Wahlpflichtvorlesung

Effiziente Algorithmen

Sommersemester 2007

Thomas Jansen
Thomas.Jansen@udo.edu

Universität Dortmund
Lehrstuhl Informatik 2
44221 Dortmund

Aktuelle Informationen zur Vorlesung findet man unter
http://ls2-www.cs.uni-dortmund.de/lehre/sommer2007/ea/.

Dieses Skript ist zur Vorlesung „Effiziente Algorithmen“ im Sommersemester 2007 neu geschrieben worden. Wie bei jedem längeren neuen Text sind leider viele Fehler enthalten. Für Geduld und Fehlermitteilungen bin ich natürlich jederzeit dankbar. Am Lehrstuhl 2 sind eine ganze Reihe von Skripten zur Vorlesung „Effiziente Algorithmen“ geschrieben worden. Viele Teile dieses Skripts sind von diesen übernommen worden. Für die freundliche Erlaubnis dazu danke ich insbesondere Thomas Hofmeister und Detlef Sieling.

Version vom 21. Oktober 2008, 8:11 Uhr

Inhaltsverzeichnis

1 Einleitung

Es gibt kaum einen Bereich der Informatik, der ohne Algorithmen auskommt. Algorithmen sind der Kern aller Softwareprodukte, sie finden sich zum Teil in Hardware realisiert auch in Geräten des Alltags. Im Grundstudium am Fachbereich Informatik der Universität Dortmund sind sie fest verankert: in der Grundvorlesung „Datenstrukturen, Algorithmen und Programmierung 2“ werden viele grundlegende Algorithmen vorgestellt, davon viele effiziente Algorithmen, die zu jeder Eingabe eines gegebenen Problems deterministisch in polynomieller Zeit eine optimale Lösung berechnen. In den Grundvorlesungen „Grundbegriffe der theoretischen Informatik“ und „Theoretische Informatik für Studierende der Angewandten Informatik“ werden Grenzen der Möglichkeit, solche effizienten Algorithmen zu entwickeln, ausgelotet. Dass die Entwicklung effizienter Algorithmen schon seit vielen Jahrzehnten in der Informatik thematisiert wird, mag zu der Annahme verleiten, dass das Thema im Wesentlichen erschöpft ist, dass für praktisch alle Probleme, für die überhaupt effiziente Algorithmen möglich sind, auch schon solche Algorithmen bekannt sind. Das ist aber eindeutig nicht so. Außerdem haben Probleme meist die unangenehme Eigenschaft, sich nicht in Luft aufzulösen, wenn ihre grundsätzliche Schwierigkeit erkannt wurde. Es lohnt sich darum, sich auch intensiv mit Algorithmen für Probleme auseinandersetzen, für die es vermutlich keine effizienten Algorithmen gibt.

Dass man Algorithmen in praktisch allen Teilgebieten der Informatik begegnet, mag Motivation genug zur intensiveren Beschäftigung mit Algorithmen sein. Es gibt aber auch noch eine ganze Reihe weiterer Gründe, dem Thema mit Interesse zu begegnen. Einsicht in die Art und Weise, wie man Probleme konkret löst, hilft bei der Einschätzung der Schwierigkeit neuer Probleme. Beim Entwurf großer und komplexer Systeme kann man zu besser handhabbaren Strukturen kommen, wenn man um Lösbarkeit und Schwierigkeiten weiß. Auch die Benutzung umfangreicher Algorithmenbibliotheken mit fertigen Problemlösungen für viele Probleme kann informierter und kompetenter geschehen, wenn man Kenntnisse grundlegender Algorithmen hat. Außerdem wird man in der Praxis häufig mit Problemen konfrontiert, die neu aussehen. Ein in der Algorithmik geübtes Auge hat bessere Chancen, solche Probleme als Varianten bekannter Probleme zu erkennen und sie durch Anpassung bekannter Algorithmen einer effizienten Lösung zuzuführen. Weil Algorithmen in der Informatik schließlich von Rechensystemen ausgeführt werden sollen, ist eine präzise Beschreibung erforderlich und Souveränität im Umgang mit formalen Notationen hilfreich. Die Fähigkeit, präzise zu denken und Problem-

lösungen präzise zu beschreiben, sind offensichtlich weit über die Algorithmik hinaus hilfreich. Schließlich kann man beobachten, dass die Algorithmik dem Teilgebiet der theoretischen Informatik zugeordnet wird. Wegen ihrer immensen praktischen Bedeutung kann man die Algorithmik mit gutem Recht als eines der praktischsten Teilgebiete der theoretischen Informatik bezeichnen. Sie bietet darum manchem, der Theorie eher skeptisch begegnet, einen gelungenen Einstieg in ein großes und spannendes Forschungsgebiet.

Man kann Algorithmen auf verschiedene Arten systematisieren und darstellen. Häufig werden Algorithmen nach Problemen gegliedert und diskutiert. Ebenso sinnvoll ist es auch, allgemeine Entwurfsparadigmen für Algorithmen zu identifizieren und Algorithmen nach diesen Entwurfsparadigmen sortiert darzustellen. Wir werden hier einen anderen Weg beschreiten und grob eine Dreiteilung vornehmen. Im ersten Teil der Vorlesung werden wir uns mit effizienten Algorithmen im engeren Sinn beschäftigen und diskutieren Probleme, für die es deterministische Algorithmen gibt, die auch im Worst Case eine optimale Lösung in Polynomialzeit berechnen. Der Schwerpunkt wird dabei bei Algorithmen für Probleme liegen, die sich als Graphproblem beschreiben lassen. Lässt sich ein Problem nicht so lösen, findet man häufig trotzdem im Worst Case in polynomieller Zeit Lösungen, die zwar nicht optimal sind, die aber von einer optimalem Lösung nicht beliebig weit entfernt sind. Für solche Approximationsalgorithmen werden wir uns im zweiten Teil der Vorlesung Beispiele mit unterschiedlichen Gütegarantien ansehen, außerdem werden wir uns bei dieser Gelegenheit intensiv mit dem wichtigen Konzept der Randomisierung auseinandersetzen. Wir wissen schon aus dem Grundstudium, dass uns der Zufall vor besonders ungünstig strukturierten Eingaben schützen kann: randomisiertes Quicksort ist ein einprägsames Beispiel. Wir werden hier einerseits diskutieren, wie Randomisierung zu effizienten Algorithmen führen kann und werden andererseits versuchen, uns vom Zufall auch wieder unabhängig zu machen und ausgehend von randomisierten Algorithmen strukturiert deterministische Algorithmen zu entwickeln, also Algorithmen zu derandomisieren. Findet man weder optimale Lösungen noch gute Approximationen in zufriedenstellender Zeit, kann man versuchen, heuristische Algorithmen zur Problemlösung einzusetzen. Vor allem mit randomisierten Suchheuristiken werden wir uns darum im dritten Teil der Vorlesung auseinandersetzen. Wir werden aber auch einen in der Praxis häufig effizienten Algorithmus mit exponentiellem Worst Case für ein praktisch sehr wichtiges Problem kennenlernen: den Simplex-Algorithmus, der das Problem der linearen Programmierung löst. Schließlich, im vierten Teil der Vorlesung, werden wir uns die Freiheit nehmen, ausgewählte Themen der Algorithmik zu betrachten, die spannend und praktisch relevant sind, die zur Allgemeinbil-

dung in der Informatik gehören und in anderen Vorlesungen vielleicht keine Erwähnung gefunden haben.
Bei der Einteilung der Vorlesung gestatten wir uns einige Freiheiten. Wenn wir ein Problem im Abschnitt über approximative Lösungen oder gar im Abschnitt über Heuristiken behandeln, bedeutet das nicht zwingend, dass es keinen deterministischen Polynomialzeitalgorithmus gibt. So lange die Beziehung zwischen P und NP nicht geklärt ist, können wir das ja auch gar nicht wissen. Es bedeutet aber nicht einmal, dass nicht ein solcher effizienter Algorithmus bekannt ist. Wir entscheiden uns aber gelegentlich für in der Praxis effiziente Algorithmen (man könnte „empirisch effizient“ sagen) und lassen dafür nachweislich effiziente Algorithmen (im Sinne von deterministisch in Polynomialzeit, man könnte „theoretisch effizient“ sagen) aus. Solche Fälle sind aber zum Glück selten: meist ist ein theoretisch effizienter Algorithmus auch praktisch effizient.
Unsere Sicht auf Algorithmen wird sich auf drei wesentliche Aspekte konzentrieren: die präzise Beschreibung des Algorithmus selbst, den Nachweis seiner Korrektheit und die Analyse seiner Laufzeit. Bei Approximationsalgorithmen schließt der Korrektheitsbeweis einen formalen Nachweis des Einhaltens der Approximationsgüte ein, bei randomisierten Algorithmen werden wir an Stelle der Laufzeit, die ja eine Zufallsvariable sein kann, den Erwartungswert dieser Zufallsvariablen, also die erwartete Laufzeit, analysieren.
In der Praxis fehlt noch zumindest ein wichtiger Schritt, die Implementierung. Grundsätzlich ist es nicht schwierig, die hier vorgestellten Algorithmen zu implementieren. Es ist allerdings oft hochgradig nichttrivial, zu einer *guten* Implementierung zu kommen. Was eine gute Implementierung ist, hängt nicht nur vom Algorithmus ab. Die verwendete Hardware-Umgebung spielt genau so eine Rolle wie zum Beispiel die Größe der Instanzen, die bearbeitet werden sollen. Für Eingaben realistischer Größe kann es vorkommen, dass der asymptotisch beste Algorithmus wesentlich langsamer als ein vielleicht viel einfacherer Algorithmus ist, der ein asymptotisch schlechteres Worst-Case-Verhalten hat. Wir werden diese Problematik hier weitgehend ausklammern und in eine Vorlesung über Algorithm Engineering verweisen.
Dieses Skript dient als Begleitmaterial zur im Sommersemester 2007 gehaltenen Vorlesung „Effiziente Algorithmen und Komplexitätstheorie mit dem Schwerpunkt Effiziente Algorithmen“. Es ist kein Lehrbuch und soll auch Lehrbücher nicht ersetzen, es ist also insbesondere nicht als Grundlage zum eigenständigen Studium und Ersatz für die Vorlesung gedacht. Es enthält kaum Beispiele, Bilder und anschauliche Erklärungen, diese werden in der Vorlesung präsentiert, man findet sie in den Folien, die vorlesungsbegleitend verfügbar gemacht werden. Zusammen mit diesen Folien soll es den Hörerinnen und Hörern als Gedächtnisstütze und zum Nachschlagen dienen; sein

wichtigster Zweck ist, das Zuhören in der Vorlesung zu ermöglichen, indem es das Mitschreiben in der Vorlesung unnötig macht. Die Vorlesung folgt keinem Lehrbuch, man kann darum auch kein einzelnes Buch nennen, das den Stoff der Vorlesung vollständig abdeckt. Es schadet aber natürlich nicht, Bücher über Algorithmen begleitend zu lesen. Als besonders gelungen sei *Introduction to Algorithms* (Second Edition) von Cormen, Leiserson, Rivest und Stein empfohlen.

Teil I

Effiziente Algorithmen

2 Starker Zusammenhang in Graphen

Viele wichtige Probleme lassen sich als Graphprobleme darstellen. Darum gehören effiziente Algorithmen für Graphprobleme zu den wichtigsten grundlegenden Algorithmen. Sie haben schon in der Vorlesung „Datenstrukturen, Algorithmen und Programmierung 2“ eine große Rolle gespielt und werden uns auch hier lange beschäftigen. Dabei wollen wir nun im Hauptstudium natürlich auf unser im Grundstudium erworbenes Wissen aufbauen. Ein besonders wichtiger Algorithmus, den wir im Grundstudium kennengelernt haben, ist die Tiefensuche (*Depth First Search* (*DFS*)). Wir betrachten zunächst zur Wiederholung die Tiefensuche in gerichteten Graphen $G = (V, E)$, die uns bekanntlich in Linearzeit $O(|V| + |E|)$ eine Nummerierung der Knoten und eine Klassifizierung der Kanten in Baum-Kanten (T-Kanten (*tree*)), Rückwärtskanten (B-Kanten (*backward*)), Vorwärtskanten (F-Kanten (*forward*)) und Querkanten (C-Kanten (*cross*)) liefert. Die Wiedergabe des uns schon bekannten Algorithmus hat auch den Sinn, die im Skript verwendete Notation von Algorithmen einzuführen.

Algorithmus 2.1 (**Tiefensuche (DFS)**).
Rahmenalgorithmus

1. ***Initialisierung***
   $i := 0$; $T := \emptyset$; $B := \emptyset$; $F := \emptyset$; $C := \emptyset$
   Für alle $v \in V$: $num(v) := 0$; $status(v) := 0$
2. *Für alle* $v \in V$: *If* $num(v) = 0$ *Then* $DFS(v)$

***DFS*(v)**

1. $status(v) := 1$; $i := i + 1$; $num(v) := i$
2. *Für alle* $w \in Adj(v)$
   If $num(w) = 0$
   Then $T := T \cup \{(v, w)\}$; $DFS(w)$
   Else If $num(w) > num(v)$
   Then $F := F \cup \{(v, w)\}$
   Else If $status(w) = 1$
   Then $B := B \cup \{(v, w)\}$
   Else $C := C \cup \{(v, w)\}$
3. $status(v) := 2$

Wir wissen schon, dass Tiefensuche ein überaus nützlicher und vielfach verwendbarer Algorithmus ist. Man kann mit Hilfe dieses Algorithmus leicht entscheiden, ob ein Graph zusammenhängend oder kreisfrei ist, er ist nützlich, wenn man einen Graphen topologisch sortieren möchte, er hilft, wenn

man eine kontextfreie Grammatik in Chomsky-Normalform bringen möchte und noch bei vielen anderen Gelegenheiten. Wir werden hier einen weiteren Einsatzzweck kennenlernen, die Berechnung der starken Zusammenhangskomponenten eines Graphen. Natürlich beginnen wir mit einer präzisen Beschreibung des Problems, das wir lösen wollen.

Definition 2.2. *Sei $G = (V, E)$ ein gerichteter Graph. Gibt es für $v, w \in V$ einen gerichteten Weg in G von v nach w (also eine Folge $(v_0 = v, v_1, v_2, \ldots, v_l = w)$ mit $(v_{i-1}, v_i) \in E$ für alle $i \in \{1, \ldots, l\}$), so schreiben wir $v \to w$. Im Fall $v = w$ ist die Bedingung leer und darum trivial erfüllt. Gilt $v \to w$ und $w \to v$, so schreiben wir $v \leftrightarrow w$.*
Zwei Knoten $v, w \in V$ heißen stark zusammenhängend*, wenn $v \leftrightarrow w$ gilt. Die Äquivalenzklassen von V bezüglich $\leftrightarrow$ heißen* starke Zusammenhangskomponenten*.*

Definition 2.2 enthält implizit die Behauptung, dass $\leftrightarrow$ eine Äquivalenzrelation ist. Andernfalls wäre der Begriff der starken Zusammenhangskomponenten gar nicht definiert. Es ist aber leicht einzusehen, dass $\leftrightarrow$ offenbar reflexiv ist (da $v \to v$ für alle $v \in V$ gerade der in der Definition erwähnte triviale Fall ist), die Symmetrieeigenschaft erfüllt ist (weil $v \leftrightarrow w$ gemäß Definition $v \to w$ und $w \to v$ impliziert, Änderung der Reihenfolge ergibt direkt $w \leftrightarrow v$). Schließlich ergibt sich Transitivität (also $u \leftrightarrow w$ als Folgerung aus $u \leftrightarrow v$ und $v \leftrightarrow w$) direkt aus der Transitivität von $\to$, was man wiederum direkt mittels Konkatenation von Pfaden erhält. Starker Zusammenhang ist trotz der formalen Definition ein überaus anschaulicher Begriff. Zwei Knoten sind stark zusammenhängend, wenn sie gegenseitig auf gerichteten Pfaden erreichbar sind. Starke Zusammenhangskomponenten sind maximale Mengen von Knoten, die alle paarweise stark zusammenhängend sind.
Unser Problem ist klar. Wir wollen zu einem gerichteten Graphen $G = (V, E)$ die starken Zusammenhangskomponenten berechnen. Wir werden zur Lösung einen sehr einfachen und auch leicht zu implementierenden Algorithmus kennenlernen, der ganz wesentlich auf der Tiefensuche in gerichteten Graphen beruht. Auch die Laufzeitanalyse wird sehr einfach sein. Der spannende und vielleicht auch überraschende Aspekt ist die Korrektheit: Auf den ersten Blick erkennt man nicht sofort, dass der Algorithmus tatsächlich die starken Zusammenhangskomponenten richtig berechnet.

