sind.
Wenn es zwei Kreuze gibt, bei denen das eine Kreuz „rechts oben“ vom anderen Kreuz liegt (in Formeln $i<i'$ und $j<j'$), dann kann man den Schedule umbauen: Man legt die Jobs i und j' zusammen sowie die Jobs i' und j.
Die maximale Füllhöhe der beiden Maschinen vor dem Umbau ist $\max\{p_i+p_j, p_{i'}+p_{j'}\} = p_i+p_j$. (Größere Jobnummer heißt kleinere Jobgröße!)
Nach dem Umbau ist die maximale Füllhöhe der beiden Maschinen $\max\{p_{i'}+p_j, p_i+p_{j'}\}$. Da $p_{i'}+p_j \le p_i+p_j$ ist (größere Jobnummer heißt kleinere Jobgröße!) und auch $p_i+p_{j'} \le p_i+p_j$ ist, ist die maximale Füllhöhe der beiden Maschinen nach dem Umbau höchstens kleiner geworden.
In Kreuzen ausgedrückt bedeutet das nun, dass wir Kreuze (i,j') und (i',j) vorliegen haben, das Kreuz (i',j) liegt also rechts unterhalb des Kreuzes (i,j').
Man überlegt sich nun leicht, dass man den Schedule insgesamt so umbauen kann, dass die Kreuze an den Stellen $(1,2m)$, $(2,2m-1)$ bis $(m,m+1)$ sind. (Also eine nach rechts unten zeigende Diagonale). Diese Kreuze geben aber genau das im Bild beschriebene Schedule wieder.
Insgesamt haben wir gezeigt: Der im Bild angegebene Schedule ist optimal. Nun müssen wir uns ansehen, was der Algorithmus LPT berechnet.

Behauptung: LPT berechnet einen Schedule, der wie im Bild aussieht.

Wir beweisen diese Behauptung nun: Dass die ersten m Jobs wie im Bild zugewiesen werden, ist offensichtlich. Wir zeigen nun: Bei der Zuweisung von Job $m+j$ ist die Maschine $m-j+1$ eine am geringsten belastete Maschine (und Job $m+j$ wird also auf diese Maschine gelegt). Wir zeigen dies durch Induktion über j. Der Induktionsanfang $j=1$ ist trivial.
Seien also die Jobs $m+1$ bis $m+j$ schon entsprechend zugewiesen. Die Maschinen m bis $m-j+1$ sind dann schon mindestens mit Last echt größer als $(2/3)\cdot opt$ ausgelastet, da sie alle jeweils zwei Jobs zugewiesen bekommen haben (und alle Jobs Größe $> (1/3)\cdot opt$ haben.)
Die anderen Maschinen 1 bis $m-j$ enthalten bislang nur einen Job, davon ist die Maschine $m-j$ die am wenigsten ausgelastete. Um die *insgesamt* am wenigsten ausgelastete Maschine zu finden, müssen wir nun also die Füllhöhe der Maschine $m-j$ abschätzen und mit $(2/3)\cdot opt$ vergleichen.
Der optimale Schedule hat den Job $m+(j+1)$ auf die Maschine $m-j$ gelegt. Da der Makespan des optimalen Schedules gleich *opt* ist, wissen wir, dass $p_{m+(j+1)}+p_{m-j} \leq opt$ ist. Da alle Jobgrößen größer als $(opt/3)$ sind, folgt $p_{m-j} \leq (2/3)\cdot opt$. Die kleinste Füllhöhe unter den Maschinen 1 bis $m-j$ ist also $\leq (2/3)\cdot opt$ und somit kleiner als die kleinste Füllhöhe der Maschinen $m-j+1$ bis m, Job Nummer $j+1$ wird also Maschine $m-j$ zugewiesen, womit der Induktionsschritt vollzogen ist.
Wenn aber auf dem angenommenen Gegenbeispiel der Algorithmus LPT eine optimale Lösung berechnet, so folgt, dass es sich doch nicht um ein Gegenbeispiel handeln kann und es somit kein Gegenbeispiel gibt. □

Eine einfache Reduktion vom Bin-Packing-Problem zeigt, dass das Makespan-Scheduling-Problem stark NP-hart ist. Es gibt also kein FPTAS für dieses Problem. Trotzdem werden wir zeigen, dass das Problem in Polynomialzeit beliebig gut approximiert werden kann, indem wir ein PTAS mit Laufzeit $n^{O(1/\varepsilon^2)}$ angeben. Für ein FPTAS wäre eine derartige Laufzeitschranke nicht zulässig, weil sie nicht polynomiell in $1/\varepsilon$ ist.

Für kleines ε ist die obige Laufzeitschranke offensichtlich nicht praktikabel. Wir wollen uns trotzdem einmal ansehen, wie ein derartiges Approximationsschema aussieht. Dieses PTAS ist aber eher unter komplexitätstheoretischen als unter praktischen Gesichtspunkten interessant. Wir benötigen zunächst die folgende Aussage:

4.14 Lemma. *Wenn die Jobgrößen $p_1,\ldots,p_n$ aus k verschiedenen Größenwerten bestehen, also $k := |\{p_i \mid i=1,\ldots,n\}|$ ist, dann kann man einen optimalen Schedule in Laufzeit $O(k\cdot n^{2k+1})$ berechnen.*

Beweis: Wenn die Maschinenzahl $m \geq n$ ist, dann geben wir den Schedule aus, der jeder Maschine höchstens einen Job zuweist. Dieser ist offensichtlich optimal. Sei also nun $m < n$.
Seien $w_1,\ldots,w_k$ die verschiedenen Größenwerte. Die Last, die man einer Maschine zuweist, kann durch einen Vektor $q=(q_1,\ldots,q_k)$ beschrieben werden, der angibt, wieviele Jobs der jeweiligen Größe einer Maschine zugewiesen werden: jeweils q_i Jobs der Größe w_i.

Sei Q die Menge aller solchen Vektoren bei n Jobs. Da alle $q_i \leq n$ sind, ist $|Q| \leq (n+1)^k = O(n^k)$. Wir definieren mit $h(q) := q_1 \cdot w_1 + \cdots + q_k \cdot w_k$ die zugehörige Füllhöhe.
Wir lösen das Makespan-Problem mit dynamischer Programmierung und definieren:
$make(n_1, \ldots, n_k, m)$ sei der optimale Makespan, den man erreichen kann, wenn für $i = 1, \ldots, k$ genau n_i Jobs der Größe w_i als Eingabe vorliegen und m Maschinen zur Verfügung stehen.
Wie berechnet man $make(n_1, \ldots, n_k, m)$? Es ist $make(n_1, \ldots, n_k, 1) = n_1 w_1 + \cdots + n_k w_k$, da alle Jobs auf (die einzige) Maschine 1 gelegt werden müssen.
Die bellmansche Optimalitätsgleichung für $m \geq 2$ lautet nun

$$make(n_1, \ldots, n_k, m) = \min_{q=(q_1,\ldots,q_k) \in Q} \max\{h(q), make(n_1 - q_1, \ldots, n_k - q_k, m-1)\}.$$

Hierbei muss beachtet werden, dass man das Minimum nur über diejenigen $q \in Q$ berechnet, für die in allen Komponenten $q_i \leq n_i$ gilt.
Die Optimalitätsgleichung erklärt sich wie folgt: Mit den Vektoren $q \in Q$ durchläuft man alle möglichen Lastzuweisungen an die erste Maschine, die dann jeweils bis zur Füllhöhe $h(q)$ gefüllt ist. Die restlichen Jobs werden optimal auf die anderen $m-1$ Maschinen verteilt, so dass dort der Makespan $make(n_1 - q_1, \ldots, n_k - q_k, m-1)$ entsteht. Der Makespan für das gesamte Problem ist dann das Maximum der Füllhöhe der ersten Maschine und der maximalen Füllhöhe der anderen Maschinen. Durch die Minimumsbildung wählen wir die Lösung mit dem kleinsten Makespan aus.
Wieviel Zeit benötigt man zum Berechnen von $make(n_1, \ldots, n_k, m)$?
Wir benutzen, wie bei der dynamischen Programmierung üblich, eine Tabelle, die im vorliegenden Fall $(n+1)^{k+1}$ Einträge hat. (Es ist $m < n$.) Für das Berechnen eines Tabelleneintrags gemäß der Optimalitätsgleichung kommen wir mit Laufzeit $O(k \cdot |Q|) = O(k \cdot n^k)$ aus. Dies liegt daran, dass wir alle möglichen Vektoren aus Q durchlaufen und die Vektoren jeweils die Länge k haben. Insgesamt erhalten wir als Laufzeit $O(k \cdot n^{2k+1})$. □

Nun kommen wir zum PTAS für das Problem Makespan-Scheduling:

Ein PTAS für MAKESPAN-SCHEDULING:
Eingabe: Jobgrößen $p_1, \ldots, p_n$, Maschinenzahl m und $\varepsilon > 0$.

Sortiere die Jobs so, dass $p_1 \geq p_2 \geq \cdots \geq p_n$ gilt.
Im folgenden sei T eine (einfach zu berechnende) untere Schranke für den Makespan des optimalen Schedules. Wir können zum Beispiel $T := p_1$ wählen.
Wir nennen einen Job i „groß“, wenn $p_i \geq \varepsilon \cdot T$ ist. Es sei $\{1, \ldots, g\}$ mit $g \geq 1$ die Menge der großen Jobs.
Phase 1: Die großen Jobs werden verteilt: Skaliere und runde die Größen der großen Jobs, d.h. setze

$$p_i^* := \frac{p_i}{\varepsilon^2 \cdot T} \text{ und } p_i' := \lceil p_i^* \rceil.$$

Berechne einen optimalen Schedule für die Jobgrößen $\{p_1', \ldots, p_g'\}$ mit Hilfe des Algorithmus, der in Lemma 4.14 zum Einsatz kam.
Phase 2: Die kleinen Jobs werden verteilt: Benutze das Verfahren Least-Loaded, um die kleinen Jobs auf das durch die großen Jobs entstandene Lastgebirge zu verteilen.

Wir zeigen zunächst, dass der durch die Gaußklammer entstehende Rundungsfehler nicht zu groß ist.

4.15 Lemma. *Für $i = 1, \ldots, g$ gilt $p_i' \leq (1+\varepsilon) \cdot p_i^*$ und somit $p_i' \leq \frac{1+\varepsilon}{\varepsilon^2 \cdot T} \cdot p_i$.*

Beweis: Für jeden Job $i \in \{1, \ldots, g\}$ gilt $p_i^* \geq \varepsilon T/(\varepsilon^2 T) = 1/\varepsilon$. Es folgt

$$\frac{p_i' - p_i^*}{p_i^*} \;\leq\; \frac{1}{1/\varepsilon} \;=\; \varepsilon.$$

□

Im folgenden sei $OPT(v_1, \ldots, v_t)$ der minimale Makespan, den man für Jobgrößen $v_1, \ldots, v_t$ erreichen kann.

4.16 Satz. *Der oben angegebene Algorithmus berechnet einen Schedule mit Makespan höchstens $(1+\varepsilon) \cdot OPT(p_1, \ldots, p_n)$. Der Algorithmus hat Laufzeit $n^{O(1/\varepsilon^2)}$.*

Beweis:
Zur Laufzeit: Die Laufzeit wird dominiert durch die Laufzeit des Algorithmus aus Lemma 4.14. Um diese zu analysieren, beobachten wir, dass die Jobgrößen $p_1', \ldots, p_n'$ natürliche Zahlen mit $1 \leq p_i' \leq \lceil 1/\varepsilon^2 \rceil$ sind, es gibt also nur $k := \lceil 1/\varepsilon^2 \rceil$ verschiedene Jobgrößen. Der Algorithmus aus Lemma 4.14 kommt mit Laufzeit $O(k \cdot n^{2k+1}) = n^{O(1/\varepsilon^2)}$ aus.
Zur Approximationsgüte:

Wegen Lemma 4.15 können wir abschätzen:

$$\begin{aligned} OPT(p'_1, \ldots, p'_g) &\leq \frac{1+\varepsilon}{\varepsilon^2 \cdot T} \cdot OPT(p_1, \ldots, p_g) \\ &\leq \frac{1+\varepsilon}{\varepsilon^2 \cdot T} \cdot OPT(p_1, \ldots, p_n). \end{aligned}$$

Wenn wir für die Jobgrößen $p'_1, \ldots, p'_g$ einen Schedule $f : \{1, \ldots, g\} \to \{1, \ldots, m\}$ mit Makespan $OPT(p'_1, \ldots, p'_g)$ haben, dann hat der gleiche Schedule mit Jobgrößen $p_1, \ldots, p_g$ einen Makespan von höchstens

$$(\varepsilon^2 \cdot T) \cdot OPT(p'_1, \ldots, p'_g) \leq (1+\varepsilon) \cdot OPT(p_1, \ldots, p_n).$$

Hier haben wir die letzte Abschätzung für $OPT(p'_1, \ldots, p'_g)$ verwendet.
Nun betrachten wir Phase 2 und unterscheiden zwei Fälle.
Fall I): Durch das Verteilen der kleinen Jobs erhöht sich der Makespan nicht. Dann haben wir in der Tat einen Schedule mit Makespan höchstens $(1+\varepsilon) \cdot OPT(p_1, \ldots, p_n)$ berechnet.
Fall II): Durch das Verteilen der kleinen Jobs erhöht sich der Makespan. Angenommen, nach der Verteilung ist der Makespan Z.
Ein kleiner Job mit Jobgröße $p_i < \varepsilon \cdot T$ ist also derjenige, der zuletzt fertig wird. Damit wissen wir, dass alle Maschinen mindestens bis zur Füllhöhe $Z - p_i > Z - \varepsilon \cdot T$ gefüllt sind, also gilt auch für den optimalen Makespan

$$OPT(p_1, \ldots, p_n) \geq Z - \varepsilon \cdot T.$$

Der Quotient aus dem Makespan Z der berechneten Lösung und dem optimalen Makespan kann also abgeschätzt werden als

$$\begin{aligned} \frac{Z}{OPT(p_1, \ldots, p_n)} &\leq \frac{OPT(p_1, \ldots, p_n) + \varepsilon \cdot T}{OPT(p_1, \ldots, p_n)} \\ &= 1 + \frac{\varepsilon \cdot T}{OPT(p_1, \ldots, p_n)} \end{aligned}$$

Da $T \leq OPT(p_1, \ldots, p_n)$ ist (T war als untere Schranke gewählt), so kann der letzte Quotient abgeschätzt werden durch $(1+\varepsilon)$. □

Wir fassen zusammen:

4.17 Satz. *Es gibt ein PTAS für das Makespan-Scheduling-Problem.*

Wegen des drastischen Einflusses von ε auf die Laufzeit kann dieses PTAS nicht als praktisch angesehen werden. Wenn wir beispielsweise mit LPT konkurrieren wollen, so müssen wir $\varepsilon = \frac{1}{3}$ setzen und erhalten $k = 9$, also eine Laufzeitschranke von $O(n^{19})$. Das PTAS ist somit zwar nicht praktikabel, aber dennoch von großem theoretischen Interesse, weil es zeigt, dass es keine untere Schranke für die beste in Polynomialzeit erreichbare Approximationsgüte geben kann.

5 Randomisierte Algorithmen

5.1 Eine kleine Erinnerung an die Wahrscheinlichkeitstheorie

Wenn wir einen Zufallsversuch durchführen, dann tritt mit bestimmten Wahrscheinlichkeiten ein bestimmtes Ereignis ein. Wenn wir zum Beispiel einen Würfel werfen, dann erhalten wir eine Zahl aus dem Bereich von 1 bis 6, jede mit Wahrscheinlichkeit 1/6.
Mathematisch modelliert man das mit Hilfe einer Zufallsvariable. Im obigen Würfelbeispiel nimmt die Zufallsvariable, nennen wir sie X, jeden der 6 Werte mit Wahrscheinlichkeit 1/6 an.
Für Zufallsvariable X, deren Werte reelle Zahlen sind, kann man den Erwartungswert $E[X]$ definieren.
Nehmen wir zunächst an, dass wir eine Zufallsvariable vorliegen haben, die Werte aus einer endlichen Menge M annehmen kann. Dann ist der Erwartungswert $E[X]$ definiert als

$$E[X] := \sum_{m \in M} Prob(X = m) \cdot m.$$

Wenn X die Zufallsvariable aus dem Würfelbeispiel ist, dann haben wir

$$E[X] = (1/6 \cdot 1 + 1/6 \cdot 2 + 1/6 \cdot 3 + \cdots + 1/6 \cdot 6) = 3.5.$$

Ebenso kann man den Erwartungswert für eine Zufallsvariable definieren, deren Werte aus einer abzählbar unendlichen Menge stammen. Hier muss man allerdings ein bisschen aufpassen, denn unter Umständen konvergiert die (unendliche) Reihe in der Definition des Erwartungswertes nicht. Falls die Zufallsvariable nur Werte annehmen kann, die positiv oder Null sind, dann konvergiert die Reihe jedoch immer und der Erwartungswert existiert (eventuell ist er aber unendlich.) Wir werden in diesem Kapitel nur mit Zufallsvariablen zu tun haben, die keine negativen Werte annehmen.
Erwartungswerte haben die schöne Eigenschaft, dass sie sich linear verhalten. Wenn man also Zufallsvariablen $X_1, \ldots, X_m$ hat, deren Erwartungswerte $E[X_1], \ldots, E[X_m]$ sind, so gilt für den Erwartungswert der Zufallsvariablen $X := X_1 + \cdots + X_m$ die folgende Eigenschaft:

$$E[X] = E[X_1] + \cdots + E[X_m].$$

Entsprechend gilt auch für Konstanten c, dass $E[X_1 + c] = E[X_1] + c$ ist.

5.1 Beispiel. *Angenommen, wir haben drei Zufallsvariablen X_1, X_2, X_3. Dabei seien X_1 und X_3 zwei Würfel, die unabhängig voneinander gewürfelt werden und X_2 sei definiert durch $X_1 + 1$. Wie ist der Erwartungswert der Summe der drei Zufallsvariablen, also*

$$E[X_1 + X_2 + X_3]?$$

Anstatt nun die Definition des Erwartungswerts zu benutzen, wo wir alle möglichen Fälle betrachten müssen, wissen wir dank der Linearität des Erwartungswertes, dass gilt:

$$E[X_1 + X_2 + X_3] = E[X_1] + E[X_2] + E[X_3] = 3.5 + 4.5 + 3.5 = 11.5.$$

Achtung: Die Linearität bezüglich der Multiplikation, also $E[X_1 \cdot X_2] = E[X_1] \cdot E[X_2]$ gilt andererseits nur, wenn die Zufallsvariablen unabhängig sind. Ein Beispiel: Sei X_1 die Zufallsvariable, die einem Würfel entspricht und sei $X_2 = X_1$. (Dies sind sicherlich zwei Zufallsvariablen, die NICHT unabhängig sind.) Dann haben wir

$$E[X_1 \cdot X_2] = 1/6 \cdot (1 \cdot 1 + 2 \cdot 2 + 3 \cdot 3 + 4 \cdot 4 + 5 \cdot 5 + 6 \cdot 6) = 91/6$$

und

$$E[X_1] \cdot E[X_2] = 3.5 \cdot 3.5 \neq 91/6.$$

5.2 Randomisierte Algorithmen

Randomisierte Algorithmen verwenden Zufallsentscheidungen während ihres Ablaufs. Dies kann man sich so vorstellen, dass im Algorithmus eine Zahl m benutzt wird und eine Zahl t aus der Menge $\{1, \ldots, m\}$ gemäß einer Wahrscheinlichkeitsverteilung ausgewürfelt wird. Die Wahrscheinlichkeitsverteilung kann man sich gegeben denken durch m Werte $p_1, \ldots, p_m$ mit $0 \leq p_i \leq 1$ für alle i und $\sum_i p_i = 1$. Das Verhalten des Algorithmus hängt von der getroffenen Wahl ab.
Die Tatsache, dass Zufallsentscheidungen getroffen werden, hat zwei mögliche Konsequenzen:

- Die Laufzeit des Algorithmus ist „zufällig“, also eine Zufallsvariable.
- Die Ausgabe des Algorithmus ist eine Zufallsvariable.

Randomisierte Algorithmen kann man sich durch einen Baum veranschaulichen. Ein randomisierter Algorithmus führt eine gewisse Zeit lang eine deterministische Rechnung durch, bis er eine Zufallsentscheidung trifft. Die deterministische Rechnung kann man sich durch einen Knoten veranschaulichen und immer dann, wenn der Algorithmus eine Zufallsentscheidung trifft, hat man eine Verzweigung. Falls eine von m möglichen Entscheidungen getroffen wird, verzweigt der entsprechende Knoten in m Söhne. Ein Ablauf des Algorithmus entspricht nun einem Durchlauf durch diesen Baum. Solch ein Baum hat eventuell unendliche Tiefe. Bei randomisierten Algorithmen kann man sich für die erwartete Laufzeit $E[T]$ interessieren. Diese ist wie folgt definiert:

$$E[T] = \sum_{v \text{ ist Knoten im Baum}} \text{Prob}[v \text{ wird durchlaufen}] \cdot (\text{Laufzeit im Knoten } v).$$

Wir betrachten zunächst zwei sehr einfache randomisierte Algorithmen:

Algorithmus A
repeat
  wirf Münze $z \in \{0, 1\}$ (gemäß der Gleichverteilung).
until $z = 0$.

Man beachte, dass der zugehörige Baum in diesem Fall ein unendlicher Baum ist, denn es könnte ja passieren, dass die Münze immer wieder 1 ist.
Wie ist die erwartete Laufzeit des Algorithmus? Wenn man den zugehörigen Baum zeichnet, dann hat jeder innere Knoten zwei Söhne, den linken (Münze = 0), der ein Blatt ist, und den rechten, an dem wieder ein unendlicher Baum hängt. Wenn wir jedem Knoten die Laufzeit 1 zuweisen, welche erwartete Laufzeit hat dann unser Algorithmus? Wir nummerieren die Ebenen des Baumes durch, mit 0 für die Wurzel beginnend. Auf jeder Ebene des Baumes, außer auf Ebene 0, gibt es zwei Knoten. Betrachten wir einen Knoten auf Ebene i. Die Wahrscheinlichkeit, diesen Knoten zu erreichen, beträgt $(1/2)^i$. Wenn wir jedem Knoten die Laufzeit 1 zuweisen, so erhalten wir nun als erwartete Laufzeit des Algorithmus:

$$1 + \sum_{i=1}^{\infty} 2 \cdot (1/2)^i = 1 + 2 \cdot \sum_{i=1}^{\infty} (1/2)^i = 1 + 2 = 3.$$

Betrachten wir nun eine Verallgemeinerung von Algorithmus A. Sei $0 < p \leq 1$.

Algorithmus B
repeat
wirf Münze $z \in \{0, 1\}$, und zwar
$z = 0$ mit Wahrscheinlichkeit p und $z = 1$ mit Wahrscheinlichkeit $1 - p$.
until $z = 0$.

Der zugrunde liegende Baum sieht genauso aus wie bei Algorithmus A. Jetzt allerdings beträgt die Wahrscheinlichkeit, dass ein Knoten auf Ebene i erreicht wird:

$(1-p)^{i-1} \cdot p$, falls der Knoten ein Blatt ist und $(1-p)^i$, falls der Knoten kein Blatt ist.

Wir ermitteln die erwartete Laufzeit: Auf Ebene $i > 0$ befindet sich jeweils ein Blatt und ein Nichtblatt. Wenn wir jedem Knoten die Laufzeit 1 zuweisen, ergibt sich als erwartete Laufzeit:

$$1 + \sum_{i=1}^{\infty} \left((1-p)^i + (1-p)^{i-1} \cdot p \right) = 1 + \sum_{i=1}^{\infty} (1-p)^{i-1} = 1 + \sum_{i=0}^{\infty} (1-p)^i.$$

Mit der bekannten Summenformel erhalten wir:

$$= 1 + \frac{1}{p}.$$

Übrigens ergibt sich daraus auch, dass der Erwartungswert der Zufallsvariable X, die zählt, wie oft wir die Münze werfen, bis der Algorithmus stoppt, sich errechnet als $E[X] = 1/p$.

Ein weiteres einfaches Beispiel:
Jemand möchte eine zufällige Lottoreihe auswürfeln und macht dies wie folgt:

Algorithmus Lotto1
a: array[1..49] of **boolean**
i: **integer**
for i:= 1 to 49 **do** a[i]=0
for i:= 1 to 6 **do**
begin
  repeat
    würfle eine Zahl z aus $\{1, \ldots, 49\}$, gemäß der Gleichverteilung.
  until a[z]=0
  a[z]=1.
end
Gib alle Zahlen i mit a[i]=1 aus.

Bevor wir uns eine Analyse dieses Algorithmus ansehen, eine Anmerkung: Selbstverständlich kann man bei einer tatsächlichen Implementierung anders vorgehen, so wie im folgenden Programmstück zum Beispiel.
Algorithmus Lotto2
a: array[1..49] of **integer**
i,h : integer
for i:= 1 to 49 **do** a[i]=i
for i:= 1 to 6 **do**
begin
    würfle eine Zahl z aus $\{i, \ldots, 49\}$, gemäß der Gleichverteilung.
    vertausche $a[i]$ und $a[z]$. (Also $h := a[i]; a[i] = a[z]; a[z] = h;$)
end
for $i := 1$ **to** 6 **do** gib $a[i]$ aus.

Auch überlasse ich dem Leser und der Leserin, nachzuvollziehen, dass auch der folgende Algorithmus das Gewünschte tut. Dieser Algorithmus kommt sogar ganz ohne Array aus:

Algorithmus Lotto3
k,n,i,x: integer
$k := 6; n := 49.$
$i := 1;$
while $i \leq 49$ **and** $k \neq 0$ **do**
begin
  Setze $x = 1$ mit Wahrscheinlichkeit k/n und sonst $x = 0$.
  Falls $x = 1$, gib i aus und setze $k := k - 1$.
  $i := i + 1; n := n - 1.$
end

Wir schauen uns den ersten Algorithmus Lotto1 aus *didaktischen* Gründen an, um zu

sehen, wie man einfach erwartete Laufzeiten ermitteln kann.
Wie häufig wird im Erwartungswert eine Zahl gewürfelt, bis der Algorithmus **Lotto1** stoppt? Wenn $i = 1$ ist, enthält das Array a noch 49 Nullen. Mit $p = 49/49$ und unserer Analyse von Algorithmus B ergibt sich also, dass bei $i = 1$ der Erwartungswert der Würfelversuche 1 ist.
Wenn $i = 2$ ist, dann sind 48 der Arrayelemente 0. Mit $p = 48/49$ ergibt sich, dass im Erwartungswert 49/48 Würfelversuche gemacht werden.
Allgemein beträgt die erwartete Anzahl der Würfelversuche $49/(50 - i)$.
Wegen der Linearität des Erwartungswerts ist die erwartete Gesamtanzahl der Würfelversuche

$$E[X] = \frac{49}{49} + \frac{49}{48} + \frac{49}{47} + \cdots + \frac{49}{44} \approx 6.33.$$

5.3 MAXSAT

MAXSAT ist eines der grundlegenden NP–vollständigen Probleme und eines der am meisten untersuchten. Es ist wie folgt beschrieben. Wir haben n Boolesche Variablen $x_1, \ldots, x_n$. Ein Literal ist eine Variable x_i oder eine negierte Variable $\overline{x_i}$.
Eine Klausel C ist ein ODER von Literalen. Zum Beispiel ist $(x_1 \vee x_4 \vee \overline{x_6})$ eine Klausel.
Eine Variablenbelegung ist gegeben durch ein Element $a \in \{0,1\}^n$. Hierbei wird die Variable x_i ersetzt durch den Wert a_i. Eine Belegung a „macht eine Klausel C wahr“ bzw. erfüllt die Klausel C, wenn nach Ersetzung der Variablen durch die entsprechenden Werte sich der logische Wert 1 ergibt. Eine Klausel heißt reduziert, wenn sie zu jedem $i \in \{1, \ldots, n\}$ maximal ein Literal auf x_i enthält.
Beispiel: Die Klauseln $(x_1 \vee x_1 \vee x_5)$ bzw. $(x_1 \vee \overline{x_1} \vee x_2)$ sind nicht reduziert, $(x_1 \vee \overline{x_3} \vee x_2)$ ist reduziert.

MAXSAT

Gegeben:	eine Liste von reduzierten Klauseln.
Gesucht:	eine Variablenbelegung, die möglichst viele Klauseln gleichzeitig erfüllt.

MAX–k–SAT ist die Variante von MAXSAT, bei der alle Klauseln Länge genau k haben.
MAXSAT ist ein NP–hartes Problem, und auch MAX–k–SAT ist für jedes $k \geq 2$ ein NP–hartes Problem.

5.3.1 Ein randomisierter Algorithmus

Wir schauen uns zunächst einen einfachen randomisierten Algorithmus an:

5.2 Satz. *Es gibt einen randomisierten Algorithmus A für das MAX–k–SAT–Problem, der folgende Eigenschaft hat: Gegeben m Klauseln $C_1, \ldots, C_m$, dann berechnet A eine Variablenbelegung, die im Erwartungswert $(1 - \frac{1}{2^k}) \cdot m$ der Klauseln erfüllt. Die Laufzeit von A ist $O(n)$.*

Beweis: Der Algorithmus A setzt der Reihe nach, jeweils mit Wahrscheinlichkeit 1/2 und unabhängig voneinander, die Variablen auf 1 bzw. 0.
Sei X die Zufallsvariable, die angibt, wieviele der Klauseln erfüllt sind. Für die i–te Klausel C_i definieren wir die Zufallsvariable X_i wie folgt:

$$X_i := \begin{cases} 1, & \text{falls die } i\text{–te Klausel erfüllt ist.} \\ 0 & \text{sonst.} \end{cases}$$

Dann ist

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

und wegen der Linearität des Erwartungswerts

$$E[X] = E[X_1] + \cdots + E[X_m].$$

Es gilt $E[X_i] = Prob[\text{die } i\text{–te Klausel ist erfüllt.}]$.
O.B.d.A. ist (nach Umbenennung der Variablen) die i–te Klausel gegeben durch $x_1 \vee x_2 \vee \cdots \vee x_k$. Diese wird nur dann nicht erfüllt, wenn $x_1 = \cdots = x_k = 0$ ist. Dies passiert mit Wahrscheinlichkeit $(\frac{1}{2})^k$. Die Wahrscheinlichkeit, dass die i–te Klausel also erfüllt wird, ist $1 - (\frac{1}{2})^k$ und somit

$$E[X_i] = 1 - \left(\frac{1}{2}\right)^k.$$

Für die Zufallsvariable X gilt damit $E[X] = \left(1 - \frac{1}{2^k}\right) \cdot m$. □

5.3.2 Ein deterministischer Algorithmus durch Derandomisierung

Derandomisierung bezeichnet den Vorgang, aus einem randomisierten Algorithmus auf recht mechanische Art und Weise einen deterministischen Algorithmus zu machen. Wir wollen dies exemplarisch an dem gerade vorgestellten einfachen MAXSAT–Algorithmus vorführen.
Dazu verallgemeinern wir zunächst unseren Algorithmus. Anstatt jede Variable mit Wahrscheinlichkeit 1/2 auf 1 zu setzen, führen wir für jede Variable i eine eigene Wahrscheinlichkeit p_i ein, mit der wir sie auf 1 setzen. Wenn diese Wahrscheinlichkeiten $p_1, \ldots, p_n$ gegeben sind, dann kann man für jede Klausel C die Wahrscheinlichkeit berechnen, dass C erfüllt wird. Zum Beispiel ist die Klausel $x_1 \vee x_2 \vee x_3$ mit Wahrscheinlichkeit $1-(1-p_1)\cdot(1-p_2)\cdot(1-p_3)$ erfüllt oder die Klausel $x_1 \vee \overline{x_2}$ mit Wahrscheinlichkeit $1-(1-p_1)\cdot p_2$.
Allgemein: Wenn $I^+ = \{i \mid x_i \text{ kommt in der Klausel positiv vor}\}$ und I^- analog für negative Vorkommen von x_i definiert ist, dann ist die Wahrscheinlichkeit, dass die Klausel erfüllt wird, genau

$$F_C(p_1, \ldots, p_n) = 1 - \prod_{i \in I^+} (1 - p_i) \cdot \prod_{j \in I^-} p_j.$$

Dies ist eine Funktion, die in jeder Variablen linear ist.
Der Erwartungswert für die erfüllte Anzahl an Klauseln, nennen wir ihn ERF, wenn die Klauseln $C_1, \ldots, C_m$ betrachtet werden, ist dann

$$ERF(p_1, \ldots, p_n) = \sum_{i=1}^{m} F_{C_i}(p_1, \ldots, p_n).$$

Als Summe von Funktionen, die in jeder Variablen linear sind, ist auch dies eine Funktion, die in jeder Variablen linear ist.
Eine Funktion, die in einer Variable x linear ist, nimmt, wenn man alle anderen Variablen fixiert lässt, ihr Minimum und Maximum an den Randwerten für x an.
Wenn man nämlich schreibt $F(x, p_2, \ldots, p_n) = x \cdot c_1 + c_2$, dann wird, falls $c_1 \geq 0$, das Maximum bei $x = 1$ angenommen und das Minimum bei $x = 0$. Falls $c_1 \leq 0$, dann wird das Maximum bei $x = 0$ und das Minimum bei $x = 1$ angenommen.
Dies zeigt uns direkt, wie wir einen deterministischen Algorithmus aus dem randomisierten Algorithmus bekommen, der *immer* eine Belegung der Variablen findet, die mindestens $(1 - \frac{1}{2^k}) \cdot m$ Klauseln erfüllt:

5.3 Algorithmus.

Setze $p_1, \ldots, p_n := 1/2$.

for $i := 1$ **to** n **do**
begin
if $ERF(p_1, \ldots, p_{i-1}, 1, p_{i+1}, \ldots, p_n) >$
$ERF(p_1, \ldots, p_{i-1}, 0, p_{i+1}, \ldots, p_n)$ **then** p_i*:=1*
else p_i*:=0*
end
Setze alle Variablen x_i *auf* 1*, für die* $p_i = 1$ *ist.*

5.4 Satz. *Auf einer MAX–k–SAT–Eingabe bestehend aus Klauseln* $C_1, \ldots, C_m$ *berechnet Algorithmus 5.3 in Polynomialzeit eine Belegung der Variablen, die mindestens* $(1 - \frac{1}{2^k}) \cdot m$ *Klauseln erfüllt.*

Beweis: Der Algorithmus startet mit $p_1, \ldots, p_n := 1/2$. Da wir

$$ERF(\frac{1}{2}, \frac{1}{2}, \ldots, \frac{1}{2}) = (1 - \frac{1}{2^k}) \cdot m$$

ausgerechnet haben, gilt zu Beginn des Algorithmus

$$ERF(p_1, \ldots, p_n) = (1 - \frac{1}{2^k}) \cdot m.$$

Unsere Überlegungen gerade haben gezeigt, dass ERF, wenn alle $p_1, \ldots, p_n$ außer p_i fixiert sind, maximal ist für $p_i = 1$ bzw. $p_i = 0$. Einer der beiden Werte

$$ERF(p_1, \ldots, p_{i-1}, 1, p_{i+1}, \ldots, p_n) \text{ bzw. } ERF(p_1, \ldots, p_{i-1}, 0, p_{i+1}, \ldots, p_n)$$

ist also mindestens genauso groß wie $ERF(p_1, \ldots, p_{i-1}, p_i, p_{i+1}, \ldots, p_n)$. Insbesondere ist das *Maximum* der beiden Werte mindestens so groß wie $ERF(p_1, \ldots, p_{i-1}, p_i, p_{i+1}, \ldots, p_n)$. Der Algorithmus setzt in der Schleife $p_i = 0$ bzw. $p_1 = 1$ je nachdem, für welchen der beiden Werte das Maximum angenommen wird. Nach dieser Modifikation gilt also immer noch

$$ERF(p_1, \ldots, p_{i-1}, p_i, p_{i+1}, \ldots, p_n) \geq (1 - \frac{1}{2^k}) \cdot m.$$

Schließlich endet der Algorithmus mit $p_1, \ldots, p_n$, wobei alle p_i nur noch aus $\{0, 1\}$ sind. Welchem randomisierten Algorithmus entspricht das? Variable x_i wird mit Wahrscheinlichkeit p_i auf 1 gesetzt. Dies bedeutet doch, dass alle x_i mit $p_i = 1$ auf 1 gesetzt und alle anderen auf 0 gesetzt werden. Dies macht der Algorithmus.
Welche Laufzeit hat der Algorithmus? Das Berechnen der Funktion ERF hat eine Laufzeit, die polynomiell in m und n ist, die Schleife wird n–mal durchlaufen. Insgesamt ist die Laufzeit ein Polynom in der Länge der Eingabe. □

Die Methode, die wir oben angewendet haben, um aus dem randomisierten Algorithmus einen deterministischen zu machen, ist die Methode der Potenzialfunktionen. Im Beispiel war die Funktion ERF die Potenzialfunktion, die den Ablauf des Algorithmus gesteuert hat.