Algorithmus 2.3 (**Starke Zusammenhangskomponenten**)**.**
1. *Führe DFS-Traversierung von $G = (V, E)$ durch, vergib dabei in absteigender Reihenfolge die f-Nummern $n, n-1, \ldots, 1$ an die Knoten, Knoten $v \in V$ erhält seine f-Nummer beim Abschluss des Aufrufs von DFS(v).*

2. *Berechne* $G' := (V, E')$ *mit* $E' := \{(v, u) \mid (u, v) \in E\}$.
3. *Führe DFS-Traversierung von* G' *durch, durchlaufe dabei im Rahmenalgorithmus (Algorithmus 2.1) die Knoten* $v \in V$ *in Reihenfolge aufsteigender* f*-Nummern, mit dem Knoten* v *mit* $f[v] = 1$ *startend.*
4. *Gib die* T*-Bäume der zweiten DFS-Traversierung als starke Zusammenhangskomponenten aus.*

Theorem 2.4. *Algorithmus 2.3 berechnet zu einem gerichteten Graphen* $G = (V, E)$ *in Zeit* $O(|V| + |E|)$ *mit Platz* $O(|V| + |E|)$ *die starken Zusammenhangskomponenten von* G.

Beweis. Die Aussagen über Lauzeit und Speicherplatz sind offensichtlich, wir müssen also nur die Korrektheit nachweisen. Wir betrachten einen Lauf von Algorithmus 2.3 auf einem beliebigen Graphen G. Am Ende der ersten DFS-Traversierung haben wir auf G einen Wald von T-Bäumen $T_1 = (V_1, E_1)$, $T_2 = (V_2, E_2)$, ..., $T_k = (V_k, E_k)$ berechnet, dabei seien die T-Bäume in der DFS-Traversierung in der Reihenfolge T_1, T_2, ..., T_k entstanden. Jede starke Zusammenhangskomponente von G ist offenbar jeweils Teilmenge eines dieser DFS-Bäume.
Alle Kanten, die einen T-Baum verlassen, haben den zweiten Knoten in einem der T-Bäume mit kleinerem Index. Es kann keine Kante $(u, v) \in E$ mit $u \in E_i$ und $v \in E_j$ mit $i < j$ geben. So eine Kante wäre in dem DFS-Aufruf DFS(v_i) gefunden worden, in dem v_i die Wurzel des DFS-Baumes T_i ist. Zu dem Zeitpunkt, zu dem die Kante (u, v) in diesem DFS-Aufruf gefunden worden wäre, wären noch alle Knoten in T_j unbesucht gewesen, es hätte also status$(v) = 0$ gegolten. Damit wäre (u, v) T-Kante in T_i geworden und Teile von T_j gehörten zu T_i. Weil das nicht der Fall ist, kann es eine solche Kante (u, v) also wie behauptet nicht geben.
In G' haben wir alle Kanten umgedreht. Die Kanten zwischen DFS-Bäumen von G verlaufen jetzt in G' in entgegengesetzter Richtung, also in Richtung aufsteigender Indizes der T-Bäume. Wir starten die DFS-Traversierung von G' im Knoten mit f-Nummer 1. Das ist der Knoten, dessen DFS-Aufruf als letzter beendet wurde. Das ist offensichtlich die Wurzel w_k des DFS-Baumes T_k, des letzten T-Baumes der DFS-Traversierung von G. Im Laufe des Aufrufs von DFS(w_k) können nur Knoten aus V_k erreicht werden, weil die Kanten zwischen den Bäumen ja nun gerade in Richtung aufsteigender Indizies der T-Bäume verlaufen, eine solche Kante T_k also nicht verlässt. Wir behaupten, dass der im Aufruf von DFS(w_k) in G' konstruierte T-Baum eine starke Zusammenhangskomponente von G ist. Das ist leicht einzusehen: Alle Knoten $v \in V$, für die in G $w_k \to v$ gilt, liegen im DFS-Baum von w_k. Die Erreichbarkeit wird ja gerade durch die T-Kanten in T_k bewiesen. Andererseits gilt nun für jeden Knoten $v \in V$, der in G' im Aufruf von DFS(w_k) erreicht

wird, $v \to w_k$. Der Weg über T-Kanten in G' zeigt $w_k \to v$, in G gilt diese Beziehung dank Umdrehens der Kanten in umgekehrter Richtung. Also gilt $w_k \leftrightarrow v$ für alle Knoten im T-Baum, der in G' durch Aufruf von DFS(w_k) konstruiert wird. Kann es noch andere Knoten geben? Offensichtlich nicht: Knoten, die in diesem Aufruf nicht erreicht werden, sind in G' von w_k aus nicht erreichbar, also ist in G von diesen Knoten aus w_k nicht erreichbar. Wir berechnen also jeweils die erste starke Zusammenhangskomponente korrekt. Nach diesen Vorbereitungen führen wir jetzt leicht einen Induktionsbeweis über die Anzahl der starken Zusammenhangskomponenten von G. Falls G nur aus einer einzigen starken Zusammenhangskomponente besteht, berechnen wir diese korrekt, wie wir uns gerade überlegt haben. Damit ist der Induktionsanfang schon gesichert. Für den Induktionsschritt haben wir uns auch schon überlegt, dass die erste starke Zusammenhangskomponente korrekt berechnet wird. Jetzt entfernen wir diese starke Zusammenhangskomponente gedanklich aus G. Was übrig bleibt, ist ein DFS-Wald, der aus T_1, T_2, …, T_{k-1} besteht sowie eventuell weiteren Bäumen, die als Rest von T_k übrig bleiben. Dieser Restgraph enthält natürlich genau eine starke Zusammenhangskomponente weniger. Gedanklich ändern wir die f-Nummern in diesem Restgraphen so, dass sie wieder bei 1 beginnen und fortlaufend aufsteigen, die Reihenfolge der f-Nummern sich aber nicht ändert. Das hat offenbar keinen Einfluss auf den Ablauf von Algorithmus 2.3. Wir sehen, dass dieser DFS-Wald als Ergebnis von Zeile 1 beim Aufruf des Algorithmus für den Restgraphen entstanden sein könnte. In diesem Graphen werden die starken Zusammenhangskomponenten korrekt berechnet, das liefert die Induktionsvoraussetzung. Also werden insgesamt die starken Zusammenhangskomponenten korrekt berechnet. □

3 Maximale Matchings

Ein interessantes und praktisch relevantes Problem ist das Zuordnungsproblem. Studentinnen und Studenten kennen es vielleicht aus Vorbesprechungen von Seminaren. Eine Menge von Studierenden und eine Menge von Vortragsthemen sollen so zugeordnet werden, dass es keine Konflikte gibt, es soll also jedes Thema nur einmal vergeben werden und alle Seminarteilnehmerinnen und -teilnehmer sollen jeweils nur ein Thema bekommen. Wir können das leicht als ein Graphproblem modellieren. Jedes Thema, jede Studentin und jeden Studenten modellieren wir mit jeweils einem Knoten. Zwischen zwei Knoten gibt es eine Kante genau dann, wenn der eine Knoten ein Thema modelliert, für das sich die oder der durch den anderen Knoten modellierte Studentin oder Student für dieses Vortragsthema interessiert.

Definition 3.1. *Sei $G = (V, E)$ ein ungerichteter Graph. Eine Menge $M \subseteq E$ heißt* Matching*, wenn $\forall e, e' \in M \colon e \cap e' = \emptyset$ gilt.*

Zu einem beliebigen ungerichteten Graphen ein Matching zu berechnen ist sehr einfach, schließlich ist die leere Menge $\emptyset$ ein Matching, wenn auch kein besonders interessantes. Bezogen auf die Zuordnung von Vortragsthemen sehen wir, dass wir wohl zugeben müssen, dass es keine Konflikte gibt, wenn wir gar keine Seminarthemen zuordnen. Aber das war nicht, was wir meinten, wir wollten natürlich möglichst vielen Studierenden ein Vortragsthema zuordnen. Die Anzahl der konfliktfreien Zuordnungen ist offenbar genau $|M|$, wenn M ein Matching ist. Natürlich wollen wir $|M|$ maximieren. Wir halten das fest.

Definition 3.2. *Beim* Matching-Problem *ist die Eingabe ein ungerichteter Graph $G = (V, E)$. Es soll ein Matching $M \subseteq E$ berechnet werden, so dass $\forall M' \subseteq E \colon (|M'| \leq |M|) \vee (M'$ kein Matching$)$ gilt, also ein Matching maximaler Kardinalität.*

Wenn wir noch einmal an unser Seminarbeispiel denken, erkennen wir, dass wir bei der Modellierung als Graphproblem einen Graphen besonderer Art erhalten. Wir können die Knotenmenge V so in zwei Mengen partitionieren, dass innerhalb dieser beiden Mengen keine Kanten verlaufen. Es gibt nämlich keine Kanten zwischen Vortragsthemen und auch keine Kanten zwischen Studierenden. Solche Graphen nennen wir bipartit, ein Begriff, den wir auch formal festhalten.

Definition 3.3. *Ein ungerichteter Graph $G = (V, E)$ heißt bipartit, wenn $\exists V_1, V_2 \subseteq V \colon (V_1 \cup V_2 = V) \wedge (V_1 \cap V_2 = \emptyset) \wedge (E \subseteq \{\{u, v\} \mid u \in V_1, v \in V_2\})$ gilt.*

Für unsere Seminarzuordnung suchen wir also ein Matching maximaler Kardinalität in einem bipartiten Graphen. Hier werden wir jetzt zunächst einen effizienten Algorithmus für dieses Problem suchen. Im Anschluss überlegen wir uns, ob das Problem viel schwieriger wird, wenn wir die Forderung, dass der Graph bipartit ist, fallen lassen.

3.1 Maximale Matchings in bipartiten Graphen

Für das Problem, in einem bipartiten Graphen ein maximales Matching zu berechnen, fällt uns direkt ein sehr einfacher Greedy-Algorithmus ein, der mit einem leeren Matching startet und dann iterativ das aktuelle Matching M vergrößert, indem er irgendeine Kante hinzufügt, welche die Matchingeigenschaft nicht verletzt. Wenn es keine solche Kante mehr gibt, hören wir auf. Für diesen sehr einfachen Algorithmus benötigen wir nicht einmal einen bipartiten Graphen. Ist dieser Algorithmus denn eigentlich gut? Oder können wir von einem Matching maximaler Kardinalität beliebig weit entfernt sein?

Algorithmus 3.4 (**Greedy-Algorithmus für das Matchingproblem**).
1. $M := \emptyset$
2. *While* $\exists e \in E$: $M \cup e$ *ist Matching*
   Do $M := M \cup e$
3. *Ausgabe* M

Theorem 3.5. *Sei $G = (V, E)$ ein beliebiger ungerichteter Graph, M_{opt} ein Matching maximaler Kardinalität für G.*
Der Greedy-Algorithms für das Matchingproblem berechnet für G in Polynomialzeit ein Matching M, für das $|M| \geq |M_{opt}|/2$ gilt. Es gibt einen Graphen, so dass $|M| = |M_{opt}|/2$ gelten kann.

Beweis. Die Aussage über die Rechenzeit ist offensichtlich. Betrachten wir M, sei V_M die Menge der Knoten, die mit Kanten aus M inzident sind. Natürlich gilt $|V_M| = 2|M|$, sonst wäre M ja kein Matching. Am Ende gilt $\forall e \in E$: $e \cap V_M \neq \emptyset$, denn sonst wäre noch eine Kante zu M hinzufügbar. Matchings haben die Eigenschaft, dass jeder Knoten des Graphen in höchstens einer Kante vorkommt, also gilt insbesondere für das optimale Matching $|M_{\text{opt}}| \leq |V_M|$. Wir haben deshalb $|M_{\text{opt}}| \leq |V_M| = 2|M|$ und die Ungleichung $|M| \geq |M_{\text{opt}}|/2$ folgt direkt.
Betrachte den Graphen $G = (V, E)$ mit $V = \{u_i, v_i \mid i \in \{1, 2, \ldots, n\}\}$ und $E = \{\{u_i, v_i\} \mid i \in \{1, 2, \ldots, n\}\} \cup \{\{v_i, v_{i+1}\} \mid i \in \{1, 2, \ldots, n-1\}\}$ für $n = 2k$ mit $k \in \mathbb{N}$. Man macht sich leicht klar, dass G tatsächlich bipartit

ist. Der Greedy-Algorithmus könnte alle $M = \{\{v_i, v_{i+1}\} \mid \{1, 3, 5, \ldots, n-1\}\}$ wählen, dann ist offensichtlich keine Kante mehr wählbar, weil alle v-Knoten in einer Kante in M vorkommen. Andererseits ist aber offensichtlich $\{\{u_i, v_i\} \mid i \in \{1, 2, \ldots, n\}\}$ offensichtlich ein Matching mit genau doppelter Kardinalität. □

Der Greedy-Algorithmus ist also höchstens um den Faktor 2 von der Kardinalität eines maximalen Matchings entfernt, das kann ihm aber sogar bei bipartiten Graphen auch passieren: Der Graph aus dem Beweis von Theorem 3.5 ist ja bipartit. Die Grundidee des Greedy-Algorithmus ist geradezu primitiv: Wenn das Matching offensichtlich durch Hinzunahme einer Kante verbessert werden kann, so wird das gemacht. Leider kann uns das in eine Sackgasse führen, in der wir von einem maximalen Matching noch deutlich entfernt sind, es aber keine mehr so einfache Verbesserungsmöglichkeit gibt. Wenn man einmal scharf hinsieht, kann man eine etwas weniger primitive, aber immer noch einfache Verbesserungsmöglichkeit entdecken. Wir schaffen uns erst ein geeignetes Vokabular, um präziser und einfacher über das Matchingproblem sprechen zu können.

Definition 3.6. *Sei $G = (V, E)$ ein ungerichteter Graph, $M \subseteq E$ ein Matching.*

- *Eine Kante $e \in M$ heißt* Matching-Kante *oder* M-Kante.
- *Eine Kante $e \notin M$ heißt* freie Kante.
- *Ein Knoten $v \in V$ mit $\exists e \in M : v \in e$ heißt* besetzt.
- *Ein Knoten $v \in V$, der nicht besetzt ist, heißt* frei.
- *Ein Pfad $P = (v_1, v_2, \ldots, v_k)$ ($k \in \mathbb{N}$, $\{v_i, v_{i+1}\} \in E$ für alle $i \in \{1, 2, \ldots, k-1\}$), bei dem sich M-Kanten und freie Kanten abwechseln, heißt* M-alternierend.
- *Ein kreisfreier M-alternierender Pfad $P = (v_1, v_2, \ldots, v_k)$, bei dem v_1 und v_k freie Knoten sind, heißt* M-verbessernd.

Der Greedy-Algorithmus (Algorithmus 3.4) fügt also immer einen M-verbessernden Pfad, der aus nur einer freien Kante besteht, zum aktuellen Matching hinzu. Der Begriff „M-verbessernder Pfad“ legt nahe, dass man auch mit längeren M-verbessernden Pfaden ein Matching M verbessern kann. Man sieht sofort, dass das tatsächlich der Fall ist. Wenn wir im Graphen G einem M-verbessernden Pfad folgen und aus den dabei benutzten freien Kanten Matching-Kanten machen und aus den dabei benutzten M-Kanten freie

Kanten, so erhalten wir ein neues Matching, das um 1 größere Kardinalität hat. Die größere Kardinalität folgt sofort, weil wir ungerade viele Kanten benutzen und die erste und letzte Kante freie Kanten sind. Es ist aber auch leicht einzusehen, dass wir die Matching-Eigenschaft nicht verletzen. An jedem Knoten einer freien Kante liegt höchstens eine Matching-Kante an, sonst wäre schon M kein Matching. Diese Matching-Kante machen wir zur freien Kante, so dass wir die alte freie Kante zur Matching-Kante machen können und weiterhin nur eine Matching-Kante an jedem Knoten anliegt.
Man kann diese Operation des Wechselns aller Kanten etwas eleganter in Mengenschreibweise beschreiben. Sei M ein Matching und seien P die Kanten eines M-verbessernden Pfades. Dann ist $M \oplus P := (M \setminus P) \cup (P \setminus M)$ ein Matching mit $|M \oplus P| = |M| + 1$.
Das liefert jetzt einen etwas aufwendigeren und hoffentlich besseren Algorithmus. Wir suchen zu unserem aktuellen Matching M jeweils einen M-verbessernden Pfad und „addieren“ ihn zu M hinzu, so dass wir M um 1 vergrößern. Das iterieren wir, bis wir keinen M-verbessernden Pfad mehr finden. Es ist interessant zu beobachten, dass diese Modifikation schon ausreicht, um tatsächlich garantiert ein maximales Matching zu finden.

Theorem 3.7. *Sei $G = (V, E)$ ein ungerichteter Graph, $M \subseteq E$ ein Matching.*
M maximal $\Leftrightarrow$ es gibt keinen M-verbessernden Pfad

Beweis. Wir beweisen die Negation, also dass es genau dann einen M-verbessernden Pfad gibt, wenn M nicht maximal ist. Eine Beweisrichtung ist sehr einfach. Wenn P die Kanten eines M-verbessernden Pfades sind, dann ist, wie wir uns schon überlegt haben, $M \oplus P$ ein Matching mit um 1 größerer Kardinalität, also war M nicht maximal.
Sei jetzt also für die Gegenrichtung M ein Matching, das nicht maximal ist. Dann gibt es ja ein Matching M' mit $|M'| > |M|$. Wir betrachten $M \oplus M' = (M \setminus M') \cup (M' \setminus M)$. Genauer gesagt betrachten wir den von $M \oplus M'$ induzierten Subgraphen G': wir entfernen aus G alle Kanten, die in $M \oplus M'$ nicht vorkommen und alle Knoten, die danach Grad 0 haben. Das Ergebnis ist der induzierte Subgraph G'. Welchen Grad haben Knoten in G'? Die zentrale Beobachtung ist, dass sie nur Grad 1 oder 2 haben können. Knoten mit Grad 0 haben wir ja explizit entfernt. Für einen größeren Grad müssten an einem Knoten mindestens zwei M-Kanten oder mindestens zwei M'-Kanten anliegen. Das kann aber nicht sein, weil M und M' beides Matchings sind. Ein Graph, in dem alle Knoten Grad 1 oder 2 haben, zerfällt offensichtlich in disjunkte kreisfreie Pfade und Kreise. Alle Kreise in diesem Graphen enthalten gleiche Anzahlen von M- und M'-Kanten, da ja weder zwei M-Kanten noch zwei M'-Kanten aneinanderstoßen können. Es

gibt aber in dem Graphen mehr M'-Kanten, als es M-Kanten gibt, weil ja $|M'| > |M|$ gilt*. Also muss es mindestens einen kreisfreien Pfad geben, der abwechselnd M- und M'-Kanten enthält und mit einer M'-Kante anfängt und aufhört. Offensichtlich ist dieser Pfad M-verbessernd. □

Wir notieren noch einmal explizit den einfachen Matching-Algorithmus, den wir uns jetzt erarbeitet haben. Dabei erinnern wir uns daran, dass der ungerichtete Graph $G = (V, E)$ bipartit ist, also $V = U \cup W$ gilt, wobei Kanten nur zwischen U und W verlaufen.

Algorithmus 3.8 (**Einfacher Matching-Algorithmus**).
1. $M := \emptyset$
2. *Konstruiere den gerichteten Graphen* $G' = (U \cup W, E')$ *mit*
   $E' = \{(u, w) \mid u \in U, w \in W, \{u, w\} \in M\}$
   $\cup\{(w, u) \mid u \in U, w \in W, \{u, w\} \in (E \setminus M)\}$.
3. *Suche mit Breitensuche beginnend in allen freien Knoten einen gerichteten Pfad zu einem freien Knoten. Sei* P *dieser Pfad.*
4. *If* P *gefunden*
   Then $M := M \oplus P$*; Weiter bei 2.*
   Else Ausgabe M*.*

Theorem 3.9. *Algorithmus 3.8 berechnet in Zeit* $O(ne) = O(n^3)$ *ein maximales Matching für einen bipartiten Graphen* $G = (U \cup W, E)$ *mit* $|U \cup W| = n$ *und* $|E| = e$*.*

Beweis. Gerichtete Pfade von freien Knoten zu freien Knoten sind offenbar gerade M-verbessernde Pfade. Wenn es einen solchen Pfad gibt, wird er durch Breitensuche auch gefunden. Das geht natürlich in Zeit $O(e)$. Wir vergrößern das aktuelle Matching in jeder Iteration der Zeilen 2–4 um 1. Natürlich ist ein maximales Matching in der Größe durch $\lfloor n/2 \rfloor$ nach oben beschränkt. □

Wir können also maximale Matchings deterministisch in Polynomialzeit berechnen. Natürlich wäre es schön, wenn das etwas schneller ginge als in Zeit $O(n \cdot e)$. Wenn man sich etwas mehr Mühe gibt (sowohl algorithmisch als auch was Einsicht in die Problemstruktur angeht), so kann man das auf Zeit $O(\sqrt{n} \cdot e)$ drücken. Wir beginnen, indem wir unsere Einsicht in die Problemstruktur verbessern. Wir verfolgen dabei zwei Ideen. Zum einen konzentrieren wir uns ausdrücklich auf möglichst kurze M-verbessernde Pfade. Zum anderen wollen wir nach Möglichkeit das aktuelle Matching in einem Schritt um mehr als 1 vergrößern. Das ist sicher möglich, wenn die M-verbessernden Pfade, die wir finden, disjunkt sind. Das motiviert unsere andere Idee, sich

*Das erklärt auch, warum der Graph G' nicht völlig leer sein kann.

auf möglichst kurze M-verbessernde Pfade zu konzentrieren. Weil wir uns immer auf die Kanten M-verbessernder Pfade konzentrieren, identifizieren wir ab jetzt einen M-verbessernden Pfad P mit den Kanten, aus denen er besteht.

Lemma 3.10. *Sei $G = (V, E)$ ein ungerichteter Graph, $M \subseteq E$ ein Matching, P ein kürzester M-verbessernder Pfad, P' ein $(M \oplus P)$-verbessernder Pfad. Es gilt $|P'| \geq |P| + |P \cap P'|$.*

Beweis. Betrachte $N := (M \oplus P) \oplus P'$. Weil $M \oplus P$ ein Matching ist, das eine Kante mehr als M enthält und P' $(M \oplus P)$-verbessernder Pfad ist, ist N ein Matching mit $|N| = |M| + 2$. Natürlich ist $M \oplus N = M \oplus (M \oplus P) \oplus P' = P \oplus P'$. Wir sehen, dass $M \oplus N$ zwei knotendisjunkte M-verbessernde Pfade P_1, P_2 enthält. Folglich gilt $|M \oplus N| = |P \oplus P'| \geq |P_1| + |P_2|$.
Wir erinnern uns daran, dass P ein kürzester M-verbessernder Pfad ist. Folglich gilt $|P| \leq |P_1|$ und $|P| \leq |P_2|$, weil P_1 und P_2 ja auch beide M-verbessernd sind. Daraus können wir schließen, dass $|P \oplus P'| \geq 2|P|$ gilt. Andererseits gilt natürlich gemäß Definition von „$\oplus$“ $|P \oplus P'| = |P| + |P'| - |P \cap P'|$. Wir haben also $|P| + |P'| - |P \cap P'| \geq 2|P|$ und sehen, dass $|P'| \geq |P| + |P \cap P'|$ folgt. □

Wir bauen die in Lemma 3.10 ausgedrückte Beobachtung aus, indem wir uns eine ganze Folge von Matchings ansehen, wie sie Algorithmus 3.8 konstruieren könnte.

Lemma 3.11. *Sei $G = (V, E)$ ein ungerichteter Graph. Sei $M_0 := \emptyset$, für $i \in \mathbb{N}_0$ sei P_i ein kürzester M_i-verbessernder Pfad, außerdem sei $M_{i+1} := M_i \oplus P_i$. Es gelten die folgenden Aussagen.*

1. *$|P_i| \leq |P_{i+1}|$*
2. *$|P_i| = |P_j|$ und $i \neq j \Rightarrow P_i$ und P_j sind knotendisjunkt*

Beweis. Die erste Aussage folgt unmittelbar aus Lemma 3.10. Wir müssen also nur die zweite Aussage beweisen. Wir führen einen Beweis durch Widerspruch und nehmen dafür an, dass wir für $i \neq j$ zwei nicht knotendisjunkte Pfade P_i, P_j haben, für die $|P_i| = |P_j|$ gilt. Wir strukturieren die Situation noch etwas stärker: Wir betrachten dazu alle Pfade P_h mit $i \leq h \leq j$. Natürlich haben alle diese Pfade Länge $|P_i| = |P_j|$. Wir wählen jetzt zwei Pfade P_k, P_l mit $i \leq k < l \leq j$, so dass einerseits P_k und P_l nicht knotendisjunkt sind, andererseits aber alle P_m mit $k < m < l$ knotendisjunkt zu P_k und P_l sind. Natürlich gibt es solche Indizes k und l, im Zweifel kann ja $l = k + 1$ gelten und die Bedingung für m wird leer und ist trivial erfüllt. Gemäß unserer Wahl

ist jetzt P_l ein $(M_k \oplus P_k)$-verbessernder Pfad. Weil P_k und P_l nicht knotendisjunkt sind, gibt es einen Knoten v, der in beiden Pfaden vorkommt. Die zu v inzidente Kante aus $M_k \oplus P_k$ muss offenbar beiden Pfaden gemeinsam sein. Damit sind also P_k und P_l auch nicht kantendisjunkt und wir haben $|P_k \cap P_l| \geq 1$. Lemma 3.10 liefert dann, dass $|P_l| \geq |P_k| + |P_k \cap P_l| > |P_k|$ gilt und wir haben einen Widerspruch zu $|P_k| = |P_l|$. □

Natürlich können knotendisjunkte M-verbessernde Pfade gleichzeitig zu M „addiert" werden, ohne dass sie sich gegenseitig beeinflussen. Wir können also gefahrlos M um mehr als 1 in einer Runde vergrößern, wenn wir mehrere kürzeste M-verbessernde Pfade finden. Diese Einsicht formulieren wir als Algorithmus.

Algorithmus 3.12 (**Algorithmus von Hopcroft und Karp**)**.**
1. *$M := \emptyset$*
2. *Berechne eine maximale Menge knotendisjunkter M-verbessernder Pfade $P_1, P_2, \ldots, P_k$ minimaler Länge.*
3. *If $k \geq 1$*
   Then $M := M \oplus P_1 \oplus P_2 \oplus \cdots \oplus P_k$; Weiter bei 2.
   Else Ausgabe M.

Theorem 3.13. *Der Algorithmus von Hopcroft und Karp (Algorithmus 3.12) berechnet in Zeit $O(\sqrt{n}e) = O(n^{5/2})$ ein maximales Matching für einen bipartiten Graphen $G = (U \cup W, E)$ mit $|U \cup W| = n$ und $|E| = e$.*

Beweis. Die Korrektheit haben wir uns ja schon vorab ausführlich überlegt. Wir müssen also nur die Aussagen zur Laufzeit beweisen. Dazu beweisen wir grob zwei Dinge. Zum einen zeigen wir, dass wir mit $O(\sqrt{n})$ Runden (Durchläufen von Zeilen 2–3) auskommen. Zum zweiten zeigen wir, dass eine dieser Runden in Zeit $O(e)$ durchgeführt werden kann.
Für eine obere Schranke zur Anzahl der Runden ist es hilfreich, eine obere Schranke für die Länge kürzester M-verbessernder Pfade zu haben. Dazu sehen wir uns zunächst den Graphen G' aus dem Beweis von Theorem 3.7 noch einmal an. Dieser Graph G' war entstanden aus $M \oplus M'$, wobei M unser aktuelles Matching und M' ein maximales Matching ist. Wir erinnern uns an die Zusammenhangskomponenten von G', die wir hier $C_i = (V_i, E_i)$ nennen, die jeweils kreisfreie Pfade oder Kreise sind. Wir hatten uns schon überlegt, dass $\delta(C_i) := |E_i \cap M'| - |E_i \cap M| \in \{-1, 0, 1\}$ gilt. Eine Zusammenhangskomponente C_i ist ein M-verbessernder Pfad genau dann, wenn $\delta(C_i) = 1$ gilt. Weil aber natürlich

$$\sum_i \delta(C_i) = |M' \setminus M| - |M \setminus M'| = |M'| - |M|$$

gilt, muss es mindestens $|M'| - |M|$ knotendisjunkte M-verbessernde Pfade geben. Diese Pfade enthalten zusammen höchstens $|M|$ Kanten aus M. Das Schubfachprinzip liefert, dass es mindestens einen M-verbessernden Pfad geben muss, der höchstens $\left\lfloor \frac{|M|}{|M'|-|M|} \right\rfloor$ M-Kanten enthält. Weil M- und M'-Kanten alternieren, ist die Länge dieses Pfades durch $2\left\lfloor \frac{|M|}{|M'|-|M|} \right\rfloor + 1$ nach oben beschränkt.
Sei nun M' ein maximales Matching für unseren Graphen G. Wir betrachten den Algorithmus von Hopcroft und Karp und zerlegen seinen Ablauf gedanklich in zwei Phasen. In der ersten Phase gilt $|M| \leq \left\lfloor |M'| - \sqrt{|M'|} \right\rfloor$. Die zweite Phase umfasst die restlichen Runden, in denen dann folglich $|M| > \left\lfloor |M'| - \sqrt{|M'|} \right\rfloor$ gilt.
In der ersten Phase ist die Länge der kürzesten M-verbessernden Pfaden nach oben durch

$$2\left\lfloor \frac{|M|}{|M'|-|M|} \right\rfloor + 1 \leq 2\left\lfloor \frac{\left\lfloor |M'| - \sqrt{|M'|} \right\rfloor}{|M'| - \left\lfloor |M'| - \sqrt{|M'|} \right\rfloor} \right\rfloor + 1$$
$$= 2\left\lfloor \frac{|M'| - \left\lceil \sqrt{|M'|} \right\rceil}{\left\lceil \sqrt{|M'|} \right\rceil} \right\rfloor + 1 = 2\left\lfloor \frac{|M'|}{\left\lceil \sqrt{|M'|} \right\rceil} - 1 \right\rfloor + 1 \leq 2\sqrt{|M'|} - 1$$

beschränkt. Nach $O\left(\sqrt{|M'|}\right)$ Runden ist also die erste Phase beendet.
In der zweiten Phase sind wir nur noch um $\sqrt{|M'|}$ von der Größe eines maximalen Matchings entfernt. Weil natürlich in jeder Runde die Matchinggröße um mindestens 1 wächst, kann auch die zweite Phase nicht mehr als $O\left(\sqrt{|M'|}\right)$ Runden andauern. Weil $n/2$ eine obere Schranke für die Größe eines maximalen Matchings ist, haben wir insgesamt die Anzahl der Runden durch $O\left(\sqrt{|M'|}\right) = O\left(\sqrt{n}\right)$ nach oben beschränkt.
Um nachzuweisen, dass jede Runde in Zeit $O(e)$ durchgeführt werden kann, betrachten wir wieder den gerichteten Graphen aus Algorithmus 3.8. Wir fügen noch zwei Knoten und einige Kanten hinzu, den Knoten q und alle Kanten (q, u) für freie Knoten $u \in U$ sowie den Knoten s und alle Kanten (w, s) für freie Knoten $w \in W$. Dabei sind U und W wieder die Partitionen der Knotenmenge des bipartiten Graphen $G = (V = U \cup W, E)$ mit Kanten ausschließlich zwischen U und W. Wir führen in Zeit $O(e)$ eine Breitensuche von q aus durch und streichen dann alle Kanten, die nicht auf kürzesten Pfaden nach s liegen. Wir erkennen, dass alle kürzesten M-verbessernden Pfade

in diesem Graphen erhalten bleiben. Wir können jetzt leicht in einer zweiten Breitensuche alle kürzesten Pfade von q nach s suchen, dabei entfernen wir jeweils alle benutzten Kanten und Knoten, so stellen wir sicher, dass die M-verbessernden Pfade, die wir erhalten, knotendisjunkt sind. Wir können die Pfade leicht zu M hinzufügen und haben so insgesamt eine Runde in Zeit $O(e)$ ausgeführt. □

3.2 Maximale Matchings in allgemeinen Graphen

Bis jetzt funktionieren unsere Algorithmen nur für bipartite Graphen. Sie nutzen diese besondere Graphstruktur direkt aus, wenn sie (kürzeste) M-verbessernde Pfade konstruieren. M-verbessernde Pfade gibt es aber in allen Graphen. Nichts in Theorem 3.7 gilt nur für bipartite Graphen. Wir könnten also in allgemeinen Graphen auch nach M-verbessernden Pfaden suchen und diese „addieren“, wie wir das in bipartiten Graphen machen. Sehen wir uns zunächst ein Beispiel an (Abbildung 1). In dem ungerichteten Graphen $G = (V, E)$ mit 12 Knoten ist ein Matching M der Größe 5 markiert, die dicker gezeichneten Kanten sind die Matching-Kanten. Es gibt zwei freie Knoten, die Knoten v_1 und v_{12}, wir fangen bei v_1 an, nach einem M-verbessernden Pfad zu suchen. Wir erreichen über alternierende Kanten den Knoten v_5 und haben die Wahl, ob wir zum Knoten v_6 oder v_7 weitergehen. Wir entscheiden uns für v_6, erreichen v_7 und haben wieder eine Wahlmöglichkeit, diesmal zwischen Knoten v_8 und v_5. Wir entscheiden uns, v_5 zu wählen und setzen dann den Weg fort, bis wir schließlich mit v_1 wieder einen freien Knoten erhalten. Wir haben so einen alternierenden Pfad konstruiert, der an einem freien Knoten beginnt und endet, es ist aber leicht einzusehen, dass der Pfad nicht M-verbessernd ist, wir können ihn nicht zu M „addieren“, um M zu vergrößern.