5.3.3 Ein weiterer Approximationsalgorithmus für MAXSAT

Eine äquivalente Formulierung für das MAXSAT–Problem ist durch folgende Formulierung gegeben:

$$\textbf{max} \sum_{j=1}^{m} z_j \text{ unter den Nebenbedingungen}$$

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

$$\sum_{x_i \text{ kommt in Klausel } j \text{ vor}} y_i + \sum_{\overline{x_i} \text{ kommt in Klausel } j \text{ vor}} (1 - y_i) \geq z_j \text{ für } j = 1, \ldots, m.$$

Dabei zählt der Ausdruck

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

genau, wieviele Literale in der j–ten Klausel wahr sind, wenn man die Variablen $x_i := y_i$ setzt. Da der Ausdruck $\sum_{j=1}^{m} z_j$ maximiert wird, gibt z_j dann an, ob die j–te Klausel erfüllt ist (also $z_j = 1$) oder nicht (also $z_j = 0$).
Da MAXSAT NP–hart ist, ist es natürlich auch NP–hart, das obige „mathematische Programm“ zu lösen.
Nun relaxieren wir das gegebene Programm und ersetzen die Bedingung $y_i, z_j \in \{0, 1\}$ durch die Bedingung $y_i, z_j \in [0..1]$.

Dadurch wird aus dem gegebenen Problem ein so genanntes Lineares Programm. Wir können die Theorie der Linearen Programme an dieser Stelle nicht in voller Breite entwickeln, wir können aber die Eigenschaften aufzählen, die für unsere Zwecke von Interesse sind:

Exkurs Lineare Programme

Ein lineares Programm ist gegeben durch eine lineare Zielfunktion $c(x_1, \ldots, x_n) := c_1 x_1 + \cdots + c_n x_n$. Die Aufgabe besteht darin, diese Zielfunktion zu minimieren unter der Randbedingung, dass der Vektor $x_1, \ldots, x_n$ eine Reihe von linearen Bedingungen erfüllt, nämlich

$$\begin{aligned} a_{11}x_1 + \cdots + a_{1n}x_n &\leq d_1 \\ a_{21}x_1 + \cdots + a_{2n}x_n &\leq d_2 \\ \vdots &\quad \vdots \\ a_{m1}x_1 + \cdots + a_{mn}x_n &\leq d_m \end{aligned}$$

Die auftretenden Koeffizienten seien dabei rational.

Am Rande sei bemerkt, dass Gleichheitsbedingungen wie zum Beispiel $x = c$ durch zwei Ungleichungen der Form $x \leq c$ und $-x \leq -c$ ausgedrückt werden können.

Es ist bekannt, dass lineare Programme in Polynomialzeit gelöst werden können, genauer: in einer Zeit, die polynomiell in der Eingabelänge des linearen Programms ist (man zählt dort alle Bits mit, die zum Kodieren der einzelnen Koeffizienten benötigt werden.) Man bekommt als Lösung nicht nur den optimalen Wert der Zielfunktion geliefert, sondern auch die zugehörige Variablenbelegung $x_1, \ldots, x_n$.

Ende des Exkurses

Wir setzen nun einen entsprechenden Algorithmus als Blackbox voraus und nehmen an, dass wir zum Linearen Programm die Lösungen $\widehat{y_i}$ und $\widehat{z_j}$ gegeben haben.

Diese Werte liegen nun alle zwischen 0 und 1. Wie bekommen wir daraus eine Belegung der Variablen? Wir wählen als Vorgehen das „Randomisierte Runden“:

for $i = 1$ **to** n **do**
begin
  setze Variable x_i mit Wahrscheinlichkeit $\widehat{y_i}$ auf 1, (und 0 sonst).
end

Wir analysieren nun, mit welcher Wahrscheinlichkeit die j–te Klausel erfüllt wird.

5.5 Lemma. *Sei die j–te Klausel eine Klausel der Länge k. Die Wahrscheinlichkeit, dass sie durch Randomisiertes Runden erfüllt wird, ist mindestens*

$$[1 - (1 - \frac{1}{k})^k] \cdot \widehat{z_j}.$$

Vor dem Beweis des Lemmas zwei Eigenschaften, die man mit ein bisschen Analysis nachweisen kann:

Lemma 1: Seien k Zahlen $a_1, \ldots, a_k \geq 0$ gegeben. Dann gilt

$$a_1 \cdots a_k \leq (\frac{a_1 + \ldots + a_k}{k})^k.$$

Lemma 2: Auf dem Intervall $x \in [0..1]$ gilt $1 - (1 - \frac{x}{k})^k \ \geq \ (1 - (1 - \frac{1}{k})^k) \cdot x$.

Beweis von Lemma 5.5:
Sei die Klausel oBdA durch $x_1 \vee \cdots \vee x_k$ gegeben. Die Klausel ist dann mit Wahrscheinlichkeit $\prod_{i=1}^{k}(1 - \widehat{y_i})$ nicht erfüllt, also mit Wahrscheinlichkeit $1 - \prod_{i=1}^{k}(1 - \widehat{y_i})$ erfüllt.
Es ist $\widehat{y_1} + \cdots + \widehat{y_k} \geq \widehat{z_j}$, also

$$(1 - \widehat{y_1}) + \cdots + (1 - \widehat{y_k}) \leq k - \widehat{z_j},$$

also nach Lemma 1:

$$1 - \prod_{i=1}^{k}(1 - \widehat{y_i}) \geq 1 - (1 - \frac{\widehat{z_j}}{k})^k.$$

Nach Lemma 2 ist dies mindestens $(1 - (1 - \frac{1}{k})^k) \cdot \widehat{z_j}$.
Man beachte, dass für alle k gilt, dass $1 - (1 - \frac{1}{k})^k \geq 1 - \frac{1}{e} \approx 0.632$ ist.

5.6 Satz. *Gegeben eine Menge von Klauseln, die eine Eingabe für das MAXSAT–Problem ist. Sei OPT die Anzahl an Klauseln, die man maximal gleichzeitig erfüllen kann.*
Dann liefert Randomisiertes Runden mit den Parametern $\widehat{y_1}, \ldots, \widehat{y_n}$ eine Variablenbelegung, die im Erwartungswert mindestens $(1 - \frac{1}{e}) \cdot OPT$ Klauseln erfüllt.

Beweis: Da das lineare Programm eine Relaxation des MAXSAT-Problems darstellt, gilt $OPT \leq \sum \widehat{z_j}$, und der Erwartungswert der Anzahl erfüllter Klauseln ist

$$\sum_j [1 - (1 - \frac{1}{k})^k] \cdot \widehat{z_j} \geq (1 - (1/e)) \cdot \sum \widehat{z_j} \geq (1 - (1/e)) \cdot OPT.$$

□

Übrigens hatten wir im Kapitel 5.3 schon einmal analysiert, wie sich Randomisiertes Runden mit den Parametern $1/2, \ldots, 1/2$ verhält.
Nun kommen wir zur Beschreibung eines Verfahrens, das Approximationsgüte 0.75 hat.

Verfahren MIX: Berechne eine Variablenbelegung a durch Randomisiertes Runden mit den Parametern $1/2, \ldots, 1/2$. Berechne eine Variablenbelegung b durch Randomisiertes Runden mit den Parametern $\hat{y}_1, \ldots, \hat{y}_n$, die sich aus der optimalen Lösung des linearen Programms ergeben. Gib diejenige Variablenbelegung von a und b aus, die mehr Klauseln erfüllt.

5.7 Satz. *Sei eine Menge von Klauseln gegeben und sei OPT die Anzahl an Klauseln, die maximal gleichzeitig erfüllt werden kann. Dann liefert das Verfahren MIX eine Variablenbelegung, die im Erwartungswert mindestens* $0.75 \cdot OPT$ *viele Klauseln erfüllt.*

Beweis: In unseren obigen Analysen haben wir als Nebenprodukt erhalten, dass bei Randomisiertem Runden mit Parametern $1/2, \ldots, 1/2$ eine Klausel der Länge k mit Wahrscheinlichkeit $A(k) := 1 - 2^{-k}$ erfüllt ist und bei Randomisiertem Runden mit den Parametern $\hat{y}_1, \ldots, \hat{y}_n$ mit Wahrscheinlichkeit mindestens $B(k) \cdot \hat{z}_j$ erfüllt ist, wenn wir $B(k) := 1 - (1 - \frac{1}{k})^k$ setzen. Wir verschaffen uns eine kleine Übersicht über die Werte $A(k)$ und $B(k)$ für einige Werte von k:

k	$A(k)$ $1 - 2^{-k}$	$B(k)$ $1 - (1 - \frac{1}{k})^k$	$(A(k) + B(k))/2$
1	0.5	1.0	0.75
2	0.75	0.75	0.75
3	0.875	0.704	0.7895
4	0.938	0.684	0.811
5	0.969	0.672	0.8205

Sei n_1 die Anzahl an Klauseln, die durch Randomisiertes Runden mit $(1/2, \ldots, 1/2)$ erfüllt werden, und sei n_2 die Anzahl an Klauseln, die durch Randomisiertes Runden mit Parametern $\hat{y}_1, \ldots, \hat{y}_n$ erfüllt werden. Der Algorithmus MIX erfüllt $\max\{n_1, n_2\}$ Klauseln.
Wir zeigen, dass $\mathrm{E}[\max\{n_1, n_2\}] \geq \frac{3}{4} \cdot \sum_j \hat{z}_j$ ist, denn dann folgt $\mathrm{E}[\max\{n_1, n_2\}] \geq \frac{3}{4} \cdot \mathrm{OPT}$.
Da $\max\{n_1, n_2\} \geq \frac{n_1+n_2}{2}$ ist („das Maximum zweier Zahlen ist mindestens so groß wie der Durchschnitt der beiden“), reicht es, zu zeigen, dass

$$\mathrm{E}\left[\frac{n_1 + n_2}{2}\right] \geq \frac{3}{4} \cdot \sum_j \hat{z}_j \text{ ist.}$$

Sei $\ell(C)$ die Länge einer Klausel C. Es ist

$$\begin{aligned} \mathrm{E}[n_1] &= \sum_{j=1}^{m} A(\ell(C_j)) \geq \sum_{j=1}^{m} A(\ell(C_j)) \cdot \widehat{z}_j \text{ sowie} \\ \mathrm{E}[n_2] &\geq \sum_{j=1}^{m} B(\ell(C_j)) \cdot \hat{z}_j. \end{aligned}$$

Somit ist

$$\mathrm{E}[n_1] + \mathrm{E}[n_2] \geq \sum_{j=1}^{m} (A(\ell(C_j)) + B(\ell(C_j))) \cdot \hat{z}_j.$$

Eine einfache Rechnung zeigt, dass $A(k) + B(k) \geq 3/2$ für alle natürlichen Zahlen k ist:

Für $k = 1$ und $k = 2$ ergibt die Summe den Wert $3/2$, für $k \geq 3$ beobachten wir $(1 - 2^{-k}) \geq 7/8$ und $1 - (1 - \frac{1}{k})^k \geq 1 - (1/e)$, damit ist die Summe mindestens

$$(7/8) + 1 - (1/e) \approx 1.507\ldots \geq 3/2.$$

Damit folgt die Aussage. □

5.4 MaxCut-Problem

Gegeben: Ein ungerichteter Graph $G = (V, E)$.
Gesucht: Eine Zerlegung der Knotenmenge V in zwei nichtleere Mengen V_1 und V_2 mit $V = V_1 \cup V_2$ und $V_1 \cap V_2 = \emptyset$, so dass die Anzahl der Kanten zwischen V_1 und V_2 *maximal* ist.

Im Gegensatz zum *MinCut-Problem* ist das *MaxCut-Problem* NP-hart, d.h. unter der Annahme $\mathcal{P} \neq \mathcal{NP}$ ist es nicht in Polynomialzeit lösbar. Zur exakten Lösung des *MaxCut-Problems* sind also keine effizienten Verfahren zu erwarten.
Man hat sich daher darauf konzentriert, Approximationsverfahren polynomieller Laufzeit zu betrachten. Ein vor wenigen Jahren erzieltes Resultat von Goemans und Williamson (1994) besagt, dass man in Polynomialzeit eine Lösung für das *MaxCut-Problem* angeben kann, die um höchstens 14% von dem Wert der optimalen Lösung entfernt ist. Andererseits gelang es 1996 Håstad, zu zeigen, dass man unter der Annahme $\mathcal{P} \neq \mathcal{NP}$ kein Polynomialzeitverfahren angeben kann, welches den Wert der optimalen Lösung bis auf 6% genau annähert. Es gibt also (falls $\mathcal{P} \neq \mathcal{NP}$) kein Polynomialzeitverfahren, das das Optimum bis auf 6% genau bestimmet, aber ein Polynomialzeitverfahren, das das Optimum auf 14% genau bestimmt.
Sehen wir uns einen einfachen Algorithmus für das MaxCut–Problem an:

5.8 Algorithmus.

for $i := 1$ **to** n **do**
begin
  Setze $x_i := 1$ *bzw.* $x_i := 0$ *mit Wahrscheinlichkeit* $1/2$.
end
Sei $V_1 := \{v_i \in V \mid x_i = 1\}$ *und* $V_2 := \{v_i \in V \mid x_i = 0\}$

Wir würfeln also für alle Knoten unabhängig voneinander aus, ob wir sie in die Menge V_1 oder in die Menge V_2 einfügen.

5.9 Satz. *Algorithmus 5.8 liefert in Zeit $O(n)$ eine Zerlegung $V_1 \cup V_2$. Für die Zufallsvariable X, die die Anzahl Kanten zwischen V_1 und V_2 zählt, gilt:*

$$E[X] = \frac{|E|}{2}.$$

Beweis: Die Aussage über die Laufzeit ist trivial. Für eine beliebige Kante $e = \{v, w\} \in E$ ist die Wahrscheinlichkeit, dass sie zwischen den beiden Mengen verläuft, genau 1/2. Es muss nämlich entweder v in die Menge V_1 und w in die Menge V_2 gewürfelt werden oder v in die Menge V_2 und w in die Menge V_1. Beide Ereignisse passieren mit Wahrscheinlichkeit 1/4.
Sei X_e die (Indikator–)Zufallsvariable, die 1 ist, wenn die Kante e zwischen den beiden Mengen verläuft und 0 sonst. Es gilt also $E[X_e] = 1/2$ für jede Kante $e \in E$.
Es gilt $X = \sum_{e \in E} X_e$, wegen der Linearität des Erwartungswertes ist also $E[X] = |E|/2$. □

Auch diesen einfachen Algorithmus kann man mit Hilfe der Methode der Potenzialfunktionen derandomisieren. Dies überlassen wir dem Leser und der Leserin als Übungsaufgabe.

5.5 MinCut

Auch hier benötigen wir die Operation „Kontraktion“, die wir schon im Kapitel 2.7 beschrieben haben.

5.5.1 Algorithmus „Randomisierte Kontraktion“

Wir stellen nun den randomisierten Algorithmus, basierend auf Kontraktionen, vor. Wenn der Eingabegraph G nicht zusammenhängend ist, dann können wir V_1 als die Menge der Knoten einer beliebigen Zusammenhangskomponente von G wählen und V_2 als alle anderen Knoten und die Partition $V_1 \dot\cup V_2$ ist dann minimal. Im folgenden sei also der Eingabegraph zusammenhängend. Durch die Kontraktionen bleibt der Zusammenhang erhalten.

Eingabe:	Ein zusammenhängender Graph $G = (V, E)$ mit $\|V\| \geq 2$, dessen Kanten mit Gewichten $w(e)$ versehen sind.
Ausgabe:	Eine Partition $V = V_1 \dot\cup V_2$ bzw. die entsprechende Menge der Kanten $e \in E$ zwischen V_1 und V_2.

$H := G$
while H hat mehr als 2 Knoten **do**
  Wähle zufällig eine Kante aus, und zwar jeweils Kante e mit
  Wahrscheinlichkeit $w(e)/W$, wenn W das Gesamtgewicht aller Kanten ist.
  $H :=$ kontraktion(H, e) („Kontrahiere Kante e“)
end of while
V_1 und V_2 sind die beiden Knotenmengen, die den zwei (Meta-) Knoten in H entsprechen.

Da der Eingabegraph G und somit auch H zusammenhängend ist, ist die Menge der Kanten, aus der wir eine Kante auswählen, nie leer während des Algorithmus.

5.10 Satz. *Für Graphen mit n Knoten kann der Algorithmus „Randomisierte Kontraktion“ so implementiert werden, dass er in Zeit $O(n^2)$ ausgeführt werden kann.*

Beweis: Es reichen folgende Beobachtungen:

- Es werden maximal $n-2$ Kontraktionen durchgeführt.
- Die Zeit für eine Kontraktion ist $O(n)$ für $|V| = n$.
- Also ist die Gesamtzeit $O(n^2)$.

Der Algorithmus „Randomisierte Kontraktion“ terminiert also nach $O(n^2)$ Schritten und liefert eine Partition $V = V_1 \dot\cup V_2$, die aber *nicht* notwendigerweise einen optimalen Cut (Zerlegung) darstellen muss. □

Es ist hier nur noch zu überlegen, wie die zufällige Auswahl einer Kante in Zeit $O(n)$ bewerkstelligt werden kann. Dies sei als Übung überlassen.

5.11 Lemma. *Es gelten folgende drei Aussagen für Eingabegraphen $G = (V, E)$ mit $|V| = n$:*

1. *Eine Partition $V = V_1 \dot\cup V_2$ ist genau dann Resultat des Algorithmus „Randomisierte Kontraktion“, wenn im Verlauf des Algorithmus keine Kante kontrahiert wurde, die zwischen V_1 und V_2 verläuft.*
2. *Ist $mincut(G) = W$, dann gilt $w(E) \geq W \cdot (n/2)$.*
3. *Für jede Kante $e \in E$ gilt: $mincut(G) \leq mincut(kontraktion(G, e))$.*

Beweis: Aussage 1 ist trivialerweise richtig. Für Aussage 2 nehmen wir an, es gäbe einen Knoten $v \in V$ mit der Eigenschaft $\text{weightedgrad}(v) := \sum_{e=\{v,\cdot\}} w(e) < W$. Dann würde die Zerlegung $V_1 = \{v\}$ und $V_2 = V \setminus \{v\}$ einen Schnitt mit Wert kleiner als W liefern und wir hätten einen Widerspruch zur Voraussetzung $mincut\,(G) = W$. Also ist $\text{weightedgrad}(v) \geq W$ für alle Knoten v. Das Gesamtgewicht $w(E) = \sum_{e \in E} w(e)$ lässt sich offensichtlich immer als $(1/2) \cdot \sum_{v \in V} \text{weightedgrad}(v)$ errechnen. Somit ergibt sich $w(E) \geq (n/2) \cdot W$, also Aussage 2. Für Aussage 3 verweisen wir auf Satz 2.27 auf Seite 23. □

5.12 Satz. *Sei $G = (V, E)$ ein Graph mit $|V| = n$, und K die Menge der Kanten zwischen zwei Mengen V_1 und V_2, die einen minimalen Cut liefern. Dann gilt:*

$$Prob\,[\text{„Randomisierte Kontraktion“ liefert } K] \geq \frac{2}{n^2}\,.$$

Die Wahrscheinlichkeit, dass der Algorithmus eine optimale Lösung für das *MinCut-Problem* liefert, ist also mindestens $\frac{2}{n^2}$.
Beweis: Wir betrachten die Zerlegung $V = V_1 \dot\cup V_2$, die zur Menge K (minimaler Cut) gehört, und die Wahrscheinlichkeit, dass keine Kante zwischen den Mengen V_1 und V_2 kontrahiert wurde.

In jeder Iteration wird eine Kante kontrahiert. Wir führen also $(n-2)$ Iterationen durch.

- Unmittelbar vor der i-ten Iteration liegen $n_i := n - i + 1$ Knoten vor. Der vor der i-ten Iteration zugrunde liegende Graph sei mit G_i bezeichnet, mit $G_1 := G$.
- Der Algorithmus liefert die Menge K genau dann, wenn keine Kante $e \in K$ im Verlauf kontrahiert wurde.

Also gilt nun (Wahrscheinlichkeitsrechnung):

$$Prob\,[\text{„Randomisierte Kontraktion“ liefert } K]$$

$$= \prod_{i=1}^{n} Prob\,[\text{In Iteration } i \text{ wird keine Kante } e \in K \text{ kontrahiert, vorausgesetzt } E_i \text{ gilt}]\,,$$

Hier bezeichnet E_i das Ereignis, dass in Iteration $1, \ldots, i-1$ keine Kante $e \in K$ kontrahiert wurde. Wurde in den Iterationen $1, 2, \ldots, i-1$ keine Kante aus K kontrahiert, so ist K auch ein Cut in G_i, also gilt nach Lemma 5.11 (3)

$$mincut(G_i) = w(K) =: W.$$

Hierbei ist W das Gewicht des Cuts K in G.

Wir nehmen an, dass in den Iterationen $1, 2, \ldots, i-1$ keine Kante aus K kontrahiert wurde. Dann gilt nach Lemma 5.11 (2), dass das Gewicht aller Kanten in G_i mindestens $W \cdot \frac{n_i}{2}$ beträgt und es folgt, dass die Wahrscheinlichkeit, in der i-ten Iteration eine Kante aus K zu kontrahieren, höchstens

$$\frac{w(K)}{W \cdot \frac{n_i}{2}} = \frac{W}{W \cdot \frac{n_i}{2}} = \frac{1}{\frac{n_i}{2}} = \frac{2}{n_i}$$

beträgt. Demnach folgt

$$Prob\,[\text{in } i\text{-ter Iteration wird keine Kante aus } K \text{ kontrahiert} \mid E_i] \geq 1 - \frac{2}{n_i}\,.$$

Also

$$\begin{aligned}
& Prob\,[\text{„Randomisierte Kontraktion“ liefert } K] \\
= \; & \prod_{i=1}^{n-2} [\text{in } i\text{-ter Iteration wird keine Kante aus } K \text{ kontrahiert} \mid E_i] \\
\geq \; & \prod_{i=1}^{n-2} \left(1 - \frac{2}{n_i}\right) = \prod_{i=1}^{n-2} \left(1 - \frac{2}{n-i+1}\right) = \prod_{j=3}^{n} \frac{j-2}{j} \\
= \; & \frac{(n-2)!}{\frac{n!}{2}} = \frac{2}{n \cdot (n-1)} \\
\geq \; & \frac{2}{n^2}\,.
\end{aligned}$$

□

Wir betrachten nun die erwartete Anzahl $E(D)$ an Durchläufen des Algorithmus „Randomisierte Kontraktion“, bis man für Graphen auf n Knoten eine optimale Lösung erhält. Dies entspricht wieder unserem Algorithmus B, diesmal mit einem p, das größer oder gleich $2/n^2$ ist. Damit ist

$$E(D) \leq n^2/2.$$

Die erwartete Anzahl Durchläufe des Algorithmus ‚Randomisierte Kontraktion‘, um für Graphen auf n Knoten eine optimale Lösung des *MinCut-Problems* zu erhalten, ist maximal $O(n^2)$. Die entsprechend erwartete Laufzeit wäre $O(n^4)$.

5.13 Satz. *Wird der Algorithmus „Randomisierte Kontraktion“ N mal ausgeführt, dann ist die Wahrscheinlichkeit jedesmal einen Cut auszugeben, der ungleich einem minimalen Cut ist, höchstens*

$$\left(1 - \frac{2}{n^2}\right)^N \leq e^{-\frac{2 \cdot N}{n^2}}.$$

Beweis: Sei K die Kantenmenge, die zu einem minimalen Cut gehört. Dann gilt für einen Durchlauf:

$$Prob\,[\text{Alg. gibt einen Cut aus, der ungleich K ist}] \leq 1 - \frac{2}{n^2}.$$

Die Wahrscheinlichkeit $Prob$, den Algorithmus N mal auszuführen, und jedes mal einen Cut zu erhalten, der ungleich K ist, berechnet sich dann als Produkt der einzelnen Wahrscheinlichkeiten, da die Durchläufe unabhängig voneinander sind, also

$$Prob \leq \left(1 - \frac{2}{n^2}\right)^N \leq e^{-\frac{2 \cdot N}{n^2}},$$

wobei die letzte Ungleichung mit der Ungleichung $1 - x \leq e^{-x}$ für alle x folgt. □

- Für $N = c \cdot n^2 \cdot \ln n$ Durchläufe, für eine Konstante $c > 0$, etwa folgt:

$$\begin{aligned} Prob &\leq e^{-\frac{2 \cdot c \cdot n^2 \cdot \ln n}{n^2}} \\ &= e^{-2 \cdot c \cdot \ln n} \\ &= n^{-2 \cdot c}, \end{aligned}$$

  also ist die Wahrscheinlichkeit, einen minimalen Schnitt auszugeben, mindestens $1 - n^{-2 \cdot c}$, also fast 1, für große Werte von n. Damit findet man bei $N = \Theta\left(n^2 \cdot \ln n\right)$ Durchläufen fast sicher einen minimalen Cut.

- Die Laufzeit bei $N = \Theta\left(n^2 \cdot \log n\right)$ Wiederholungen des Algorithmus ist dann

$$O\left(n^4 \cdot \log n\right).$$

5.14 Lemma. *Würde der Algorithmus ‚Randomisierte Kontraktion' abbrechen, wenn genau t Knoten übrig sind, dann ist die Wahrscheinlichkeit Prob, dass keine Kante eines minimalen Cuts kontrahiert wird, mindestens*

$$\frac{t \cdot (t-1)}{n \cdot (n-1)} .$$

Beweis: Wie im Beweis von Satz 5.12 bestimmen wir die Wahrscheinlichkeit, dass während der ersten $n - t$ Iterationen keine Kante von K kontrahiert wird, zu:

$$\begin{aligned} \geq \prod_{i=1}^{n-t} \left(1 - \frac{2}{n_i}\right) &= \prod_{i=1}^{n-t} \left(1 - \frac{2}{n-i+1}\right) \\ &= \prod_{i=1}^{n-t} \frac{n-i-1}{n-i+1} \\ &= \frac{(n-2) \cdot (n-3) \cdots (t-1)}{n \cdot (n-1) \cdots (t+1)} \\ &= \frac{t \cdot (t-1)}{n \cdot (n-1)} . \end{aligned}$$

□

Nach dieser Analyse liegt der Schluss nahe, dass ein neuer Algorithmus das Problem besser lösen könnte, wenn das Verfahren ‚Randomisierte Kontraktion' früher abgebrochen wird. Der resultierende Algorithmus ‚Fast Cut' wird im folgenden Abschnitt behandelt.

5.5.2 Algorithmus ‚Fast Cut'

Eingabe: Ein zusammenhängender Graph $G = (V, E)$.
Ausgabe: Eine Partition $V = V_1 \dot\cup V_2$ bzw. die entsprechende Menge der Kanten $e \in E$ zwischen V_1 und V_2.

1. $n := |V|$
2. **if** $n \leq 6$ **then** berechne minimalen Cut C auf irgendeine triviale Weise. **return** C.
3. **else begin**
4. $t := \left\lceil 1 + \frac{n}{\sqrt{2}} \right\rceil$
5. Wende Algorithmus ‚Randomisierte Kontraktion' zweimal an, Abbruch jeweils bei t Knoten. Man erhält zwei Graphen H_1 und H_2 auf t Knoten.

   Wende rekursiv den Algorithmus ‚Fast Cut' auf H_1 und H_2 an und gib den kleineren der beiden berechneten Cuts aus.

6. **end.**

Der Abbruch bei $n \leq 6$ (Zeile 2) ist nötig, da dann $t \geq n$ wird, denn für $n = 6$ gilt zum Beispiel:

$$t = \left\lceil 1 + \frac{n}{\sqrt{2}} \right\rceil = \left\lceil 1 + \frac{6}{\sqrt{2}} \right\rceil = 6 .$$

5.15 Satz. *Der Algorithmus ‚Fast Cut' hat für Eingabegraphen auf n Knoten eine Laufzeit von $O\left(n^2 \cdot \log n\right)$. Er liefert dabei einen minimalen Schnitt mit einer Wahrscheinlichkeit von mindestens $\Omega(\frac{1}{\log n})$.*

Beweis:
Analyse der Laufzeit
Sei $T(n)$ die Laufzeit von ‚Fast Cut' auf Eingabegraphen mit n Knoten. Die Zeit für den Algorithmus ‚Randomisierte Kontraktion' kann durch $O(n^2)$ abgeschätzt werden, um H_1 und H_2 zu berechnen. Um unseren Beweis zu vereinfachen, nehmen wir an, dass die Zeit genau n^2 ist, dadurch verändern wir die Laufzeit höchstens um einen konstanten Faktor. Aus dem gleichen Grund können wir $T(6) = 1$ annehmen. Wir haben also:

$$T(n) = 2 \cdot T\left(\left\lceil 1 + \frac{n}{\sqrt{2}} \right\rceil\right) + n^2 \text{ sowie } T(6) = 1.$$

Wir wollen nun $T(n)$ sauber abschätzen, d.h. auch die Gaußklammer in der Analyse berücksichtigen. Die üblichen Annahmen, die einem sonst das Rechnen mit den Gaußklammern ersparen, wie zum Beispiel „Sei n eine Zweierpotenz" kann man hier nicht verwenden.
Dazu betrachten wir den Verlauf der Werte, d.h., wir definieren eine Folge $n_0 := n$ und $n_{i+1} = \lceil 1 + \frac{n_i}{\sqrt{2}} \rceil$.
Diese endet mit dem ersten n_i, das gleich 6 ist. Sei also die Folge $n_0, \ldots, n_r = 6$ gegeben. Mit dieser Folge erhalten wir:

$$T(n) = T(n_0) = 2 \cdot T(n_1) + n_0^2 = 4 \cdot T(n_2) + 2n_1^2 + n_0^2 = \ldots$$

Induktiv bekommt man also

$$T(n) = 2^r \cdot T(6) + \sum_{i=0}^{r-1} 2^i \cdot n_i^2.$$

Wir beobachten nun, dass nach Definition $\frac{n_{i+1}^2}{n_i^2} \geq 1/2$ ist, also

$$2^i \cdot n_i^2 \leq 2^{i+1} \cdot n_{i+1}^2 \leq \cdots \leq 2^r \cdot n_r^2 = 36 \cdot 2^r.$$

Damit erhalten wir:

$$T(n) \leq 2^r \cdot T(6) + r \cdot (2^r \cdot 36) = O(r \cdot 2^r).$$

Wir müssen nun also nur noch den Wert r abschätzen. Dazu schauen wir uns zunächst eine Abschätzung für die n_i an:

5.16 Lemma. *Sei* $q := \frac{1}{\sqrt{2}}$. *Dann gilt* $n_i \leq \dfrac{2}{1-q} + n \cdot q^i$ *für alle* $i \geq 0$. *Dabei ist* $\frac{2}{1-q} \approx 6.8284\ldots$

Beweis: Durch Induktion: Für $i = 0$ ist die linke Seite $n_0 = n$ und die rechte Seite ist $\frac{2}{1-q} + n$.
Im Induktionsschritt beobachten wir, dass nach Definition

$$n_{i+1} = \lceil 1 + n_i \cdot q \rceil \leq 2 + n_i \cdot q$$

ist und somit

$$n_{i+1} \leq 2 + \frac{2q}{1-q} + nq^{i+1} = \frac{2}{1-q} + n \cdot q^{i+1},$$

was zu zeigen war. □

Nun zur Abschätzung von r. Wenn für ein i die Schranke $\frac{2}{1-q} + n \cdot q^i$ kleiner als 7 ist, dann ist n_i kleiner als 7 und somit $i \geq r$.
Für welches i ist die Schranke kleiner als 7?

$$\frac{2}{1-q} + n \cdot q^i < 7$$
$$\Leftrightarrow$$
$$n \cdot q^i < 7 - \frac{2}{1-q}$$
$$\Leftrightarrow$$
$$\log n + i \cdot \log q < \log(7 - \frac{2}{1-q})$$

Da $\log q = -1/2$:

$$\Leftrightarrow$$
$$-\frac{i}{2} < \log(7 - \frac{2}{1-q}) - \log n$$
$$\Leftrightarrow$$
$$i > 2\log n - 2 \cdot \log(7 - \frac{2}{1-q})$$

Also: Wenn $i > 2\log n + 5.09$ ist, dann ist die Schranke kleiner als 7 und somit $i \geq r$.
Also:

$$r \leq 2\log n + 7.$$

Eingesetzt in die Abschätzung für $T(n)$ erhalten wir nun $T(n) = O(n^2 \log n)$.

Analyse der Erfolgswahrscheinlichkeit

Sei K ein minimaler Schnitt. ‚Fast Cut' liefert als Ausgabe einen minimalen Schnitt, wenn zum Beispiel gilt:

Bei der Kontraktion zu H_1 wird keine Kante aus K kontrahiert UND auf H_1 liefert ‚Fast Cut' einen minimalen Schnitt ODER Bei der Kontraktion zu H_2 wird keine Kante aus K kontrahiert UND auf H_2 liefert ‚Fast Cut' einen minimalen Schnitt.

Im folgenden sei $P(n)$ die Wahrscheinlichkeit, dass der Algorithmus ‚Fast Cut' auf einem Graphen mit n Knoten einen minimalen Schnitt findet.
Nach Lemma 5.14 existieren in H_i $(i = 1, 2)$ nach Wahl von t alle Kanten aus K mit einer Wahrscheinlichkeit p von

$$p \geq \frac{t \cdot (t-1)}{n \cdot (n-1)} \geq \frac{1}{2} .$$

Die Wahrscheinlichkeit, dass bei Kontraktion des Graphen G zu H_1 (bzw. H_2) keine Kante aus K kontrahiert wurde, ist also mindestens $1/2$. H_1 (bzw. H_2) haben daher mit Wahrscheinlichkeit mindestens $1/2$ immer noch einen Schnitt der Größe k.
Die Wahrscheinlichkeit, dass auf H_1 (bzw. H_2) ein minimaler Schnitt berechnet wird, ist mindestens $P(t)$, da H_1 (bzw. H_2) genau t Knoten hat. Die Wahrscheinlichkeit, dass H_1 (bzw. H_2) einen Schnitt der Größe k hat und dieser in der Rekursion auch berechnet wird, ist also mindestens $\frac{1}{2} \cdot P(t)$.
Damit ist die Wahrscheinlichkeit, dass weder auf dem Weg über H_1 noch auf dem Weg über H_2 ein minimaler Schnitt berechnet wird, höchstens

$$\left(1 - \frac{1}{2} \cdot P(t)\right)^2 .$$

Die Wahrscheinlichkeit, dass ‚Fast Cut' einen minimalen Schnitt liefert, ist also

$$P(n) \geq 1 - \left(1 - \frac{1}{2} \cdot P(t)\right)^2 .$$

Zur Berechnung von $P(n)$ führen wir folgende Transformation durch:
Transformation:
Sei r die Anzahl der Iterationen und $p(r)$ eine untere Schranke für die Erfolgswahrscheinlichkeit, einen minimalen Schnitt zu berechnen ($\rightarrow p(0) = 1$). Dann gilt für alle $i \leq r - 1$:

$$\begin{aligned} p(i+1) &\geq 1 - \left(1 - \frac{1}{2} \cdot p(i)\right)^2 \\ &= p(i) - \frac{p(i)^2}{4} . \end{aligned}$$

Die Funktion $f(x) := x - \frac{x^2}{4}$ ist eine nach unten offene Parabel, die symmetrisch um $x = 2$ liegt, daher ist f für $x \in [0, 1]$ eine monoton steigende Funktion.
Wir wollen zeigen:

$$p(i) \geq \frac{1}{d} \quad \Longrightarrow \quad p(i+1) \geq \frac{1}{d+1} \, .$$

Dies geschieht wie folgt:

$$p(i+1) \geq f(p(i)) \geq f(1/d).$$

(Letzte Ungleichung gilt wegen der Monotonie von f und wegen der Annahme, dass $p(i) \geq 1/d$.

$$f(1/d) = \frac{1}{d} - \frac{1}{4 \cdot d^2}.$$

Es ist

$$\begin{aligned} f(1/d) \geq \frac{1}{d+1} \quad &\Leftrightarrow \quad \frac{1}{d} - \frac{1}{4 \cdot d^2} \geq \frac{1}{d+1} \Leftrightarrow (d+1) - \frac{d+1}{4d} \geq d \\ &\Leftrightarrow \quad \frac{d+1}{4d} \leq 1 \Leftrightarrow d + 1 \leq 4d \\ &\Leftrightarrow \quad d \geq 1/3, \end{aligned}$$

was offensichtlich erfüllt ist.
Mit $p(0) = 1 \geq \frac{1}{1}$ folgt nun per Induktion $p(r) \geq \frac{1}{r+1}$.
Also ist $p(r) = \Omega\left(\frac{1}{r}\right)$. Da wir r bereits ausgerechnet haben, ergibt sich

$$P(n) = \Omega\left(\frac{1}{\log n}\right) \, .$$

□

Anmerkung für den Leser/die Leserin, die den Beweis sehr gut verstanden haben: Beim Algorithmus „Randomisierte Kontraktion“ haben wir für *jeden* minimalen Schnitt K die Wahrscheinlichkeit, dass K berechnet wird, als mindestens $2/n^2$ berechnet.
Beim Algorithmus „FastCut“ können wir *nicht* aussagen, dass für jeden minimalen Schnitt K gilt, dass er mit Wahrscheinlichkeit mindestens $\Omega(1/\log n)$ berechnet wird. Wir können nur aussagen, dass *ein* minimaler Schnitt mit Wahrscheinlichkeit mindestens $\Omega(1/\log n)$ berechnet wird. Die Leserin und der Leser mögen sich überlegen, warum das so ist. Ein Hinweis dazu: FastCut liefert von den beiden auf H_1 und H_2 berechneten Schnitten den mit kleinerem Wert zurück. Wenn H_1 den minimalen Schnitt K zurückliefert und H_2 einen anderen minimalen Schnitt K^*, dann hängt es von der Auswahlregel ab, ob K oder K^* zurückgeliefert wird.
Nun wollen wir kurz diskutieren, wie klein man den Fehler durch Wiederholung machen kann:

Wir wissen, dass $P(n) = \Omega(\frac{1}{\log n})$ ist. Nehmen wir an, dass die versteckte Konstante C ist, also

$$P(n) \geq \frac{C}{\log n}.$$

Wenn man den Algorithmus FastCut T mal wiederholt, ist die Wahrscheinlichkeit, jedes Mal keinen minimalen Schnitt zu bekommen, höchstens

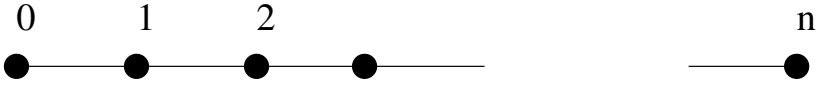
$$(1 - P(n))^T \leq e^{-P(n)\cdot T} \leq e^{-T\cdot\frac{C}{\log n}}.$$

Bei Wahl von $T := \frac{100}{C}\cdot\log n$ erhält man, dass diese Wahrscheinlichkeit höchstens $e^{-100} \approx 10^{-43}$, also zu vernachlässigen ist, wir werden also mit ziemlicher Sicherheit mindestens einmal einen minimalen Schnitt berechnen. (Anmerkung: Die Wahrscheinlichkeit, dass die Hardware, auf der unser Programm arbeitet, einen Fehler begeht, ist vermutlich schon größer als die Wahrscheinlichkeit 10^{-43}...)
Insgesamt haben wir also nun einen randomisierten Algorithmus, der (bei $T := \frac{100}{C} \cdot \log n$ Wiederholungen) Laufzeit $O(n^2 \log^2 n)$ hat und Fehlerwahrscheinlichkeit höchstens 10^{-43}. Verglichen mit der Laufzeit des deterministischen Algorithmus aus Kapitel 4, der Laufzeit $O(n\cdot e+n^2 \log n) = O(n^3)$ hatte, sind wir also asymptotisch wesentlich schneller.

5.6 Ein einfacher randomisierter 2SAT–Algorithmus

5.6.1 Ein random walk

Wir starten mit der Analyse eines einfachen „Spiels“. Gegeben haben wir eine Kette mit den Positionen 0 bis n:



Wir setzen ein Teilchen an Position i der Kette und starten ein Zufallsexperiment, einen so genannten „random walk“. Das Teilchen spaziert zufällig durch die Kette. Das Ganze läßt sich wie folgt formulieren:
$pos := i$
repeat
  if $pos = 0$ **then** STOP.
  if $pos = n$ **then** $pos := n - 1$.
    else $pos := pos - 1$ bzw. $pos := pos + 1$ (je mit Wahrscheinlichkeit 1/2.)
until false

Das Teilchen wird also an Position n „reflektiert“, in Position 0 stoppt es und an allen anderen Stellen geht es mit Wahrscheinlichkeit 1/2 nach links bzw. rechts.
Wir interessieren uns für die folgende Frage: Wie lange dauert es durchschnittlich, bis das Teilchen an Stelle 0 angekommen ist, wenn wir es an Position i starten? Genauer:

5.17 Definition. *Sei E_i die erwartete Anzahl an Schritten, die das Teilchen macht, wenn wir es an Position i starten.*

5.18 Satz. $E_i = i \cdot (2n - i)$.

Damit ist zum Beispiel $E_1 = 2n - 1$ bzw. $E_n = n^2$.
Beweis: Es gelten die folgenden 3 Aussagen:

$$E_0 = 0, E_n = E_{n-1} + 1$$

sowie

$$E_i = \frac{1}{2} \cdot E_{i-1} + \frac{1}{2} E_{i+1} + 1 \quad \text{für } i = 1, \ldots, n-1.$$

Die ersten beiden Aussagen sind klar, die dritte gilt, weil das Teilchen in Position $1 \leq i \leq n-1$ einen Schritt macht und dann mit Wahrscheinlichkeit $1/2$ in Position $i-1$ bzw. $i+1$ ist.[2] Wenn wir in der dritten Gleichung auf beiden Seiten $(1/2) \cdot E_i$ subtrahieren und umstellen, erhalten wir die Aussage:

$$\frac{1}{2}(E_i - E_{i-1}) = \frac{1}{2} \cdot (E_{i+1} - E_i) + 1 \quad \text{für } i = 1, \ldots, n-1.$$

Diese Gleichung können wir anders hinschreiben, wenn wir D_i für $i = 1, \ldots n$ definieren durch

$$D_i := E_i - E_{i-1},$$

denn dann ist $D_n = E_n - E_{n-1} = 1$ und:

$$D_i = D_{i+1} + 2.$$

Somit ist

$$D_n = 1, D_{n-1} = 3, D_{n-2} = 5, \ldots, D_1 = 2n - 1$$

Wie bekommen wir aus den D_i–Werten die uns eigentlich interessierenden E_i–Werte? Dazu beobachten wir:

$$D_1 + D_2 + \ldots + D_i = (E_1 - E_0) + (E_2 - E_1) + \cdots + (E_i - E_{i-1}) = E_i - E_0 = E_i.$$

Also ist

$$E_i = D_1 + \cdots + D_i = (2n-1) + (2n-3) + \cdots + \cdots \text{(genau } i \text{ Terme)}.$$

$$= 2n \cdot i - (1 + 3 + 5 + \cdots + 2i - 1) = 2n \cdot i - i^2 = i \cdot (2n - i).$$

Die vorletzte Gleichung gilt dabei, weil, wie man aus der Schule weiß, die Summe der ersten i ungeraden Zahlen den Wert i^2 ergibt. (Z.B. $1 + 3 + 5 + 7 + 9 = 25 = 5^2$.) $\square$

[2] Streng genommen müssten wir zunächst zeigen, dass die Erwartungswerte endlich sind, damit wir solche Gleichungen schreiben können, aber wir wollen uns an dieser Stelle diese kleine mathematische Nachlässigkeit gönnen.

5.6.2 Anwendung auf 2SAT

Das 2–SAT–Problem ist gegeben durch eine Liste $C_1, \ldots, C_m$ von reduzierten Klauseln der Länge 2. Gesucht ist eine Belegung a^* der Variablen, die alle Klauseln gleichzeitig erfüllt. Das Problem 2SAT ist deterministisch in Polynomialzeit lösbar. Wir wollen uns hier einen sehr einfachen randomisierten Algorithmus ansehen, der einen random walk durchführt mit Schrittanzahl $O(n^2)$ und der mit sehr großer Wahrscheinlichkeit eine erfüllende Belegung findet, wenn es denn eine gibt. (Auch hier kann man die Fehlerwahrscheinlichkeit kleiner wählen als die Wahrscheinlichkeit, dass wir wegen eines Hardwareproblems eine falsche Lösung bekommen.)
Wir nehmen zunächst an, dass wir eine Klauselmenge vorliegen haben, die auf jeden Fall von einer Belegung a^* erfüllt werden kann.

Algorithmus A
$a := (0, \ldots, 0)$
for $t := 1$ **to** ∞ **do**
begin
  if a erfüllt alle Klauseln **then STOP**
    else wähle eine Klausel C, die nicht erfüllt ist.
      Seien x_i und x_j die beiden Variablen,
      die in der Klausel (negiert oder unnegiert) vorkommen.
      Ändere a an der i–ten oder j–ten Stelle, und zwar genau
      eins von beiden mit Wahrscheinlichkeit 1/2.
end

Zur Analyse des Algorithmus benötigen wir die Definition der Hammingdistanz:

5.19 Definition. *Für* $a, b \in \{0,1\}^n$ *sei* $d(a,b) := |\ \{i \mid a_i \neq b_i\}|$.

Die Hammingdistanz von zwei Vektoren gibt also die Anzahl der Stellen an, in denen sie sich unterscheiden.
Wir analysieren, wie sich $d(a, a^*)$ im Verlaufe des Algorithmus verhält, wenn a^* eine beliebige, aber feste, erfüllende Belegung ist. Diese Hammingdistanz ist bei n Variablen eine Zahl zwischen 0 und n.
Der Algorithmus A stoppt spätestens bei $d(a, a^*) = 0$. Natürlich kann er auch vorher stoppen, wenn er eine andere erfüllende Belegung gefunden hat, aber wir wollen ja hier eine obere Schranke für die erwartete Anzahl an Schritten berechnen.
Wir sagen, dass der Algorithmus in Zustand i ist, wenn $d(a, a^*) = i$ ist und analysieren den Zustandsverlauf des Algorithmus.
Wenn a eine Belegung ist, die nicht alle Klauseln erfüllt, dann gibt es eine Klausel C, die von a nicht erfüllt ist. Wenn z.B. $C = x_1 \vee x_2$ ist, dann wäre $a_1 = 0$ und $a_2 = 0$. Für die erfüllende Belegung a^* muss aber $a_1^* = 1$ oder $a_2^* = 1$ gelten, da sonst die Klausel C nicht erfüllt wäre.
Da wir a an einer Stelle ändern, verändern wir die Hammingdistanz von a zu a^*. Entweder, wir erhöhen sie um 1, oder wir erniedrigen sie um 1. Mindestens eine der beiden

Änderungen, die wir an a vornehmen, verringert aber die Hammingdistanz zu a^*. Also: Mit einer Wahrscheinlichkeit $p \geq 1/2$ verringern wir den Zustand des Algorithmus um 1, mit einer Wahrscheinlichkeit $1 - p \leq 1/2$ erhöhen wir den Zustand um 1. Der schlechteste Fall für die Anzahl Schritte ist natürlich der, wo $p = 1/2$ ist.
Dann haben wir aber den Fall der Kette vorliegen, die wir analysiert haben. Der Erwartungswert, um aus Zustand i in Zustand 0 zu gelangen, ist für die analysierte Kette $i \cdot (2n - i) \leq n^2$.
Damit ist die erwartete Anzahl an Schritten, die Algorithmus A durchführt, bevor $a = a^*$ gilt, höchstens n^2.
Wir schauen uns nun Algorithmus B an, der sich von Algorithmus A nur dadurch unterscheidet, dass wir ∞ durch $2n^2$ ersetzen.
Die Wahrscheinlichkeit, dass Algorithmus A nach mehr als $2n^2$ Schritten eine erfüllende Belegung findet, wenn es denn eine gibt, ist höchstens $1/2$, denn ansonsten wäre die erwartete Schrittzahl größer als n^2. Also findet Algorithmus B mit Wahrscheinlichkeit $1/2$ eine erfüllende Belegung, wenn es eine gibt.
Wenn man Algorithmus B zum Beispiel 100mal iteriert, dann ist die Wahrscheinlichkeit, keine erfüllende Belegung zu finden, obwohl es eine gibt, höchstens $(1/2)^{100} \approx 8 \cdot 10^{-31}$, also ziemlich klein. Der Algorithmus sollte also dann als Ausgabe liefern: „Vermutlich gibt es keine erfüllende Belegung.“.
Moderne SAT–Algorithmen basieren fast alle auf solchen random walks.

5.7 Ein einfacher randomisierter 3-SAT-Algorithmus

Nun kommen wir zu einem 3-SAT-Algorithmus, der im Prinzip schon länger bekannt ist. Allerdings wurde erst im Jahre 1999 durch eine überraschende Analyse gezeigt, dass er eine erwartete Laufzeit von $O(1.3334^n)$ hat. Zuerst definieren wir den Begriff der Entropie: Für $0 \leq \alpha \leq 1$ sei

$$H(\alpha) := -\alpha \cdot \log \alpha - (1 - \alpha) \cdot \log(1 - \alpha).$$

Anmerkungen:

a) Eigentlich ist $\log 0$ nicht definiert, aber aus Stetigkeitsgründen definiert man $H(0) = H(1) = 0$.

b) Es gilt $2^{-H(\alpha)} = \alpha^\alpha \cdot (1 - \alpha)^{1-\alpha}$.

Wir werden die Entropie benutzen, um Binomialkoeffizienten geeignet abzuschätzen. Insbesondere verwenden wir die Abschätzung aus dem folgenden Satz, wobei wir bemerken wollen, dass es sogar genauere Abschätzungen gibt. Wir verzichten hier auf den Beweis dieses Satzes und begnügen uns mit dem Hinweis, dass dieser recht elementar zu führen ist.

5.20 Satz. *Für alle* $0 \leq k \leq n$ *gilt* $\frac{1}{n+1} \cdot 2^{H(k/n) \cdot n} \leq \binom{n}{k} \leq 2^{H(k/n) \cdot n}$.

Wir stellen nun den 3-SAT-Algorithmus RandomWalk vor. Sei a eine initiale Variablenbelegung.

Algorithmus `RandomWalk(`a`)`
DO $3n+1$ times {
    IF a erfüllend THEN STOP
    ELSE {
        Wähle eine Klausel C, die unter der Belegung a
        nicht erfüllt ist, o.B.d.A. sei $C = x_1 \vee x_2 \vee x_3$.
        Wähle $i \in \{1,2,3\}$ gemäß der Gleichverteilung und flippe a_i.
    }
}
output: Es gibt (vermutlich) keine erfüllende Belegung.

Wir beweisen zunächst folgende Aussage:

5.21 Satz. *Sei $a \in \{0,1\}^n$ und a^* eine erfüllende Belegung. Dann gilt:*

$$\textbf{Prob}\,(\textit{RandomWalk(a) findet eine erfüllende Belegung}) \geq \frac{1}{3n+1}\left(\frac{1}{2}\right)^{d(a,a^*)},$$

wenn $d(v,w)$ die Hammingdistanz zweier Vektoren v und w bezeichnet.

Beweis: Wir betrachten den Verlauf der Hammingdistanz $d(a,a^*)$. Mit der gleichen Begründung wie bei der Analyse des 2-SAT Algorithmus verringert sich diese in jedem Schritt mit einer Wahrscheinlichkeit von mindestens $1/3$. Mit der Sichtweise „Hammingdistanz um 1 verringern“ entspricht „Schritt nach links“ und „Hammingdistanz um 1 vergrößern“ entspricht „Schritt nach rechts“ erhalten wir, wenn wir auch $j := d(a,a^*)$ abkürzen:

$$\begin{gathered}\textbf{Prob}\,(\text{in } 3n \text{ Schritten wird eine erfüllende Belegung gefunden})\\ \geq \textbf{Prob}\,(\text{in } 3j \text{ Schritten wird eine erfüllende Belegung gefunden})\\ \geq\\ \textbf{Prob}\,(\text{bei } 3j \text{ Schritten werden mind. } 2j \text{ Schritte nach links und höchstens } j \text{ nach rechts gemacht})\end{gathered}$$

$$
\begin{aligned}
&\geq \binom{3j}{2j}\left(\frac{1}{3}\right)^{2j}\left(\frac{2}{3}\right)^{j} \\
&\geq \frac{1}{3j+1}2^{H(2/3)\cdot 3j}\left(\frac{1}{3}\right)^{2j}\left(\frac{2}{3}\right)^{j} \\
&= \frac{1}{3j+1}2^{H(2/3)\cdot 3j}\left[\left(\frac{1}{3}\right)^{1/3}\left(\frac{2}{3}\right)^{2/3}\right]^{3j}\left(\frac{1}{2}\right)^{j} \\
&= \frac{1}{3j+1}2^{H(2/3)\cdot 3j}\left[2^{-H(2/3)}\right]^{3j}\left(\frac{1}{2}\right)^{j} \\
&= \frac{1}{3j+1}\left(\frac{1}{2}\right)^{j} \\
&\geq \frac{1}{3n+1}\left(\frac{1}{2}\right)^{j}
\end{aligned}
$$

□

Man beachte übrigens, dass die erfüllende Belegung, die der Algorithmus in der Aussage des Satzes findet, nicht unbedingt die Belegung a^* sein muss.
Als Korollar erhalten wir übrigens aus dem Satz, dass der Algorithmus

$$\text{RandomWalk}(00\cdots 0);\ \text{RandomWalk}(11\cdots 1);$$

mit Wahrscheinlichkeit mindestens

$$\frac{1}{3n+1}\cdot\left(\frac{1}{2}\right)^{n/2} = \Omega(1.42^{-n})$$

eine erfüllende Belegung findet, da a^* entweder zum Einsvektor $11\cdots 1$ oder zum Nullvektor $00\cdots 0$ einen durch $n/2$ beschränkten Hammingabstand hat. Mit dem üblichen Argument bekäme man einen 3SAT-Algorithmus mit erwarteter Laufzeit $O(1.42^n)$. Wir gehen aber „intelligenter“ vor und würfeln das initiale a nach der Gleichverteilung aus. Der gesamte Algorithmus RW sieht wie folgt aus:

Algorithmus RW:
Wähle gemäß Gleichverteilung $a \in \{0,1\}^n$ und starte RandomWalk(a).

Zur Analyse von RW definiere zunächst die Zufallsvariable $X_i := d(a_i, a_i^*)$. Diese ist 1, wenn sich (das gewürfelte) a und a^* in der i-ten Stelle unterscheiden und 0 sonst. Man kann nun

$$d(a, a^*) = X_1 + \cdots + X_n$$

schreiben. Wie groß ist die Erfolgswahrscheinlichkeit P_{Erfolg}, dass Algorithmus RW eine erfüllende Belegung findet? Es ist

$$
\begin{aligned}
P_{\text{Erfolg}} &\geq \sum_{a\in\{0,1\}^n} \mathbf{Prob}\left(a \text{ gewürfelt}\right) \cdot \frac{1}{3n+1} \cdot \left(\frac{1}{2}\right)^{d(a,a^*)} \\
&= \frac{1}{3n+1} \cdot \sum_{a\in\{0,1\}^n} \mathbf{Prob}\left(a \text{ gewürfelt}\right) \cdot \left(\frac{1}{2}\right)^{d(a,a^*)} \\
&= \frac{1}{3n+1} \cdot \mathrm{E}\left[\left(\frac{1}{2}\right)^{d(a,a^*)}\right] \\
&= \frac{1}{3n+1} \cdot \mathrm{E}\left[\left(\frac{1}{2}\right)^{X_1+\cdots+X_n}\right].
\end{aligned}
$$

Schließlich gilt $\mathrm{E}\left[\left(\frac{1}{2}\right)^{X_1+\cdots+X_n}\right] = \mathrm{E}\left[\left(\frac{1}{2}\right)^{X_1}\right] \cdots \mathrm{E}\left[\left(\frac{1}{2}\right)^{X_n}\right]$, da $\mathrm{E}\left[X \cdot Y\right] = \mathrm{E}\left[X\right] \cdot \mathrm{E}\left[Y\right]$ für unabhängige Zufallsvariablen X und Y gilt. Hier sind die X_i und somit die $(\frac{1}{2})^{X_i}$ unabhängig.

Da $X_i \in \{0,1\}$, ist $\mathrm{E}\left[\left(\frac{1}{2}\right)^{X_i}\right] = (1/2) \cdot \left(\frac{1}{2}\right)^1 + (1/2) \cdot \left(\frac{1}{2}\right)^0 = 3/4$.
(Man beachte übrigens wieder einmal, dass *nicht* $\mathrm{E}\left[\left(\frac{1}{2}\right)^{X_i}\right] = (1/2)^{\mathrm{E}\left[X_i\right]}$ ist!)
Insgesamt haben wir erhalten:

$$P_{\text{Erfolg}} \geq \frac{1}{3n+1} \cdot (3/4)^n.$$

Mit dem üblichen Argument erhalten wir also einen randomisierten 3-SAT-Algorithmus mit erwarteter Laufzeit $(4/3)^n \cdot poly(n)$.
Nun kommen wir zu einer Verbesserung aus dem Jahre 2001/02. Bei der Initialisierung ignorieren wir ja bislang die Klauseln völlig. Falls beispielsweise $C = x_1 \vee x_2 \vee x_3$ ist, dann werden mit Wahrscheinlichkeit 1/8 alle drei Variablen auf Null gesetzt. Kann man das nicht vermeiden?

Die prinzipielle Idee besteht darin, die Variablen in Dreiergruppen (entsprechend den Klauseln) auszuwürfeln und dabei zu vermeiden, alle drei gleichzeitig auf Null zu setzen. Das Problem dabei ist aber, dass Klauseln Variablen gemeinsam haben können und es daher Überlappungen zwischen den Dreiergruppen gibt. Wir können aber versuchen, möglichst viele unabhängige Dreiergruppen zu finden. Dies führt zu folgender Definition:

5.22 Definition. *Zwei Klauseln heißen* unabhängig, *wenn sie keine Variablen gemeinsam haben.*

Beispiel: $x_1 \vee x_2 \vee \overline{x}_5$ und $x_3 \vee x_4 \vee x_5$ sind *nicht* unabhängig.

Wir starten unseren neuen Algorithmus zunächst damit, dass wir eine inklusionsmaximale Menge unabhängiger Klauseln berechnen. Wenn wir eine Menge M von Klauseln

als Eingabe für das 3-SAT-Problem gegeben haben, dann heißt eine Teilmenge $T \subseteq M$ der Klauseln maximal unabhängig, wenn je zwei von ihnen unabhängig sind und wir keine Klausel aus $M \setminus T$ zu T hinzufügen können, ohne diese Eigenschaft zu verletzen. Es sollte klar sein, dass man eine solche maximale Menge von unabhängigen Klauseln greedy berechnen kann. Sei im folgenden $\widehat{m}$ die Anzahl der unabhängigen Klauseln, die man gefunden hat. Nach geeigneter Umbenennung kann man davon ausgehen, dass diese $\widehat{m}$ Klauseln von der Form sind

$$\begin{aligned} C_1 &= x_1 \vee x_2 \vee x_3 \\ C_2 &= x_4 \vee x_5 \vee x_6 \\ &\vdots \\ C_{\widehat{m}} &= x_{3\widehat{m}-2} \vee x_{3\widehat{m}-1} \vee x_{3\widehat{m}}. \end{aligned}$$

Wir initialisieren die Variablen in den unabhängigen Klauseln (also $x_1, \ldots, x_{3\widehat{m}}$) anders als vorhin, mit Hilfe einer parametrisierten Prozedur namens Ind-Clauses-Assign. Die Parameter p_1, p_2, p_3 müssen dabei die Gleichung $3p_1 + 3p_2 + p_3 = 1$ erfüllen.

Ind-Clauses-Assign(p_1, p_2, p_3)
Für jede Klausel $C = x_i \vee x_j \vee x_k \in \{C_1, \ldots, C_{\widehat{m}}\}$ **do:**
Setze die drei Variablen mit Wahrscheinlichkeit:

p_1 auf $(0,0,1)$	p_2 auf $(0,1,1)$	p_3 auf $(1,1,1)$
p_1 auf $(0,1,0)$	p_2 auf $(1,1,0)$	
p_1 auf $(1,0,0)$	p_2 auf $(1,0,1)$	

Unser neuer Algorithmus, nennen wir ihn RW*, sieht wie folgt aus, die Parameter p_1, p_2, p_3 wählen wir dabei erst später:

Algorithmus RW$^*(p_1, p_2, p_3)$
Ind-Clauses-Assign(p_1, p_2, p_3)
Initialisiere die übrigen Variablen, also $x_{3\widehat{m}+1}, \ldots, x_n$, jeweils auf 0 bzw. 1, jeweils mit Wahrscheinlichkeit 1/2.
Nun sind alle Variablen gesetzt, wir haben also eine Belegung a erhalten.
Starte RandomWalk(a).