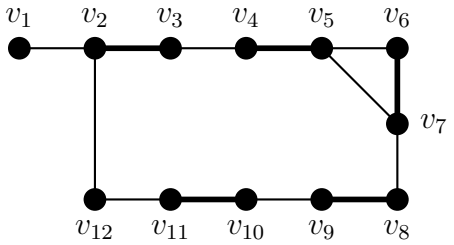

Abbildung 1: Beispielgraph mit Matching.

Das Problem ist leicht erkennbar: wir haben einen Kreis gefunden, genauer einen alternierenden Kreis ungerader Länge. Solche Kreise, die es in bipar-

titen Graphen offensichtlich nicht geben kann, nennt man *Blüten**, weil sie uns fälschlich vorgaukeln, M-verbessernd zu sein. Weil wir auch für allgemeine (nicht bipartite) Graphen das Matchingproblem effizient lösen wollen, werden wir uns überlegen müssen, wie wir mit Blüten umgehen können. Wir beschreiben dazu jetzt grob einen Algorithmus, den wir im Anschluss noch einmal präzise angeben wollen. Die Grundidee wird dabei unverändert vom Algorithmus von Hopcroft und Karp (Algorithmus 3.12) übernommen: wir starten mit dem leeren Matching, finden in einer Phase mit aktuellem Matching M möglichst viele knotendisjunkte kürzeste M-verbessernde Pfade und vergrößern M, indem wir diese Pfade alle gleichzeitig „addieren“. Wenn wir uns den Beweis von Theorem 3.13 noch einmal ansehen, erkennen wir, dass die Analyse der Anzahl der Phasen für allgemeine Graphen gilt. Wir brauchen also weiterhin nur $O(\sqrt{n})$ solche Phasen. Offen ist nur die Frage, wie wir in allgemeinen Graphen möglichst effizient eine maximale Menge knotendisjunkter kürzester M-verbessernder Pfade finden. Es sei vorweg eingestanden, dass das schwierig, aufwendig und auch etwas kompliziert wird. Aber der Aufwand lohnt sich: Wir werden am Ende sehen, dass wir eine Phase in allgemeinen Graphen in Zeit $O(e)$ durchführen können. Ein erstaunliches Resultat, wir sind also in allgemeinen Graphen asymptotisch nicht langsamer als in bipartiten Graphen.
Wir beginnen mit einer formalen Definition von Begriffen, die wir bei der Beschreibung des Algorithmus brauchen werden. Nicht von jedem Begriff ist vielleicht sofort einsehbar, warum er sich als nützlich erweisen wird. Wir ziehen es trotzdem vor, die Definition hier zentral zu sammeln und sie nicht jeweils verstreut dort zu definieren, wo wir das erste Mal Gebrauch von ihnen machen.

Definition 3.14. *Sei $G = (V, E)$ ein Graph, M ein Matching in G.*
Für einen Knoten $v \in V$ sei

- $\mathit{geradeTiefe}(v) := \min\{l \mid (v_1, v_2, \ldots, v_l, v)$ *alternierender Pfad,* v_1 *frei,* l *gerade* $\} \cup \{\infty\}$,
- $\mathit{ungeradeTiefe}(v) := \min\{l \mid (v_1, v_2, \ldots, v_l, v)$ *alternierender Pfad,* v_1 *frei,* l *ungerade* $\} \cup \{\infty\}$,
- $\mathit{Tiefe}(v) := \min\{\mathit{geradeTiefe}(v), \mathit{ungeradeTiefe}(v)\}$.

Für Knoten $v \in V$ mit $\mathit{Tiefe}(v) < \infty$ nennen wir

- *den Knoten v* äußeren Knoten*, wenn $\mathit{Tiefe}(v) = \mathit{geradeTiefe}(v)$ gilt*

*Den Blütenbegriff werden wir später noch etwas allgemeiner fassen, darum halten wir ihn hier nicht in einer formalen Definition fest.

- *den Knoten v* inneren Knoten *sonst.*

Für einen äußeren Knoten v ist ungeradeTiefe(v) seine andere Tiefe*, entsprechend ist für einen inneren Knoten v geradeTiefe(v) seine* andere Tiefe*. Eine Kante $\{s,t\} \in E$ heißt* Brücke*, wenn geradeTiefe(s) + geradeTiefe(t) $< \infty$ oder ungeradeTiefe(s) + ungeradeTiefe(t) $< \infty$ gilt.*
Für eine Brücke $\{s,t\} \in E$ sei Gewicht($\{s,t\}$) $:=$ min{geradeTiefe(s) +geradeTiefe(t), ungeradeTiefe(s) +ungeradeTiefe(t)} + 1.

Wir bemerken, dass für freie Knoten v immer Tiefe(v) $= 0$ gilt und freie Knoten also immer äußere Knoten sind. Weil M-verbessernde Pfade ungerade Länge haben, ist jede Kante eines solchen Pfades eine Brücke: alle Knoten des M-verbessernden Pfades, die vom freien Knoten am einen Ende über eine gerade Anzahl Kanten erreicht werden, werden vom anderen freien Knoten aus über eine ungerade Anzahl Kanten erreicht und umgekehrt. Wenn G (mit M) keine Blüten enthält, dann liegt natürlich auch umgekehrt jede Brücke auf einem M-verbessernden Pfad. Wir sehen jetzt, dass Gewicht($\{s,t\}$) gerade die Länge eines kürzesten M-verbessernden Pfades ist, der die Brücke $\{s,t\}$ enthält.
Die Ideen für den Algorithms liegen jetzt klar auf der Hand. Wir verbessern das aktuelle Matching phasenweise durch „Addition“ knotendisjunkter kürzester M-verbessernder Pfade. Wir finden solche Pfade, indem wir Brücken minimalen Gewichtes suchen. So eine Brücke wird dann entweder zu einem kürzesten M-verbessernden Pfad ausgebaut oder ist Bestandteil einer Blüte, die dann geeignet behandelt werden muss.
In einer Phase des Algorithmus werden drei zentrale Funktionen benutzt: Die Funktion `Suche` sucht gleichzeitig von allen freien Knoten aus mit Breitensuche nach verbessernden Pfaden, dabei bedeutet gleichzeitig, dass alle Knoten mit Tiefe i gefunden werden, bevor ein Knoten mit Tiefe $i+1$ gefunden wird. Für jede in Tiefe i gefundene Brücke wird eine Funktion `BlüteOderVerbesserung` aufgerufen, die entweder M verbessern oder eine Blüte finden soll. Wenn in Tiefe i Brücken gefunden werden, wird Tiefe $i+1$ nicht mehr betreten und die aktuelle Phase endet mit den Aufrufen von `BlüteOderVerbesserung`. Wenn gar kein Knoten der Tiefe i gefunden wird, endet das Programm, das aktuelle Matching ist dann maximal. Um mit Blüten effizient umgehen zu können, werden in `Suche` noch weitere Informationen berechnet und an den Knoten abgespeichert. Wenn es zu einem besetzten Knoten $v \in V$ einen anderen Knoten $v' \in V$ gibt, so dass entweder v ein äußerer Knoten und $\{v,v'\}$ Matchingkante ist oder v ein innerer Knoten und UngeradeTiefe(v) = GeradeTiefe(v') + 1 gilt, dann heißt v' *Vorgänger* von v und `Suche` speichert v' in einer Liste von Vorgängern im Knoten v. Außerdem wird entsprechend eine Liste von Nachfolgern gespeichert. Wir nennen einen

Knoten v' eine *Anomalie* für $v \in V$, wenn v innerer Knoten ist, v' äußerer Knoten ist, $\{v, v'\} \in E \setminus M$ und GeradeTiefe(v') > GeradeTiefe(v) gilt. Auch solche Anomalien speichern wir in einer Liste in v.
Die Funktion `BlüteOderVerbesserung`, die eine Brücke $\{s, t\}$ als Parameter erhält, benutzt Tiefensuche ausgehend von den beiden Knoten der Brücke. Wenn der aktuelle Knoten des „linken“ DFS-Aufrufs v_L (initial $v_L = s$) und der des „rechten“ DFS-Aufrufs v_R (initial $v_R = r$) ist, so gilt initial Tiefe(v_L) = Tiefe(v_R) und es wird immer in dem Knoten der DFS-Aufruf fortgesetzt, der größere Tiefe hat. Wir merken uns den Knoten mit kleinster Tiefe, der sowohl im linken als auch rechten DFS-Aufruf gefunden wurde. Anfangs gibt es natürlich noch keinen solchen Knoten und es muss auch nicht zwingend ein Knoten gemeinsam gefunden werden. Wir markieren Knoten als benutzt (und an anderer Stelle noch als gelöscht), betrachtet werden in den DFS-Aufrufen nur Knoten, die weder bereits benutzt noch bereits gelöscht sind, außerdem werden als nächste aktuelle Knoten nur solche Knoten betrachtet, die in der aktuellen Liste der Nachfolger zu finden sind. Wir merken uns, welchen Weg wir gegangen sind, indem wir den vorangegangenen Knoten als Elter speichern, außerdem markieren wir die Knoten als links oder rechts, je nachdem in welchem DFS-Aufruf sie betreten wurden. Wenn auf diese Weise zwei freie Knoten gefunden werden, ist ein kürzester M-verbessernder Pfad gefunden worden. Falls die DFS-Aufrufe sich an einem Knoten treffen, so darf zunächst der linke DFS-Aufruf diesen Knoten als links markieren und der rechte DFS-Aufruf sucht nach einer Alternative gleicher Tiefe. Dabei kann der Elter zum Backtracking benutzt werden. Falls das nicht funktioniert, wird der Knoten als rechts markiert und der linke DFS-Aufruf sucht auf gleiche Art nach einer Alternative gleicher Tiefe. Falls das auch fehlschlägt, wurde eine Blüte gefunden. Der tiefste gemeinsam gefundene Knoten, den wir uns ja merken, ist dann der Eintrittspunkt in diese Blüte, auch Basis genannt. Damit die Laufzeit durch das Backtracking nicht zu groß wird, merken wir uns einen Knoten, der *Schranke* genannt wird und über den hinaus keine Backtracking-Schritte mehr erlaubt sind. Wir definieren diese Schranke für den rechten DFS-Aufruf und setzen darum initial v_R als Schranke ein. Im Lauf wird immer der gemeinsam gefundene Knoten neue Schranke, bei dem ein Backtracking-Durchlauf des rechten DFS-Durchlaufs erfolglos war.
Wir müssen uns Gedanken über Blüten machen: An einer Brücke $\{s, t\}$ sprechen wir von einer Blüte, wenn es einen Knoten w gibt, der über Vorgänger von s und t erreichbar ist und kein anderer so erreichbarer Knoten gleiche Tiefe wie w hat. Betrachten wir alle solche Knoten w, die noch keiner Blüte zugeordnet wurden. Unter diesen Knoten sei b der Knoten mit maximaler Tiefe; wir nennen b Basis der Blüte B. Diese Blüte B besteht aus all den Knoten v, die noch keiner anderen Blüte zugeordnet wurden und für die ent-

weder $v \in \{s, t, v' \mid v' \text{ über Vorgänger von } s \text{ oder } t \text{ erreichbar}\}$ oder b über einen Vorgänger von v erreichbar ist. Wir nennen s und t die *Spitzen* von B. Wir präsentieren jetzt den vollständigen Algorithmus. Wir machen das modular, indem wir die wesentlichen Funktionen jeweils gesondert präsentieren. Durch unsere Überlegungen im Vorfeld ist der Algorithmus gut verständlich. An einigen Stellen finden sich zusätzliche Kommentare, die jeweils $\{* \; so \; *\}$ gekennzeichnet sind.

Algorithmus 3.15 (**Algorithmus von Micali und Vazirani**).

Rahmenalgorithmus

1. $M := \emptyset$
2. *Repeat*
3. *Für alle* $v \in V$
4. $geradeTiefe(v) := ungeradeTiefe(v) := \infty$
5. $Blüte(v) := -$ {∗ *Name der Blüte, zu der* v *gehört* ∗}
6. $Vorgänger(v) := Nachfolger(v) := Anomalien(v) := \emptyset$
7. $num(v) := 0$ {∗ *Anzahl nicht gelöschter Vorgänger* ∗}
8. $gelöscht(v) := besucht(v) := links(v) := rechts(v) := false$
9. *Für alle* $\{u, v\} \in E$
10. $benutzt(\{u, v\}) := besucht(\{u, v\}) := false$
11. *Für* $i \in \{1, 2, \ldots, |V|\}$ {∗ *Suchlevel* ∗}
12. $Kandidaten(i) := \emptyset$
13. $Brücken(i) := \emptyset$
14. *VerbesserungGefunden:=Suche()*
15. *Until VerbesserungGefunden* $=$ *false*

Suche()

1. $i := 0$*; Verbesserung* := *false; Für alle* $v \in \{v' \in V \mid v'$ *frei*$\}$
2. *geradeTiefe*$(v) := 0$*; Füge* v *in Kandidaten*(0) *ein.*
3. *While Kandidaten*$(i) \neq \emptyset$ *und Verbesserung* = *false*
4. *If* i *gerade*
5. *Für alle* $v \in$ *Kandidaten*(i)
6. *Für alle Nachbarn* u *von* v *mit gelöscht*$(u) =$ *benutzt*$(u) =$ *false,* $\{u, v\}$ *frei*
7. *If geradeTiefe*$(u) < \infty$
8. *Then Füge* $\{u, v\}$ *in Brücken*$((geradeTiefe(u) + geradeTiefe(v))/2)$ *ein.*
9. *Else*
10. *If ungeradeTiefe*$(u) = \infty$ *Then ungeradeTiefe*$(u) = i + 1$
11. *If ungeradeTiefe*$(u) = i + 1$
12. *Then*
13. *num*$(u) :=$ *num*$(u) + 1$*; Füge* u *in Nachfolger*(v) *ein.*
14. *Füge* v *in Vorgänger*(u) *ein; Füge* u *in Kandidaten*$(i + 1)$ *ein.*
15. *If ungeradeTiefe*$(u) < i$ *Then Füge* v *in Anomalien*(u) *ein.*
16. *Else* $\{*$ i *ungerade* $*\}$
17. *Für alle* $v \in$ *Kandidaten*(i) *mit Blüte*$(v) = -$
18. $u :=$ *Partner von* v $\{*$ u *ist besetzt* $*\}$
19. *If ungeradeTiefe*$(u) < \infty$
20. *Then Füge* $\{u, v\}$ *in Brücken*$((ungeradeTiefe(u) + ungeradeTiefe(v))/2)$ *ein.*
21. *If geradeTiefe*$(u) = \infty$
22. *Then*
23. *Nachfolger*$(v) := \{u\}$*; Vorgänger*$(u) := \{v\}$*; num*$(u) := 1$
24. *geradeTiefe*$(u) := i + 1$*; Füge* u *in Kandidaten*$(i + 1)$ *ein.*
25. *Für alle* $\{s, t\} \in$ *Brücken*(i)
26. *If gelöscht*$(s) =$ *gelöscht*$(t) =$ *false*
27. *Then Verbesserung* := *BlüteOderVerbesserung*$(\{s, t\})$
28. $i := i + 1$
29. Exit mit Rückgabe von Verbesserung

***BlüteOderVerbesserung*($\{s,t\}$)** $\{*$ $\{s,t\}$ *ist Brücke* $*\}$

1. *If* $Blüte(s) = Blüte(t)$ *und* $Blüte(s) \neq -$ *Then Exit.*
2. *If* $Blüte(s) \neq -$ *Then* $v_L := Basis(Blüte(s))$ *Else* $v_L := s$
3. *If* $Blüte(t) \neq -$ *Then* $v_R := Basis(Blüte(t))$ *Else* $v_R := t$
4. $links(v_L) := true$; $rechts(v_R) := true$; $DCV := -$; $Schranke := v_R$
5. *While (*v_L *nicht frei oder* v_R *nicht frei) und* $BlüteGefunden = false$
6. $Tiefe_L := \min\{geradeTiefe(v_L), ungeradeTiefe(v_L)\}$
7. $Tiefe_R := \min\{geradeTiefe(v_R), ungeradeTiefe(v_R)\}$
8. *If* $Tiefe_L \geq Tiefe_R$
   Then $BlüteGefunden := DFSLinks()$
   Else $BlüteGefunden := DFSRechts()$
9. *If* $BlüteGefunden = false$ *Then*
10. $P_L := FindePfad(s, v_L, -)$; $P_R := FindePfad(t, v_R, -)$;
11. *Verbessere Matching mit Pfad* $P := P_L, \{s,t\}, P_R$.
12. *Repeat*
13. *Entferne* y *aus* P; $gelöscht(y) := true$;
14. *Für alle* $z \in Nachfolger(y)$ *mit* $gelöscht(z) = false$
15. $num(z) := num(z) - 1$
16. *If* $num(z) = 0$ *Then Füge* z *zu* P *hinzu*
17. *Until* P *leer.*
18. *Exit mit Rückgabe von true;*
19. *Else*
20. $rechts(DCV) := false$;
21. *Erzeuge Blüte* B *aus allen in diesem Aufruf von BlüteOderVerbesserung als links oder rechts markierten Knoten,* $Blüte(v) := B$ *für diese* v. *Speichere* r *und* s *als Spitzen von* B.
22. $Basis(B) := DCV$
23. *Für alle* $y \in B$
24. *If* $\min\{geradeTiefe(y), ungeradeTiefe(y)\}$ *gerade*
25. *Then* $ungeradeTiefe(y) := 2i + 1 - geradeTiefe(y)$;
26. *Else*
27. $geradeTiefe(y) := 2i + 1 - ungeradeTiefe(y)$;
28. *Füge* y *in* $Kandidaten(ungeradeTiefe(y))$ *ein.*
29. *Für alle* $z \in Anomalien(y)$
30. $j := (geradeTiefe(y) + geradeTiefe(z))/2$
31. *Füge* $\{y,z\}$ *in* $Brücken(j)$ *ein;* $benutzt(\{y,z\}) := true$