Welche Erfolgswahrscheinlichkeit hat RW$^*(p_1, p_2, p_3)$?
Wieder müssen wir $\mathrm{E}\left[(1/2)^{d(a,a^*)}\right]$ analysieren. Aber diesmal sind nicht alle Bits unabhängig gewählt. Daher definiere:

$$\begin{aligned} X_{1,2,3} &:= d(\ (a_1, a_2, a_3), (a_1^*, a_2^*, a_3^*)\ ), \\ X_{4,5,6} &:= d(\ (a_4, a_5, a_6), (a_4^*, a_5^*, a_6^*)\ ), \text{ etc.} \end{aligned}$$

Damit ist dann

$$d(a, a^*) = X_{1,2,3} + X_{4,5,6} + \cdots + X_{3\widehat{m}-2,3\widehat{m}-1,3\widehat{m}} + X_{3\widehat{m}+1} + X_{3\widehat{m}+2} + \cdots + X_n$$

und es ergibt sich:

$$P_{\text{Erfolg}} \geq \frac{1}{3n+1} \quad \cdot \quad \mathrm{E}\left[(\frac{1}{2})^{X_{1,2,3}}\right] \cdot \mathrm{E}\left[(\frac{1}{2})^{X_{4,5,6}}\right] \cdots \mathrm{E}\left[(\frac{1}{2})^{X_{3\widehat{m}-2,3\widehat{m}-1,3\widehat{m}}}\right] \cdot \prod_{i=3\widehat{m}+1}^{n} \mathrm{E}\left[(\frac{1}{2})^{X_i}\right].$$

Das hintere Produkt $\prod_{i=3\widehat{m}+1}^{n} \mathrm{E}\left[(\frac{1}{2})^{X_i}\right]$ können wir genauso analysieren, wie wir das vorhin getan haben, es ergibt sich, dass sein Wert $(3/4)^{n-3\widehat{m}}$ ist.
Bleibt die Aufgabe, $\mathrm{E}\left[(\frac{1}{2})^{X_{1,2,3}}\right]$ zu analysieren. (Die anderen Terme ergeben sich analog.) Dieser Erwartungswert hängt von a_1^*, a_2^*, a_3^* ab. Wir müssen drei Fälle betrachten, je nachdem, wieviele Einsen die drei a_1^*, a_2^*, a_3^* enthalten:

Fall $a_1^* + a_2^* + a_3^* = 3$.
Wir müssen alle sieben Möglichkeiten für die Belegung der a_1, a_2, a_3 durchgehen. Wenn wir z.B. $(a_1, a_2, a_3) = (0, 0, 1)$ wählen (was mit Wahrscheinlichkeit p_1 passiert), dann haben wir $X_{1,2,3} = 2$. Wenn wir $(a_1, a_2, a_3) = (0, 1, 1)$ wählen (was mit Wahrscheinlichkeit p_2 passiert), dann haben wir $X_{1,2,3} = 1$.
Gehen wir alle sieben Möglichkeiten durch, so erhalten wir:

$$\begin{aligned} \mathrm{E}\left[(\frac{1}{2})^{X_{1,2,3}}\right] &= (\frac{1}{2})^0 \cdot p_3 + (\frac{1}{2})^1 \cdot 3p_2 + (\frac{1}{2})^2 \cdot 3p_1 \\ &= p_3 + \frac{3}{2}p_2 + \frac{3}{4}p_1. \end{aligned}$$

Nun betrachten wir noch die anderen beiden Fälle:

Fall $a_1^* + a_2^* + a_3^* = 1$:

$$\mathrm{E}\left[(\frac{1}{2})^{X_{1,2,3}}\right] = (\frac{1}{2})^0 \cdot p_1 + (\frac{1}{2})^1 \cdot 2p_2 + (\frac{1}{2})^2 \cdot (2p_1 + p_3) + (\frac{1}{2})^3 \cdot p_2 = \frac{p_3}{4} + \frac{9}{8} \cdot p_2 + \frac{3}{2} \cdot p_1.$$

Fall $a_1^* + a_2^* + a_3^* = 2$:

$$\mathrm{E}\left[(\frac{1}{2})^{X_{1,2,3}}\right] = (\frac{1}{2})^0 \cdot p_2 + (\frac{1}{2})^1 \cdot (p_3 + 2p_1) + (\frac{1}{2})^2 \cdot 2p_2 + (\frac{1}{2})^3 \cdot p_1 = \frac{p_3}{2} + \frac{3}{2} \cdot p_2 + \frac{9}{8} \cdot p_1.$$

Wählen wir nun $p_1 := 4/21, p_2 := 2/21, p_3 := 3/21$, dann stellt sich heraus, dass der Erwartungswert $\mathrm{E}\left[(\frac{1}{2})^{X_{1,2,3}}\right]$ in allen drei Fällen gleich ist, nämlich $3/7$. Damit ergibt sich als Erfolgswahrscheinlichkeit für $\mathrm{RW}^*(4/21, 2/21, 3/21)$:

$$\frac{1}{3n+1} \cdot \left(\frac{3}{4}\right)^{n-3\widehat{m}} \cdot \left(\frac{3}{7}\right)^{\widehat{m}} = \left(\frac{3}{4}\right)^{n} \cdot \left(\frac{64}{63}\right)^{\widehat{m}} \cdot \frac{1}{3n+1}.$$

Diese Erfolgswahrscheinlichkeit wächst mit $\widehat{m}$, ist aber nur bei großen $\widehat{m}$ asymptotisch besser als beim alten Algorithmus!
Der Ausweg: Wir brauchen einen Algorithmus, der bei 3SAT-Instanzen mit *kleinem* $\widehat{m}$ erfolgreich ist. Den gibt es aber: Wenn man die Variablen $x_1, \ldots, x_{3\widehat{m}}$ belegt hat, dann bleibt nur noch eine 2SAT-Instanz übrig: Da die Klauselmenge *maximal* unabhängig war, enthält jede Klausel in der Eingabeinstanz eine der Variablen $x_1, \ldots, x_{3\widehat{m}}$. 2SAT kann jedoch in polynomieller Zeit gelöst werden. Dies führt zu folgendem Algorithmus RED2:

RED2 (q_1, q_2, q_3)
Ind-Clauses-Assign(q_1, q_2, q_3)
Löse 2SAT auf den restlichen Klauseln.

RED2$(1/7, 1/7, 1/7)$ ist z.B. dann erfolgreich, wenn die Belegungen $a_1, \ldots, a_{3\widehat{m}}$ alle mit $a_1^*, \ldots, a_{3\widehat{m}}^*$ übereinstimmen. Dies passiert mit Wahrscheinlichkeit $(1/7)^{\widehat{m}}$. Wir kombinieren beide Algorithmen:

MIX
RW$^*(\frac{4}{21}, \frac{2}{21}, \frac{3}{21})$.
RED2$(\frac{1}{7}, \frac{1}{7}, \frac{1}{7})$.

Nun ist die Erfolgswahrscheinlichkeit mindestens

$$\frac{1}{3n+1} \cdot \max\left[(\frac{1}{7})^{\widehat{m}}, \quad \left(\frac{3}{4}\right)^{n-3\widehat{m}} \cdot \left(\frac{3}{7}\right)^{\widehat{m}}\right].$$

Da die eine Funktion monoton wachsend in $\widehat{m}$ ist und die andere monoton fallend in $\widehat{m}$, erhalten wir eine untere Schranke, wenn wir das $\widehat{m}$ berechnen, wo beide den gleichen Wert ergeben: Es ergibt sich Gleichheit für $\widehat{m} \approx 0.1466525 \cdot n$. Dieses $\widehat{m}$ eingesetzt zeigt, dass die Erfolgswahrscheinlichkeit mindestens $P_{\text{Erfolg}} \geq 1.330258^{-n}$ ist.
Eine weitere Verbesserung auf die Erfolgswahrscheinlichkeit $P_{\text{Erfolg}} \geq 1.330193^{-n}$ kann man bekommen, wenn man die Wahl der Parameter p_1, p_2, p_3 noch optimiert, darauf wollen wir aber in der Vorlesung nicht eingehen.
Wir halten fest (vgl. unsere Vorüberlegungen vom Beginn dieses Abschnitts):

5.23 Satz. *Es gibt einen randomisierten Algorithmus, der das 3SAT-Problem in worst-case-Laufzeit $O(1.3302^n)$ löst. Genauer: auf Instanzen, die nicht erfüllbar sind, gibt er immer „nicht erfüllbar“ aus, und auf Instanzen, die erfüllbar sind, gibt er mit Wahrscheinlichkeit mindestens $1 - 2^{-n}$ eine erfüllende Belegung aus.*

Die Wahrscheinlichkeit $1 - 2^{-n}$ erhält man dabei mit den üblichen Amplifikationsargumenten.

5.8 Randomisierte Suchheuristiken

5.8.1 Motivation

Bei der Lösung von Optimierungsproblemen wünschen wir uns Algorithmen, die mit geringer worst case Rechenzeit stets optimale Lösungen liefern. Dieses weitgehende Ziel wird allerdings selten erreicht, ein positives Beispiel ist die Berechnung kürzester Wege, zwei weitere Problemklassen, die auf diese Weise effizient lösbar sind, werden wir in Kapitel 6 kennen lernen. Ansonsten haben wir bisher drei mögliche Auswege untersucht. Einerseits haben wir Algorithmen wie Branch-and-Bound-Algorithmen entworfen, die zwar die Berechnung einer optimalen Lösung garantieren, die aber bezüglich der Rechenzeit heuristisch sind. Wir können nur hoffen, dass sie auf vielen für uns interessanten Eingaben effizient sind. Mit Approximationsalgorithmen haben wir das Ziel der Berechnung optimaler Lösungen aufgegeben. Wir können gute Rechenzeiten garantieren und in vielen Fällen auch eine Schranke für die Approximationsgüte. Es lässt sich auch die Situation von garantierter Rechenzeit und nicht garantierter Approximationsgüte vorstellen. Dann hoffen wir, dass wir für viele für uns interessante Eingaben Lösungen berechnen, deren Wert nahe dem Wert optimaler Lösungen liegt. Für viele Approximationsalgorithmen erhalten wir für viele Eingaben Lösungen mit wesentlich größerer Qualität als sie durch die Abschätzung der worst case Güte garantiert wird.
Der dritte Ausweg ist Randomisierung. Die Idee besteht darin, dass es in vielen Fällen hilft, eine Entscheidung zufällig zu treffen, wenn es uns nicht möglich ist, eine gute Entscheidung zu berechnen. Randomisierung hilft, wenn es genügend viele gute Optionen gibt, eine gute Option durch gezielte Suche aber nicht effizient gefunden wird. Randomisierung hilft aber auch bei der Exploration schlecht strukturierter Suchräume oder von Suchräumen, über die wir wenig wissen.
Hier wollen wir randomisierte Suchheuristiken untersuchen, für die wir weder eine gute Rechenzeit noch eine gute Qualität der Lösung „garantieren“ können. Wir erwarten zumindest, dass unsere Lösungen „lokal“ optimal sind, wobei lokal „fair“ definiert ist. Wenn die Nachbarschaft jedes Suchpunktes der ganze Suchraum ist, fallen die Begriffe lokal optimal und global optimal zusammen. Dennoch führen auch vernünftige Lokalitätsbegriffe zu so großen Nachbarschaften, dass es nicht mehr effizient ist, die gesamte Nachbarschaft systematisch zu durchsuchen. Beim 3-Opting für das TSP sind die Nachbarn einer Tour alle Touren, die durch das Aufbrechen der Tour in drei Teilstücke und ein beliebiges Zusammenkleben der Teile entstehen können. Damit hat die Nachbarschaft eine Größe von $\Theta(n^3)$. In so einem Fall werden Nachbarn zufällig erzeugt. Ist der neue Suchpunkt mindestens so gut wie der alte, wird er aktueller Suchpunkt. Der Suchprozess wird gestoppt, wenn eine Zeitschranke überschritten wird, längere Zeit sich der Wert des aktuellen Suchpunktes nicht oder nur wenig verbessert hat oder ein anderes Kriterium greift.
Wir können einen derartigen Suchprozess an zufälligen Suchpunkten starten oder an von anderen Algorithmen berechneten vermutlich guten oder sogar garantiert ziemlich guten Suchpunkten. Die Güte der erzeugten Lösung kann durch eine Multistartvariante verbessert werden, bei der die randomisierte lokale Suche mehrfach unabhängig

voneinander durchgeführt wird.
Ein gravierender Nachteil der lokalen Suche ist, dass lokale Optima nicht verlassen werden können. Falls die betrachtete Funktion viele lokale Optima hat, von denen viele schlechte Werte haben, müssen wir es randomisierten Suchheuristiken erlauben, lokale Optima zu verlassen. Simulated-Annealing-Algorithmen akzeptieren mit im Laufe der Zeit fallender Wahrscheinlichkeit auch schlechtere Suchpunkte, während evolutionäre Algorithmen Suchoperatoren, also Operatoren zur Erzeugung neuer Suchpunkte, benutzen, bei denen jeder Suchpunkt gewählt werden kann, aber Punkte in größerer Nähe zum aktuellen Suchpunkt eine größere Wahrscheinlichkeit bekommen, gewählt zu werden. Darüber hinaus arbeiten sie mit Mengen aktueller Suchpunkte, die sich beeinflussen (Populationen). Dies erlaubt Suchoperatoren, die neue Suchpunkte aus mehreren aktuellen Suchpunkten konstruieren (Kreuzungen, Crossover). Wir wollen in diesem Kapitel am Beispiel evolutionärer Algorithmen untersuchen, ob und wie das Verhalten randomisierter Suchheuristiken analysiert werden kann.
Zunächst wollen wir zwei Szenarien für den Einsatz randomisierter Suchheuristiken unterscheiden. Wir können sie auf gut strukturierte Probleme anwenden, für die die bekannten Methoden versagen oder keine befriedigenden Lösungen berechnen. In diesem Fall sollte die Wahl der Suchoperatoren problemspezifisch sein. Es ist nicht sinnvoll, Strukturwissen über das Problem nicht einzusetzen. Hier gibt es eine Vielzahl problemspezifischer randomisierter Suchheuristiken, deren Analyse auch durch die problemspezifischen Besonderheiten kompliziert wird.
Daneben gibt es das Szenario der Black-Box-Optimierung. Dabei ist der endliche Suchraum S gegeben, die zu optimierende Funktion $f : S \to \mathbb{R}_0^+$ aber unbekannt. Wir können Wissen über f nur durch Stichproben (sampling) gewinnen. Eine randomisierte Suchheuristik kann den ersten Suchpunkt x_1 nach einer frei zu wählenden Verteilung zufällig bestimmen. Allgemein sind bei der Wahl des t-ten Suchpunktes x_t die ersten $t-1$ Suchpunkte $x_1, \ldots, x_{t-1}$ und ihre Funktionswerte $f(x_1), \ldots, f(x_{t-1})$ (oft Fitnesswerte genannt) bekannt und die Wahrscheinlichkeitsverteilung, nach der x_t gewählt wird, kann auf dieses Wissen zurückgreifen. Wieso ist dieses Szenario sinnvoll? In vielen Situationen der Praxis sind nicht genügend Ressourcen (Zeit, Geld, Know-how) vorhanden, um einen problemspezifischen Algorithmus zu entwickeln. Dann ist eine „robuste“ Suchheuristik gefragt, die auf vielen Eingaben vieler Probleme gut arbeitet. In dieser Situation darf man natürlich nicht hoffen, problemspezifische Algorithmen zu übertreffen. Es gibt auch Situationen, in denen die Einstellung freier Parameter eines technischen Systems zu optimieren ist. Diese Systeme sind so komplex, dass die Funktion f, die die Güte von Parametereinstellungen beschreibt, tatsächlich unbekannt ist. Dann lassen sich f-Werte nur experimentell (oder mit Hilfe von Simulationen) berechnen. In dieser Situation bilden randomisierte Suchheuristiken den einzigen Ausweg.
Wir werden randomisierte Suchheuristiken auf folgende Weise bewerten. Es sei X_f die Zufallsvariable, die das kleinste t beschreibt, so dass x_t optimal (oder fast optimal) ist. Beim Ablauf des Algorithmus können wir X_f nicht messen, da wir ja im Black-Box-Szenario nicht wissen, ob ein Suchpunkt optimal ist. Dennoch ist X_f aussagekräftig. Die meisten randomisierten Suchheuristiken benutzen Stoppkriterien, die den Suchprozess recht schnell stoppen, wenn ein optimaler Suchpunkt gefunden wurde. Allerdings

ist nicht ausgeschlossen, dass der Suchprozess „zu früh“ gestoppt wird. Wir sind an der erwarteten Optimierungszeit (oder Suchzeit) $E(X_f)$ ebenso interessiert wie an der Erfolgswahrscheinlichkeit $s(t) = \mathbf{Prob}(X_f \leq t)$. Wenn $s(p_1(n)) \geq 1/p_2(n)$ ist, hat eine Multistartstrategie mit $h(n)p_2(n)$ unabhängigen, gleichzeitigen Starts mit jeweils $p_1(n)$ Schritten eine Erfolgswahrscheinlichkeit von mindestens

$$1 - \left(1 - \frac{1}{p_2(n)}\right)^{h(n)p_2(n)} \approx 1 - e^{-h(n)}.$$

Exemplarisch wollen wir in diesem Kapitel einige Analysemethoden für randomisierte Suchheuristiken vorstellen. Im Mittelpunkt wird der einfachste evolutionäre Algorithmus stehen (Populationsgröße 1, nur Mutation), der so genannte $(1+1)$EA (aus einem Suchpunkt, dem Elter, wird ein neuer Suchpunkt, das Kind, erzeugt, aus beiden wird der bessere Suchpunkt als neuer aktueller Suchpunkt gewählt).

5.24 Algorithmus ($(1+1)$EA für die Maximierung von Funktionen $f : \{0,1\}^n \to \mathbb{R}$)**.**

1.) Wähle $x \in \{0,1\}^n$ gemäß der Gleichverteilung.

2.) Wiederhole den folgenden Mutationsschritt (bis ein Stoppkriterium greift):
Es sei $x' = (x'_1, \ldots, x'_n)$ mit

$$x'_i := \begin{cases} x_i & \text{mit Wahrscheinlichkeit } 1 - 1/n \\ 1 - x_i & \text{mit Wahrscheinlichkeit } 1/n \end{cases}$$

Jedes Bit flippt also mit Wahrscheinlichkeit $1/n$.

Es wird x genau dann durch x' ersetzt, wenn $f(x') \geq f(x)$ ist.

Die Wahl von $1/n$ als Mutationswahrscheinlichkeit ist die Standardwahl. Im Durchschnitt ändert sich ein Bit. Bei kleineren Werten passiert in vielen Schritten nichts und bei größeren Werten ist es schwierig, optimale Suchpunkte genau zu treffen. Allerdings gibt es Probleme, bei denen andere Werte (oder sogar wechselnde Werte) für die Mutationswahrscheinlichkeit besser sind. In Kapitel 5.8.2 beschreiben wir eine einfache Analysetechnik für den $(1+1)$EA und wenden sie auf mehrere Probleme an.

5.8.2 Eine obere Schranke für die erwartete Optimierungszeit des $(1+1)$EA

Im Folgenden nutzen wir aus, dass beim $(1+1)$EA der Fitnesswert des jeweils aktuellen Suchpunktes nie fällt.

5.25 Definition. *Für $A, B \subseteq \{0,1\}^n$ gilt $A <_f B$, wenn $f(x) < f(y)$ für alle $x \in A$ und $y \in B$ gilt. Eine f-basierte Partition von $\{0,1\}^n$ ist eine Partition $A_1, \ldots, A_m$ mit $A_1 <_f A_2 <_f \cdots <_f A_m$, so dass A_m genau die Suchpunkte mit maximalem f-Wert enthält. Es sei $s(a)$ für $a \in A_i, i < m$, die Wahrscheinlichkeit, dass aus a durch Mutation ein Suchpunkt $b \in A_j, j > i$, entsteht und $s(i) = \min\{s(x) \mid x \in A_i\}$.*

5.26 Lemma. *Für jede f-basierte Partition $A_1, \dots, A_m$ gilt*

$$E(X_f) \le 1 + \sum_{1 \le i \le m-1} p(A_i) \left(s(i)^{-1} + \cdots + s(m-1)^{-1} \right) \le 1 + s(1)^{-1} + \cdots + s(m-1)^{-1},$$

wobei $p(A_i)$ die Wahrscheinlichkeit ist, dass die Suche mit einem Suchpunkt aus A_i beginnt.

Beweis: Die zweite Ungleichung ist offensichtlich richtig. In der ersten Ungleichung steht der Summand 1 für den ersten Suchpunkt. Allgemein gilt nach dem Satz von der vollständigen Wahrscheinlichkeit

$$E(X_f) = \sum_{1 \le i \le m} p(A_i) E(X_f \mid x_1 \in A_i),$$

wobei x_1 den zufälligen ersten Suchpunkt bezeichnet. Wenn $x_1 \in A_i$ und $i < m$ ist, muss durch Mutation aus einem Suchpunkt aus A_i ein Suchpunkt aus $A_{i+1} \cup \cdots \cup A_m$ entstehen. Solange der aktuelle Suchpunkt in A_i ist, ist diese Erfolgswahrscheinlichkeit mindestens $s(i)$ und daher die erwartete Wartezeit durch $1/s(i)$ beschränkt. Wenn wir in A_i starten, müssen wir jede der Mengen $A_i, \dots, A_{m-1}$ maximal einmal verlassen. □

Diese sehr einfache Schranke ist in erstaunlich vielen Fällen sehr gut, obwohl wir so tun, als könnten wir nicht von A_i direkt nach A_j mit $j - i > 1$ springen. Allerdings muss dazu die f-basierte Partition geeignet gewählt werden. Wir beginnen mit zwei einfachen Anwendungen.

5.27 Satz. *Es sei $f(x) = x_1 + \cdots + x_n$ (oft ONEMAX genannt), dann gilt für den $(1+1)$EA: $E(X_f) \le 1 + e \cdot n(\ln n + 1)$. (Hierbei ist e die Eulersche Zahl $2,718\dots$)*

Beweis: Die Menge A_i soll alle x mit $f(x) = i$ enthalten. Dann ist $A_0, \dots, A_n$ eine f-basierte Partition. Es sei $x \in A_i$. Dann gibt es $n - i$ 1-Bit-Mutationen, die zu einem neuen Suchpunkt aus A_{i+1} führen. Jede 1-Bit-Mutation hat eine Wahrscheinlichkeit von $\frac{1}{n}\left(1 - \frac{1}{n}\right)^{n-1} \ge \frac{1}{en}$. Also gilt

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

und

$$E(X_f) \le 1 + en \sum_{0 \le i \le n-1} \frac{1}{n-i} \le 1 + en(\ln n + 1).$$

□

5.28 Definition. *Eine Funktion $f : \{0,1\}^n \to \mathbb{R}$ heißt unimodal, wenn es für jeden nicht optimalen Suchpunkt einen (bezüglich des Hammingabstands) benachbarten Suchpunkt mit größerem f-Wert gibt. (Meistens wird zusätzlich gefordert, dass es nur einen global optimalen Punkt gibt, aber wir können unser Ergebnis auch unter dieser abgeschwächten Forderung beweisen.)*

5.29 Satz. *Es sei f eine unimodale Funktion, die $I(f)$ verschiedene Funktionswerte annimmt. Dann gilt für den $(1+1)$EA: $E(X_f) \leq 1 + en(I(f)-1)$.*

Beweis: Wir betrachten die f-basierte Partition mit $I(f)$ Mengen, für die alle Suchpunkte in A_i denselben Funktionswert haben. Dann ist für alle A_i, die nicht die optimalen Suchpunkte enthalten, $s(i) \geq 1/(en)$, da es nach Definition mindestens eine 1-Bit-Mutation gibt, die den Funktionswert vergrößert. Nun folgt der Satz aus Lemma 5.26, da wir maximal $I(f)-1$ viele A_i-Mengen verlassen müssen. □

ONEMAX ist eine lineare Funktion, die unimodal ist. Aus Satz 5.29 erhalten wir aber nur die schlechtere Schranke $O(n^2)$, da wir nicht berücksichtigen, dass es für viele A_i viele gute 1-Bit-Mutationen gibt. Wir betrachten nun den allgemeinen Fall linearer Funktionen (eigentlich affiner Funktionen):

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

Wenn wir x_i durch $1 - x_i$ ersetzen, ändert sich die erwartete Optimierungszeit nicht, da der $(1+1)$EA Nullen und Einsen gleich behandelt. Daher können wir annehmen, dass alle Gewichte $w_i, i \geq 1$, nicht negativ sind. Außerdem können wir $w_0 = 0$ annehmen, da konstante additive Terme den $(1+1)$EA nicht beeinflussen. Nun ist es leicht zu sehen, dass lineare Funktionen unimodal sind. Allerdings kann $I(f)$ sehr groß sein, für $w_i = 2^{n-i}$ (x wird als Binärzahl bewertet, daher BV = binary value genannt) erhalten wir den größtmöglichen Wert 2^n. Aus Satz 5.29 folgt dann die wenig nützliche Schranke $O(n2^n)$. Wir können mit einer besseren f-basierten Partition eine wesentlich bessere Schranke zeigen.

5.30 Satz. *Für lineare Funktionen f und den $(1+1)$EA gilt $E(X_f) \leq 1 + en^2$.*

Beweis: Nach Umnummerierung der Variablen können wir annehmen, dass $w_1 \geq \cdots \geq w_n \geq 0$ ist. Außerdem können wir uns auf den Fall $w_0 = 0$ beschränken. Dann definieren wir für $i \in \{1, \ldots, n+1\}$ und $w_{n+1} := 1$

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

Für $a \in A_i$ gibt es ein $j \leq i$ mit $a_j = 0$. Da $w_j \geq w_i$ ist, erzeugt die 1-Bit-Mutation, die nur das j-te Bit flippt, einen Suchpunkt aus $A_{i+1} \cup \cdots \cup A_{n+1}$. Also ist $s(i) \geq 1/en$ und $E(X_f) \leq 1 + en^2$. □

Polynome mit durch k beschränktem Grad sind Summen von Termen $w x_{i_1} x_{i_2} \cdots x_{i_j}$ mit $j \leq k$.

5.31 Satz. *Für Polynome f mit durch k beschränktem Grad und N Termen, deren Gewichte alle positiv sind, und den $(1+1)$EA gilt $E(X_f) \leq 1 + eNn^k$.*

Beweis: Wir gehen wie im Beweis von Satz 5.30 vor und sortieren die Terme nach fallenden Gewichten $w_1 \geq \cdots \geq w_N > 0$. Die A_i-Mengen, $1 \leq i \leq N+1$, werden auf gleiche Weise gebildet. Für $a \in A_i$ gibt es ein $j \leq i$, so dass der zu w_j gehörige Term den Wert 0

hat. Es gibt dann eine m-Bit-Mutation, $m \leq k$, die genau die Nullen in dem betrachteten Term zu Einsen macht. Sie erhöht den Funktionswert, da sie diesen Term „aktiviert“. Da kein Gewicht negativ ist, liefert kein anderer Term weniger als zuvor. Die Wahrscheinlichkeit für die betrachtete Mutation beträgt $\left(\frac{1}{n}\right)^m \left(1-\frac{1}{n}\right)^{n-m} \geq \left(\frac{1}{n}\right)^m / e \geq 1/(en^k)$. Daher ist $1/s(i) \leq en^k$ und die Behauptung folgt. □

In den Beweisen von Satz 5.30 und 5.31 haben wir nicht berücksichtigt, dass wir unter Umständen mehr günstige Terme haben, deren Aktivierung dazu dient, A_i zu verlassen. In der Tat kann die Schranke in Satz 5.30 für lineare Funktionen auf $O(n \log n)$ gesenkt werden. Außerdem haben wir im Beweis von Satz 5.31 m durch k abgeschätzt und damit etwas verschenkt. Mit verbesserten Beweismethoden kann die Schranke aus Satz 5.31 für $k \leq \log n$ auf $O(Nn2^k/k)$ gesenkt werden.

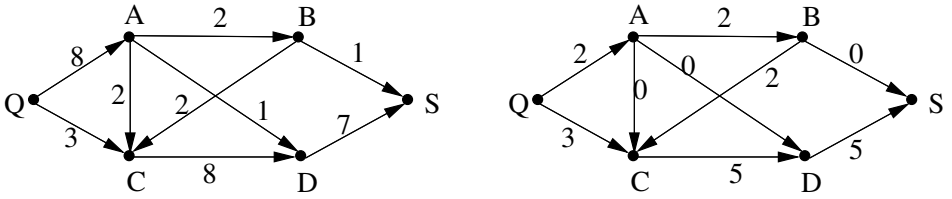
6 Maximierung von Flüssen in Netzwerken

6.1 Definitionen

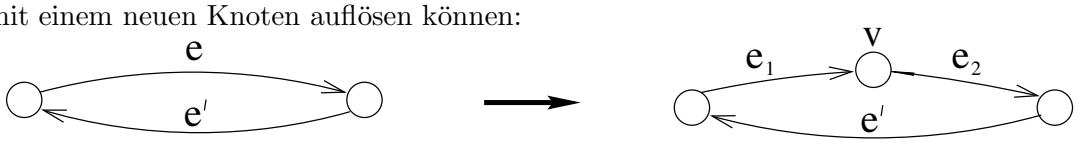
Der BVB muss in der Bundesliga gegen Werder Bremen antreten. Wie kann der Besucherstrom von Bremen nach Dortmund optimal gelenkt werden?

6.1 Definition. *Ein Netzwerk ist ein gerichteter, asymmetrischer*[3] *Graph $G = (V, E, c)$ mit zwei ausgewählten Knoten $Q, S \in V$ und einer Kapazitätsfunktion $c : E \to \mathbb{N}_0$. Die Quelle Q hat Eingangsgrad 0, und die Senke S hat Ausgangsgrad 0.*

Wir werden sinnvollerweise davon ausgehen, dass in den vorliegenden Netzwerken jeder Knoten $v \neq S$ mindestens eine ausgehende Kante und jeder Knoten $v \neq Q$ mindestens eine eingehende Kante hat. Dann ist die Kantenzahl $|E| \geq n - 1$.
Im linken Teil des folgenden Bildes sehen wir ein Beispiel für ein Netzwerk. Das rechte Bild zeigt bereits einen Fluss in diesem Netzwerk, diesen Begriff definieren wir aber erst ein paar Zeilen tiefer.



Anschaulich kann man sich die Knoten als Verkehrspunkte und die Kanten als Straßen vorstellen. Die Kapazität einer Straße misst den maximal möglichen Verkehrsstrom. Die Asymmetrie wird vorausgesetzt, da wir Kanten (x, y) manchmal auch als Rückwärtskanten (y, x) betrachten. Sie bedeutet keine echte Einschränkung, da wir jede Doppelkante mit einem neuen Knoten auflösen können:



$c(e_1) = c(e_2) = c(e)$. Den neuen Knoten v berühren keine weiteren Kanten.

6.2 Definition. *i) $\varphi : E \to \mathbb{R}_0^+$ heißt Fluss (im Netzwerk G), wenn die beiden folgenden Bedingungen erfüllt sind:*
a) Für alle $x \in V \setminus \{Q, S\}$ gilt die „Kirchhoff-Regel“

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

Diese besagt, dass in jeden Knoten (ausser der Quelle und der Senke) genausoviel hinein fließt wie aus ihm heraus fließt.

b) Für jede Kante e gilt, dass $\varphi(e) \leq c(e)$ ist. φ beachtet also die Kapazitätswerte.

ii) Der Fluss φ heißt ganzzahlig, wenn $\varphi(e) \in \mathbb{N}_0$ für alle $e \in E$ ist.

[3] Asymmetrie bedeutet, dass aus $(x, y) \in E$ folgt, dass $(y, x) \notin E$ ist.

iii) Der Wert *eines Flusses φ ist definiert als $w(\varphi) = \sum_{e=(Q,\cdot)} \varphi(e)$.*

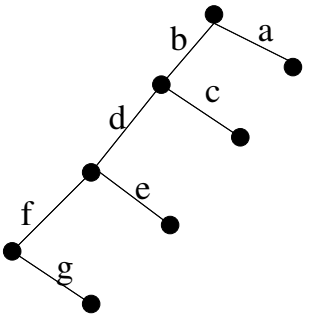
iv) Der Fluss φ heißt maximal, wenn $w(\varphi') \leq w(\varphi)$ für alle Flüsse φ' gilt.

Wegen der Kirchhoff-Regel ist $w(\varphi) = \sum_{e=(\cdot,S)} \varphi(e)$. Der Wert des Flusses misst also tatsächlich den Fluss von Q nach S. Ziel unserer Betrachtungen ist die effiziente Berechnung maximaler Flüsse. Bevor wir dies tun, betrachten wir zunächst eine Beispielanwendung für das Flussproblem.

6.2 Eine Anwendung: maximale Matchings in bipartiten Graphen

6.3 Definition. *Ein Matching M in einem ungerichteten Graphen $G = (V, E)$ ist eine Kantenmenge $M \subseteq E$, so dass je zwei Kanten aus M keinen Knoten gemeinsam haben.*

Gesucht ist ein maximales Matching, das heißt, ein Matching mit möglichst großer Kantenzahl.
Wir schauen uns zunächst das Verhalten eines Greedyalgorithmus für die Berechnung von Matchings an. Der Greedyalgorithmus startet mit einem leeren Matching M und wählt aus den Kanten, die noch zu M hinzugefügt werden können, so dass es ein Matching bleibt, eine aus, und fügt diese zu M hinzu. Das folgende Bild zeigt uns, dass dieser Greedyalgorithmus nicht immer ein maximales Matching berechnet:



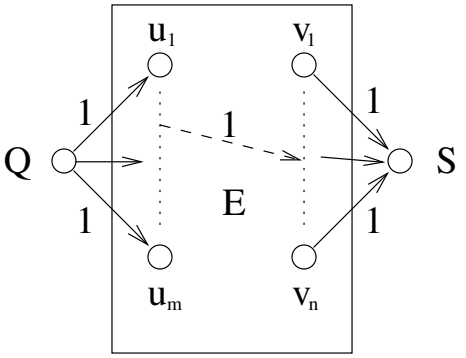
Der Greedyalgorithmus könnte zunächst die Kanten b und f auswählen. Dann kann keine weitere Kante gewählt werden. Hingegen wäre die Kantenmenge $\{a, c, e, g\}$ ein maximales Matching gewesen. Übrigens kann man über den Greedyalgorithmus wenigstens folgendes sagen:

6.4 Satz. *Der Greedyalgorithmus berechnet in einem Graphen $G = (V, E)$ stets ein Matching M_{greedy} mit $|M_{greedy}| \geq \frac{|M_{opt}|}{2}$, wenn M_{opt} ein maximales Matching ist.*

Beweis: Sei V_{greedy} die Menge der Knoten, die zu einer Kante aus M_{greedy} inzident sind. Natürlich ist $|V_{greedy}| = 2 \cdot |M_{greedy}|$. Jede Kante aus E ist zu mindestens einem Knoten aus V_{greedy} inzident. Wenn nämlich eine Kante e' zu keinem Knoten aus V_{greedy} inzident wäre, so hätte der Greedyalgorithmus diese Kante wählen können.
Also ist auch jede Kante aus M_{opt} zu mindestens einem Knoten aus V_{greedy} inzident. Also ist $|M_{opt}| \leq |V_{greedy}| = 2 \cdot |M_{greedy}|$. Die Aussage des Satzes ergibt sich somit. □

6.5 Definition. *Ein ungerichteter Graph $G = (V, E)$ heißt bipartit, wenn es eine disjunkte Zerlegung $V = V_1 \cup V_2$ gibt, so dass jede Kante $e \in E$ einen Knoten in V_1 und einen Knoten in V_2 hat.*

Der Leser und die Leserin mögen sich überzeugen, dass der Beispielgraph von gerade auch bipartit ist, also berechnet der Greedyalgorithmus noch nicht einmal auf bipartiten Graphen immer maximale Matchings.
Wieso betrachtet man überhaupt Matchings in bipartiten Graphen? Ein Matching ist eine Paarbildung. Wenn die Knoten eines Graphen Personen darstellen und Kanten Sympathie ausdrücken, besteht ein Matching aus Paaren, die sich mögen. Wir suchen nach maximalen Matchings, also Matchings mit maximaler Kantenzahl. Bipartite Graphen bilden eine wichtige Teilklasse. Wenn Sympathie durch „mögliche Heirat“ ersetzt wird und nur heterosexuelle Heiraten denkbar sind, erhalten wir bipartite Graphen. Ein maximales Matching entspricht der maximalen Anzahl von Ehepaaren, die sich mögen. „Wichtiger“ sind natürlich andere Zuordnungsprobleme, z. B. zwischen Personen und Tätigkeiten, die sie verrichten können.
Das Matchingproblem hat zahlreiche Anwendungen in Optimierungsproblemen. In der Praxis noch relevanter sind gewichtete Matchingprobleme, bei denen jede Kante ein Gewicht $w(e) \in \mathbb{R}_0^+$ trägt und nach Matchings M gesucht wird, für die die Summe aller $w(e)$ mit $e \in M$ maximal ist. Dieses Problem werden wir allerdings nicht untersuchen, da bereits die „einfachere“ Variante interessante Algorithmen zulässt.
Wir wollen zunächst maximale Matchings auf bipartiten Graphen mit Hilfe eines Flussalgorithmus berechnen. Für bipartite Graphen nehmen wir an, dass die Zerlegung der Knotenmenge vorgegeben ist: $G = (U \dot\cup V, E)$ und $|e \cap U| = |e \cap V| = 1$ für $e \in E$. Eine solche Zerlegung kann ansonsten leicht berechnet werden (Übungsaufgabe).
Zu $G = (U \dot\cup V, E)$ definieren wir ein Netzwerk $N(G)$. G legt fest, welche Kanten im umrandeten Teil (siehe Bild) des Netzwerks vorhanden sind.