DFSLinks()

1. *While* $\exists u \in Vorgänger(v_L)$ *mit* $benutzt(\{u, v_L\}) = gelöscht(u) = false$
2. $benutzt(\{u, v_L\}) := true$
3. *If* $Blüte(u) \neq -$ *Then* $u := Basis(Blüte(u))$
4. *If* $links(u) = rechts(u) = false$ *Then*
5. $links(u) := true$; $Elter(u) := v_L$; $v_L := u$
6. *Exit mit Rückgabe von false* $\{*$ *keine Blüte* $*\}$
$\{*$ *u schon links oder rechts* $\rightsquigarrow$ *Backtracking* $*\}$
7. *If* $v_L = s$ *Then Exit mit Rückgabe von true* $\{*$ *Blüte* $*\}$
8. *Else* $v_L := Elter(v_L)$; *Exit mit Rückgabe von false* $\{*$ *Backtracking* $*\}$

DFSRechts()

1. *While* $\exists u \in Vorgänger(v_R)$ *mit* $benutzt(\{u, v_R\}) = gelöscht(u) = false$
2. $benutzt(\{u, v_R\}) := true$
3. *If* $Blüte(u) \neq -$ *Then* $u := Basis(Blüte(u))$
4. *If* $links(u) = rechts(u) = false$ *Then*
5. $rechts(u) := true$; $Elter(u) := v_R$; $v_R := u$
6. *Exit mit Rückgabe von false* $\{*$ *keine Blüte* $*\}$
$\{*$ *u schon links oder rechts* $\rightsquigarrow$ *Backtracking* $*\}$
7. *Else If* $u = v_L$ *Then* $DCV := u$
8. *If* $v_R = Schranke$ *Then*
9. $v_R := DCV$; $Schranke := DCV$; $links(v_R) := false$; $rechts(v_R) := true$;
10. $v_L := Elter(v_L)$ $\{*$ *Backtracking links* $*\}$
11. *Else* $v_R := Elter(v_R)$ $\{*$ *Backtracking rechts* $*\}$
12. *Exit mit Rückgabe von false*

***FindePfad*(h, l, B)**

1. *If* $h = l$ *Then* $P := h$; *Exit mit Rückgabe von* P
2. $v := h$; *Repeat*
3. *While* $\nexists u \in$ *Vorgänger*(v) *mit benutzt*$(\{u, v\}) =$ *false*
4. $v :=$ *Elter*(v)
5. *If Blüte*$(v) = B$ *oder Blüte*$(v) = -$
6. *Then Wähle* $u \in$ *Vorgänger*(v) *mit benutzt*$(\{u, v\}) =$ *false; benutzt*$(\{u, v\}) :=$ *true*
7. *Else* $u :=$ *Basis*(*Blüte*(v))
8. *If besucht*$(u) =$ *false und gelöscht*$(u) =$ *false und* $\min\{$*ungeradeTiefe*(u), *geradeTiefe*$(u)\} > \min\{$*ungeradeTiefe*(l), *geradeTiefe*$(l)\}$ *und links*$(u) =$ *links*(h) *und rechts*$(u) =$ *rechts*(h)
9. *Then besucht*$(u) :=$ *true; Elter*$(u) := v$; $v := u$
10. *Until* $u = l$
11. *Sei* $P = x_1, x_2, \ldots, x_m$ *Pfad über Elter-Zeiger von* h *nach* l.
12. *Für alle* $j \in \{1, 2, \ldots, m-1\}$
13. *If Blüte*$(x_j) \neq B$ *und Blüte*$(x_j) \neq -$
14. *Then Ersetze* x_j *und* x_{j+1} *durch Ausgaben von Öffne*(x_j)
15. *Exit mit Rückgabe von* P

***Öffne*(x)**

1. $B :=$ *Blüte*(x); $b :=$ *Basis*(B)
2. *If* $\min\{$*ungeradeTiefe*(x), *geradeTiefe*$(x)\}$ *gerade*
3. *Then Exit mit Rückgabe FindePfad*(x, b, B)
4. $p_L :=$ *linke Spitze von* B; $p_R :=$ *rechte Spitze von* B
5. *If links*$(x) =$ *true*
6. *Then* $P :=$ *FindePfad*(l, x, B), *FindePfad*(r, b, B)
7. *Else* $P :=$ *FindePfad*(r, x, B), *FindePfad*(l, b, B)
8. *Exit mit Rückgabe von* P

Theorem 3.16. *Der Algorithmus von Micali und Vazirani (Algorithmus 3.15) berechnet in Zeit $O(\sqrt{n}e)$ ein maximales Matching für einen beliebigen Graphen $G = (V, E)$ mit $|V| = n$ und $|E| = e$.*

Beweis. Nach unseren Vorüberlegungen reicht es, wenn wir uns überlegen, dass Algorithmus 3.15 in jeder Runde eine maximale Menge knotendisjunkter kürzester M-verbessernder Pfade berechnet und das in Zeit $O(m)$ schafft. Knotendisjunktheit der M-verbessernden Pfaden ist offensichtlich, da verwendete Knoten als gelöscht markiert und im Anschluss nicht mehr betrachtet werden. Dass nur kürzeste M-verbessernde Pfade gefunden werden, folgt unmittelbar aus dem Ablauf der Suche, die Ebene für Ebene vorgeht. Weil wir alle nicht benutzten Kanten passender Tiefe erschöpfend betrachten, ist auch gesichert, dass wir keinen kürzesten M-verbessernden Pfad übersehen. Für die Laufzeit je Runde machen wir uns klar, dass jeder Listenzugriff in konstanter Zeit realisiert werden kann, das gilt auch für die Abfrage der Attribute von Knoten und Kanten. Für Graphen ohne Blüten ist klar, dass wir jede Kante und jeden Knoten nur $O(1)$ mal betrachten. Für Blüten ist entscheidend, dass wir Blüten finden und Knoten mit der Zugehörigkeit zu ihrer Blüte markieren. In den Aufrufen von DFSLinks und DFSRechts werden bekannte Blüten, die man bei der Blüte trifft, implizit „geschrumpft“: Wir wechseln jeweils in Zeile 3 vom aktuellen Knoten u in die Basis der Blüte, zu der u gehört. Das stellt sicher, dass wir Blüten nur einmal durchlaufen, so dass wir auch Knoten und Kanten in Blüten nur $O(1)$ mal betrachten. Daraus folgt die Laufzeitschranke $O(n + m) = O(m)$ je Phase. □

4 Das Flussproblem

Wir wollen uns ein weiteres Graphproblem ansehen, das viele praktische Anwendungen hat. Wir betrachten dazu eine spezielle Klasse von Graphen, die wir *Netzwerke* nennen. Ein Netzwerk ist ein gerichteter Graph mit einem „Startknoten“, den wir *Quelle* nennen, und einem „Zielknoten“, den wir *Senke* nennen. Die gerichteten Kanten sind gewichtet, konkret mit nicht-negativen ganzen Zahlen versehen. Wir wollen jetzt möglichst viel einer kontinuierlich teilbaren Substanz vom Start zum Ziel (besser: von der Quelle zur Senke) bringen, dabei darf über jede Kanten nicht mehr als ihr Gewicht geschickt werden, negative Substanzmengen sind verboten; außerdem können die Knoten nichts speichern, alles, was wir in einen Knoten hineinleiten, muss ihn also auch wieder verlassen – das gilt nur für die Senke nicht. Anwendungen liegen auf der Hand, man kann zum Beispiel an die Lenkung von Verkehrsströmen denken, bei ausreichend großem Verkehrsaufkommen ist die Voraussetzung der kontinuierlichen Teilbarkeit nicht zu weit von der Realität entfernt. Es gibt auch weniger naheliegende Anwendungen; bevor wir uns ein Beispiel ansehen, wollen wir zunächst den Begriff des Netzwerks und das Flussproblem formal definieren.

Definition 4.1. *Ein* Netzwerk *ist ein gerichteter Graph $G = (V, E)$ mit einer* Quelle $Q \in V$*, einer* Senke $S \in V$ *und einer* Kapazitätsfunktion $c\colon E \to \mathbb{N}_0$*, der* asymmetrisch *ist, für den also*

$$\forall u, v \in V\colon (u, v) \in E \Rightarrow (v, u) \notin E$$

gilt. Außerdem ist $\{(v, Q), (S, v) \mid v \in V\} = \emptyset$.
Ein Fluss Φ *ist eine Abbildung $\Phi\colon E \to \mathbb{R}_0^+$, die jeder Kante einen Fluss zuweist, wobei die Kapazitäten respektiert werden ($\forall e \in E\colon \Phi(e) \leq c(e)$) und die* Kirchhoff-Regel *gilt ($\forall v \in V \setminus \{S, Q\}\colon \sum_{e=(u,v)\in E} \phi(e) = \sum_{e=(v,w)\in E} \phi(e)$).*
Ein Fluss *heißt* ganzzahlig*, wenn $\forall e \in E\colon \Phi(e) \in \mathbb{N}_0$ gilt.*
Der Wert *w eines Flusses ist definiert als $w(\Phi) = \sum_{e=(Q,v)\in E} \phi(e)$.*
Ein Fluss Φ heißt maximal*, wenn $w(\Phi') \leq w(\Phi)$ für alle Flüsse Φ' gilt.*

Beim *Flussproblem* ist für ein Netzwerk ein maximaler Fluss zu bestimmen. Die Einschränkung auf asymmetrische Graphen ist offensichtlich etwas künstlich, sie wird sich später bei der Beschreibung von Algorithmen aber als praktisch erweisen. Wir wollen uns das Leben nicht unnötig dadurch erschweren, hier auf sie zu verzichten. Man macht sich leicht klar, dass es sich nicht

um eine wesentliche Einschränkung handelt. Wenn man tatsächlich mit einem Graphen G konfrontiert ist, der (u, v) und (v, u) gleichzeitig enthält, so löst man das Flussproblem stattdessen für ein Netzwerk G', dass sich als lokale Modifikation von G ergibt. Die Knotenmenge wird einfach um einen neuen Knoten x erweitert, aus der Kantenmenge entfernt man die Kante $e = (v, u)$ und führt stattdessen die Kanten $e_1 = (v, x)$ und $e_2 = (x, u)$ mit $c(e_1) = c(e_2) = c(e)$ ein. Der Graph wächst dadurch offensichtlich nicht wesentlich und wir können aus einem maximalen Fluss für den modifizierten Graphen G' direkt einen maximalen Fluss für den Ausgangsgraphen G ablesen.

Als Beispiel für die Anwendbarkeit des Flussproblems in anderen Bereichen denken wir noch einmal an die Berechnung maximaler Matchings in bipartiten Graphen zurück (Abschnitt 3.1). Wir haben einen bipartiten Graphen $G = (U \uplus V, E)$ und suchen ein maximales Matching. Wir übersetzen das in ein Flussproblem, indem wir ein Netzwerk $G' = (V', E')$ definieren mit $V' := U \uplus V \uplus \{Q, S\}$, $E' := \{(u, v) \mid u \in U, v \in V, \{u, v\} \in E\} \uplus \left(\bigcup_{u \in U} (Q, u)\right) \uplus \left(\bigcup_{v \in V} (v, S)\right)$ und $c(e) := 1$ für alle $e \in E$. Wenn wir einen *ganzzahligen* maximalen Fluss für G' haben, können wir direkt ein maximales Matching für G davon ablesen. Weil der Fluss ganzzahlig ist, haben alle Kanten entweder Fluss 0 oder Fluss 1. Wir wählen die Kanten mit Fluss 1, die sowohl in G als auch G' vorhanden sind, als Matching aus. Offenbar erhalten wir ein gültiges Matching: Weil jeder Knoten nur eine Kante mit Kapazität 1 zur Senke bzw. von der Quelle hat, kann höchstens eine mit ihm inzidente Kante aus $E \cap E'$ Fluss 1 haben. Die Größe des Matchings ist offensichtlich gleich dem Wert des Flusses. Warum ist das Matching maximal? Wenn wir die Perspektive wechseln, sehen wir direkt, dass wir ganz analog aus einem beliebigen Matching auf G einen ganzzahligen Fluss machen können: Die Matching-Kanten bekommen Fluss 1, die übrigen Kanten Fluss 0, die Kanten in $E' \setminus E$ können passend ergänzt werden. Wieder stimmen Größe des Matchings und Wert des ganzzahligen Flusses überein. Gäbe es also ein größeres Matching, wäre der ganzzahlige Fluss im Widerspruch zu unserer Voraussetzung nicht maximal gewesen. Wir können also tatsächlich so das Matchingproblem für bipartite Graphen lösen.

4.1 Der Algorithmus von Ford und Fulkerson

Wir haben im letzten Abschnitt das Matchingproblem in bipartiten Graphen auf das Flussproblem reduziert, dabei haben wir aber voraussetzen müssen,

dass es in dem von uns konstruierten Graphen einen ganzzahligen maximalen Fluss muss. Ist das möglich oder kann es vielleicht sein, dass es keinen ganzzahligen maximalen Fluss gibt? Wir werden uns konstruktiv davon überzeugen, dass das nicht so ist: Wir beschreiben einen Algorithmus, der nur ganzzahlige Flüsse berechnet und weisen nach, dass er einen maximalen Fluss berechnet.

Wir beginnen auch hier mit einer trivialen Idee. Wir können im Netzwerk mit dem leeren Fluss (Fluss 0 auf allen Kanten) starten. Wenn wir irgendeinen Fluss haben, dann suchen wir einen gerichteten Weg von Q nach S, wobei wir nur Kanten berücksichtigen, die noch über Restkapazität verfügen, also Kanten e mit $c(e) - \Phi(e) > 0$. Finden wir einen solchen Weg P, können wir auf allen Kanten in P den Fluss um $\min\{c(e)-\Phi(e) \mid e \in P\}$ vergrößern. Wir haben weiterhin einen Fluss, der Wert ist um $\min\{c(e)-\Phi(e) \mid e \in P\}$ größer geworden. Wenn wir keinen solchen Weg mehr finden, hören wir auf. Es ist nicht schwer einzusehen, dass das im Allgemeinen nicht zu einem maximalen Fluss führt. Im kleinen Netzwerk $G = (V, E)$ mit $V = \{Q, S, A, B\}$ mit den Kanten $E = \{(Q, A), (Q, B), (A, B), (A, S), (B, S)\}$ und $c(e) = 1$ für alle $e \in E$ gibt es einen offensichtlich maximalen Fluss Φ mit Wert 2. Wenn wir aber mit dem leeren Fluss starten und den Weg $Q \rightsquigarrow A \rightsquigarrow B \rightsquigarrow S$ wählen, erhalten wir einen Fluss mit Wert 1, der sich mit unserem Algorithmus nicht mehr verbessern lässt.

Wir sehen, dass zwar jeder Weg von der Quelle zur Senke mit positiven Restkapazitäten dazu geeignet ist, den aktuellen Fluss zu vergrößern, nicht jeder Weg aber schließlich zu einem maximalen Fluss führt. In diesem Sinn kann man sich also bei der Wahl des Weges falsch entscheiden. Es liegt nahe, einem Algorithmus zu erlauben, solche ungünstigen Entscheidungen wieder zurückzunehmen. Dann muss man aber aufpassen, dass das Verfahren nicht in endlose Schleifen geraten kann. Wir werden uns aber eine einfache Methode ansehen, mit der die Rücknahme solcher Entscheidungen möglich ist, das Verfahren aber garantiert endlich bleibt.