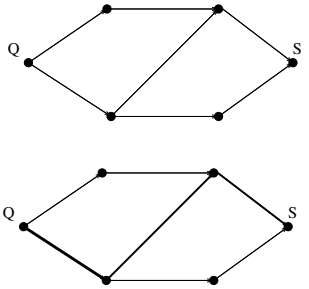
Das Netzwerk $N(G)$ hat die Knoten aus U und V und zusätzlich eine Quelle Q und eine Senke S. Es existieren die Kanten (Q, u) für $u \in U$, (v, S) für $v \in V$ und (u, v) für $\{u, v\} \in E$. Alle Kanten haben Kapazität 1.

6.6 Lemma. *Sei G ein ungerichteter bipartiter Graph und φ ein maximaler, ganzzahliger Fluss auf dem gerade konstruierten Netzwerk $N(G)$. Dann bilden die Kanten $\{u, v\}$ mit $\varphi(u, v) = 1$ ein maximales Matching M in G.*

Beweis: Falls $\varphi(u,v) = \varphi(u,v') = 1$ für $v \neq v'$ ist, so kann φ kein Fluss sein. Der Knoten u wird nur von der Kante (Q,u) erreicht. Also kann maximal 1 durch u fließen. Analog kann $\varphi(u,v) = \varphi(u',v) = 1$ für $u \neq u'$ widerlegt werden. Also ist M ein Matching. Es genügt nun zu zeigen, dass es zu jedem Matching M' einen ganzzahligen Fluss mit Wert $|M'|$ gibt. Sei $M' = \{\{u_1,v_1\},\ldots,\{u_k,v_k\}\}$ für $k = |M'|$. Dann sei $\varphi'(Q,u_i) = \varphi'(u_i,v_i) = \varphi'(v_i,S) = 1$ für $1 \leq i \leq k$ und $\varphi'(e) = 0$ für alle anderen Kanten. φ' ist ein ganzzahliger Fluss mit Wert $k = |M'|$. □

6.3 Der Restgraph

Der Algorithmus von Ford und Fulkerson wird versuchen, Wege zu finden, entlang derer der aktuelle Fluss vergrößert werden kann. Man könnte zunächst auf die Idee kommen, immer nur „vorwärts“ nach Kanten zu suchen, über die man noch mehr Fluss schicken kann. Diese einfache Idee scheitert jedoch, wie wir uns an folgendem Graphen klarmachen wollen:



Das obere Netzwerk soll auf allen Kanten Kapazität 1 haben, die Kanten sollen von links nach rechts gerichtet sein. In dem unteren Bild sehen wir einen Fluss φ in diesem Netzwerk (auf den dicker gezeichneten Kanten), der Wert 1 hat. Wenn man nun von der Quelle aus nur Kanten verfolgt, entlang derer man den Fluss noch erhöhen kann, so findet man keinen Weg zur Senke mehr. Dennoch ist dieser Fluss nicht maximal, denn der Fluss, der eine Einheit über den „oberen Weg“ schickt und eine Einheit über den „unteren Weg“, hat Wert 2.
Im Fluss φ hat man sich entschieden, eine Einheit 1 über die Querkante von links unten nach rechts oben zu schicken. Wenn man dies tut, kann man tatsächlich auch nur eine Einheit von Q nach S schicken. Man bräuchte also die Möglichkeit, die Entscheidung, über die Querkante eine Einheit zu schicken, rückgängig zu machen. Dies geschieht im Algorithmus von Ford und Fulkerson dadurch, dass man Kanten auch „rückwarts“ durchlaufen darf, also Fluss wieder zurücknehmen darf. Dieses motiviert die folgenden beiden Definitionen:

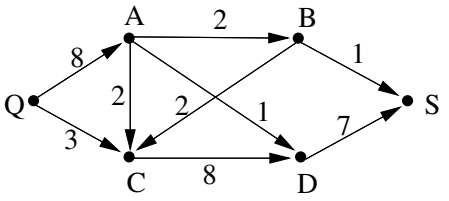
6.7 Definition. *Wenn $e = (x,y)$ eine Kante in einem gerichteten Graphen $G = (V,E)$ ist, dann bezeichnen wir mit $rev(e)$ die umgedrehte Kante, also $rev((x,y)) = (y,x)$.*

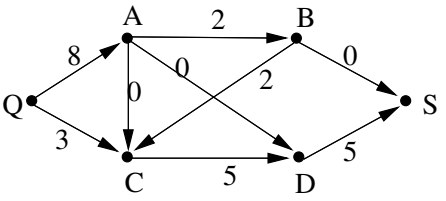
6.8 Definition. *(Restgraph)*
Gegeben ein Netzwerk $G = (V,E)$ und eine Abbildung $\varphi : E \to \mathbb{N}_0$. Der „Restgraph“

$Rest_\varphi = (V, E_\varphi)$ hat die gleiche Knotenmenge wie G und die folgenden Kanten:
Für jede Kante $e \in E$ mit $\varphi(e) < c(e)$ enthält E_φ die Kante e mit der Kapazität $r_\varphi(e) := c(e) - \varphi(e)$.
Für jede Kante $e \in E$ mit $\varphi(e) > 0$ enthält E_φ die Kante $e' = rev(e)$ mit der Kapazität $r_\varphi(e') := \varphi(e)$.

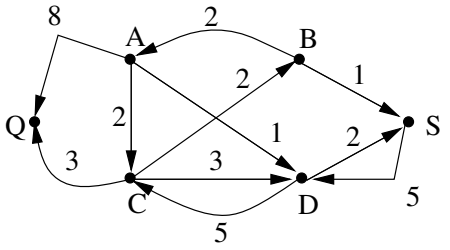
Hier wird auch klar, warum es günstig ist, sich auf asymmetrische Graphen zu beschränken. Wenn wir im Restgraphen eine Kante e sehen, wissen wir immer automatisch, ob sie aufgrund der ersten oder der zweiten Regel in den Restgraph aufgenommen worden ist.

Beispiel Restgraph: Links oben ein Netzwerk mit Kapazitäten an den Kanten. Rechts daneben eine Abbildung φ in diesem Netzwerk. (In den Knoten A fließen 8 Einheiten hinein und nur 2 hinaus, deswegen ist φ hier kein Fluss.) Darunter der zugehörige Restgraph.





Der Restgraph:



Wir wollen an dieser Stelle anmerken, dass es konzeptionell einfacher ist, über den Restgraphen zu reden und sich vorzustellen, dass die Algorithmen, die man entwirft, auf dem Restgraphen operieren. Bei tatsächlichen Implementierungen und sehr grossen Netzwerken wäre es allerdings eine Verschwendung, zu gegebenem Netzwerk und Fluss den Restgraphen explizit zu konstruieren. Man geht dann so vor, dass man im Speicher nur das eigentliche Netzwerk hält und z.B. das Vorhandensein einer Kante im Restgraphen immer dann aus dem Netzwerk und dem Fluss berechnet, wenn man diese Information benötigt.

6.9 Definition. *Ein Weg von der Quelle Q zur Senke S im Restgraphen heißt FV–Weg. (FV steht für Flussvergrößerung.)*

6.4 Der Algorithmus von Ford und Fulkerson

Da wir diesen Algorithmus später verfeinern, beschränken wir uns hier auf eine grobe Beschreibung. Es sollte klar sein, dass der Algorithmus zum Beispiel mit dem DFS–Ansatz in Zeit $O(n+e)$ durchgeführt werden kann.

6.10 Algorithmus. *Markiere zu Beginn nur den Knoten Q. Weitere Knoten können nur aufgrund der folgenden Regel markiert werden. Die Reihenfolge, in der die Regel angewendet wird, ist unerheblich.*
Regel: Wenn x markiert ist und es gibt im Restgraph eine Kante (x, y) zu einem unmarkierten Knoten y, dann markiere y mit dem Vermerk „wurde von x erreicht“.

6.11 Lemma. *Wenn Algorithmus 6.10 die Senke S markiert, dann ist φ nicht maximal. Ein Fluss φ' mit $w(\varphi') > w(\varphi)$ kann dann in $O(n)$ Schritten berechnet werden.*

Beweis: Wir starten an der Senke S und verfolgen zurück, über welchen Weg S von der Quelle aus erreicht worden ist. Insgesamt erhalten wir so einen Weg von Q nach S, sei dieser $a_0 = Q \to a_1 \to \cdots \to a_k = S$. Die Kante, die dabei den Knoten a_i verlässt, heiße e_i.
Wir berechnen δ als Minimum aller $r_\varphi(e_i)$ Nach Konstruktion ist $\delta > 0$. Nun erhöhen wir den Fluss entlang des gefundenen Weges. Genauer:
Falls $e_i \in E$ ist, so erhöhe den Fluss auf e_i um δ: $\varphi'(e_i) = \varphi(e_i) + \delta$. (Sprechweise: „$e_i$ wird vorwärts durchlaufen“.)
Falls $rev(e_i) \in E$ ist, so erniedrige den Fluss auf $rev(e_i)$ um δ: $\varphi'(rev(e_i)) = \varphi(rev(e_i)) - \delta$. (Sprechweise: „$rev(e_i)$ wird rückwärts durchlaufen“.)
Die Flusswerte auf den anderen Kanten lasse unverändert.
Nach Konstruktion ist $0 \le \varphi'(e) \le c(e)$ für alle $e \in E$. Wir überprüfen die Kirchhoff-Regel. Sie könnte an einem Knoten a_i mit $1 \le i \le k-1$ verletzt sein. Der Knoten a_i wird von e_{i-1} und e_i berührt. Dies sind die einzigen Kanten, die a_i berühren und einen anderen φ'-Wert als φ-Wert haben. Es gibt 4 Fälle abhängig davon, ob e_{i-1} bzw. e_i in E vorhanden sind bzw. ob ihre umgedrehte Version $rev(e_{i-1})$ bzw. $rev(e_i)$ in E vorhanden sind.

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

2.) Rück/Rück: $\times \xleftarrow{-\delta} \times \xleftarrow{-\delta} \times$. Der Fluss durch a_i verringert sich um δ.

3.) Vor/Rück: $\times \xrightarrow{+\delta} \times \xleftarrow{-\delta} \times$. Der Fluss durch a_i bleibt gleich groß.

4.) Rück/Vor: $\times \xleftarrow{-\delta} \times \xrightarrow{+\delta} \times$. Der Fluss durch a_i bleibt gleich groß.

In jedem Fall ist (Q, a_1) Vorwärtskante, und $w(\varphi') = w(\varphi) + \delta$. Die Rechenzeit ist $O(n)$. □

Der Algorithmus von Ford und Fulkerson startet mit $\varphi \equiv 0$, der stets ein Fluss ist. Mit Hilfe von Algorithmus 6.10 verbessern wir den jeweils vorliegenden Fluss solange, bis die Senke nicht mehr markiert wird. Alle φ während des Algorithmus sind Flüsse, und sie sind stets ganzzahlig, da alle $c(e)$ ganzzahlig sind. Es sei $B := \sum_{e=(Q,\cdot)} c(e)$. Dann gilt für jeden Fluss φ: $w(\varphi) \le B$, der Algorithmus von Ford und Fulkerson gerät

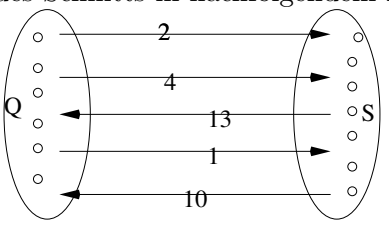
also nach spätestens B Schritten in die Situation, dass die Senke nicht markiert wird. Beim Beweis, dass dann ein maximaler Fluss vorliegt, wird der folgende berühmte Satz „MAXFLOW=MINCUT“ hilfreich sein. Wir definieren zunächst, was wir unter einem „Q–S–Schnitt“ verstehen.

6.12 Definition. *Ein Q–S–Schnitt ist eine disjunkte Zerlegung $V = V_Q \cup V_S$ der Knotenmenge mit $Q \in V_Q$ und $S \in V_S$. Der Wert eines Q–S–Schnittes ist definiert als*

$$wert(V_Q, V_S) = \sum_{e \in E \cap (V_Q \times V_S)} c(e),$$

also als Summe der Kapazitäten aller Kanten, die von V_Q nach V_S führen.

Hier ein Beispiel: Der Wert des Schnitts in nachfolgendem Bild beträgt $2 + 4 + 1 = 7$.



Verschiedene Q–S–Schnitte können verschiedene Werte haben. Ein Q–S–Schnitt heißt minimal, wenn es keinen Q–S-Schnitt gibt, dessen Wert kleiner ist. Wir kommen zum berühmten Satz:

6.13 Satz. (MAXFLOW=MINCUT)
Der Wert eines maximalen Flusses ist gleich dem Wert eines minimalen Q–S–Schnitts.

Beweis: Wir zeigen zunächst:
Sei φ ein Fluss und $V_Q \cup V_S$ ein Q–S–Schnitt. Dann gilt $w(\varphi) \leq wert(V_Q, V_S)$.
Diese Aussage ist intuitiv klar, da über die Kanten des Schnittes nicht mehr fließen kann als die Summe der entsprechenden Kapazitäten. Formal zeigen wir dies wie folgt:

$$w(\varphi) = \sum_{e=(Q,\cdot) \in E} \varphi(e) - \sum_{e=(\cdot,Q) \in E} \varphi(e).$$

(Die zweite Summe ist leer, also gleich Null.) Wegen der Kirchhoff-Regel gilt für alle $v \in V_Q \setminus \{Q\}$: $\sum_{e=(v,\cdot) \in E} \varphi(e) - \sum_{e=(\cdot,v) \in E} \varphi(e) = 0$, also ist („Addition von 0“):

$$w(\varphi) = \sum_{v \in V_Q} \left(\sum_{e=(v,\cdot) \in E} \varphi(e) - \sum_{e=(\cdot,v) \in E} \varphi(e) \right).$$

Kanten, die in V_Q verlaufen, tauchen einmal mit positivem und einmal mit negativem Vorzeichen in dieser Summe auf. Daher erhalten wir:

$$w(\varphi) = \sum_{e \in (V_Q \times V_S) \cap E} \varphi(e) - \sum_{e \in (V_S \times V_Q) \cap E} \varphi(e). \tag{2}$$

Da die erste Summe höchstens $\sum_{e\in(V_Q\times V_S)\cap E} c(e) = wert(V_Q, V_S)$ ist und die zweite größer oder gleich 0, ist $w(\varphi) \leq wert(V_Q, V_S)$.
Um zu zeigen, dass es einen Q–S–Schnitt gibt und einen Fluss, bei denen Gleichheit $w(\varphi) = wert(V_Q, V_S)$ gilt, betrachten wir den Markierungsalgorithmus 6.10 in der Situation, wo die Senke S nicht markiert wird. Wir wählen als V_Q die Menge der Knoten, die der Algorithmus markiert hat und als V_S die Menge der restlichen Knoten. Da die Markierungsregeln nicht mehr anwendbar sind, gilt

- $(x, y) \in E,\ x \in V_Q,\ y \notin V_Q \Rightarrow \varphi(x, y) = c(x, y)$.
- $(x, y) \in E,\ x \notin V_Q,\ y \in V_Q \Rightarrow \varphi(x, y) = 0$.

Somit gilt für den Fluss φ gemäß Gleichung (2):

$$w(\varphi) = \sum_{e\in(V_Q\times V_S)\in E} c(e) = wert(V_Q, V_S),$$

der Fluss ist also maximal. □

In dem letzten Beweis haben wir auch folgendes Lemma bewiesen.

6.14 Lemma. *Wenn Algorithmus 6.10 die Senke S nicht markiert, ist φ maximal.*

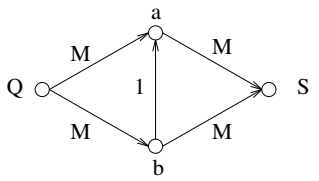
Wir haben mit der Einteilung $V = V_Q \cup V_S$ einen „Flaschenhals“ gefunden, der von φ maximal belastet wird. Kein anderer Fluss kann mehr durch diesen Flaschenhals schicken. Da der Algorithmus von Ford und Fulkerson nach maximal B Flussvergrößerungen die Senke nicht mehr markiert, wissen wir, dass folgender Satz gilt:

6.15 Satz. *Es sei B die Summe aller $c(e)$ mit $e = (Q, \cdot) \in E$. Maximale Flüsse können in $O((n+e) \cdot B)$ Schritten berechnet werden.*

Üblicherweise wird die Eingabelänge in Bits gemessen. Da mit b Bits Zahlen der Größe $2^b - 1$ dargestellt werden können, ist die Rechenzeit $(n+e)B$ nicht polynomiell in der Eingabelänge. Falls aber alle Kapazitäten klein sind, $c(e) \leq p(n)$ für ein Polynom p, ist die Rechenzeit des Algorithmus von Ford und Fulkerson polynomiell.

6.16 Definition. *Algorithmen, deren Laufzeit ein Polynom in der Eingabelänge und in der Größe der größten in der Eingabe vorkommenden Zahl ist, heißen pseudopolynomiell.*

Der Algorithmus von Ford und Fulkerson ist pseudopolynomiell, da $B \leq n \cdot \max\{\ c(e) \mid e \in E\}$ ist. Pseudopolynomielle Algorithmen sind für Eingaben, die nur kleine Zahlen enthalten, effizient.
Vielleicht haben wir nur die Rechenzeit unseres Algorithmus ungeschickt abgeschätzt, und der Algorithmus ist sogar polynomiell. Dies ist nicht der Fall, wie folgendes Beispiel zeigt.



6.17 Beispiel. *Wenn wir als FV-Wege abwechselnd die Wege* (Q, b, a, S) *und* (Q, a, b, S) *(mit Rückwärtskante* (a, b)*) wählen, so brauchen wir* $2M$ *Flussvergrößerungen, um den maximalen Fluss zu berechnen. Die Eingabe hat Länge* $\Theta(\log M)$*.*

Der Algorithmus von Ford und Fulkerson ist der Urvater von Generationen von Flussalgorithmen. Obwohl schon Mitte der 70er Jahre effiziente Algorithmen, die sich in der Praxis bewährt haben, zur Verfügung standen, wurde von vielen Forschern und Forscherinnen an weiteren Verbesserungen gearbeitet. Aus der Sicht der Praxis waren manche Fortschritte nur marginal, andere Verbesserungen waren asymptotische Verbesserungen, die bei Netzwerken üblicher Größe wegen der Verwendung komplizierter Datenstrukturen oder größerer Konstanten in der Rechenzeit gar zu schlechteren Ergebnissen führten. Dennoch sind alle Arbeiten von hohem Wert. Das Flussproblem hat sich als Problem erwiesen, an dem Methoden zum Entwurf effizienter Algorithmen entdeckt und neue Datenstrukturen entwickelt wurden. Darüber hinaus hat jeder neue Algorithmus neue Einsichten in die Struktur des Problems geliefert. Im folgenden werden die wichtigsten Arbeiten mit ihren Ergebnissen bis 1990 aufgelistet. Dabei ist n die Zahl der Knoten, e die Zahl der Kanten und $U = \max\{c(e) \mid e \in E\}$.
Algorithmen ohne Verbesserung der Laufzeit, die hier erwähnt werden, haben stets einen wesentlich neuen Ansatz gebracht.

Jahr	Autoren	Zeit gemessen in n, e, U	Zeit, wenn $e = \Omega(n^2)$
1969	Edmonds/Karp	$O(ne^2)$	$O(n^5)$
1970	Dinic	$O(n^2e)$	$O(n^4)$
1974	Karzanov	$O(n^3)$	$O(n^3)$
1977	Cherkasky	$O(n^2e^{1/2})$	$O(n^3)$
1978	Malhotra/Pramodh Kumar/ Maheshvari	$O(n^3)$	$O(n^3)$
1978	Galil	$O(n^{5/3}e^{2/3})$	$O(n^3)$
1978	Galil/Naamad sowie Shiloach	$O(ne\log^2 n)$	$O(n^3\log^2 n)$
1980	Sleator/Tarjan	$O(ne\log n)$	$O(n^3\log n)$
1982	Shiloach/Vishkin	$O(n^3)$	$O(n^3)$
1983	Gabow	$O(ne\log U)$	$O(n^3\log U)$
1984	Tarjan	$O(n^3)$	$O(n^3)$
1985	Goldberg	$O(n^3)$	$O(n^3)$
1986	Goldberg/Tarjan	$O(ne\log(n^2/e))$	$O(n^3)$
1986	Ahuja/Orlin	$O(ne + n^2\log U)$	$O(n^3 + n^2\log U)$
1989	Ahuja/Orlin/Tarjan	$O(ne + n^2\log U/\log\log U)$ $O(ne + n^2\log^{1/2} U)$ $O(ne\log(\frac{n}{e}\log^{1/2} U + 2))$	
1989	Cheriyan/Hagerup (rand.)	$O(ne + n^2\log^3 n)$	
	det. Version von Alon	$O(\min(ne\log n, ne + n^{8/3}\log n))$	
	det. Version von Tarjan	$O(\min(ne\log n, ne + n^2\log^2 n))$	
1990	Cheriyan/Hagerup/Mehlhorn	$O(n^3/\log n)$	
	rand.	$O(\min(ne\log n, ne + n^2\log^2 n, n^3/\log n))$	

Das Problem für jeden Neuling auf diesem Gebiet besteht darin, dass es nicht ausreicht, eine der neueren Arbeiten zu lesen. Vieles baut aufeinander auf. Es ist nötig, einen großen Teil der Arbeiten zu lesen, um die neuesten Arbeiten verstehen und würdigen zu können. In einigen Jahren wird das Gebiet vielleicht so gut aufgearbeitet sein, dass es einen einfacheren Zugang gibt. Wir wollen dennoch versuchen, einige Meilensteine in der Entwicklung nachzuvollziehen.

6.5 Exkurs Distanzen in Graphen

Gegeben ein gerichteter Graph $G = (V, E)$ sowie ein ausgezeichneter Knoten $Q \in V$. Wir bezeichnen mit $d(v)$ die Länge des kürzesten Weges in G von Q nach v, wobei die Länge eines Weges hier die Anzahl der Kanten auf dem Weg bezeichnet. Wir stellen uns die Frage, wie sich diese Distanzen verändern, wenn wir Kanten zum Graphen hinzufügen bzw. Kanten entfernen.

6.18 Lemma. *Gegeben seien zwei Knoten v und w.*

a) Wenn $d(v) \geq d(w) - 1$ ist, dann bleiben durch Hinzufügen der Kante $v \to w$ alle Distanzen unverändert.

b) Durch Wegnahme einer Kante $v \to w$ bleiben die Distanzen all derjenigen Knoten v' unverändert, für die $d(v') \leq d(w) - 1$ galt und alle anderen Distanzen erhöhen sich höchstens.

Beweis: G' bezeichne den Graph, den wir aus G nach der Veränderung bekommen und d' bezeichne die Distanzen in G'.
a): Das Hinzufügen von Kanten kann prinzipiell Distanzen kleiner machen. Wir zeigen, dass dies hier nicht passieren kann. Als erstes beobachten wir $d'(v) = d(v)$, da kein kürzester Weg von Q nach v in G' die Kante $v \to w$ benutzt. Außerdem ist $d'(w) = d(w)$, da für jeden Weg von Q nach w in G', der die Kante $v \to w$ benutzt, die Länge mindestens $d'(v) + 1 = d(v) + 1 \geq d(w)$ ist. Sei nun v' ein Knoten, der weder der Knoten v noch der Knoten w ist. Angenommen, in G' gibt es einen Weg von Q nach v', der die Kante $v \to w$ benutzt und Länge l hat. Der Teilweg von Q nach w hat Länge mindestens $d(w)$. Wir ersetzen diesen Teilweg durch den Weg der Länge $d(w)$, den es auch in G schon gibt. Damit haben wir auch in G einen Weg der Länge höchstens l, also ist $d(v') = d'(v')$.
b): Durch das Entfernen von Kanten können Distanzen nur größer werden. Ein kürzester Weg, der die Kante $v \to w$ benutzt, hat Länge mindestens $d(w)$. Damit sind die kürzesten Wege zu allen v' mit $d(v') \leq d(w) - 1$ im Graphen G' immer noch vorhanden. □

6.6 Der Algorithmus von Dinic

Wenn wir auf Beispiel 6.17 zurückblicken, dann fällt auf, dass die hohe Zahl von Flussvergrößerungen nicht nötig ist. Wenn wir die „richtigen“ FV–Wege wählen, nämlich (Q, a, S) und dann (Q, b, S), genügen 2 FVs zur Berechnung des maximalen Flusses. Man kann also bei der Wahl der FV-Wege Glück haben. Wir werden nun das Glück steuern und nur solche Flussvergrößerungen betrachten, die über kürzeste FV-Wege laufen.
Der Algorithmus von Dinic, den wir nun vorstellen, behandelt implizit alle kürzesten FV–Wege gleichzeitig.
Der Algorithmus ermittelt zunächst mit Hilfe eines BFS-Durchlaufs im Restgraphen zu jedem Knoten v des Netzwerks dessen Distanz $d(v)$ von der Quelle. Knoten $v \neq S$, die eine Distanz größer oder gleich $d(S)$ von der Quelle haben, können nicht auf kürzesten FV-Wegen von der Quelle zur Senke liegen und werden von der Betrachtung ausgeschlossen. Wir definieren „Niveaus“, die aus Knoten bestehen, wie folgt:

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

sowie $V_i = \{S\}$ für $i = d(S)$.
Wir definieren nun ein sogenanntes Niveaunetzwerk zu einem gegebenen Fluss φ. Dieses enthält nur noch die Kanten aus dem Restgraph, die zwischen benachbarten Niveaus verlaufen, genauer:

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

Wie schon vorhin beim Restgraph soll auch hier $r_\varphi(e)$ den Spielraum auf der Kante e angeben.
Außerdem sei E_φ die Vereinigung aller E_i. Der Markierungsalgorithmus hat gezeigt:

6.19 Bemerkung. *Zu einem Netzwerk G und Fluss φ kann (mit einem BFS-Algorithmus) das zugehörige Niveaunetzwerk in Zeit $O(n+e)$ berechnet werden.*

Wir benötigen folgende Definition:

6.20 Definition. *Sei ψ ein Fluss im Niveaunetzwerk. Eine Kante e heißt saturiert durch ψ, wenn $\psi(e) = r_\varphi(e)$ ist. Der Fluss ψ heißt Sperrfluss, wenn auf jedem Q–S–Weg im Niveaunetzwerk mindestens eine saturierte Kante liegt.*

Wir werden nun zeigen, dass Sperrflüsse effizient berechenbar sind. Wir können den Sperrfluss ψ dann zu φ „addieren“, wobei der Fluss $\psi(e)$ für Rückwärtskanten subtrahiert wird.
Danach zeigen wir, dass Sperrflüsse das gewünschte Verhalten haben, d. h. die Länge kürzester FV-Wege von Q nach S ist nach Addition des Sperrflusses um mindestens 1 gewachsen.

6.21 Algorithmus. *Konstruktion eines Sperrflusses ψ für ein Niveaunetzwerk.*
Methode: Backtracking. Der Algorithmus hat stets einen aktuellen Weg

$$Q = v_0 \to v_1 \to \cdots \to v_i$$

vorliegen und versucht, diesen zur Senke zu erweitern.

0.) $\psi(e) := 0$ *für alle Kanten e im Niveaunetzwerk.*

1.) Der aktuelle Weg ist der Weg der Länge 0 von Q nach Q, d. h. $i = 0$ und $v_0 = Q$.

2.) Falls $v_i = S$, gehe zu Schritt 3.
Ansonsten durchlaufe im Niveaunetzwerk die Adjazenzliste von v_i. Wenn die Adjazenzliste als nächstes die Kante $e = (v_i, w)$ enthält, so verlängere den Pfad um diese Kante, d. h. $i := i + 1; v_i = w$. Gehe zu Schritt 2.
Wenn die Adjazenzliste leer geworden ist, dann kann S von v_i aus im Niveaunetzwerk nicht mehr erreicht werden. Entferne v_i aus dem Niveaunetzwerk sowie die zu v_i inzidenten Kanten.
if $i = 0$ **then** *STOP* **else** $i := i - 1$ *und gehe zu Schritt 2.*

3.) Sei W der gefundene Weg von Q nach S und $\delta := \min\{r_\varphi(e) - \psi(e) \mid e \in W\}$. Ersetze für alle $e \in W$ den Wert des Sperrflusses $\psi(e)$ durch $\psi(e)+\delta$. Entferne alle nun saturierten Kanten (es gibt mindestens eine) und mache in Schritt 1 weiter.

6.22 Satz. *Algorithmus 6.21 konstruiert in Zeit $O(n \cdot e)$ einen Sperrfluss ψ.*

Beweis: Der Algorithmus terminiert nur, wenn es keinen Q-S-Weg mehr gibt. Kanten werden nur entfernt, wenn sie saturiert sind oder von einem ihrer inzidenten Knoten klar ist, dass S nicht mehr erreicht werden kann. Also ist am Ende des Algorithmus ψ ein Sperrfluss.
Nun zur Laufzeit: Jeder Versuch, einen Q–S-Weg zu konstruieren, endet mit der Entfernung einer Kante, also gibt es höchstens e Versuche. Q–S-Wege im Niveaunetzwerk haben Länge höchstens $n-1$. Daher ist nach höchstens n Schritten ein Versuch erfolgreich oder erfolglos. Somit ist die Rechenzeit durch $O(e \cdot n)$ beschränkt. □

6.23 Algorithmus. *Konstruktion eines maximalen Flusses φ.*

0.) Setze $\varphi(e) := 0$ für alle $e \in E$.

1.) Berechne das zum Fluss φ gehörige Niveaunetzwerk. Wenn dabei festgestellt wird, dass die Senke überhaupt nicht mehr von der Quelle erreicht werden kann, so stoppt der Algorithmus. Der Fluss φ ist dann maximal.

2.) Berechne einen Sperrfluss ψ für das Niveaunetzwerk.