Wir betrachten ein Netzwerk $G = (V, E)$ und einen Fluss Φ auf G. Um eine übersichtlichere Notation zu bekommen, definieren wir uns jetzt einen neuen gerichteten Graphen, der alle Wege enthalten soll, die wir in G benutzen können, um einen solchen Q-S-Weg zu konstruieren. Neben den Kanten mit positiven Restkapazitäten könnten wir ja auf Kanten mit positivem Fluss einen Teil dieses Flusses oder auch den ganzen Fluss auf der Kante wieder wegnehmen. Das symbolisieren wir in unserem neuen Graphen, indem wir für eine Kante $e = (v, w)$ mit $\Phi(e) > 0$ eine Kante (w, v) mit Kapazität $\Phi(e)$ in den neuen Graphen aufnehmen. Die Kante (w, v) nennen wir *Rückwärtskante*, wir schreiben $\mathrm{rev}(e) = (w, v)$ für eine Kante $e = (v, w)$. Wir halten

diese Konstruktion, die sich als nützlich erweisen wird, auch formal in einer Definition fest.

Definition 4.2 (Restgraph)**.** *Zu einem Netzwerk (G, c) ($G = (V, E)$) und einem Fluss $\Phi\colon E \to \mathbb{R}_0^+$ auf G definieren wir den* Restgraphen *$Rest_\Phi = ((V, E'), c')$ mit Kantenmenge*

$$E' := \{e = (v, w) \mid e \in E \text{ und } c(e) - \Phi(e) > 0\} \cup \{(w, v) \mid e = (v, w) \in E \text{ und } \Phi(e) > 0\}$$

und Restkapazitäten $r_\Phi\colon E' \to \mathbb{R}_0^+$ mit

$$r_\Phi(e) := \begin{cases} c(e) - \Phi(e) & \text{falls } e \in E, \\ \Phi(e) & \text{falls } rev(e) \in E. \end{cases}$$

Die Definition des Restgraphen setzt voraus, dass das Netzwerk G asymmetrisch ist. Sonst wäre in der Definition von E' die Vereinigung nicht notwendig disjunkt und (viel schlimmer) die Fallunterscheidung in der Definition von r_Φ so nicht möglich. Einen gerichteten Q-S-Weg im Restgraphen nennen wir einen *flussvergrößernden Weg*. Darin steckt natürlich implizit die Behauptung, dass wir aus einem flussvergrößernden Weg in Rest_Φ einen Fluss mit größerem Wert als $w(\Phi)$ für G bekommen können. Wir halten diese wichtige Aussage fest.

Lemma 4.3. *Sei $(G = (V, E), c)$ ein Netzwerk, Φ ein Fluss auf G, $Rest_\Phi = ((V, E'), r_\Phi)$ der Restgraph dazu, $P = (e_1, e_2, \ldots, e_l)$ ein einfacher gerichteter Weg in $Rest_\Phi$, der in Q beginnt und S endet. Sei $r := \min\{r_\Phi(e) \mid e \in P\}$. Betrachte $\Phi'\colon E \to \mathbb{R}_0^+$ mit*

$$\Phi'(e) := \begin{cases} \Phi(e) + r & \text{falls } e \in P, \\ \Phi(e) - r & \text{falls } rev(e) \in P, \\ \Phi(e) & \text{sonst.} \end{cases}$$

Φ' ist ein Fluss für (G, c) mit $w(\Phi') = w(\Phi) + r > w(\Phi)$.

Beweis. Es folgt direkt aus der Definition des Restgraphen (Definition 4.2), dass $r > 0$ gilt. Weil P ein einfacher Weg ist, führt genau eine Kante aus Q heraus. Weil in G keine Kante in Q hineinführt, kann es sich dabei nicht um eine Rückwärtskante handeln. Es ist also offensichtlich $w(\Phi') = w(\Phi) + r > w(\Phi)$ wie behauptet. Wir müssen also nur noch zeigen, dass es sich bei Φ' tatsächlich um einen Fluss handelt.

Die Definition von r stellt offenbar sicher, dass Φ' die Kantenkapazitäten respektiert und nicht negativ ist. Damit bleibt gemäß Definition eines Flusses (Definition 4.1) nur die Einhaltung der Kirchhoff-Regel zu überprüfen.
Für Q und S ist nichts zu prüfen, für Knoten $v \notin P$ ändert sich offenbar nichts. Wir betrachten also einen Knoten $v \in V \setminus \{Q, S\}$, der in P vorkommt. Es gib also ein i, so dass $e_i = (u, v)$ und $e_{i+1} = (v, w)$ mit $u \in V \setminus \{S\}$ und $w \in V \setminus \{Q\}$. Bezüglich Φ ist die Kirchhoff-Regel in v erfüllt, wir müssen nachrechnen, dass das bezüglich Φ' auch der Fall ist. Wir erinnern uns daran, dass die Kirchhoff-Regel Gleichheit der Summe der Flüsse der eingehenden Kanten und der Summe der Flüsse der ausgehenden Kanten verlangt. Jetzt führen wir eine vollständige Fallunterscheidung nach der Art der Kanten e_i und e_{i+1} durch.

1. Fall: $e_i \in E$ und $e_{i+1} \in E$
Wenn beide Kanten in E vorhanden sind, ist der Fluss Φ' auf beiden Kanten im Vergleich zu Φ um r vergrößert. Beide Summen vergrößern sich um r, die Gleichheit bleibt also erhalten.

2. Fall: $e_i \notin E$ und $e_{i+1} \notin E$
Wenn beide Kanten in E nicht vorhanden sind, es sich also bei beiden Kanten um Rückwärtskanten handelt, ist der Fluss Φ' auf den beiden Kanten $\mathrm{rev}(e_i)$ und $\mathrm{rev}(e_{i+1})$ im Vergleich zu Φ um r verkleinert. Beide Summen verkleinern sich um r, die Gleichheit bleibt also erhalten.

3. Fall: $e_i \in E$ und $e_{i+1} \notin E$
Nur die Kanten e_{i+1} ist Rückwärtskante, wir vergrößern also den Fluss auf e_i um r und verkleinern ihn auf $\mathrm{rev}(e_{i+1})$ um r. Die Kante e_i taucht in der Summe der Flüsse der eingehenden Kanten auf. Weil $e_{i+1} = (v, w)$ aus v hinausführt, taucht auch $\mathrm{rev}(e_{i+1})$ in der Summe der Flüsse der eingehenden Kanten auf und diese Summe ändert sich nicht. Weil auch die andere Summe sich nicht ändert, bleibt die Gleichheit erhalten.

4. Fall: $e_i \notin E$ und $e_{i+1} \in E$
Der Fall ist symmetrisch zum 3. Fall: Nur die Kante e_i ist Rückwärtskante, wir vergrößern also den Fluss auf e_{i+1} um r und verkleinern ihn auf $\mathrm{rev}(e_i)$ um r. Die Kante e_{i+1} taucht in der Summe der Flüsse der ausgehenden Kanten auf. Weil $e_i = (u, v)$ in v hineinführt, taucht auch $\mathrm{rev}(e_i)$ in der Summe der Flüsse der ausgehenden Kanten auf und diese Summe ändert sich nicht. Weil auch die andere Summe sich nicht ändert, bleibt die Gleichheit erhalten. □

Wir bauen jetzt die Erkenntnis, dass wir aus einem Fluss mit Hilfe eines flussvergrößernden Weges einen Fluss mit größerem Wert machen können, zu einem Algorithmus aus. Dieser Algorithmus ist der am Anfang dieses Abschnitts versprochene konstruktive Beweis, dass es in Netzwerken mit ausschließlich ganzzahligen Kapazitäten immer einen ganzzahligen maximalen

Fluss gibt. Natürlich ergibt sich die Aussage erst aus dem Korrektheitsbeweis des Algorithmus.

Algorithmus 4.4 (**Algorithmus von Ford und Fulkerson**).
1. $\Phi := \emptyset$
2. *Berechne Restgraph* $Rest_\Phi$.
3. *Markiere* Q.
4. *While* S *nicht markiert*
5. *If* $\exists$ *in* $Rest_\Phi$*:*
   Knoten x *markiert, Knoten* y *nicht markiert, Kante* (x, y)
6. *Then Markiere* y *mit „erreicht von* x*“.*
7. *Else Exit mit Ausgabe* Φ.
8. *Betrachte markierten Weg* P *von* Q *nach* S.
9. $r := \min\{r_\Phi(e) \mid e \in P\}$
10. *Für alle* $e \in E \cap P$: $\Phi(e) := \Phi(e) + r$
11. *Für alle* $rev(e) \in E \cap P$: $\Phi(e) := \Phi(e) - r$
12. *Weiter bei 2.*

Wir beobachten, dass ein Durchlauf der Zeilen 2–12 in Zeit $O(|V| + |E|)$ zu bewerkstelligen ist. Eine Graphtraversierung (zum Beispiel in Form einer Tiefen- oder Breitensuche) reicht offensichtlich aus. Weil P ein einfacher Weg ist, können wir die eigentliche Flussvergrößerung (Zeilen 8–11) sogar in Zeit $O(|V|)$ durchführen. Lemma 4.3 garantiert, dass Φ am Anfang und Ende jeden Durchlaufs ein Fluss ist. Wir sehen auch, dass nur ganzzahlige Flüsse berechnet werden: Die Kapazitäten sind ganzzahlig, initial sind alle Flüsse 0, also auch ganzzahlig. Die einzigen Rechenoperationen sind die Minimumbildung, Addition und Subtraktion, dabei können offenbar nur ganze Zahlen erzeugt werden. Auch dass der Algorithmus endlich ist, ist leicht einzusehen. In jeder Runde wächst der Fluss Φ um mindestens 1. Der Fluss Φ kann aber nicht beliebig wachsen, er ist ja sowohl durch $\sum_{e=(Q,\cdot)} c(e)$ als auch durch $\sum_{e=(\cdot,S)} c(e)$ trivial nach oben beschränkt. Also ist die Anzahl der Durchläufe ebenso nach oben beschränkt. Leider sieht man nicht so direkt, dass am Ende der Fluss auch tatsächlich maximal ist. Wir werden uns darum als Erstes kümmern und uns dann überlegen, ob sich vielleicht auch eine befriedigende Laufzeitschranke gewinnen lässt.
Für den Korrektheitsbeweis gehen wir scheinbar einen Umweg, indem wir zunächst einen neuen Begriff einführen. Für ein Netzwerk (G, c) definieren wir einen Q-S-Schnitt und seinen Wert.

Definition 4.5. *Sei* $(G = (V, E), c)$ *ein Netzwerk mit Quelle* Q *und Senke* S*. Eine Partitionierung* (V_Q, V_S) *von* V *(also* $V = V_Q \cup V_S$*) mit* $Q \in V_Q$ *und*

$S \in V_S$ *heißt* Q-S-Schnitt. *Der* Wert $w(V_Q, V_S)$ *eines* Q-S-*Schnittes* (V_Q, V_S) *ist durch*

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

definiert.

Q-S-Schnitte erscheinen uns zunächst als Umweg, der mit maximalen Flüssen nicht viel zu tun hat. Es gibt aber eine tiefe Beziehung zwischen Q-S-Schnitten und Flüssen, die wir in Form des Max-Flow-Min-Cut-Theorems festhalten.

Theorem 4.6 (Max-Flow = Min-Cut)**.** *Sei* (G, c) *ein Netzwerk. Der Wert eines maximalen Flusses für* (G, c) *ist gleich dem Wert eines minimalen* Q-S-*Schnittes für* (G, c)*.*

Beweis. Wir zeigen zunächst für alle Flüsse Φ und alle Q-S-Schnitte (V_Q, V_S), dass jedenfalls $w(\Phi) \leq w(V_Q, V_S)$ gilt. Wir erinnern uns zunächst an die Definition des Wertes eines Flusses:

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

Außerdem erinnern wir uns an die Kirchhoff-Regel:

$$\forall v \in V \setminus \{Q, S\} \colon \sum_{e=(v,\cdot)} \Phi(e) = \sum_{e=(\cdot,v)} \Phi(e) \tag{1}$$

Natürlich können wir Gleichung (1) äquivalent zu

$$\forall v \in V \setminus \{Q, S\} \colon \sum_{e=(v,\cdot)} \Phi(e) - \sum_{e=(\cdot,v)} \Phi(e) = 0 \tag{2}$$

umformen. Wenn wir die Differenz aus Gleichung (2) für alle Knoten $v \in V_Q$ betrachten, so erhalten wir einzig für den Knoten $Q \in V_Q$ nicht 0. Weil keine Kanten in Q hineinführen (siehe Definition 4.1), erhalten wir für $v = Q$ gerade $w(\Phi)$ als Differenz. Wir können also

$$\sum_{v \in V_Q} \left(\sum_{e=(v,\cdot)} \Phi(e) - \sum_{e=(\cdot,v)} \Phi(e) \right) = w(\Phi) \tag{3}$$

schreiben. Wir betrachten die beiden Summen über die Kanten und spalten sie gedanklich auf nach der Art der Kanten, über die summiert wird. Wir unterscheiden Kanten $e \in E \cap (V_Q \times V_Q)$, Kanten $e \in E \cap (V_Q \times V_S)$ und

Kanten $e \in E \cap (V_S \times V_Q)$. Kanten, die ausschließlich innerhalb von V_S verlaufen, tauchen in den Summen offensichtlich nicht auf.
Die Kanten $e \in E \cap (V_Q \times V_S)$ tauchen ausschließlich in der ersten, der positiven Summe auf. Die Kanten $e \in E \cap (V_S \times V_Q)$ tauchen ausschließlich in der zweiten, der negativen Summe auf. Die Kanten $e \in E \cap (V_Q \times V_Q)$ schließlich tauchen in beiden Summen auf, einmal positiv und einmal negativ. Wir verändern also den Wert der Summe nicht, wenn wir diese Kanten einfach auslassen. Darum erhalten wir die folgende Gleichung (4).

$$\sum_{e \in E \cap (V_Q \times V_S)} \Phi(e) - \sum_{e \in E \cap (V_S \times V_Q)} \Phi(e) = w(\Phi) \tag{4}$$

Wir sind jetzt fast am Ziel. Gemäß Definition 4.5 gilt

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

und es folgt

$$w(V_Q, V_S) \geq \sum_{e \in E \cap (V_Q \times V_S)} \Phi(e),$$

weil $c(e) \geq \Phi(e)$ für alle Kanten $e \in E$ gilt. Außerdem gilt

$$- \sum_{e \in E \cap (V_S \times V_Q)} \Phi(e) \leq 0,$$

weil $\Phi(e) \geq 0$ für allen Kanten $e \in E$ gilt. Es folgt also $w(\Phi) \leq w(V_Q, V_S)$ wie behauptet für beliebige Flüsse Φ und beliebige Q-S-Schnitte (V_Q, V_S).
Jetzt zeigen wir für einen ausgewählten Fluss Φ^* und einen ausgewählten Q-S-Schnitt (V_Q^*, V_S^*) die Gleichheit der Werte, also $w(\Phi^*) = w(V_Q^*, V_S^*)$. Weil für beliebige Flüsse und beliebige Q-S-Schnitte immer $w(\Phi) \leq w(V_Q, V_S)$ gilt, muss es sich bei Φ^* um einen maximalen Fluss und bei (V_Q^*, V_S^*) um einen minimalen Schnitt handeln.
Die Wahl von Φ^* fällt uns nicht schwer, wir betrachten den Fluss, den der Algorithmus von Ford und Fulkerson (Algorithmus 4.4) am Ende ausgibt. Wir verlassen damit den Algorithmus aber noch nicht, er wird uns zusätzlich dabei helfen, den Q-S-Schnitt (V_Q^*, V_S^*) geeignet zu definieren. Wir betrachten dazu den Restgraphen Rest_Φ aus der letzten Runde von Algorithmus 4.4. Wir wissen, dass die Quelle Q markiert ist und die Senke S nicht markiert ist, andernfalls hätte es eine weitere Runde im Algorithmus gegeben. Es kann noch weitere markierte und auch noch weitere nicht markierte Knoten geben, wir benutzen nun diese Markierungen zur Definition von (V_Q^*, V_S^*): Alle markierten Knoten fassen wir in V_Q^* zusammen, alle nicht markierten Knoten gehören

zu V_S^*. Offensichtlich haben wir so einen Q-S-Schnitt definiert. Betrachten wir nun eine Kante $e = (v, w) \in E$ mit $v \in V_Q^*$ und $w \in V_S^*$, es ist also v markiert und w nicht markiert. Wir haben $\Phi(e) = c(e)$, denn andernfalls hätte es in Rest_Φ eine Kante $e = (v, w)$ mit Restkapazität $r_\Phi(e) = c(e) - \Phi(e) > 0$ gegeben. Über diese Kante wäre aber vom markierten Knoten v aus der nicht markierte Knoten w erreichbar gewesen und folglich wäre w markiert worden – im Widerspruch zur Voraussetzung, dass w nicht markiert ist.
Wenn wir an Gleichung (4) zurückdenken, sehen wir, dass das so weit schon einmal günstig ist für die Gleichheit von $w(\Phi^*)$ und $w(V_Q^*, V_S^*)$: Für die Kanten, über die wir sowohl im Wert des Q-S-Schnittes also auch im Wert des Flusses addieren, sind Kapazitäten und Flüsse auf den Kanten gleich. Allerdings wird für die Berechnung des Wertes des Flusses der Fluss auf den von V_S^* nach V_Q^* verlaufenden Kanten noch abgezogen. Wir betrachten jetzt also eine Kante $e = (v, w) \in E$ mit $v \in V_S^*$ und $w \in V_Q^*$, es ist also v nicht markiert und w markiert. Wir sehen, dass $\Phi(e) = 0$ gilt, denn andernfalls hätte es in Rest_Φ eine Kante $\text{rev}(e) = (w, v)$ gegeben mit Restkapazität $r_\Phi(e) = \Phi(e) > 0$. Über diese Kante wäre aber vom markierten Knoten w aus der nicht markierte Knoten v erreichbar gewesen und folglich wäre v markiert worden – im Widerspruch zur Voraussetzung, dass v nicht markiert ist.
Insgesamt (also auch Gleichung (4) berücksichtigend) haben wir

$$\begin{aligned} w(\Phi^*) &= \sum_{e \in E \cap (V_Q^* \times V_S^*)} \Phi(e) - \sum_{e \in E \cap (V_S^* \times V_Q^*)} \Phi(e) \\ &= \sum_{e \in E \cap (V_Q^* \times V_S^*)} c(e) - \sum_{e \in E \cap (V_S^* \times V_Q^*)} 0 \\ &= \sum_{e \in E \cap (V_Q^* \times V_S^*)} c(e) = w(V_Q^*, V_S^*) \end{aligned}$$

bewiesen und die Behauptung folgt wie oben erläutert. □

Wir wissen jetzt also, dass der Algorithmus von Ford und Fulkerson tatsächlich einen maximalen Fluss berechnet und das Flussproblem löst. Wir halten dieses Ergebnis fest.