*3.) „Addiere den Sperrfluss ψ zum aktuellen Fluss φ." Bei dieser Addition muss natürlich wieder die Richtung der Kante(n) berücksichtigt werden.
Gehe zu Schritt 1.*

6.24 Satz. *Mit Algorithmus 6.23 können maximale Flüsse in Zeit $O(n^2e) = O(n^4)$ berechnet werden.*

Beweis: Nach allen Vorüberlegungen ist der Algorithmus korrekt. Die Kosten für die einzelnen Schritte betragen $O(e)$ für Schritt 0, $O(n+e)$ für Schritt 1 (Bemerkung 6.19), $O(ne)$ für Schritt 2 (Satz 6.22) und $O(e)$ für Schritt 3. Es genügt also zu zeigen, dass stets $n-1$ Iterationen genügen.
Zu diesem Zweck schauen wir uns Wege im Restgraphen an. Wir sind gestartet mit dem Fluss φ. Dazu haben wir das Niveaunetzwerk konstruiert, in dem die Senke Entfernung ℓ von der Quelle hatte. Wir haben zu φ einen Sperrfluss ψ „addiert", um den Fluss φ' zu bekommen.
Wir vergleichen die beiden Restgraphen $Rest_\varphi$ und $Rest_{\varphi'}$. Kanten im Graphen $Rest_\varphi$, die in einem Knoten starten, der die Entfernung x von der Quelle hat, führen zu Knoten, die Entfernung $\leq x+1$ von der Quelle haben. Welche Kanten enthält $Rest_{\varphi'}$ neu? Da wir den Fluss im Restgraphen nur entlang von Kanten erhöhen, die für irgendein j von Niveau j zu Niveau $j+1$ verlaufen, enthält $Rest_{\varphi'}$ nur solche Kanten neu, die für irgendein j von Niveau $j+1$ zu Niveau j verlaufen. Damit sollte klar sein, dass Wege von der Quelle zur Senke in $Rest_{\varphi'}$, die mindestens eine neue Kante benutzen, Länge mindestens $\ell+2$ besitzen.
Wege, die keine neue Kante benutzen und Länge genau ℓ besitzen, sind durch den Sperrfluss zerstört worden. Also ist die Entfernung im Restgraphen $Rest_{\varphi'}$ von der Quelle zur Senke mindestens $\ell+1$. □

6.7 Sperrflussberechnung in Laufzeit $O(n^2)$ – der Algorithmus von Malhotra, Kumar, Maheshwari

Gegeben sei das Niveaunetzwerk zum Restgraphen $Rest_\varphi$ mit den Niveaus $V_0 = \{Q\}, \ldots, V_r = \{S\}$. Die Kanten sind mit $r_\varphi(e)$ beschriftet.
Wir wollen wieder einen Sperrfluss konstruieren und starten mit dem Nullfluss $\psi \equiv 0$ in diesem Niveaunetzwerk. Diesen Fluss ψ erhöhen wir nach und nach, bis wir einen Sperrfluss erhalten. Für einen Knoten v im Niveaunetzwerk und den aktuellen Fluss ψ definieren wir einen Potenzialwert $pot(v)$. Für $v \notin \{Q, S\}$ ist das Potenzial $pot(v)$ definiert als:

$$pot(v) := \min \left\{ \sum_{e=(v,\cdot)} r_\varphi(e) - \psi(e), \sum_{e=(\cdot,v)} r_\varphi(e) - \psi(e) \right\}.$$

Außerdem ist $pot(Q) := \sum_{e=(Q,\cdot)} r_\varphi(e) - \psi(e)$ und $pot(S) := \sum_{e=(\cdot,S)} r_\varphi(e) - \psi(e)$.
Die Zahl $pot(v)$ gibt anschaulich den Wert an, um den man den Fluss durch den Knoten v maximal noch erhöhen kann.
Der folgende Algorithmus geht nun so vor, dass er von allen Knoten den minimalen *pot*–Wert berechnet und von dem zugehörigen Knoten aus den Fluss erhöht. Dabei schiebt man den Fluss vor sich her. Im Array Überschuss[v] merkt man sich, wieviel Fluss der Knoten v noch weiterreichen muss. Man benötigt zwei Prozeduren `Forward(v)` und `Backward(v)`, von denen wir nur die eine beschreiben, weil die andere „spiegelsymmetrisch“ dazu ist.

`Forward(v)`
Überschuss[v] = $pot(v)$ und für alle $w \neq v$ sei Überschuss[w] = 0.
Initialisiere eine Queue mit dem Knoten v.
while queue not empty **do**
begin
  Entferne das oberste Element aus der Queue und nenne es v.
  Sei e die erste von v ausgehende Kante im Niveaunetzwerk.
  while Überschuss[v] > 0 **do**
  begin
    Sei $e = (v, w)$. Sei $\delta = \min\{r_\varphi(v, w) - \psi(v, w), \text{Überschuss}[v]\}$.
    Erhöhe den Fluss ψ auf der Kante (v, w) um δ und aktualisiere $pot(v)$, $pot(w)$.
    Falls Überschuss[w] = 0 **and** $\delta > 0$ **and** $w \neq S$ **then** füge w in die Queue ein.
    Überschuss[w] :=Überschuss[w] + δ. Überschuss[v] :=Überschuss[v] − δ.
    Falls nun $r_\varphi(v, w) = \psi(v, w)$ ist,
    so entferne die Kante (v, w) aus dem Niveaunetzwerk.
    Sei e nun die nächste von v ausgehende Kante im Niveaunetzwerk.
  end;
end.

In der Queue sollen immer alle Knoten mit positivem Überschuss stehen. Damit kein Knoten zum zweiten Mal in die Queue eingefügt wird, testet man, ob die Erhöhung

um δ den Überschuss des Knotens zum ersten Mal von 0 weg ändert und nimmt ihn nur dann in die Queue auf. Analog zur Prozedur `Forward` kann man sich eine Prozedur `Backward(v)` denken, die Fluss nach hinten, in Richtung zur Quelle zurückgibt.
Nun kann man zur Berechnung eines Sperrflusses wie folgt vorgehen:

6.25 Algorithmus. *(Forward–Backward–Propagation)*
Berechne alle pot(v).
while *Netzwerk enthält noch Q und S* **do**
begin
Berechne v_{min}, einen Knoten mit minimalem Potenzialwert.
Falls pot(v_{min}) > 0: `Forward(`v_{min}`)`, `Backward(`v_{min}`)`.
Entferne v_{min} aus dem Netzwerk.
end;

Die Wahl des Knotens mit dem kleinsten Potenzial garantiert, dass man tatsächlich den Fluss von v_{min} aus zur Senke und von v_{min} zurück zur Quelle propagieren kann. Anschließend kann man v_{min} aus dem Niveaunetzwerk entfernen, weil entweder alle seine ausgehenden Kanten oder alle seine eingehenden Kanten saturiert sind.
Welche Laufzeit hat nun dieser Algorithmus? Die Prozedur `Forward(v)` hat die Laufzeit $O(n+$ Anzahl entfernter Kanten$)$. Der Grund dafür ist der folgende. Da jeder Knoten nur maximal einmal in die Queue aufgenommen wird, hat die **while**–Schleife (bis auf die Kantenschleife innerhalb dieser) Laufzeit $O(n)$. Für die Schleife innerhalb der while-Schleife beobachten wir, dass eine Kante entweder saturiert wird oder als saturiert festgestellt wird, dann wird sie entfernt. Es gibt maximal eine Kante, die im Durchlauf nicht saturiert wird. Das ist nämlich die letzte betrachtete Kante. Dann wird die Schleife abgebrochen, weil der Überschuss des Knotens v abgebaut worden ist.
Wenn wir nun über alle maximal n Iterationen der Prozedur `Forward(v)` summieren, so erhalten wir als Gesamtlaufzeit $O(n \cdot n+$ Anzahl aller entfernten Kanten$)$. Da man jedoch insgesamt nicht mehr als $2e$ Kanten entfernen kann, ist die Gesamtlaufzeit $O(n^2 + e) = O(n^2)$.

6.26 Satz. *Der Algorithmus Forward–Backward–Propagation berechnet einen Sperrfluss in einem Niveaunetzwerk in Laufzeit $O(n^2)$. Durch iterierte Sperrflussberechnung kann man somit einen maximalen Fluss in Laufzeit $O(n^3)$ berechnen.*

6.8 Die Algorithmenfamilie von Goldberg und Tarjan – „Preflow–Push“

Die bisherigen Algorithmen erhöhten den Fluss entlang von FV-Wegen bzw. addierten sogar einen ganzen Sperrfluss zum aktuellen Fluss hinzu. Goldberg und Tarjan[4] betrachten das Verändern des Flusses jedoch noch atomarer, sie verändern den Fluss jeweils nur auf einer Kante und benutzen eine Extrainformation, die so genannten d–Werte, um festzulegen, entlang welcher Kante der Fluss verändert werden kann. Der

[4] A new approach to the maximum–flow problem, Journal of the ACM 35, (1988), 921–940.

Algorithmus von Goldberg und Tarjan ist ein Rahmenalgorithmus, der noch eine Reihe von Wahlmöglichkeiten lässt, die gesamte Algorithmenfamilie ist unter dem Namen „Preflow-Push-Algorithmen“ bekannt geworden.

6.27 Definition. *Für eine Abbildung $\varphi : E \to \mathbb{N}_0$ sei*

$$e(v) := \sum_{(\cdot,v)\in E} \varphi(\cdot, v) - \sum_{(v,\cdot)\in E} \varphi(v, \cdot)$$

der „Stau“ oder „Überschuss“ (flow excess) am Knoten v. Ein „Preflow“ φ muss die folgenden Eigenschaften erfüllen:

$$\begin{aligned} 0 \le \varphi(v,w) \le c(v,w) \quad & \text{für alle } (v,w) \in E \text{ sowie} \\ e(v) \ge 0 \quad & \text{für alle } v \neq Q. \end{aligned}$$

Im Vergleich zur Definition von Flüssen haben wir also die Kirchhoff–Regel, die in der obigen Notation $e(v) = 0$ lauten würde, ein wenig abgeschwächt.

6.28 Definition. *Ein Knoten $v \notin \{Q, S\}$ heißt aktiv, wenn $e(v) > 0$ gilt.*

6.29 Lemma. *Sei φ ein Preflow und v ein aktiver Knoten. Dann gibt es in $Rest_\varphi = (V, E_\varphi)$ einen gerichteten Weg von v nach Q.*

Beweis: Sei V^* die Menge der von v in $Rest_\varphi$ erreichbaren Knoten und $\overline{V^*} := V \setminus V^*$. Natürlich ist $v \in V^*$. Nehmen wir an, dass $Q \notin V^*$ ist. Für $(v^*, v') \in V^* \times \overline{V^*}$ gilt (da v' nicht erreicht werden kann), dass $(v^*, v') \notin E_\varphi$ ist. Wir analysieren die folgende Summe:

$$\sum_{w\in V^*} e(w).$$

Eine Kante $(v_1^*, v_2^*) \in V^* \times V^*$ trägt zu dieser Summe 0 bei, da $\varphi(v_1^*, v_2^*)$ einmal mit positivem Vorzeichen (in $e(v_2^*)$) und einmal mit negativem Vorzeichen (in $e(v_1^*)$) vorkommt. Somit ist der Wert der Summe gleich

$$\sum_{(v',v^*)\in E\cap(\overline{V^*}\times V^*)} \varphi(v', v^*) - \sum_{(v^*,v')\in E\cap(V^*\times\overline{V^*})} \varphi(v^*, v'). \tag{3}$$

Da es in $Rest_\varphi$ keine Kante von V^* nach $\overline{V^*}$ gibt, ist für jede Kante $(v', v^*) \in E \cap \overline{V^*} \times V^*$ der Wert $\varphi(v', v^*) = 0$ und für jede Kante $(v^*, v') \in E \cap V^* \times \overline{V^*}$ der Wert $\varphi(v^*, v') = c(v^*, v')$. Damit ist der Ausdruck (3) kleiner oder gleich Null. Da alle $e(w)$ für $w \neq Q$ nicht negativ sind, ist die Summe aber auch größer oder gleich Null, somit ist sie gleich Null und es folgt, dass für alle $w \in V^*$ gilt, dass $e(w) = 0$ ist. Dies ist ein Widerspruch zu $e(v) > 0$. □

Bislang haben wir die Entfernungen $d(v)$ auf FV–Wegen benutzt, um festzulegen, wie der Fluss nach und nach vergrößert wird. Goldberg und Tarjan verwenden ebenfalls eine d–Funktion, die steuert, entlang welcher Kanten wir den Fluss vergrößern sollten.

6.30 Definition. *Eine gültige Knotenmarkierung für einen Preflow φ ist eine Abbildung $d : V \to \mathbb{N}_0$ mit folgenden Eigenschaften: $d(Q) = n$, $d(S) = 0$ und $d(v) \leq d(w) + 1$ für alle $(v, w) \in E_\varphi$.*

6.31 Lemma. *Sei φ ein Preflow, für den es eine gültige Markierung d gibt. Dann gilt:*

a) Für alle $v \neq w \in V$ gilt:
Jeder Weg in $Rest_\varphi$ von v nach w hat Länge mindestens $d(v) - d(w)$.

b) In $Rest_\varphi$ gibt es keinen Weg von Q nach S.

Beweis: a) Für einen Weg $v = v_0 \to v_1 \to \cdots \to v_k = w$ der Länge k ist $d(v) = d(v_0) \leq d(v_1) + 1 \leq d(v_2) + 2 \leq \cdots \leq d(w) + k$. Es folgt $k \geq d(v) - d(w)$.
b) Wegen $d(S) = 0$ und $d(Q) = n$ folgt aus a), dass jeder Weg von Q nach S Länge mindestens n hat. Es kann also keinen Weg von Q nach S geben. □

(Im Hinterkopf sollten wir die Interpretation von d als Distanz behalten. Der Algorithmus abstrahiert diese Idee jedoch und reduziert sie auf einfache Rechenregeln.
Wenn man sich den Exkurs in die Distanzen (Kapitel 6.5) in Erinnerung ruft, sollte auch die folgende Definition nicht unbedingt verblüffen:)

6.32 Definition. *Eine Kante $(v, w) \in E_\varphi$ heißt wählbar, wenn $d(v) = d(w) + 1$ ist.*

Wir kommen zum Rahmenalgorithmus. Dieser benutzt zwei Basisoperationen: Push und Relabel. Zur Effizienzsteigerung werden wir später genauer spezifizieren, in welcher Reihenfolge diese ausgewählt werden.

6.33 Algorithmus. *(Generischer Preflow–Push–Algorithmus)*

1.) (Initialisierung)
Für alle $v \in V$: $\{e(v) := 0; d(v) := 0\}$; $d(Q) := n$.
Für alle $(v, w) \in E$: $\varphi(v, w) := 0$.

2.) (Weitere Initialisierung)
Für alle $(Q, v) \in E$: $e(v) := c(Q, v)$ und $\varphi(Q, v) := c(Q, v)$.

3.) Solange es aktive Knoten gibt: Wähle einen aktiven Knoten v aus und führe eine auf v anwendbare Basisoperation aus. Basisoperationen sind Push und Relabel.

Push(v, w)	(anwendbar, wenn v aktiv und $(v, w) \in E_\varphi$ wählbar ist.)
	$\delta := \min\{e(v),\ r_\varphi(v, w)\}$.
	Falls $(v, w) \in E$, dann $\varphi(v, w) := \varphi(v, w) + \delta$,
	sonst $\varphi(w, v) := \varphi(w, v) - \delta$.
	$e(v) := e(v) - \delta, e(w) := e(w) + \delta$.
Relabel(v)	(anwendbar, wenn v aktiv und keine Kante $(v, \cdot) \in E_\varphi$ wählbar ist.)
	$d(v) := \min\{d(w) + 1 \mid (v, w) \in E_\varphi\}$.

In Relabel(v) ist die Menge, aus der das Minimum gewählt wird, nicht leer, da nach

Lemma 6.29 ein Weg von v zur Quelle Q in $Rest_\varphi$ existiert, da v ja aktiv ist. Und ein Weg in $Rest_\varphi$ beginnt mit einer Kante aus E_φ.

6.34 Lemma. *Wenn d eine gültige Markierung ist, dann wird durch Relabel(v) die Zahl $d(v)$ mindestens um 1 erhöht.*

Beweis: Die Anwendbarkeit von Relabel(v) impliziert, dass keine Kante (v, w) aus dem Restgraph wählbar ist. Somit ist $d(v) \neq d(w) + 1$ für alle $(v, w) \in E_\varphi$. Da d gültig ist, folgt zusätzlich $d(v) \leq d(w) + 1$ und somit insgesamt $d(v) \leq d(w)$ für alle $(v, w) \in E_\varphi$. Also ist

$$\min\{d(w) + 1 \mid (v, w) \in E_\varphi\} \geq d(v) + 1$$

und $d(v)$ wird um mindestens 1 erhöht. □

6.35 Definition. *Eine Push–Operation Push(v, w) heißt* saturierend*, wenn in der Operation $\delta = r_\varphi(v, w)$ ist, die Kante also durch den Push voll ausgelastet wird. Andere Push–Operationen werden* nicht–saturierend *genannt.*

Wir zeigen zunächst, dass auch bei beliebiger Auswahl der Reihenfolge der Basisoperationen der Algorithmus korrekt und endlich ist.

6.36 Lemma. *Während des Algorithmus ist d stets eine gültige Markierung für φ.*

Beweis: Die Anfangsmarkierung ist gültig, da die Bedingung „$d(v) \leq d(w) + 1$“ zu Beginn nur für $v = Q$ verletzt sein könnte. Da jedoch alle Q verlassenden Kanten saturiert sind und somit keine Kante der Form (Q, w) in E_φ ist, folgt die Gültigkeit.
Durch die Anwendung einer Basisoperation werden gültige Markierungen nicht zerstört:
Relabel(v): bereitet für Kanten (v, w) keine Probleme, da $d(v)$ den größtmöglichen erlaubten Wert erhält. Auch für Kanten (w, v) können keine Probleme entstehen, da $d(v)$ durch Relabel(v) nicht kleiner wird (Lemma 6.34).
Push(v, w): Durch diesen Aufruf kann in $Rest_\varphi$ die Kante (v, w) eliminiert werden oder die Kante (w, v) neu entstehen. Kanteneliminationen bereiten keine Probleme. Betrachten wir also das Entstehen einer neuen Kante (w, v). Da Push(v, w) anwendbar ist, ist $d(w) = d(v) - 1 \leq d(v) + 1$, und die neue Kante (w, v) bereitet keine Probleme. □

6.37 Lemma. *Wenn der Algorithmus stoppt, dann ist φ ein maximaler Fluss.*

Beweis: Der Algorithmus stoppt nur, wenn es keinen aktiven Knoten mehr gibt. Dann ist $e(v) = 0$ für alle $v \notin \{Q, S\}$. Damit ist φ ein Fluss. Außerdem ist d nach Lemma 6.36 eine gültige Markierung. Lemma 6.31 b) stellt sicher, dass es keinen FV-Weg von Q nach S gibt. Damit ist φ maximal. □

Für den Beweis der Endlichkeit machen wir eine Rechenzeitanalyse.

6.38 Lemma.
- *Der Wert von $d(v)$ sinkt niemals.*

- *Stets gilt* $d(v) \leq 2n-1$.

Beweis: $d(v)$ wird nur durch Relabel(v) verändert, in Lemma 6.34 haben wir gezeigt, dass $d(v)$ dabei um mindestens 1 größer wird.
Die zweite Behauptung ist trivial für $v \in \{Q, S\}$, denn da d eine gültige Markierung ist, gilt stets $d(S) = 0$ und $d(Q) = n$. Die Markierung eines Knotens $v \notin \{Q, S\}$ kann sich nur verändern, wenn v aktiv, also $e(v) > 0$ ist. Nach Lemma 6.29 gibt es also einen gerichteten Weg von v nach Q (der Länge höchstens $n-1$) in $Rest_\varphi$. Wenn dieser Weg die Form $v \to v' \to \cdots \to Q$ hat, dann hat der Weg von v' nach Q die Länge höchstens $n-2$. Wegen Lemma 6.31, Teil a), folgt $d(v') - d(Q) \leq n-2$ und somit $d(v') \leq 2n-2$. Da $(v, v') \in E_\varphi$ ist, ergibt sich, dass

$$\min\{d(w) + 1 \mid (v, w) \in E_\varphi\} \leq d(v') + 1 \leq 2n - 1$$

ist. □

6.39 Lemma. *a) Es finden höchstens* $2n^2$ *Relabel-Aufrufe statt.*

b) Es finden höchstens $2ne$ *saturierende Push–Operationen statt.*

c) Es finden höchstens $4n^2e$ *nicht-saturierende Push–Operationen statt.*

Beweis: a): Da jeder Relabel(v)-Aufruf den Wert $d(v)$ um mindestens 1 erhöht und $d(v) \leq 2n-1$ ist (Lemma 6.38), gibt es für jeden Knoten v höchstens $2n-1$ Relabel-Aufrufe. Aussage a) folgt, da es n Knoten gibt.
b): Für $(v, w) \in E_\varphi$ betrachten wir saturierende Push–Operationen von v nach w. Zwischen zwei saturierenden Push–Operationen von v nach w muss ein Push von w nach v durchgeführt worden sein. Wegen der Voraussetzungen an die Anwendbarkeit von Push und da d-Markierungen nur wachsen, muss $d(w)$ um mindestens 2 gewachsen sein und $d(v)$ muss auch um mindestens 2 gewachsen sein. Vor dem ersten saturierenden Push gilt $d(v) + d(w) \geq 1$, vor dem letzten saturierenden Push nach Lemma 6.38 $d(v) + d(w) \leq 4n-3$. Also kann es höchstens n saturierende (v, w)-Push–Operationen geben. Da der Restgraph maximal $2e$ Kanten enthält, folgt die Aussage des Lemmas.
c): Wir verwenden eine „Potenzialfunktion“, die vom „Zustand des Algorithmus“ abhängt. Sei

$$\Phi := \underbrace{(\text{Anzahl saturierender Pushs}) \cdot (2n-2)}_{\Phi_1} + \underbrace{\sum_{t \in V} d(t)}_{\Phi_2} - \underbrace{\sum_{t \in V,\ t \text{ aktiv}} d(t)}_{\Phi_3}.$$

Nach der Initialisierungsphase ist $\Phi = n$. Wir betrachten, wie sich Φ im Laufe des Algorithmus verändert. Da Push(v, w) nur anwendbar ist, wenn $d(v) = d(w) + 1$ ist, ist $d(w) = d(v) - 1 \leq 2n-2$. Wir unterscheiden wieder saturierende und nicht-saturierende Pushs:
Bei einem nicht-saturierenden Push(v, w) bleiben Φ_1 und Φ_2 unverändert. Es wird $e(v) = 0$ und somit v inaktiv. Damit wird Φ_3 um $d(v)$ kleiner, eventuell zusätzlich aber auch

um $d(w)$ größer, wenn w neu aktiviert wird. Wegen $d(v) = d(w) + 1$ wird auf jeden Fall Φ_3 um mindestens 1 kleiner, Φ also mindestens 1 größer.
Bei einem saturierenden Push kann im Unterschied zum nicht-saturierenden Push der Knoten v anschließend immer noch aktiv sein, somit kann Φ_3 um $d(w) \leq 2n - 2$ größer werden. Andererseits wird Φ_1 um $2n - 2$ größer, da die Anzahl der saturierenden Pushs um 1 gewachsen ist. Insgesamt erhalten wir, dass ein saturierender Push das Potenzial Φ nicht kleiner macht.
Durch eine Relabel-Operation wird $d(v)$ um eine Zahl $h \geq 1$ größer. Da v aktiv ist, wird somit Φ_3 um h größer, allerdings wird auch Φ_2 um h größer. Der Gesamteffekt ist der, dass Φ unverändert bleibt.
Φ ist am Ende des Algorithmus nicht größer als

$$\begin{aligned}(2ne) \cdot (2n-2) + n \cdot (2n-1) &\leq 4n^2e - 4ne + 2n^2 \\ &= 4n^2e + 2n \cdot (n - 2e).\end{aligned}$$

Da $e \geq n - 1 \geq n/2$ ist, ist $n - 2e \leq 0$ und somit ist Φ höchstens $4n^2e$, also kann es höchstens $4n^2e$ viele nicht-saturierende Push–Operationen geben. □

Egal, in welcher Reihenfolge wir Basisoperationen auswählen, auf jeden Fall reichen $O(n^2e)$ Basisoperationen aus. Eine Push–Operation ist in konstanter Zeit ausführbar, wenn eine anwendbare Basisoperation bereits gefunden wurde. Wie kommt man schnell an eine anwendbare Basisoperation heran? Das klären wir nun.

PUSH/RELABEL(v) (Voraussetzung: v ist aktiv.)
Versuche Push(v, w) für die aktuelle Kante (v, w). Wenn die Kante nicht wählbar ist, gehe in $Adj(v)$ einen Schritt weiter, die neue Kante wird aktuell. Wird auf diese Weise das Ende der Liste erreicht, Relabel(v). Für den ersten Knoten w in $Adj(v)$ wird (v, w) aktuell. (Sprich: Man springt über das Ende der Liste wieder an den Anfang der Liste zurück.)

6.40 Lemma. *Die Operation Push/Relabel(v) führt Relabel(v) nur durch, wenn diese Operation anwendbar ist.*

Beweis: Nach Voraussetzung ist v aktiv. Wir betrachten eine Kante (v, w) und gehen zurück zu dem Zeitpunkt, als (v, w) das letzte Mal aktuell war. Irgendwann war (v, w) nicht mehr wählbar und wir sind in der Adjazenzliste von v einen Schritt weitergegangen. Zwischen diesem Zeitpunkt und dem Aufruf von Relabel hat es möglicherweise eine Reihe von Operationen gegeben. Wir müssen uns davon überzeugen, dass dadurch die Kante (v, w) nicht wieder wählbar wurde. Man überzeugt sich von folgendem. Durch eine Relabel(v')–Operation werden nur Kanten möglicherweise wählbar, die im Knoten v' starten. Push–Operationen fügen keine Kante zur Menge der wählbaren Kanten hinzu: Sie verändern die d–Werte nicht, also wird keine schon im Restgraphen vorhandene Kante wählbar. Andererseits fügt Push(v', w') die Kante (w', v') eventuell neu zum Restgraphen hinzu, allerdings gilt dann folgendes: nach Voraussetzung der Push–Operationen

war die Kante (v', w') wählbar und somit $d(v') = d(w') + 1$. Damit ist die Kante (w', v') nicht wählbar. □

Wir erhalten folgenden Gesamtalgorithmus:

6.41 Algorithmus.

while *die Menge A der aktiven Knoten ist nicht leer* **do**
begin
  wähle einen beliebigen Knoten $v \in A$.
  Push/Relabel(v).
end

Auch dies ist wieder ein generischer Algorithmus, weil wir die Wahl haben, wie wir den „beliebigen" Knoten tatsächlich wählen.

6.42 Satz. *Der generische Algorithmus 6.41 berechnet in Zeit* $O(n^2e)$ *einen maximalen Fluss. Mit Ausnahme der nicht saturierenden Push–Operationen genügt Zeit* $O(ne)$.

Beweis: Wenn beim Betrachten einer aktuellen Kante (v, w) eine Push–Operation durchgeführt wird, dann zählen wir diese Kosten bei der Anzahl der Basisoperationen mit. Eventuell wird eine aktuelle Kante aber „vergeblich" betrachtet, wenn nämlich keine Push–Operation entlang der Kante durchgeführt wird. Eine bestimmte Kante (v, w) kann aber nur $O(n)$ mal vergeblich betrachtet werden, da zwischen zwei vergeblichen Betrachtungen von (v, w) ein Relabel(v)-Aufruf liegt. Daher sind die Extrakosten, die durch vergebliche Kantenbetrachtungen entstehen, durch $O(ne)$ beschränkt.
Push–Operationen dauern Zeit $O(1)$, Relabel–Operationen brauchen länger: Bei Aufruf von Relabel(v) wird zur Bestimmung des Minimums die Liste der Kanten, die den Knoten v enthalten, noch einmal durchlaufen. Da jede Kante jedoch nur in maximal $2n - 1$ solchen Berechnungen betrachtet wird, genügt auch hier die Extrazeit $O(n \cdot e)$.
Man beachte, dass man für die Auswahl des Knotens v, auf dem Push/Relabel durchgeführt wird, die Menge der aktiven Knoten mit einem Stack verwalten kann. □

Bisher haben wir die Laufzeit des Algorithmus von Dinic erreicht. Es ist aber leicht, durch eine festgelegte Reihenfolge der Basisoperationen auch die Laufzeit $O(n^3)$ zu erreichen.

6.43 Algorithmus. *(FIFO-Strategie)*

1.) Initialisiere eine Queue der aktiven Knoten.

2.) **while** *Queue nicht leer:*
i) v sei der oberste Knoten in der Queue. Entferne ihn aus der Queue.
ii) **repeat**
Push/Relabel(v). Falls ein Knoten w dabei aktiv wird, füge w in die Queue ein.
until *(e(v) = 0) oder Relabel(v) wurde aufgerufen.*
iii) falls v noch aktiv ist, füge v in die Queue ein.
endwhile

Dieser Algorithmus hat wieder Laufzeit $O(n^3)$. Er ist einfach zu implementieren und bietet Raum für weitere Verbesserungen. Zur Analyse definieren wir sogenannte Durchgänge von Algorithmus 6.43. Der erste Durchgang enthält die Aufrufe für die Knoten, die während der Initialisierungsphase in die Queue eingefügt worden sind. Der $(i+1)$-te Durchgang enthält die Aufrufe für Knoten, die während Durchgang i in die Queue eingefügt worden sind.

6.44 Satz. *Mit der FIFO-Strategie kann ein maximaler Fluss in Zeit $O(n^3)$ berechnet werden.*

Beweis: Nach Satz 6.42 genügt es, die Zahl der nicht saturierenden Push–Operationen abzuschätzen. Ein nicht saturierender Push für Knoten v führt dazu, dass $e(v) = 0$ ist. Also kann es für jeden Knoten in jedem Durchgang nur einen nicht saturierenden Push geben. Daher ist es ausreichend, die Zahl der Durchgänge durch $4n^2$ abzuschätzen. Wieder arbeiten wir mit einer Potenzialfunktion.

$$\Phi := \underbrace{2 \cdot \sum_{v \in V} d(v)}_{\Phi_1} - \underbrace{\max\{d(v) \mid v \text{ ist aktiv}\}}_{\Phi_2}.$$

Nach der Initialisierung des Algorithmus gilt $\Phi = 2n$ und am Ende gilt $\Phi \leq 4n^2$, da es keine aktiven Knoten mehr gibt.
Wir beobachten die Veränderung von Φ innerhalb eines Durchganges. Unser Ziel ist es, zu zeigen, dass Φ pro Durchgang um mindestens eins größer wird. Wir analysieren daher die Auswirkungen der Operationen auf die Potenzialfunktion Φ.
Durch eine Relabel(v)–Operation wächst $d(v)$ um einen Wert $h \geq 1$. Dann wird Φ_1 um $2h$ größer, Φ_2 aber höchstens um h, also wird Φ um mindestens $2h - h = h \geq 1$ größer.
Durch eine Push–Operation verändert sich Φ_1 nicht, da sie keine d-Werte verändert. Allerdings kann eine Push(v,w)–Operation den Knoten v deaktivieren bzw. den Knoten w aktivieren. Durch eine Deaktivierung wird Φ_2 nicht größer. Die Aktivierung macht auch keine Probleme, da vor der Push–Operation $\Phi_2 \geq d(v)$ war und $d(w) = d(v) - 1$ gilt. Somit ist Φ_2 nach der Push–Operation nicht größer geworden. Fazit: Φ wird nicht kleiner.

Schauen wir uns nun die Situation am Ende eines Durchgangs an: Entweder wurde für irgendeinen bearbeiteten Knoten v die Operation Relabel(v) aufgerufen. Mit der Argumentation von gerade ist Φ mindestens um 1 gewachsen.
Oder es wurde keine einzige Relabel–Operation aufgerufen. Dann ist für jeden Knoten v der Wert $d(v)$ gleich geblieben und somit ist Φ_1 gleich geblieben. Für jeden Knoten v wurde die repeat-Schleife also mit der Bedingung $e(v) = 0$ abgebrochen und daher war v in diesem Moment inaktiv. Nach Beendigung des Durchgangs sind also nur noch Knoten aktiv, die in dem Durchgang durch eine Push–Operation aktiviert wurden. Push–Operationen entlang einer Kante (v, w) aktivieren aber nur Knoten w, für die $d(w) = d(v) - 1$ und v aktiv ist. Daher werden in einem Durchgang nur Knoten aktiviert, deren d-Wert kleiner oder gleich $\Phi_2 - 1$ ist. Am Ende eines Durchgangs ist dann also Φ_2 um mindestens eins kleiner geworden. Zusammen erhalten wir, dass Φ pro Durchgang um mindestens 1 wächst, und da $\Phi \leq 4n^2$, gibt es also höchstens $4n^2$ Durchgänge. □

6.9 Die Highest–Level Selection Rule

Nur am Rande wollen wir die „Highest–Level Selection Rule“ erwähnen. Diese schreibt vor, dass als aktiver Knoten, für den eine Basisoperation ausgeführt werden soll, jeweils derjenige gewählt werden soll, dessen d–Wert maximal ist.
Mit Hilfe einer einfachen Potenzialfunktionsanalyse kann man zeigen, dass bei dieser Auswahl die Laufzeit zur Berechnung eines maximalen Flusses beschränkt ist durch $O(n^2\sqrt{e})$. Wer sich für diesen Beweis interessiert, kann sich den Artikel „An analysis of the highest–level selection rule in the preflow–push max–flow algorithm“ von Cheriyan und Mehlhorn ansehen, erschienen in Information Processing Letters 69 (1999), S. 239–242.