Theorem 4.7. *Der Algorithmus von Ford und Fulkerson (Algorithmus 4.4) berechnet für ein beliebiges Netzwerk $(G = (V, E), c)$ einen maximalen Fluss in Zeit $O(B \cdot (|V| + |E|))$, dabei ist $B = \min\Big\{ \sum_{e=(Q,\cdot) \in E} c(e), \sum_{e=(\cdot,S) \in E} c(e), |V| \cdot \max\{c(e) \mid e \in E\} \Big\}$.*

Beweis. Die Korrektheit folgt wie gesehen direkt aus Lemma 4.3 und Theorem 4.6. Die Aussage über die Laufzeit haben wir ebenfalls schon diskutiert. Der Fluss wächst in jeder Runde um mindestens 1, jede Runde kann in Zeit $O(|V| + |E|)$ durchgeführt werden. Wir starten mit dem leeren Fluss, der Wert 0 hat, B ist eine triviale obere Schranke für den Wert eines maximalen Flusses: Wir können nicht mehr Fluss von Quelle zu Senke bringen, als wir an Fluss aus der Quelle herausbekommen oder in die Senke hineinbekommen, $|V| \cdot \max\{c(e) \mid e \in E\}$ ist offenbar für beides eine triviale obere Schranke. □

Die Laufzeitschranke aus Theorem 4.7 ist im Allgemeinen nicht polynomiell: Sie hängt von den in der Eingabe vorkommenden Zahlen ab, ist polynomiell nur dann, wenn alle in der Eingabe vorkommenden Zahlen polynomiell in der Eingabelänge beschränkt sind. Man spricht von einer *pseudopolynomiellen* Laufzeit, wie wir sie vielleicht schon von einem Algorithmus für das Rucksackproblem kennen. In der Praxis mag es sein, dass wir nur mit Netzwerken mit kleinen Kapazitäten zu tun haben und der Algorithmus von Ford und Fulkerson ausreichend schnell ist, befriedigend ist so eine Laufzeit aber sicher nicht.

Es könnte sein, dass nur unsere Abschätzung ein pseudopolynomielles Resultat hat, die Laufzeit des Algorithmus von Ford und Fulkerson sich in Wahrheit aber doch polynomiell nach oben abschätzen lässt. Wir können die Schranke B leicht auf den Wert eines maximalen Flusses $w(\Phi_{\text{opt}})$ verbessern. Es ist klar, dass der Wert eines maximalen Flusses tatsächlich exponentiell groß in der Länge der Eingabe (also der Länge der Codierung des Netzwerkes) sein kann: Das kleine Netz mit $V = \{Q, S\}$, $E = \{(Q, S)\}$ und $c((Q, S)) = k$ hat offenbar diese Eigenschaft. Es ist aber denkbar, dass unsere triviale Abschätzung für die Differenz zweier im Algorithmenablauf zeitlich aufeinanderfolgender Flüsse zu pessimistisch ist und Algorithmus 4.4 doch immer in Polynomialzeit einen maximalen Fluss findet; für das gerade beschriebene Mini-Netzwerk ist das offensichtlich der Fall. Leider ist das im Allgemeinen aber nicht immer so, wie man sich an einem wiederum sehr einfachen Beispiel klar machen kann. Wir betrachten das Netzwerk $(G = (V, E), c)$ mit $V = \{Q, A, B, S\}$, $E = \{(Q, A), (Q, B), (A, B), (A, S), (B, S)\}$, $c(e) = M$ für $e \in E \setminus \{(A, B)\}$ und $c((A, B)) = 1$. Offensichtlich gibt es einen maximalen Fluss mit Wert $2M$. Man sieht aber leicht ein, dass man im Restgraphen immer die Kante (A, B) bzw. die Kante (B, A) in den zu konstruierenden flussvergrößernden Weg aufnehmen kann, so dass sich der aktuelle Fluss in jeder Runde um genau 1 vergrößert. Weil M beliebig groß werden kann, sehen wir, dass die Laufzeit tatsächlich exponentiell groß werden kann.

4.2 Der Algorithmus von Dinic

Wir können mit flussvergrößernden Wegen maximale Flüsse berechnen, das funktioniert mit dem Algorithmus von Ford und Fulkerson aber leider nicht unbedingt besonders effizient. Das erinnert an die Situation bei Matchings: Dort konnten wir mit M-verbessernden Pfaden maximale Matchings berechnen, allerdings musste man die M-verbessernden Pfade geschickt suchen, um wirklich effizient zu sein. Dass es so leicht war, das Matchingproblem für bipartite Graphen auf das Flussproblem zu übertragen, zeigt uns, dass es tatsächlich inhaltliche Beziehungen zwischen dem Matchingproblem und dem Flussproblem gibt. Es erscheint darum vernünftig zu sein, wenn wir versuchen, die wesentlichen Ideen aus den guten Matchingalgorithmen (Algorithmus von Hopcroft und Karp (Algorithmus 3.12) für bipartite Graphen und Algorithmus von Micali und Vazirani (Algorithmus 3.15) für allgemeine Graphen) zu übertragen. Welche Ideen waren das? Wir hatten zwei zentrale Ideen: Zum einen hat es sich als günstig erwiesen, sich auf kürzeste M-verbessernde Wege zu konzentrieren. Außerdem war es sinnvoll, immer möglichst viele M-verbessernde Pfade gleichzeitig zur Matchingverbesserung zu benutzen. Übertragen wir das auf das Flussproblem, so bedeutet das, dass wir kürzeste flussvergrößernde Pfade suchen und so viele flussvergrößernde Pfade wie möglich gleichzeitig zur Flussvergrößerung benutzen wollen. Bevor wir das umsetzen, machen wir uns noch ein paar Gedanken zu kürzesten Pfaden in Netzwerken. Als Maß für die Länge eines Weges verwenden wir wie bei Matchings einfach die Anzahl der Kanten. Weil alle Wege in der Quelle Q starten, ist es sinnvoll, für jeden Knoten v den Abstand zur Quelle zur Quelle $d(v)$ zu betrachten, also die minimale Anzahl Kanten, die man benutzen muss, um auf einem gerichteten Weg von der Quelle Q zum Knoten v zu kommen. Wir erinnern uns daran, dass wir nicht im Netzwerk nach flussvergrößernden Wegen suchen, sondern im Restgraphen. Der Restgraph hängt nicht nur vom Netzwerk, sondern auch vom aktuellen Fluss ab. Es können Kanten wegfallen (wenn wir eine Kante im Netzwerk ganz ausschöpfen und oder von einer Kante im Netzwerk allen Fluss herunternehmen) und auch Kanten hinzukommen (wenn wir von einer ausgeschöpften Kante im Netzwerk Fluss herunternehmen oder auf eine Kante ohne Fluss Fluss addieren). Wir wollen uns vorab überlegen, wie sich die Abstände von der Quelle durch das Wegnehmen und Zufügen einzelner Kanten ändern können.

Lemma 4.8. *Sei $G = (V, E)$ ein gerichteter Graph mit $Q \in V$. Sei $d\colon V \to \mathbb{N}_0$ so gegeben, dass für alle $v \in V$ die Funktion $d(v)$ die Länge eines kürzesten gerichteten Weges von Q nach v bezeichnet.*

Seien $v, w \in V$, sei $G^+ := (V, E^+)$ mit $E^+ := E \cup \{(v, w)\}$, gebe $d^+ \colon V \to \mathbb{N}_0$ für alle $u \in V$ die Länge eines kürzesten gerichteten Weges von Q nach u in G^+ an, sei $G^- := (V, E^-)$ mit $E^- := E \setminus \{(v, w)\}$, gebe $d^- \colon V \to \mathbb{N}_0$ für alle $u \in V$ die Länge eines kürzesten gerichteten Weges von Q nach u in G^- an.

1. $(d(v) \geq d(w) - 1) \Rightarrow (\forall u \in V \colon d^+(u) = d(u))$
2. $\forall u \in V$ *mit* $d(u) \leq d(w) - 1 \colon d^-(u) = d(u)$
3. $\forall u \in V \colon d^-(u) \geq d(u)$

Beweis. Wenn die Kante (v, w) in G schon vorhanden ist, so sind G und G^+ gleich, so dass die Gleichheit von d und d^+ nicht gezeigt werden braucht. Ebenso muss die Gleichheit von d und d^- nicht gezeigt werden, wenn (v, w) in G nicht vorhanden ist. Wir beschränken uns also auf die Fälle, in denen sich die zwei jeweils betrachteten Graphen unterscheiden.

1. Wir haben $d^+(u) \leq d(u)$ für alle $u \in V$, da durch das Hinzufügen einer Kante Distanzen nur kürzer werden können. Wird für einen Knoten $u \in V$ die Distanz kürzer, gilt also $d^+(u) < d(u)$, so liegt die neue Kante (v, w) auf dem kürzesten Weg von Q nach u in G^+. Dieser kürzeste Weg setzt sich zusammen aus dem kürzesten Weg von Q nach v, der Kante (v, w) und dem kürzesten Weg von w nach u. In G gab es schon einen kürzesten Weg von Q nach w, der nach u fortgesetzt werden kann. Der kürzeste Weg von w nach u ist in G und G^+ gleich, weil die Kante (v, w) nicht benutzt werden kann. Der kürzeste Weg in G^+ über v nach w hat Länge $d(v) + 1$ und wir haben $d(v) + 1 \geq d(w)$ nach Voraussetzung. Es folgt $d^+(u) \geq d(u)$ für alle Knoten $u \in V$ und mit $d^+(u) \leq d(u)$ folgt die Behauptung.

2. Auf kürzesten Wegen von Q nach u können nur Knoten x mit $d(x) \leq d(u) - 1$ vorkommen. Wir haben $d(u) \leq d(w) - 1$ gemäß Voraussetzung und sehen, dass w auf einem kürzesten Weg zu u nicht vorkommen kann. Folglich kann die Kante (v, w) auf kürzesten Wegen zu solchen Knoten u nicht vorkommen und das Entfernen der Kante kann die Distanz nicht verändern, wir haben also $d^-(u) = d(u)$ wie behauptet.

3. Wir sehen direkt, dass durch das Entfernen einer Kante Distanzen nur länger werden können.

□

Wir wollen jetzt im Restgraphen kürzeste Q-S-Wege suchen. Dabei können wir offensichtlich alle Knoten $v \in V \setminus \{S\}$ mit $d(v) \geq d(S)$ ignorieren. Wir

fassen gedanklich alle Knoten mit gleicher Distanz zu Q zu einer Ebene zusammen und sehen direkt ein, dass kürzeste Wege nur von einer solchen Ebene zur nächsten führen. Wir fassen diese Erkenntnis in einer Definition zusammen: Unser Ziel ist es, einen besser strukturierten Ersatz für den Restgraphen zu bekommen.

Definition 4.9. *Sei $(G = (V, E), c)$ ein Netzwerk, Φ ein Fluss auf G, $Rest_\Phi$ der Restgraph dazu, gebe $d\colon V \to \mathbb{N}_0$ für alle $v \in V$ die Länge eines kürzesten Weges von Q nach v.*
Für $i \in \{0, 1, \ldots, d(S)\}$ sei das i-te Niveau V_i *durch*

$$V_i := \begin{cases} \{v \in V \mid d(v) = i\} & \text{für } i < d(S) \\ \{S\} & \text{für } i = d(S) \end{cases}$$

definiert. Das Niveaunetzwerk $N_\Phi := (V_\Phi, E_\Phi, r_\Phi)$ *ist definiert durch*

$$\begin{aligned} V_\Phi &:= \bigcup_{i=0}^{d(S)} V_i \\ E_i &:= \{(v, w) \in V_{i-1} \times V_i \mid (v, w) \in Rest_\Phi\} \\ E_\Phi &:= \bigcup_{i=1}^{d(S)} E_i \end{aligned}$$

und r_Φ wie im Restgraphen.

Natürlich kann bei gegebenem Netzwerk und Fluss das Niveaunetzwerk in Zeit $O(|V| + |E|)$ konstruiert werden. Wir erinnern uns daran, dass wir nicht nur einen flussvergrößernden Weg addieren wollen, wir möchten gerne den Fluss in jeder Runde um möglichst viel vergrößern. Dazu werden wir den Begriff des flussvergrößernden Weges durch einen anderen Begriff ersetzen. Bisher haben wir flussvergrößernde Wege im Restgraphen gesucht, das könnten wir natürlich auch im Niveaunetzwerk machen. Wenn wir einen solchen flussvergrößernden Weg im Niveaunetzwerk gefunden haben und *dort* betrachten, so erkennen wir, dass er ein Fluss für das Niveaunetzwerk ist. Diese Einsicht legt nahe, dass man auch einen beliebigen Fluss *im Niveaunetzwerk* berechnen könnte. Wir sehen analog zu Lemma 4.3 ein, dass wir einen solchen Fluss im Niveaunetzwerk N_Φ direkt in das ursprüngliche Netzwerk übertragen und dort zum aktuellen Fluss Φ „addieren“ können, dabei ist natürlich wieder auf die unterschiedliche Behandlung von Kanten und Rückwärtskanten zu achten. Das Ergebnis dieser Operation ist dann wiederum ein Fluss, dessen Wert im Vergleich zu Φ genau um den Wert des im Niveaunetzwerks

konstruierten Flusses gewachsen ist. Zunächst scheint durch diese Einsicht nicht viel gewonnen: Wir wollen einen größeren Fluss in einem Netzwerk finden und schlagen dafür vor, einen möglichst großen Fluss in einem anderen Netzwerk, dem Niveaunetzwerk, zu finden. Die Idee klingt zirkulär. Sie ist aber durchaus sinnvoll, wenn wir im Niveaunetzwerk einen einfachen Algorithmus benutzen können und dann trotzdem im eigentlichen Netzwerk gute Fortschritte machen. Wir benutzen unsere Idee der flussvergrößernden Wege als Ausgangspunkt. Wenn wir einen flussvergrößernden Weg P haben, benutzen wir $\min\{r_\Phi(e) \mid e \in P\}$ als Wert, der im ursprünglichen Netzwerk auf den Fluss aufaddiert wird. Die Begründung dafür ist leicht zu sehen: Es handelt sich um den größten Wert, der auf diesem Weg addiert werden kann, es gibt eine Kante, die nicht mehr Flusszunahme zulässt. Im Restgraphen schöpfen wir mindestens eine Kante also voll aus, wir sagen, diese Kante ist *saturiert*. Wir verallgemeinern diese Idee, indem wir so lange *im Niveaunetzwerk* nach flussvergrößernden Wegen suchen, bis auf jedem solchen Weg mindestens eine Kante saturiert ist. Den so erhaltenen Fluss nennen wir Sperrfluss. Wir halten das jetzt auch formal in einer Definition fest.