7 Ein effizienter randomisierter Primzahltest

7.1 Kryptographische Systeme

Die Zahlentheorie zählt zu den ältesten mathematischen Disziplinen. Allerdings befasst sich die mathematische Zahlentheorie vor allem mit strukturellen Aspekten. Die Entwicklung effizienter Algorithmen für Probleme der Zahlentheorie gelangte erst mit der Entwicklung der Informatik in den Mittelpunkt des Interesses. Besonders die Kryptographie, die sich mit Chiffriermethoden befasst, stellt ein Bindeglied zwischen der Zahlentheorie und dem Entwurf effizienter Algorithmen dar.
Der Vorteil effizienter Algorithmen in der Zahlentheorie, z. B. bei der Berechnung des größten gemeinsamen Teilers zweier Zahlen oder bei dem Test, ob eine gegebene Zahl prim ist, zeigt sich vor allem bei großen Zahlen. Zur Motivation der in diesem Kapitel behandelten Probleme stellen wir daher zunächst das RSA-System vor. Dieses System ist das am häufigsten benutzte, mit öffentlich bekannten Schlüsseln arbeitende kryptographische Verfahren. Es benötigt effiziente Algorithmen für den Umgang mit sehr langen Binärzahlen und die meisten grundlegenden Probleme der Elementaren Zahlentheorie. In Kapitel 7.2 stellen wir effiziente Algorithmen für die Berechnung von a^n, des größten gemeinsamen Teilers zweier Zahlen und der multiplikativen Inversen modulo m vor. Kapitel 7.4 befasst sich mit der effizienten Berechnung des Jacobi-Symbols, und in Kapitel 7.6 wird schließlich ein effizienter Primzahltest vorgestellt.
Die Kryptographie ist eine sehr alte Wissenschaft, die bis in die 70er Jahre hinein fast ausschließlich von militärischen Anforderungen geprägt wurde. Schon Cäsar verschlüsselte Nachrichten, allerdings auf sehr einfache Weise. Heutzutage bedient sich neben dem Militär der Datenschutz kryptographischer Verfahren. Um den unberechtigten Zugriff auf schutzwürdige Daten zu verhindern, werden Daten verschlüsselt. Mit der zunehmenden Bedeutung des Internets für den Geschäftsverkehr (zum Beispiel „Begleichen einer Rechnung durch Angabe der Kreditkartennummer“) wächst auch hier die Bedeutung von sicheren kryptographischen Verfahren.
Da über einen langen Zeitraum und von vielen Benutzern das gleiche kryptographische System verwendet werden soll, wird davon ausgegangen, dass das benutzte kryptographische Verfahren allgemein bekannt ist. Die Eigenschaft, dass die Sicherheit eines Kryptosystems nicht auf der Geheimhaltung des zugrunde liegenden Algorithmus beruhen darf, ist übrigens auch unter dem Namen „Kerkhoffs Prinzip“ bekannt.

7.1 Definition. *Ein kryptographisches System besteht aus folgenden Komponenten:*

- *der endlichen Schlüsselmenge K (z. B. $\{0,1\}^{300}$).*
- *der Nachrichtenmenge M (z. B. $\{0,1\}^*$, der Menge der endlichen Folgen über $\{0,1\}$).*
- *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* $m \in M$ *und* $k \in K$ *ist.*

Wie wird ein solches kryptographisches System benutzt? Wenn zwei Benutzer kommunizieren wollen, vereinbaren sie einen Schlüssel $k \in K$. Der Absender einer Nachricht m verschlüsselt diese mit Hilfe des Schlüssels, indem er das Kryptogramm $y = E(k, m)$ berechnet. Das Kryptogramm wird über einen unter Umständen unsicheren Kommunikationskanal an den Empfänger gesendet. Der Empfänger kann das Kryptogramm entschlüsseln, indem er $m = D(k, y)$ berechnet. Damit die Verwendung des kryptographischen Systems die Kommunikation nicht wesentlich verlangsamt, sollten die notwendigen Operationen, also die Vereinbarung eines Schlüssels und die Anwendung der Chiffrier- und Dechiffrierfunktion, effizient durchführbar sein.
Darüber hinaus soll die Nachricht bei der Kommunikation geschützt sein. Ein Dritter, hier Gegner genannt, der das Kryptogramm y abfängt, soll daraus (fast) keinen Informationsgewinn erlangen können. Daher muss ihm auf jeden Fall der Schlüssel k unbekannt bleiben, denn sonst könnte er das Kryptogramm wie der legale Benutzer dechiffrieren. Wegen der Redundanz jeder natürlichen Sprache, die sich in den Nachrichten widerspiegelt, sind alle kryptographischen Systeme, die mit endlich vielen verschiedenen Schlüsseln arbeiten, im Prinzip zu „knacken“. Der Gegner kann, wenn das Kryptogramm lang genug ist, die Nachricht m und den verwendeten Schlüssel k mit hoher Wahrscheinlichkeit korrekt errechnen. Die Sicherheit kryptographischer Systeme beruht darauf, dass der Gegner zur Berechnung von m und/oder k so viel Aufwand treiben muss, dass sich die Berechnung für ihn nicht lohnt oder dass die benötigten Ressourcen praktisch nicht verfügbar sind, da er z. B. 10000 Jahre CPU-Zeit auf dem schnellsten Rechner benötigt. Ein sicheres kryptographisches System muss also so beschaffen sein, dass die von den legalen Benutzern auszuführenden Operationen effizient durchführbar sind, während es für die Kryptanalyse durch einen Gegner keinen effizienten Algorithmus gibt.
Bis zum Jahr 1976 waren nur kryptographische Systeme mit geheimen Schlüsseln (secret key cryptosystems) bekannt. Dabei tauschen die Kommunikationspartner den von ihnen verwendeten Schlüssel auf einem sicheren Kommunikationsweg aus, z. B. mit Hilfe eines vertrauenswürdigen Kuriers. Im militärischen Bereich sind derartige Systeme gut einsetzbar. Auch zur Datensicherung, an der nur ein Benutzer oder ein Rechner beteiligt ist, eignen sich Systeme mit geheimen Schlüsseln. In postalischen Netzen (Telefon, Internet, usw.) sind solche Systeme aber ungeeignet. Das Wesen dieser Kommunikationssysteme besteht ja gerade darin, dass jeder Benutzer mit jedem anderen, ihm zuvor eventuell unbekannten Benutzer kommunizieren kann. Die Vereinbarung eines Schlüssels auf sicherem Weg ist dann nicht möglich, wenn man die Post nicht für einen vertrauenswürdigen Kurier hält.
Diffie und Hellman (1976) haben die Benutzung kryptographischer Systeme mit öffentlich bekannten Schlüsseln (public key cryptosystems) vorgeschlagen, ein auf den ersten Blick undurchführbarer Vorschlag. Wenn alle Informationen öffentlich sind, muss doch der Gegner genauso wie der Empfänger einer Nachricht das Kryptogramm entschlüsseln können. Der faszinierende Trick in dem Vorschlag von Diffie und Hellman lässt sich einfach beschreiben. Jeder Benutzer B erzeugt sich einen Schlüssel k_B und eine mit dem

Schlüssel verbundene zusätzliche Information $I(k_B)$. Der Schlüssel k_B wird öffentlich bekannt gemacht. Wenn Benutzer A eine Nachricht m an B senden will, benutzt er den Schlüssel k_B und berechnet das Kryptogramm $y = E(k_B, m)$. Die Zusatzinformation $I(k_B)$ muss nun so beschaffen sein, dass die Dechiffrierung, also die Berechnung von $m = D(k_B, y)$, mit Hilfe von y, k_B und $I(k_B)$ auf effiziente Weise möglich ist, während eine effiziente Berechnung von m aus y und k_B, ohne $I(k_B)$ zu kennen, unmöglich ist.
Rivest, Shamir und Adleman (1978) haben ein vermutlich sicheres kryptographisches System mit öffentlich bekannten Schlüsseln entworfen. Wir wollen dieses nach den Anfangsbuchstaben ihrer Erfinder benannte RSA-System vorstellen, bevor wir den Zusatz „vermutlich“ im vorigen Satz erläutern.
Jeder Benutzer B erzeugt sich zwei Primzahlen $p, q \geq 2^\ell$ mit je $\ell + 1$ Bits. Heutzutage muss ℓ mindestens 500 sein. Zur Erzeugung von p und q werden Zufallszahlen darauf getestet, ob sie Primzahlen sind. Da es genügend viele Primzahlen gibt, müssen im Durchschnitt nicht zu viele Versuche unternommen werden. Für diesen Schritt stellen wir in Kapitel 7.6 einen effizienten, randomisierten Primzahltest vor, der mit sehr kleiner Wahrscheinlichkeit auch Zahlen, die keine Primzahlen sind, als Primzahlen klassifiziert. Allerdings kann man diese Wahrscheinlichkeit so klein wählen, dass die Wahrscheinlichkeit eines Fehlers in der Hardware verglichen dazu viel größer ist. Durch einen derartigen Fehler erhält der Gegner keine zusätzliche Information, allerdings wird die Kommunikation zwischen den beiden legalen Benutzern gestört. In diesem seltenen Fall müsste B einen neuen Schlüssel erzeugen. Im Normalfall berechnet B nun $n = pq$ und den Wert der Euler-Funktion $\varphi(n)$. Hierbei ist $\varphi(n)$ die Anzahl aller $a \in \{1, \ldots, n-1\}$, die relativ prim zu n sind. Für Primzahlen $p \neq q$ ist $\varphi(pq) = (p-1)(q-1)$. Außerdem erzeugt sich B eine Zufallszahl $e \in \{2, \ldots, \varphi(n)-1\}$, die teilerfremd zu $\varphi(n)$ ist, und berechnet $d \equiv e^{-1} \bmod \varphi(n)$. In Kapitel 7.2 geben wir effiziente Algorithmen für die Berechnung von größten gemeinsamen Teilern und von multiplikativen Inversen modulo m an. Der öffentlich bekannt gegebene Schlüssel k besteht aus dem Paar (n, e), während das Paar $(\varphi(n), d)$ die geheime Zusatzinformation von B ist.
Nehmen wir nun an, dass A eine Nachricht $m \in \{0,1\}^*$ an B senden will. Dann zerlegt A die Nachricht m in Blöcke m_i der Länge 2ℓ. Jeder Block wird auch als Binärzahl aufgefasst, deren Wert ebenfalls mit m_i bezeichnet wird. Es ist $0 \leq m_i < 2^{2\ell}$. Die Chiffrierfunktion E ist definiert durch

$$E((n, e), m_i) \equiv m_i^e \bmod n$$

und kann (s. Kapitel 7.2) effizient berechnet werden. Sei $D((n, d), y) \equiv y^d \bmod n$, dann ist D ebenfalls effizient berechenbar. Es ist $D((n, d), E((n, e), m_i)) \equiv m_i^{ed} \bmod n$. Da nach Konstruktion $ed \equiv 1 \bmod \varphi(n)$ ist, folgt aus der Elementaren Zahlentheorie $m_i^{ed} \equiv m_i \bmod n$. (Einen Beweis hierzu findet man im Anhang des Skripts.) Da $n \geq 2^{2\ell}$, haben wir mit $m_i \bmod n$ auch m_i eindeutig berechnet.[5]

[5]Für die von uns verwendeten und nicht bewiesenen Aussagen der Elementaren Zahlentheorie und für weitere Resultate empfehlen wir die Bücher „An Introduction to the Theory of Numbers“ von Niven und Zuckerman, „Einführung in die Zahlentheorie“ von P. Bundschuh oder „Algorithmic Number Theory, Volume I, Efficient Algorithms“ von Eric Bach und Jeffrey Shallit.

Die für die folgenden Abschnitte angekündigten Resultate implizieren also, dass die legalen Benutzer das RSA-System effizient benutzen können. Es gibt bisher keinen effizienten Algorithmus für die Kryptanalyse, also keinen effizienten Algorithmus, mit dem der Gegner das RSA-System knacken kann. Wenn er in der Lage wäre, Zahlen effizient zu faktorisieren, d.h. in ihre Primfaktoren zu zerlegen, dann könnte er aus (n, e) auf effiziente Weise die geheime Zusatzinformation $(\varphi(n), d)$ berechnen.
Umgekehrt kann man zeigen, dass man die Zahl $n = p \cdot q$ faktorisieren kann, wenn man $\varphi(n)$ kennt:

7.2 Lemma. *Sei n von der Form $n = p \cdot q$ für zwei Primzahlen $p \neq q$. Aus $\varphi(n)$ und n kann man in Polynomialzeit die beiden Faktoren p und q errechnen.*

Beweis: Es ist

$$\begin{aligned} \varphi(n) &= (p-1) \cdot (q-1) \\ &= (p-1) \cdot (\frac{n}{p} - 1) \\ &= n - p - \frac{n}{p} + 1. \end{aligned}$$

Damit ist $p \cdot \varphi(n) = np - p^2 - n + p$ und folglich p Nullstelle der quadratischen Gleichung $x \cdot \varphi(n) = nx - x^2 - n + x$ bzw.

$$x^2 + x \cdot (\varphi(n) - n - 1) + n = 0$$

und kann in Polynomialzeit berechnet werden. □

7.3 Beispiel. *Bekannt sei $n = 84773093$ und $\varphi(n) = 84754668$. Die quadratische Gleichung wird dann zu $x^2 - x \cdot (18426) + 84773093 = 0$. Nullstellen davon sind 9539 und 8887. Dies sind gerade die beiden Faktoren p und q in $n = p \cdot q$.*

Also beruht die Sicherheit des RSA-Systems letztendlich darauf, dass es schwierig ist, Zahlen zu faktorisieren.
Man vermutet, dass es wesentlich schwieriger ist, eine Zahl zu faktorisieren, als zu entscheiden, ob sie Primzahl ist. Daher liefern effiziente Primzahltests für Zahlen, die keine Primzahlen sind, im allgemeinen keine Teiler. Wir wollen jedoch nicht tiefer in die Diskussion über die Sicherheit des RSA-Systems einsteigen, da dieses System in unserem Kontext seine Aufgabe, die Beschäftigung mit Problemen der Elementaren Zahlentheorie für sehr große Zahlen zu motivieren, bereits erfüllt hat.

7.2 Potenzen, multiplikative Inverse und größte gemeinsame Teiler

Wir benutzen nun die arithmetischen Operationen Addition, Subtraktion, Multiplikation und (ganzzahlige) Division als Elementaroperationen. Dann ist auch die Berechnung von $a \bmod m$ in konstanter Zeit möglich. In $\mathbb{Z}_m$ können wir also Addition, Subtraktion und Multiplikation als Elementaroperationen ansehen.

Im RSA-System müssen Potenzen $x^n \bmod m$ für sehr große n berechnet werden. Der folgende Algorithmus zeigt, dass man x^n mit sehr viel weniger als n Operationen berechnen kann. Bei der Berechnung von $x^n \bmod m$ kann (und sollte) man natürlich schon die Zwischenergebnisse modulo m nehmen.

7.4 Algorithmus. *„Potenzieren“*
$POT(x, n)$
if n=1 **then** **return** x;
if n *gerade* **then** $temp = POT(x, \frac{n}{2})$; **return** $temp \cdot temp$;
else $temp = POT(x, \frac{n-1}{2})$; **return** $temp \cdot temp \cdot x$;

Man mache sich als einfache Übung klar, dass man diesen Algorithmus auch iterativ umsetzen kann. Wir analysieren die Laufzeit des Algorithmus.

7.5 Satz. *Algorithmus 7.4 berechnet x^n mit höchstens $2 \log n$ Multiplikationen.*

Beweis: Wir führen den Beweis durch Induktion über n. Für $n = 1$ gilt die Behauptung offensichtlich. Unabhängig davon, ob n gerade oder ungerade ist, der Algorithmus verwendet im rekursiven Aufruf nach Induktionsvoraussetzung höchstens $2 \log \frac{n}{2}$ viele Multiplikationen und anschließend höchstens zwei Multiplikationen. Es ist $2 \log \frac{n}{2} + 2 = 2 \log n$. □

Im RSA-System besteht die Chiffrierung und Dechiffrierung jedes Nachrichtenblocks in der Berechnung einer Potenz $m^e \bmod n$ bzw. $y^d \bmod n$. Dabei können e und d bis zu 600-stellige Binärzahlen sein. Nach Satz 7.5 ist die Chiffrierung und Dechiffrierung mit je höchstens 1200 Multiplikationen und Divisionen mit Rest von Zahlen der Länge 1200 möglich. Hier zeigt sich auch, dass Multiplikations- und Divisionsverfahren, die erst für große Zahlen andere Verfahren schlagen, ihre Bedeutung in der Praxis haben.
Der in Kapitel 7.6 zu besprechende Primzahltest wird u.a. für die zu testende Zahl n und Zufallszahlen a entscheiden, ob sie einen gemeinsamen Teiler größer als 1 haben. Dies geschieht durch die Berechnung des größten gemeinsamen Teilers $\mathrm{ggT}(a, n)$ der Zahlen a und n. Mit dem Euklidischen Algorithmus steht seit langem ein effizienter Algorithmus zur Berechnung des größten gemeinsamen Teilers zur Verfügung.

7.6 Algorithmus. *(Euklidischer ggT–Algorithmus)*
Eingabe: a, b mit $a \geq b \geq 1$.
$ggT(a, b)$
begin
  $c := a \bmod b$. *(Dann ist $0 \leq c \leq b-1$.)*
  if $c = 0$ **then return** b **else return** $ggT(b, c)$.
end

Diesen Algorithmus kennt man schon aus der Schule oder aus dem ersten Semester, wo auch die Korrektheit des Algorithmus nachgewiesen wird. Wir wollen daher nur die Laufzeit dieses Algorithmus analysieren.

7.7 Satz. *Der Euklidische Algorithmus berechnet den größten gemeinsamen Teiler von a und b mit $O(\log(a+b))$ arithmetischen Operationen. Die Zahl der wesentlichen Operationen, d.h. die Zahl der modulo–Operationen, ist nicht größer als $\lfloor \log_{3/2}(a+b) \rfloor$.*

Beweis: Durch Induktion über $a+b$. Wenn $a+b \leq 2$ ist, dann folgt $a = b = 1$ und die Aussage gilt trivialerweise. Nun weisen wir nach, dass $(b+c) \leq \frac{2}{3}(a+b)$ gilt:

$$\left.\begin{array}{l} \text{Wegen } c \leq a-b \text{ folgt } a \geq b+c. \\ \text{Es ist } b = \frac{2b}{2} \geq \frac{b+c}{2} \text{ wegen } b \geq c \text{ und somit } b \geq \frac{b+c}{2}. \end{array}\right\} \Rightarrow\ a+b \geq \tfrac{3}{2} \cdot (b+c).$$

Der Aufruf $\mathrm{ggT}(b,c)$ kommt also nach Induktionsvoraussetzung mit $\lfloor \log_{3/2}((2/3)\cdot(a+b)) \rfloor = \lfloor \log_{3/2}(a+b) \rfloor - 1$ wesentlichen Operationen aus, insgesamt haben wir wegen der Berechnung $c := a \bmod b$ höchstens $\lfloor \log_{3/2}(a+b) \rfloor$ wesentliche Operationen. □

Man kann die Laufzeit noch präziser analysieren (Stichwort: Fibonacci–Folge), aber uns reicht schon die obige „grobe“ Abschätzung. Der Euklidische Algorithmus hat also logarithmische Laufzeit bezogen auf die Größe der beiden Inputs. Üblicherweise beziehen wir die Laufzeit von Algorithmen jedoch auf die Länge der Inputs, hier also auf $\lceil \log(a+1) \rceil + \lceil \log(b+1) \rceil$. Bezogen auf die Länge der Zahlen benötigt der Euklidische Algorithmus lineare Rechenzeit und ist ein effizienter sequentieller Algorithmus. Wenn a und b z. B. 200-stellige Binärzahlen sind, genügen also 341 wesentliche Operationen zur Berechnung von $\mathrm{ggT}(a,b)$.
Nun schauen wir uns an, wie man (multiplikative) Inverse berechnen kann. Aus der Mathematik sollte man wissen, dass das Inverse von b modulo a existiert, wenn $\mathrm{ggT}(a,b) = 1$ ist.

7.8 Algorithmus. *(Berechnung von Inversen)*
Eingabe: Natürliche Zahlen $a \geq b \geq 1$ mit $ggT(a,b) = 1$.
Ausgabe: Das Inverse $b^{-1} \bmod a$.

```
INV(a, b)
begin
   if b=1 then return 1;
   c :=  a mod b.
   c* :=  INV(b, c).
   return (1−c*a)/b mod a.
end
```

7.9 Satz. *Algorithmus 7.8 berechnet das Inverse $b^{-1} \bmod a$ mit $O(\log(a+b))$ vielen arithmetischen Operationen.*

Beweis: Zunächst beweisen wir, dass tatsächlich das multiplikative Inverse berechnet wird. Wir machen eine Induktion über den zweiten Parameter, b. Falls $b = 1$ ist, dann wird die 1 als das Inverse korrekt zurückgeliefert. Sei also $b > 1$.
Der Algorithmus liefert für $w := \frac{1-c^*a}{b}$ die Zahl $w \bmod a$ zurück. Da $w \cdot b \equiv 1 \bmod a$ ist, müssen wir nur noch zeigen, dass w eine ganze Zahl ist, dann ist klar, dass $w \bmod a$ das Inverse von b modulo a ist.

Nach Wahl von c gibt es ein k, so dass wir schreiben können $a = k \cdot b + c$. Die Zahl c ist größer als 0, denn ansonsten wäre a ein Vielfaches von b und somit wegen $b > 1$ der ggT von a und b größer als 1. Da $\text{ggT}(a, b) = \text{ggT}(b, c) = 1$ ist, $c \geq 1$ und $b \geq c$ ist, ist der Aufruf $\text{INV}(b, c)$ zulässig und c^* ist nach Induktionsvoraussetzung das Inverse $c^{-1} \bmod b$. Also gibt es ein k', so dass man schreiben kann $c \cdot c^* = k' \cdot b + 1$. Wir rechnen aus:

$$\begin{aligned} 1 - c^* \cdot a &= 1 - c^* \cdot k \cdot b - c^* \cdot c \\ &= -c^* \cdot k \cdot b - k' \cdot b \\ &= -b \cdot (c^* \cdot k + k'). \end{aligned}$$

Damit ist $w = -(c^* \cdot k + k')$ eine ganze Zahl, was zu zeigen war. Die Laufzeit weist man wie beim ggT–Algorithmus nach. □

Betrachten wir ein Beispiel zur Inversenberechnung:

Beispiel: $\text{INV}(11, 5)$. Der ggT der beiden Zahlen ist 1, wir können den Algorithmus also starten. Zunächst ist $c = 1$ und durch den Aufruf $c^* :=\text{INV}(5, 1)$ wird $c^* = 1$. Dann wird $\frac{1-11}{5} \bmod 11 = -2 \bmod 11 = 9$ als Inverses zurückgeliefert. Wir überprüfen: $9 \cdot 5 = 45 \equiv 1 \bmod 11$ und somit ist in der Tat $5^{-1} \bmod 11 = 9$.

7.3 Zahlentheoretische Grundlagen

Bevor wir den Algorithmus von Solovay und Strassen (1977) beschreiben, der eine Zahl daraufhin testet, ob sie prim ist, wollen wir die Erinnerung des Lesers und der Leserin an einige zahlentheoretische Grundkenntnisse auffrischen. Auch führen wir einige eventuell unbekannte Begriffe ein.

7.10 Definition.

- $\mathbb{Z}_n$ *ist die Menge der Zahlen* $\{0, \ldots, n-1\}$.
- $\mathbb{Z}_n^*$ *ist die Menge aller Zahlen* a *aus* $\{1, \ldots, n-1\}$, *die teilerfremd zu* n *sind, für die also* $ggT(a, n) = 1$ *gilt.*
- *Für eine Zahl* $a \in \mathbb{Z}_n^*$ *bezeichnet* $ord_n(a)$ *die Ordnung von* a, *also*

$$ord_n(a) := \min\{i \geq 1 \mid a^i \equiv 1 \bmod n\}.$$

- *Eine Zahl* $a \in \mathbb{Z}_n^*$ *ist ein* quadratischer Rest modulo n, *Notation* $a \in QR(n)$, *falls die Gleichung* $x^2 \equiv a \bmod n$ *eine Lösung* $x \in \mathbb{Z}_n^*$ *hat.*
- *Ein Element* $g \in \mathbb{Z}_n^*$ *heißt* erzeugendes Element *von* $\mathbb{Z}_n^*$, *wenn* $\mathbb{Z}_n^* = \{g^i \mid 0 \leq i\}$ *ist.*
- *Wenn* g *ein erzeugendes Element ist, dann ist der Index* $index_g(a)$ *das kleinste* $i \geq 0$, *so dass* $g^i = a$ *ist.*

- *Eine Zahl n heißt quadratfrei, wenn in der Primfaktorzerlegung von n keine Primzahl mit einem Exponenten größer als 1 vorkommt.*
- *Die Zahl $\lambda(m)$ bezeichnet das kleinste $i \geq 1$, so dass für alle $a \in \mathbb{Z}_m^*$ gilt, dass $a^i \equiv 1 \bmod m$ ist. $\lambda(\cdot)$ wird auch manchmal als Carmichael–Funktion bezeichnet.*

Zahlen $a \in \mathbb{Z}$ heißen quadratischer Rest modulo n, wenn es ein $x \in \mathbb{Z}_n^*$ gibt mit $a \equiv x^2 \bmod n$. Dies ist natürlich genau dann der Fall, wenn $a \bmod n$ quadratischer Rest ist. Wir bemerken außerdem, dass $\mathbb{Z}_p^*$ immer ein erzeugendes Element besitzt, wenn p eine Primzahl ist.

7.11 Beispiel. *2 ist ein quadratischer Rest modulo 7, da $3^2 \equiv 2 \bmod 7$. Die Zahl 14 ist quadratfrei, die Zahl 50 nicht. Es ist $QR(9) = \{1, 4, 7\}$. 3 ist ein erzeugendes Element von $\mathbb{Z}_5^*$, da $\{3, 3^2 \bmod 5, 3^3 \bmod 5, 3^4 \bmod 5\} = \{3, 4, 2, 1\}$. Es ist $index_3(4) = 2$ in $\mathbb{Z}_5^*$. Es ist $\lambda(9) = 6$.*

In der Zahlentheorie kennt man folgende (elementar zu beweisende) Aussagen für die Carmichael–Funktion $\lambda(\cdot)$:

7.12 Satz.
- *$\lambda(p^e) = p^{e-1} \cdot (p-1)$, wenn p eine ungerade Primzahl ist und $e \geq 1$.*
- *$\lambda(2) = 1, \lambda(4) = 2$ und $\lambda(2^e) = 2^{e-2}$ für $e \geq 3$*
- *$\lambda(p_1^{\alpha_1} \cdots p_r^{\alpha_r}) = \mathrm{kgV}_{i=1\ldots r}\ \lambda(p_i^{\alpha_i})$ (wenn $p_i \neq p_j$ für $i \neq j$)*

Hier einige weitere Eigenschaften:

7.13 Satz. *Wenn $p \geq 3$ eine Primzahl ist, dann ist $|QR(p)| = |\mathbb{Z}_p^*|/2 = \frac{p-1}{2}$.*

Beweis: Es ist $x^2 \equiv y^2 \bmod p \Leftrightarrow (x-y)\cdot(x+y) \equiv 0 \bmod p$. Wegen der Nullteilerfreiheit folgt $x \equiv y \bmod p$ oder $x \equiv -y \bmod p$. Damit ist die Menge der quadratischen Reste genau die Menge

$$\{x^2 \mid x = 1, \ldots, \frac{p-1}{2}\}.$$

□

7.14 Satz. *Sei $a \in \mathbb{Z}_n^*$. Es gilt $a^e \equiv 1 \bmod n \Leftrightarrow e$ ist Vielfaches von $ord_n(a)$.*

Beweis: Zerlege $e = k \cdot ord_n(a) + j$ mit $0 \leq j < ord_n(a)$.
Es ist $a^{k \cdot ord_n(a)+j} \equiv (a^{ord_n(a)})^k \cdot a^j \equiv 1 \cdot a^j \bmod n$. Dies ist äquivalent zu 1 modulo n genau dann, wenn $a^j \equiv 1 \bmod n$. Da $j < ord_n(a)$, folgt aus der Definition der Ordnung, dass $j = 0$ ist. □

Auch bekannt sein dürfte:

7.15 Satz. *(Kleiner Satz von Fermat) Sei p eine Primzahl. Für alle $a \in \mathbb{Z}_p^*$ gilt $a^{p-1} \equiv 1 \bmod p$.*

Man kann auch den Satz von Fermat schon benutzen, um für manche Zahlen nachzuweisen, dass sie nicht prim sind. Zum Beispiel kann man mit der effizienten Potenzierungsmethode schnell ausrechnen, dass $2^{50} \equiv 4 \bmod 51$ ist. Aus dem Satz von Fermat folgt sofort, dass damit 51 keine Primzahl sein kann. Die Zahl $a = 2$ kann man also quasi als „Zeugen“ betrachten, die nachweist, dass $n = 51$ keine Primzahl ist. Dummerweise gibt es Zahlen, die keine Primzahl sind und für die kein derartiger Zeuge (nach dem kleinen Satz von Fermat) existiert. Dies sind die sogenannten Carmichael–Zahlen:

7.16 Definition. *Eine Zahl $n \geq 2$ heißt „Carmichael–Zahl“, wenn n keine Primzahl ist und für alle $a \in \mathbb{Z}_n^*$ gilt, dass $a^{n-1} \equiv 1 \bmod n$ ist.*

Der Primzahltest von Solovay und Strassen wird wegen der Existenz solcher Carmichael–Zahlen auf einer Verschärfung des kleinen Satzes von Fermat basieren.
Übrigens ist die kleinste Carmichael–Zahl $561 = 3 \cdot 11 \cdot 17$, und unter allen Zahlen, die kleiner als 10^{12} sind, gibt es gerade mal 8241 Carmichael–Zahlen. Erst 1994 wurde bewiesen, dass es unendlich viele Carmichael–Zahlen gibt. Man kann Carmichael–Zahlen wie folgt charakterisieren:

7.17 Satz. *Eine zusammengesetzte Zahl n ist Carmichael–Zahl genau dann, wenn $n-1$ ein Vielfaches von $\lambda(n)$ ist.*

Beweis: „$\Leftarrow$“: Sei $n-1 = k \cdot \lambda(n)$, dann ist $a^{n-1} \equiv (a^{\lambda(n)})^k \equiv 1 \bmod n$.
„$\Rightarrow$“: Aus Satz 7.14 ergibt sich:
$a^e \equiv 1 \bmod n$ für alle $a \in \mathbb{Z}_n^*$ gilt genau dann, wenn e Vielfaches von $ord_n(a)$ für alle $a \in \mathbb{Z}_n^*$ ist. Dies ist genau dann der Fall, wenn e Vielfaches des kgV aller $ord_n(a)$ ist.
Das kleinste gemeinsame Vielfache aller $ord_n(a)$ ist aber gerade $\lambda(n)$.
Wenn n eine Carmichael-Zahl ist, dann ist also $n-1$ ein Vielfaches von $\lambda(n)$. □

Folgende Eigenschaft der Carmichael–Zahlen werden wir später beim Beweis der Korrektheit des Primzahltests benötigen.

7.18 Satz. *Wenn n eine Carmichael–Zahl ist, dann ist n ungerade und quadratfrei.*

Beweis: Wenn $n > 2$ ist, dann ist $ord_n(n-1) = ord_n(-1) = 2$, somit ist $\lambda(n)$ ein Vielfaches von 2. Nach Satz 7.17 ist somit auch $n-1$ gerade, also n ungerade.
Angenommen, n ist nicht quadratfrei, dann gibt es eine ungerade Primzahl p mit „p^2 teilt n“. Aus Satz 7.12 folgt, dass p die Zahl $\lambda(n)$ teilt. Also (Satz 7.17) teilt p auch $n-1$.
Die Zahl p kann aber nicht gleichzeitig Teiler von n und $n-1$ sein, ein Widerspruch. □

Man kann auch zeigen, dass jede Carmichael–Zahl durch mindestens drei verschiedene Primzahlen teilbar ist (z.B. $561 = 3 \cdot 11 \cdot 17$), diese Eigenschaft benötigen wir jedoch nicht.

7.4 Das Jacobi-Symbol

Nachdem wir uns einige einfache Eigenschaften zurechtgelegt haben, kommen wir zur Definition des Jacobi-Symbols und zu dem Problem, dieses algorithmisch effizient zu berechnen.

7.19 Definition. (Jacobi-Symbol)

i) Für Primzahlen p und $a \in \mathbb{Z}$ mit $ggT(a,p)=1$ ist das Jacobi–Symbol $\left(\frac{a}{p}\right)$ wie folgt definiert:

$$\left(\frac{a}{p}\right) = \begin{cases} +1, & \text{falls } a \text{ quadratischer Rest modulo } p \text{ ist.} \\ -1, & \text{sonst.} \end{cases}$$

ii) Für ungerade Zahlen $q \geq 3$ mit der Primfaktorzerlegung $q = q_1 \cdots q_k$ (Achtung: q_i=q_j für $i \neq j$ möglich) ist das Jacobi-Symbol $\left(\frac{a}{q}\right)$ von a bzgl. q (mit $ggT(a,q) = 1$) definiert als

$$\left(\frac{a}{q}\right) = \left(\frac{a}{q_1}\right) \cdot \left(\frac{a}{q_2}\right) \cdots \left(\frac{a}{q_k}\right).$$

7.20 Beispiel. *Zum Beispiel ist $\left(\frac{2}{7}\right) = +1$, weil $2 \in QR(7)$ ist, denn $3^2 = 9 \bmod 7 = 2 \bmod 7$. Auch ist $\left(\frac{2}{3}\right) = -1$ einfach einzusehen, da $QR(3) = \{1\}$ ist. Aus beiden zusammen folgt $\left(\frac{2}{21}\right) = -1$.*

Wenn n eine Primzahl ist, dann wird das Symbol $\left(\frac{a}{n}\right)$ auch als „Legendre-Symbol“ bezeichnet.
Wir werden einen effizienten Algorithmus zur Berechnung des Jacobi-Symbols entwerfen. Der Primzahltest wird für die zu testende Zahl n und eine Zufallszahl a mit $\mathrm{ggT}(a,n) = 1$ das Jacobi-Symbol $\left(\frac{a}{n}\right)$ mit dem effizienten Algorithmus berechnen und dies mit der Zahl $a^{\frac{n-1}{2}}$ vergleichen. Für Primzahlen n liefern beide Rechnungen das gleiche Resultat. Für zusammengesetzte Zahlen hängt es von a ab, ob beide Rechnungen das gleiche Resultat liefern. Wenn also die Ergebnisse verschieden sind, muss n eine zusammengesetzte Zahl sein.
Wir listen nun Rechenregeln für das Jacobi-Symbol auf. Auf diesen Rechenregeln wird der effiziente Algorithmus zur Berechnung des Jacobi-Symbols beruhen.

7.21 Lemma. *Für ungerade Zahlen n und $a, b \in \mathbb{Z}_n^*$ gilt*

$$\left(\frac{ab}{n}\right) = \left(\frac{a}{n}\right)\left(\frac{b}{n}\right).$$

Ein berühmtes Resultat aus der Mathematik ist das sogenannte „Quadratische Reziprozitätsgesetz“. Für dieses Ergebnis gibt es eine Unmenge an verschiedenartigen Beweisen. Wir benötigen das Reziprozitätsgesetz in der folgenden Form.

7.22 Satz. *Es seien $m, n \geq 3$ ungerade und $ggT(m,n) = 1$.*

i) $\left(\frac{m}{n}\right)\left(\frac{n}{m}\right) = (-1)^{(m-1)(n-1)/4}$.

ii) $\left(\frac{2}{n}\right) = (-1)^{(n^2-1)/8}$.

7.23 Lemma. *Für ungerade n ist $\left(\frac{2}{n}\right) = 1$ genau dann, wenn $n \bmod 8 \in \{1,7\}$ ist und -1 sonst.*

Beweis: Um zu entscheiden, ob $(-1)^{(n^2-1)/8}$ gleich 1 oder gleich -1 ist, muss man nur wissen, ob $(n^2-1)/8$ gerade oder ungerade ist. Die Zahl $(n^2-1)/8$ ist genau dann gerade, wenn n^2-1 durch 16 teilbar ist. Dies ist genau dann der Fall, wenn $n \bmod 16 \in \{1,7,9,15\}$ ist. Und dies ist äquivalent zu $n \bmod 8 \in \{1,7\}$. Man braucht also nur die letzten 3 Bits in der Binärdarstellung von n, um t aus n zu berechnen. □

Der Leser und die Leserin mögen sich überlegen, dass man den Ausdruck $(-1)^{(m-1)(n-1)/4}$ mit Hilfe ähnlicher Überlegungen berechnen kann.
Nun haben wir alle Hilfsmittel bereitgestellt, um einen effizienten Algorithmus zur Berechnung des Jacobi-Symbols zu entwerfen.

7.24 Algorithmus. *(Berechnung des Jacobi-Symbols)*
Input: $q^* > 1$ *ungerade und* p^* *mit* $ggT(p^*, q^*) = 1$.
Output: $s = \left(\frac{p^*}{q^*}\right)$.
Setze $s := 1$; $p := p^*$; *und* $q := q^*$.
Kommentar: Es soll stets $\left(\frac{p^*}{q^*}\right) = s \cdot \left(\frac{p}{q}\right)$ *gelten.*
while $p \neq 1$ **do**
begin
1.) $p :=$ $p \bmod q$.
2.) *Falls* $q \bmod 8 \in \{1,7\}$, *dann* $t := 1$, *sonst* $t := -1$. *#Damit ist* $t = \left(\frac{2}{q}\right)$.
3.) **while** p *gerade* **do begin** $p := p/2; s := t \cdot s;$ **end;**
4.) *Falls* $p \neq 1$, *vertausche* p *mit* q *und setze* $s := s \cdot (-1)^{(p-1)(q-1)/4}$.
end;
5.) Output s;

Wir untersuchen nun die Korrektheit und Laufzeit des gerade beschriebenen Algorithmus.

7.25 Satz. *Algorithmus 7.24 berechnet das Jacobi-Symbol* $\left(\frac{p^*}{q^*}\right)$ *in* $O(\log(p^* + q^*))$ *Schritten.*

Beweis: Wir zeigen zunächst induktiv, dass die Beziehung $\left(\frac{p^*}{q^*}\right) = s \cdot \left(\frac{p}{q}\right)$ nach jedem Schritt des Algorithmus gilt.
Schritt 1: Es sei $\hat{p} \equiv p \bmod q$. $\left(\frac{p}{q}\right) = \left(\frac{\hat{p}}{q}\right)$ folgt direkt aus der Definition des Jacobi–Symbols.
Schritt 2 und 3: Wie oben angedeutet ist $t = \left(\frac{2}{q}\right)$. In Schritt 3 wird dann die Eigenschaft $\left(\frac{2x}{q}\right) = \left(\frac{2}{q}\right) \cdot \left(\frac{x}{q}\right)$ ausgenutzt.
Schritt 4: p ist nun ungerade. Die Zahl q wird nur in Schritt 4 verändert. Da q zu Beginn ungerade ist und p nach Schritt 3 ungerade ist, bleibt q stets ungerade. In Schritt 4 sind p und q ungerade und es kann die Aussage i) aus Satz 7.22 in der Form

$$\left(\frac{p}{q}\right) = \left(\frac{q}{p}\right) \cdot (-1)^{(p-1)(q-1)/4}$$

angewendet werden.
Schritt 5: Da $p = 1$, ist $\left(\frac{p^*}{q^*}\right) = s\left(\frac{1}{q}\right)$. Da $1 \in QR(q_i)$ für alle Primfaktoren q_i von q ist, ist $\left(\frac{1}{q}\right) = 1$.

Wenn der Algorithmus stoppt, ist also $s = \left(\frac{p^*}{q^*}\right)$.
Nun zur Laufzeit: Die Summe $p + q$ wird während des Algorithmus nur verkleinert. In Schritt 4 und 1 wird das Paar (p, q) mit $p < q$ durch das Paar $(q \bmod p, p)$ ersetzt. Dies ist genau ein Iterationsschritt des Euklidischen Algorithmus zur Berechnung von $\mathrm{ggT}(p, q)$. Da q in Schritt 3 stets ungerade ist, wird der größte gemeinsame Teiler von p und q durch das Abspalten der Faktoren 2 in p nicht verändert. Da nach Voraussetzung $\mathrm{ggT}(p, q) = 1$ ist, würde der Euklidische Algorithmus mit $p = 0$ und $q = 1$ stoppen. Ein Iterationsschritt vorher ist $p = 1$ und $q > 1$. Algorithmus 7.24 stoppt in dieser Situation und produziert die Ausgabe $\left(\frac{p^*}{q^*}\right)$.
Nach Satz 7.7 wird die Schleife weniger als $\lfloor \log_{3/2}(p^* + q^*) \rfloor$-mal durchlaufen. Die Schritte 1, 2 und 4 verursachen jeweils Kosten $O(1)$. In Schritt 3 wird insgesamt (über alle Iterationen betrachtet) höchstens $(\lfloor \log p^* \rfloor + \lfloor \log q^* \rfloor)$-mal ein Faktor 2 von den Zahlen abgespaltet, also sind auch die Gesamtkosten von Schritt 3 durch $O(\log(p^* + q^*))$ beschränkt. □

7.5 Der Chinesische Restsatz

7.26 Satz. *Gegeben natürliche Zahlen $m_1, \ldots, m_k$ mit $ggT(m_i, m_j) = 1$ für $i \neq j$. Gegeben außerdem natürliche Zahlen $a_1, \ldots, a_k$. Das folgende Gleichungssystem hat genau eine Lösung x mit $x \in \{0, \ldots, m_1 \cdot m_2 \cdots m_k - 1\}$:*

$$\begin{aligned} x &\equiv a_1 \bmod m_1 \\ x &\equiv a_2 \bmod m_2 \\ &\cdots \\ x &\equiv a_k \bmod m_k. \end{aligned}$$

Wir beweisen diesen Satz, indem wir ein Verfahren angeben, das die Lösung x berechnet.
Beweis: Es sollte klar sein, dass wir uns auf den Fall beschränken können, wo für alle a_i gilt, dass $0 \leq a_i < m_i$ ist. Es gibt $m_1 \cdots m_k$ viele solche Gleichungssysteme. Kein x kann zwei solcher Gleichungssysteme erfüllen. Wenn wir also nachweisen, dass für jedes Gleichungssystem mindestens eine Lösung x existiert, dann ist wegen $|\{0, \ldots, m_1 \cdot m_2 \cdots m_k - 1\}| = m_1 \cdots m_k$ klar, dass jedes Gleichungssystem *genau* eine Lösung x hat. Wir zeigen die Existenz einer Lösung nun durch Induktion über k. Falls k=1 ist, so ist $a_1 \bmod m_1$ eine Lösung. Für den Induktionsschritt nehmen wir an, dass wir eine Zahl L haben mit $0 \leq L \leq m_1 \cdots m_{k-1} - 1$, die das folgende Gleichungssystem, das wir (*)

nennen wollen, erfüllt:

$$\begin{array}{rcll} x & \equiv & a_1 & \bmod m_1 \\ x & \equiv & a_2 & \bmod m_2 \\ & \cdots & & \\ x & \equiv & a_{k-1} & \bmod m_{k-1} \end{array}$$

Wir zeigen, wie man eine Zahl L^* findet, die das Gleichungssystem (*) erfüllt und für die zusätzlich $L^* \equiv a_k \bmod m_k$ gilt.
Da L eine Lösung des Gleichungssystems (*) ist, sind auch alle $L+t\cdot m_1 \cdots m_{k-1}$ Lösungen des Gleichungssystems (*). Wir suchen ein t, so dass zusätzlich $L+t\cdot m_1 \cdots m_{k-1} \equiv a_k \bmod m_k$ gilt. Man wählt ein $0 \leq t \leq m_k - 1$ mit

$$t \equiv (a_k - L) \cdot (m_1 \cdots m_{k-1})^{-1} \bmod m_k.$$

Diese Zahl existiert, da das Inverse von $(m_1 \cdots m_{k-1})$ modulo m_k existiert, da alle m_i relativ prim zueinander sind.
Mit dieser Wahl von t gilt also, dass $L^* := L + t \cdot m_1 \cdots m_{k-1}$ Lösung des Gleichungssystems (*) ist und zusätzlich $L^* \equiv a_k \bmod m_k$ erfüllt ist. Da $0 \leq t \leq m_k - 1$ ist, ist $0 \leq L^* \leq m_1 \cdots m_k - 1$. □

Beispiel: Wir wollen das Gleichungssystem

$$\begin{array}{rcl} x & \equiv & 1 \bmod 2 \\ x & \equiv & 2 \bmod 3 \\ x & \equiv & 9 \bmod 11 \end{array}$$

lösen. Angenommen, wir kennen schon die Lösung $L = 5$, die die ersten beiden Gleichungen erfüllt. Wir berechnen:

$$t \equiv (9 - 5) \cdot (2 \cdot 3)^{-1} \bmod 11.$$

Das Inverse von 6 modulo 11 ist 2, also erhalten wir $t \equiv 4 \cdot 2 \bmod 11 \equiv 8 \bmod 11$. Damit errechnet sich $L^* = 5+8\cdot(2\cdot 3) = 53$ und wir haben die Lösung 53 des Gleichungssystems gefunden.

7.6 Der probabilistische Primzahltest von Solovay und Strassen

Der hier vorgestellte probabilistische Primzahltest wurde von Solovay und Strassen (1977) entworfen. Der Parameter k gibt die Zahl der Tests an, die auf die Eingabezahl angewendet werden. Je höher k gewählt wird, umso geringer wird die Irrtumswahrscheinlichkeit des Algorithmus.
Der Primzahltest nutzt die Eigenschaften aus dem folgenden Satz aus, den wir in mehreren Schritten beweisen werden:

7.27 Satz. *Sei $n \geq 3$ ungerade und $E(n) := \{a \in \mathbb{Z}_n^* \mid a^{\frac{n-1}{2}} \equiv \left(\frac{a}{n}\right)\}$. Dann gilt:*

$$\begin{array}{rcl} n \text{ prim} & \Rightarrow & E(n) = \mathbb{Z}_n^* \\ n \text{ nicht prim} & \Rightarrow & |E(n)| \leq |\mathbb{Z}_n^*|/2 \end{array}$$

Wir beginnen mit dem Beweis der Aussage für n prim:

7.28 Satz. *Für Primzahlen $p \neq 2$ und $a \in \mathbb{Z}_p^*$ gilt $a^{(p-1)/2} \equiv \left(\frac{a}{p}\right) \bmod p$, es ist also $E(p) = \mathbb{Z}_p^*$.*

Beweis: Da p ungerade ist, ist $(p-1)/2$ eine ganze Zahl und der Ausdruck $a^{(p-1)/2}$ wohldefiniert.
Nach dem kleinen Satz von Fermat ist $a^{p-1} \equiv 1 \bmod p$. Die Zahl 1 hat aber modulo p nur zwei Wurzeln, nämlich 1 und -1 (siehe der Beweis zu Satz 7.13). Da $a^{(p-1)/2}$ eine Wurzel von 1 ist, ist somit gezeigt, dass $a^{(p-1)/2} \equiv 1 \bmod p$ oder $a^{(p-1)/2} \equiv -1 \bmod p$ ist.
Da $\left(\frac{a}{p}\right) = 1$ genau dann gilt, wenn a ein quadratischer Rest modulo p ist, reicht es zum Beweis dieses Satzes aus, zu zeigen:

$$a \text{ ist quadratischer Rest modulo } p \Leftrightarrow a^{\frac{p-1}{2}} \equiv 1 \bmod p.$$

„$\Rightarrow$“: Dann ist $a \equiv x^2 \bmod p$ für ein $x \in \mathbb{Z}_p^*$ und

$$a^{\frac{p-1}{2}} \equiv (x^2)^{\frac{p-1}{2}} \equiv x^{p-1} \equiv 1 \bmod p.$$

„$\Leftarrow$“: Da p eine Primzahl ist, existiert ein erzeugendes Element g modulo p. Wir können schreiben:

$$1 \equiv a^{\frac{p-1}{2}} \equiv \left(g^{index_g(a)}\right)^{\frac{p-1}{2}} \bmod p,$$

also $g^{index_g(a) \cdot \frac{p-1}{2}} \equiv 1 \bmod p$, also muss $index_g(a) \cdot \frac{p-1}{2}$ ein Vielfaches von der Ordnung von g sein, also ein Vielfaches von $p-1$. Also ist $index_g(a) = 2 \cdot k$ für ein k und somit ist $(g^k)^2 \equiv a \bmod p$ und daher ist a ein quadratischer Rest. □

Wir beschreiben zunächst den Algorithmus und beweisen dann seine Korrektheit.

7.29 Algorithmus. *(von Solovay und Strassen (1977))*
Input: $n \geq 3,\ k \geq 1$.
Output: „n ist vermutlich Primzahl“ oder „n ist keine Primzahl“.

Falls n gerade: Output „n ist keine Primzahl“, STOP.
for $i := 1$ **to** k **do**
begin
Wähle Zufallszahl $a \in \{1, \ldots, n-1\}$ *gemäß der Gleichverteilung.*
Berechne $d := ggT(a, n)$.
Falls $d \neq 1$*: Output „n ist keine Primzahl“, STOP.*
Berechne $b \equiv a^{(n-1)/2} \bmod n$.
Falls $b \not\equiv \{-1, 1\} \bmod n$*: Output „n ist keine Primzahl“, STOP.*
(*) Berechne $c := \left(\frac{a}{n}\right)$*. (Das Jacobi-Symbol.)*
(*) Falls $b \not\equiv c \bmod n$*: Output „n ist keine Primzahl“, STOP.*
end
Output „n ist vermutlich Primzahl“.

7.30 Satz. *i) Algorithmus 7.29 ist in Zeit* $O(k \log n)$ *und damit für konstantes* k *in linearer Zeit bezogen auf die Inputlänge durchführbar.*

ii) Für Primzahlen $n \geq 3$ *liefert Algorithmus 7.29 den Output „n ist (vermutlich) Primzahl“.*

iii) Für zusammengesetzte Zahlen n liefert Algorithmus 7.29 den Output „n ist keine Primzahl“ mit einer Wahrscheinlichkeit von mindestens $1 - (1/2)^k$.

Das folgende Lemma ist die zentrale Aussage, die im Algorithmus von Solovay und Strassen ausgenutzt wird.

7.31 Lemma. *Sei* $n \geq 3$ *ungerade. Dann ist*

$$E(n) = \mathbb{Z}_n^* \Leftrightarrow \quad n \text{ ist prim.}$$

Beweis: Die Richtung „$\Leftarrow$“ gilt nach Satz 7.28. Für die andere Richtung („Widerspruchsbeweis“) nehmen wir an, dass $E(n) = \mathbb{Z}_n^*$, aber n zusammengesetzt ist. Da $\left(\frac{a}{n}\right)^2 \equiv 1 \bmod n$ gilt, ist nach Definition n eine Carmichael–Zahl. Nach Lemma 7.18 ist n also quadratfrei und wir können schreiben $n = p \cdot r$ für eine Primzahl p und ein $r > 1$ mit $\mathrm{ggT}(r, p) = 1$. Sei g ein quadratischer Nichtrest modulo p. (So einen gibt es nach Satz 7.13.) Nach dem Chinesischen Restsatz gibt es ein $a \in \mathbb{Z}_n$, so dass

$$a \equiv g \bmod p \quad \text{und} \quad a \equiv 1 \bmod r.$$

Natürlich ist dann $\mathrm{ggT}(a, n) = 1$ und somit $a \in \mathbb{Z}_n^*$.
Wir zeigen nun, dass $a \notin E(n)$ ist: Nach der Definition des Jacobi-Symbols ist $\left(\frac{a}{n}\right) = \left(\frac{a}{pr}\right) = \left(\frac{a}{p}\right) \cdot \left(\frac{a}{r}\right) = \left(\frac{g}{p}\right) \cdot \left(\frac{1}{r}\right) = (-1) \cdot (+1) = -1$.

Andererseits ist $a^{\frac{n-1}{2}} \equiv 1 \bmod r$ und da $a^{\frac{n-1}{2}} \bmod n$ nur den Wert 1 oder -1 haben kann, folgt aus dem chinesischen Restsatz, dass $a^{\frac{n-1}{2}} \equiv 1 \bmod n$ sein muss. □

Beweis: (von Satz 7.30) Zu i): Wir haben in den Sätzen 7.5, 7.7 und 7.25 gezeigt, dass die einzelnen Schritte des Algorithmus für jedes i in $O(\log n)$ Schritten durchführbar sind.
Zu ii): Der Output „n ist zusammengesetzt“ wird nur erzeugt, wenn n gerade ist, wenn $\mathrm{ggT}(a_i, n) \neq 1$ für ein $i \in \{1, \ldots, k\}$ oder $a_i^{(n-1)/2} \not\equiv \left(\frac{a_i}{n}\right) \bmod n$ für ein $i \in \{1, \ldots, k\}$ ist. Für Primzahlen ist (s. Satz 7.28) keine dieser Bedingungen erfüllt.
Zu iii): Für gerades n ist die Wahrscheinlichkeit, dass der Output „n ist zusammengesetzt“ erzeugt wird, sogar 1. Falls $\mathrm{ggT}(a_i, n) \neq 1$ für eine der Zufallszahlen a_i ist, so wird n ebenfalls als zusammengesetzt entlarvt. Ansonsten sind die Zufallszahlen $a_1, \ldots, a_k$ alle in $\mathbb{Z}_n^*$ enthalten.
$E(n)$ kann gesehen werden als die Menge der Zahlen, die als Testzahlen die Eingabe n nicht als zusammengesetzt entlarven. Wir zeigen, dass $|E(n)| \leq |\mathbb{Z}_n^*|/2$ für zusammengesetzte Zahlen n ist. Die Wahrscheinlichkeit, dass alle zufällig gewählten Testzahlen $a_1, \ldots, a_k$ schlecht, also in $E(n)$ enthalten sind, ist dann höchstens $(1/2)^k$.
Es ist eine elementare Aussage der Theorie endlicher Gruppen, dass die Ordnung einer Untergruppe (Anzahl der Elemente der Untergruppe) die Ordnung der Gruppe teilt. $\mathbb{Z}_n^*$ ist (s. Kapitel 7.2) eine Gruppe bezüglich der Multiplikation modulo n. Wir zeigen, dass $E(n)$ eine echte Untergruppe von $\mathbb{Z}_n^*$ ist. Da kein echter Teiler von $|\mathbb{Z}_n^*|$ größer als $|\mathbb{Z}_n^*|/2$ ist, folgt daraus der Satz.
Da n zusammengesetzt ist, folgt aus Lemma 7.31, dass $E(n) \neq \mathbb{Z}_n^*$ ist. Auch ist $E(n)$ nicht leer, da $1 \in E(n)$ ist. Bleibt also die (einfache) Aufgabe, zu zeigen, dass $E(n)$ eine Untergruppe von $\mathbb{Z}_n^*$ ist. Aus der Mathematik kennen wir folgendes einfaches Kriterium: *Eine nichtleere endliche Teilmenge U einer Gruppe G ist genau dann eine Untergruppe, wenn mit allen $a, b \in U$ auch ab in U liegt.*
Sei also $a, b \in E(n)$. Es ist

$$(ab)^{(n-1)/2} = a^{(n-1)/2} b^{(n-1)/2} = \left(\frac{a}{n}\right)\left(\frac{b}{n}\right) = \left(\frac{ab}{n}\right) \bmod n.$$

Die letzte Gleichung gilt dabei wegen Lemma 7.21. □

Fazit

Unser probabilistischer Primzahltest kann zusammengesetzte Zahlen entlarven. Die Antwort „n ist zusammengesetzt“ ist irrtumsfrei. Für Primzahlen kommt es mit Sicherheit zur Aussage „n ist vermutlich Primzahl“. Diese Aussage wird bei Tests mit k Zufallszahlen (z. B. $k = 100$) für zusammengesetzte Zahlen n nur mit der (verschwindend kleinen) Wahrscheinlichkeit $(1/2)^k$ erzeugt. Diese geringe Fehlerwahrscheinlichkeit ist für die meisten kommerziellen Anwendungen kryptographischer Systeme tolerabel. Die Tests für die einzelnen Zufallszahlen können natürlich auch parallel ausgeführt werden.

Für die legalen Benutzer des RSA-Systems stehen also sowohl für die Schlüsselerzeugung als auch für die Chiffrierung und Dechiffrierung der einzelnen Nachrichtenblöcke effiziente Algorithmen zur Verfügung.
Die in diesem Kapitel vorgestellten effizienten Algorithmen für die Elementare Zahlentheorie haben lineare Laufzeit bezogen auf die Länge der eingegebenen Zahlen. Für den Primzahltest gilt dies nur, solange k konstant ist.

Beispiele

Für die Zahl $n = 161$ soll entschieden werden, ob sie Primzahl ist. Wir wollen mit einer Zuverlässigkeitsquote von $1 - 2^{-5} \approx 97\%$ zufrieden sein und setzen $k = 5$. Offensichtlich ist 161 nicht gerade. Wir erzeugen zunächst die Zufallszahl $a = 2$. Der Euklidische Algorithmus liefert uns die Aussage $\mathrm{ggT}(2, n) = 1$. Mit Hilfe unseres effizienten Potenzierungsalgorithmus berechnen wir $2^{(n-1)/2} = 2^{80} \equiv 123 \bmod 161$. Dies ist weder äquivalent zu 1 mod 161 noch äquivalent zu -1 mod 161. Damit ist klar, dass 161 keine Primzahl sein kann. (Allerdings hat uns der Test keinen Teiler von 161 verraten.)
Als nächstes testen wir $n = 71$ mit der Folge der Zufallszahlen $a = 2, 3, 5, 9, 10$. Der ggT von a und n ist für alle diese Wahlen gleich 1. Es ergibt sich außerdem, dass für alle diese Zahlen a die Zahl b im Algorithmus gleich 1 ist. Auch für alle Jacobi–Symbole $\left(\frac{a}{n}\right)$ errechnet sich der Wert 1 für die obigen Werte von a.
Wenn unsere Zufallszahlen echte Zufallszahlen wären, könnten wir nun mit gutem Gewissen vermuten, dass 71 eine Primzahl ist. Wäre 71 keine Primzahl, wäre die Wahrscheinlichkeit, unter 5 Zufallszahlen keinen Zeugen für diese Tatsache zu finden, höchstens 1/32.

Variationen

Es gibt eine ganze Reihe von Variationen zum Algorithmus von Solovay und Strassen. Zum Beispiel können wir in dem Algorithmus die beiden mit (*) markierten Zeilen weglassen, wenn $n \equiv 3 \bmod 4$ ist. Wir brauchen dann also nicht mehr das Jacobi–Symbol zu berechnen. Den Grund dafür wollen wir hier skizzieren:
Sei $G(n) := \{x \in \mathbb{Z}_n^* \mid x^{\frac{n-1}{2}} \equiv \pm 1 \bmod n\}$. Die Menge $G(n)$ ist eine Untergruppe von $\mathbb{Z}_n^*$ (Beweis ähnlich wie vorhin) und außerdem trifft folgende Aussage zu:

$$\text{Wenn } n \equiv 3 \bmod 4 \text{ ist, dann gilt:} \quad G(n) = \mathbb{Z}_n^* \iff n \text{ ist prim.}$$

Beweis: Die Richtung „$\Leftarrow$“ folgt aus Satz 7.28.
Wenn n nicht prim ist, dann folgt zunächst, daß n eine Carmichael–Zahl ist. Damit ist n quadratfrei und wir können $n = n_1 \cdot n_2$ schreiben für zwei Zahlen $n_1, n_2 \geq 3$ mit $\mathrm{ggT}(n_1, n_2)$=1. Nach dem Chinesischen Restsatz können wir ein a wählen aus $\mathbb{Z}_n^*$, für das $a \equiv -1 \bmod n_1$ und $a \equiv 1 \bmod n_2$ gilt. n ist äquivalent zu 3 mod 4, also ist $(n-1)/2$ ungerade, also ist $a^{\frac{n-1}{2}} \equiv -1 \bmod n_1$ und $a^{\frac{n-1}{2}} \equiv 1 \bmod n_2$. Da $n_1, n_2 \neq 2$ sind (n ist ungerade), ist $a^{\frac{n-1}{2}}$ also nicht $\equiv \pm 1 \bmod n$, also ist a nicht in $G(n)$, $G(n)$ ist also eine echte Untergruppe von $\mathbb{Z}_n^*$. □

Die Forderung in der obigen Aussage, dass $n \equiv 3 \bmod 4$ sein sollte, ist übrigens essenziell, denn zum Beispiel gilt für die Zahl $n = 1729 = 7 \cdot 13 \cdot 19$, die nicht äquivalent zu 3 modulo 4 ist, gilt $G(n) = \mathbb{Z}_n^*$. Es ist vielleicht eine gute Übungsaufgabe, dies nachzuweisen, ohne für alle Elemente a aus $\mathbb{Z}_n^*$ den Wert $a^{\frac{1729-1}{2}}$ auszurechnen.

8 Anhang

8.1 Schubfachprinzip

Das Schubfachprinzip besagt folgendes:

8.1 Satz. *Wenn man r Zahlen $a_1, \ldots, a_r$ hat, deren Summe S ist, dann gibt es ein i mit $a_i \leq S/r$.*

Beweis: Wenn alle $a_i > S/r$ wären, so wäre $a_1 + \cdots + a_r > (S/r) + \cdots + (S/r) = S$, im Widerspruch dazu, dass die Summe gleich S ist. □

Natürlich kann man analog zeigen, dass es auch ein i mit $a_i \geq S/r$ gibt.
Manchmal hilft auch folgende Aussage:

8.2 Satz. *Gegeben seien r Quotienten $\frac{c_1}{t_1}, \ldots, \frac{c_r}{t_r}$ mit $t_i > 0$ für alle i. Sei S_c die Summe der Zähler, also $S_c := c_1 + \cdots + c_r$ und S_t die Summe der Nenner, also $S_t := t_1 + \cdots + t_r$. Dann gibt es ein i mit $c_i/t_i \leq S_c/S_t$.*

Beweis: Angenommen, $c_i/t_i > S_c/S_t$ für alle i. Dann wäre $c_i > t_i \cdot S_c/S_t$ für alle i und somit

$$\begin{aligned} c_1 + \cdots + c_r &> \frac{t_1 S_c}{S_t} + \cdots + \frac{t_r S_c}{S_t} \\ &= \frac{(t_1 + \cdots + t_r) \cdot S_c}{S_t} = S_c \end{aligned}$$

Also ein Widerspruch. □

8.2 Eine Rekursionsgleichung

Bei der Analyse des Algorithmus FastCut haben wir eine Rekursionsgleichung der Form $t(0) = 1$ und $t(r+1) = t(r) \cdot (1 - \frac{t(r)}{4})$ analysiert und $t(r) \geq 1/(r+1)$ nachgewiesen. Für die Erfolgswahrscheinlichkeit $p(r) \geq t(r)$ folgte dann auch $p(r) = \Omega(1/r)$. War das eine zumindest asymptotisch gute Abschätzung? An dieser Stelle wollen wir die Rekursionsgleichung interessehalber ein wenig genauer analysieren.

8.3 Satz. *Wenn $t(0) = 1$ und $t(r+1) = t(r) \cdot (1 - t(r)/4)$ definiert ist, dann gilt für alle $i \geq 0$:*

$$\frac{1}{1 + (i/3)} \leq t(i) \leq \frac{1}{1 + (i/4)}.$$

Es ist also $t(n) = \Theta(1/n)$.

Beweis: Wir definieren zunächst $q(i) := 1/t(i)$ für alle $i \geq 0$. Dann ist $q(0) = 1$, $q(i) \geq 1$ für alle i und

$$\begin{aligned} q(r+1) &= \frac{1}{t(r+1)} \\ &= \frac{1}{t(r) \cdot (1 - t(r)/4)} \\ &= q(r) \cdot \frac{1}{1 - 1/(4q(r))} \\ &= \frac{q(r)^2}{q(r) - (1/4)} \\ &= q(r) + (1/4) + \frac{1/16}{q(r) - (1/4)} \end{aligned}$$

Die letzte Gleichung ergibt sich, wenn man nach der Schulmethode die ersten beiden Terme des Quotienten ausrechnet. Da $\frac{1/16}{q(r)-(1/4)} = \frac{1}{16q(r)-4} \leq \frac{1}{12}$ ist, folgt $q(r+1) \geq q(r) + (1/4)$ und $q(r+1) \leq q(r) + (1/4) + (1/12) = q(r) + (1/3)$. Induktiv erhalten wir

$$1 + \frac{i}{4} \leq q(i) \leq 1 + \frac{i}{3}$$

und da $t(i) = 1/q(i)$ ist, erhalten wir die Abschätzung aus dem Satz. □

8.3 Zur eindeutigen Dekodierung beim RSA-System

Im folgenden sei $n = p \cdot q$ für zwei verschiedene Primzahlen p und q. Außerdem seien e und d zwei Zahlen mit $e \cdot d \equiv 1 \bmod \varphi(n)$.
Wir wollen zeigen, dass für alle $m \in \{0, \ldots, n-1\}$ gilt, dass $(m^e)^d \equiv m \bmod n$ ist.
Bekannt ist der
Satz von Fermat: Für Primzahlen p und Zahlen a mit $\mathrm{ggT}(a, p) = 1$ gilt:

$$a^{p-1} \equiv 1 \bmod p.$$

Daraus ergibt sich:

8.4 Korollar. *Sei p prim. Für alle ganzen Zahlen k und a gilt:*

$$a^{1+k\cdot(p-1)} \equiv a \bmod p.$$

Beweis: Für $a \equiv 0 \bmod p$ ist es trivial, da die Gleichung dann $0 \equiv 0 \bmod p$ aussagt. Für $a \not\equiv 0 \bmod p$ gilt $\mathrm{ggT}(a, p)$=1 und somit folgt aus dem Satz von Fermat:

$$\begin{aligned} a^{1+k\cdot(p-1)} &\equiv a \cdot (a^{p-1})^k \bmod p \\ &\equiv a \cdot 1^k \bmod p \\ &\equiv a \bmod p. \end{aligned}$$

□

Für $n = p \cdot q$ ist $\varphi(n) = (p-1) \cdot (q-1)$ und somit $e \cdot d \equiv 1 \bmod (p-1) \cdot (q-1)$. Für eine ganze Zahl t können wir also schreiben:

$$e \cdot d = 1 + t \cdot (p-1) \cdot (q-1).$$

Nun zeigen wir, dass die Nachricht $m \in \{0, \ldots, n-1\}$, die zu $y := m^e \bmod n$ kodiert wird, eindeutig zu dekodieren ist. Wir betrachten zunächst y^d modulo p:

$$\begin{aligned} y^d \equiv (m^e)^d &\equiv m^{1+t\cdot(p-1)\cdot(q-1)} \bmod p \\ &\equiv m \bmod p. \end{aligned}$$

Die letzte Gleichheit folgt dabei aus Korollar 8.4.
Da auch q eine Primzahl ist, können wir auf die gleiche Art und Weise zeigen, dass $y^d \equiv m \bmod q$ ist.
Aus dem Chinesischen Restsatz folgt, dass es im Bereich von 0 bis $p \cdot q - 1 = n - 1$ genau eine Zahl x gibt, die die Kongruenzen $x \equiv m \bmod p$ und $x \equiv m \bmod q$ erfüllt. Die Zahl m stammt aus diesem Bereich und erfüllt die Kongruenzen. Auch y^d erfüllt die Kongruenzen. Daher ist $m = y^d \bmod n$.