Definition 4.10. *Sei $(G = (V, E), c)$ ein Netzwerk, Φ ein Fluss auf G, N_Φ das Niveaunetzwerk dazu, Ψ ein Fluss auf N_Φ.*
Eine Kante $e \in E_\Phi$ heißt saturiert*, wenn $\Psi(e) = r_\Phi(e)$ gilt.*
Der Fluss Ψ heißt Sperrfluss*, wenn auf jedem gerichteten Weg von Q nach S in N_Φ mindestens eine saturierte Kante liegt.*

Unser nächstes Ziel liegt jetzt auf der Hand, wir wollen Sperrflüsse berechnen. Dazu können wir recht naiv vorgehen: Wir berechnen einen Q-S-Weg P, ziehen für alle Kanten $e \in P$ von $r_\Phi(e)$ genau $\min\{r_\Phi(e) \mid e \in P\}$ ab (addieren also genau diesen Fluss zu der Kante), so dass mindestens eine Kante saturiert ist. Dann wiederholen wir das Ganze und stoppen erst dann, wenn wir keinen Q-S-Weg mehr finden.

Algorithmus 4.11 (**Sperrflussberechnung**).
1. $\Psi := 0$
2. $i := 0$*;* $v_i := Q$
3. *While* $v_i \neq S$
4. *If* $\exists(v_i, w)$
5. *Then* $i := i + 1$*;* $v_i := w$
6. *Else*
7. *Entferne v_i und alle Kanten $(\cdot, v_i)$ aus N_Φ.*
8. $i := i - 1$*; If* $i < 0$ *Then Exit mit Ausgabe von* Ψ*.*
9. $r := \min\{r_\Phi((v_j, v_{j+1})) - \Psi((v_j, v_{j+1})) \mid j \in \{0, 1, \ldots, i-1\}\}$
10. *Für alle* $j \in \{0, 1, \ldots, i-1\}$

11. $\Psi((v_j, v_{j+1})) := \Psi((v_j, v_{j+1})) + r$
12. *If* $\Psi((v_j, v_{j+1})) = r_\Phi((v_j, v_{j+1}))$ *Then Entferne* (v_j, v_{j+1}) *aus* N_Φ.
13. Weiter bei 2.

Lemma 4.12. *Sei* $(G = (V, E), c)$ *ein Netzwerk,* Φ *ein Fluss auf* G*,* N_Φ *das Niveaunetzwerk dazu. Algorithmus 4.11 berechnet in Zeit* $O(n \cdot e)$ *einen Sperrfluss zu* N_Φ*, dabei ist* $n = |V|$ *und* $e = |E|$*.*

Beweis. Es ist leicht einzusehen, dass Algorithmus 4.11 tatsächlich einen Sperrfluss berechnet. Offenbar wird korrekt nach einem Q-S-Weg gesucht, es werden nur Knoten und Kanten entfernt, die entweder in Sackgassen liegen oder saturiert sind. Darum gibt es keinen Q-S-Weg über ausschließlich nichtsaturierte Kanten mehr, wenn der Algorithmus terminiert, und es wurde ein Sperrfluss berechnet.
Wir müssen noch nachweisen, dass die Sperrflussberechnung in Zeit $O(n \cdot e)$ gelingt. Wir regeln zunächst unseren Sprachgebrauch und legen fest, dass eine Pfadkonstruktion alle Schritte umfasst, die ausgeführt werden zwischen dem Ausführen von Zeile 2 ($v_i := Q$) und entweder dem Ende der While-Schleife (also dem Erreichen von S) oder dem Entfernen von v_i in Zeile 7. Jede solche Pfadkonstruktion führt also dazu, dass mindestens eine Kante aus N_Φ entfernt wird. Weil N_Φ höchstens e Kanten enthält, kann es also nicht mehr als e Pfadkonstruktionen geben. Ein Q-S-Pfad hat höchstens Länge n, weil wir ausschließlich kürzeste Q-S-Pfade konstruieren und kürzeste Pfade natürlich kreisfrei sind. Darum dauert jede Pfadkonstruktion auch nur $O(n)$ Schritte. Das Backtracking (Zeile 8) stört dabei nicht, da ein solcher Backtracking-Schritt immer direkt nach Ende einer erfolglosen Pfadkonstruktion erfolgt, keine Pfadkonstruktion also mehrere Backtracking-Schritte umfassen kann. Zusammen ergibt sich so die behauptete Laufzeitschranke von $O(n \cdot e)$. □

Die Sperrflussberechnung dient der Vergrößerung eines aktuellen Flusses, sie gleicht in dieser Hinsicht der Berechnung eines flussvergrößernden Weges im Algorithmus von Ford und Fulkerson (Algorithmus 4.4). Wir binden sie jetzt in den naheliegenden Rahmenalgorithmus ein und erhalten einen vollständigen Algorithmus für das Flussproblem.

Algorithmus 4.13 (**Algorithmus von Dinic**)**.**
1. $\Phi := 0$
2. Repeat
3. *Berechne das Niveaunetzwerk* N_Φ*.*
4. *Berechne einen Sperrfluss* Ψ *mit Algorithmus 4.11.*
5. $\Phi := \Phi + \Psi$
6. Until $\Psi = 0$
7. Ausgabe Φ

Theorem 4.14. *Sei $(G = (V, E), c)$ ein Netzwerk. Der Algorithmus von Dinic (Algorithmus 4.13) berechnet zu G einen maximalen Fluss in Zeit $O(n^2 \cdot e) = O(n^4)$, dabei sind $|V| = n$ und $|E| = e$.*

Beweis. Die Korrektheit des Algorithmus ist offensichtlich. Wir erinnern uns daran, dass eine Sperrflussberechnung in Zeit $O(n \cdot e)$ durchführbar ist (Lemma 4.12). Wir müssen also nur zeigen, dass wir mit $O(n)$ Sperrflussberechnungen auskommen.
Wir betrachten zwei Flüsse Φ und Φ', wobei Φ' im Algorithmus von Dinic aus Φ durch „Addition“ des Sperrflusses Ψ entstanden ist. Wir gehen davon aus, dass $\Phi \neq \Phi'$ gilt, nehmen also an, dass Φ nicht schon maximal ist. Andernfalls stoppt der Algorithmus ja ohnehin nach dieser Runde.
Wir betrachten zu Φ und Φ' die Restgraphen Rest_Φ und $\text{Rest}_{\Phi'}$, die beiden Niveaunetzwerke N_Φ und $N_{\Phi'}$, in denen der Algorithmus den Sperrfluss berechnet, sind ja jeweils Teilgraphen davon. Wir betrachten einen kürzesten Weg von Q nach S im neuen Restgraphen $\text{Rest}_{\Phi'}$ und beobachten, dass dieser Weg nicht auch kürzester Weg im Restgraphen Rest_Φ gewesen sein kann. Andernfalls hätte Ψ mindestens eine Kante dieses Weges saturiert und der Weg wäre so in $\text{Rest}_{\Phi'}$ gar nicht mehr vorhanden. Kürzeste Wege in $\text{Rest}_{\Phi'}$ müssen also entweder länger sein als kürzestes Wege in Rest_Φ oder eine Kante enthalten, die in Rest_Φ noch nicht vorhanden war. Wir erinnern uns daran, dass wir alle Knoten im Niveaunetzwerk mit einem Niveau (ihrem Abstand von der Quelle Q) versehen hatten und alle Kanten im Niveaunetzwerk jeweils von einem Knoten eines Niveaus zu einem Knoten auf dem direkten Nachfolgeniveau führen. Es gibt also nur Kanten (u, v) in N_Φ mit $d(v) = d(u) + 1$. Wenn in $\text{Rest}_{\Phi'}$ eine Kante vorhanden ist, die in Rest_Φ nicht vorhanden war, so kann das nur durch Benutzung einer solchen Kante (u, v) in Ψ geschehen. Wenn (u, v) eine Rückwärtskante ist, so kann die Kante (v, u) gefehlt haben, da sie saturiert war. Es wird dann also eine Kante (v, u) eingefügt, die in Rest_Φ von einem Niveau $i + 1$ zu einem Niveau i führt. Wenn (u, v) nicht Rückwärtskante ist, so kann die Rückwärtskante (v, u) eingefügt werden. Aber auch diese Kante verläuft in Rest_Φ von Niveau $i + 1$ zu Niveau i. Wir sehen direkt, dass durch Einfügen solcher Kanten kürzeste Wege in Rest_Φ nicht kürzer werden können. Wer das nicht direkt einsieht, überzeugt sich leicht davon, dass das eine direkt Folge aus Lemma 4.8 ist. Also kann es insgesamt in $\text{Rest}_{\Phi'}$ nur längere kürzeste Wege als in Rest_Φ geben.
Jetzt sind wir am Ziel. Wir haben gezeigt, dass in jeder Runde des Algorithmus von Dinic die Länge kürzester Q-S-Wege um mindestens 1 wächst. Weil kürzeste Wege kreisfrei sind, ist die Länge kürzester Wege aber durch $n - 1$ nach oben beschränkt. Also kann es nicht mehr als $n - 1$ Runden geben und die Laufzeit von $O(n^2 e) = O(n^4)$ folgt. □

4.3 Der Algorithmus von Malhotra, Pramodh Kumar und Maheshwari

Der Algorithmus von Dinic (Algorithmus 4.13) zeigt uns, dass das Flussproblem deterministisch in Polynomialzeit lösbar ist. Allerdings ist eine Laufzeit von $O(n^2e) = O(n^4)$ sicher nichts, womit man glücklich und zufrieden sein kann. Wir werden uns darum weiter mit dem Flussproblem auseinandersetzen und versuchen, schnellere Algorithmen zu finden. Wir behalten die Ideen, die wir im Algorithmus von Dinic umgesetzt haben, bei: Wir starten mit dem leeren Fluss, berechnen in jeder Runde zum aktuellen Fluss erst das Niveaunetzwerk und dann einen Sperrfluss darin, dann verbessern wir den aktuellen Fluss durch „Addition“ des Sperrflusses. Wir hatten gezeigt, dass wir mit $O(n)$ Sperrflussberechnungen auskommen und brauchen für die Sperrflussberechnung mit Algorithmus 4.11 Zeit $O(n \cdot e)$ für eine Sperrflussberechnung. Das ist ziemlich lang, darum werden hier nach schnelleren Wegen suchen. Wir halten noch einmal explizit fest, was wir schon implizit verwenden. Wir betrachten immer das Flussproblem für ein Netzwerk $(G = (V, E), c)$ mit $|V| = n$ und $|E| = e$.

Die Sperrflussberechnung in Algorithmus 4.11 ist recht naiv, sie sucht einfach irgendeinen kürzesten Q-S-Weg. Dass die naive Wahl eines Weges sehr ungünstig sein kann, hatten wir in extremer Weise beim Algorithmus von Ford und Fulkerson (Algorithmus 4.4) gesehen. Wir wollen uns darum jetzt etwas geschickter anstellen. Wir erlauben uns einen „globalen“ Blick auf das gesamte Netzwerk und versuchen herauszubekommen, welcher Knoten besonders „kritisch“ ist. Dazu schätzen wir, wie viel Fluss wir durch einen Knoten aktuell noch höchstens zusätzlich schicken können. Wir sprechen von einer Schätzung, weil wir die Situation rein lokal am jeweiligen Knoten betrachten und zum Beispiel unberücksichtigt lassen, ob man überhaupt so viel Fluss zu diesem Knoten bringen kann. Knoten, die einen positiven Schätzwert erhalten, sind potentiell interessant. Als besonders „kritisch“ empfinden wir den Knoten, bei dem unser Schätzwert besonders klein ist, er scheint eine Art Flaschenhals in der aktuellen Situation darzustellen. Wir kümmern uns darum speziell um diesen Knoten und hoffen so, insgesamt einen möglichst großen Sperrfluss zu bekommen. Wir formalisieren diese Ideen jetzt und beginnen damit, unseren Schätzwert, den wir künftig *Potenzial* eines Knotens nennen wollen, formal zu definieren.

Definition 4.15. *Sei $(G = (V, E), c)$ ein Netzwerk, $\Phi\colon E \to \mathbb{R}_0^+$ ein Fluss dazu, Rest_Φ das zugehörige Restnetzwerk, $N_\Phi = (V_\Phi, E_\Phi, r_\Phi)$ das zugehörige Niveaunetzwerk, Ψ ein Fluss für N_Φ.*

Für $v \in V_\Phi$ bezeichnet

$$pot(v) := \begin{cases} \min\left\{ \sum\limits_{e=(v,\cdot)\in E_\Phi} r_\Phi(e) - \Psi(e), \right. & \\ \qquad \left. \sum\limits_{e=(\cdot,v)\in E_\Phi} r_\Phi(e) - \Psi(e) \right\} & \text{für } v \in V_\Phi \setminus \{Q,S\} \\ \sum\limits_{e=(v,\cdot)\in E_\Phi} r_\Phi(e) - \Psi(e) & \text{für } v = Q \\ \sum\limits_{e=(\cdot,v)\in E_\Phi} r_\Phi(e) - \Psi(e) & \text{für } v = S \end{cases}$$

das Potenzial *des Knotens $v \in V_\Phi$.*

Wir können unsere Idee jetzt etwas präziser beschreiben. Wir berechnen zu jedem Knoten das Potenzial und wählen einen Knoten mit minimalem Potenzial. Von diesem Knoten aus versuchen wir, Fluss in der Größe des Potenzials zur Senke zu treiben. Das muss natürlich nicht funktionieren; was wir nicht zur Senke treiben können, treiben wir dann in umgekehrter Richtung zur Quelle. Insgesamt müssen wir den Fluss tatsächlich so „loswerden“ können; das liegt einfach daran, dass wir einen Knoten mit minimalem Potenzial gewählt haben. Danach können wir sicher sein, alle Wege über diesen Knoten „gesperrt“ zu haben, also mindestens eine saturierte Kante auf solchen Wegen zu finden; schließlich haben wir ja zusätzlichen Fluss im Wert des gesamten Potenzials dieses Knotens „bewegt“. Wir iterieren dieses Verfahren, bis wir einen Sperrfluss berechnet haben. Wir geben hier eine formale Beschreibung, die zwei Routinen `Forward()` und `Backward()` verwendet, in denen wie besprochen der Fluss vorwärts zur Senke bzw. rückwärts zur Quelle getrieben wird. Der Algorithmus arbeitet in einem Niveaunetzwerk N_Φ und berechnet dazu einen Sperrfluss Ψ.

Algorithmus 4.16 (**Forward-Backward-Propagation**)**.**
1. $\Psi := 0$
2. *Für alle $v \in V$ berechne $pot[v]$.*
3. *While $\{Q,S\} \subseteq V$*
4. *Wähle Knoten $v \in V$ mit minimalem Potenzial; $p_v := pot[v]$*
5. *If $pot[v] > 0$ Then Forward(v); Backward(v, p_v)*
6. *Entferne v aus V und Kanten $(\cdot, v)$, $(v, \cdot)$ aus E.*

Forward(v)
1. *Für alle $w \in V$ Überschuss$[w] := 0$*
2. *Überschuss$[v] := pot[v]$*
3. *$Qu := \emptyset$; $Qu.Enqueue(v)$*

4. *While* $Qu \neq \emptyset$
5. $v := Qu.Dequeue()$
6. *While* $\ddot{U}berschuss[v] > 0$
7. *Für alle* $e = (v, w) \in E$ $\{*$ *nur für* $\ddot{U}berschuss[v] > 0$ $*\}$
8. $\delta := \min\{r_\Phi(e) - \Psi(e), \ddot{U}berschuss[v]\}$
9. $\Psi(e) := \Psi(e) + \delta$
10. $pot[v] := pot[v] - \delta$; $pot[w] := pot[w] + \delta$
11. *If* $\ddot{U}berschuss[w] = 0$ *und* $\delta > 0$ *und* $w \neq S$
Then $Qu.Enqueue(w)$
12. $\ddot{U}berschuss[w] := \ddot{U}berschuss[w] + \delta$; $\ddot{U}berschuss[v] := \ddot{U}berschuss[v] - \delta$
13. *If* $\Psi(e) = r_\Phi(e)$ *Then Entferne* e *aus* E.

Backward(v, p_v)
1. *Für alle* $w \in V$ $\ddot{U}berschuss[w] := 0$
2. $\ddot{U}berschuss[v] := p_v$
3. $Qu := \emptyset$; $Qu.Enqueue(v)$
4. *While* $Qu \neq \emptyset$
5. $v := Qu.Dequeue()$
6. *While* $\ddot{U}berschuss[v] > 0$
7. *Für alle* $e = (w, v) \in E$ $\{*$ *nur für* $\ddot{U}berschuss[v] > 0$ $*\}$
8. $\delta := \min\{r_\Phi(e) - \Psi(e), \ddot{U}berschuss[v]\}$
9. $\Psi(e) := \Psi(e) + \delta$
10. $pot[v] := pot[v] - \delta$; $pot[w] := pot[w] + \delta$
11. *If* $\ddot{U}berschuss[w] = 0$ *und* $\delta > 0$ *und* $w \neq Q$
Then $Qu.Enqueue(w)$
12. $\ddot{U}berschuss[w] := \ddot{U}berschuss[w] + \delta$; $\ddot{U}berschuss[v] := \ddot{U}berschuss[v] - \delta$
13. *If* $\Psi(e) = r_\Phi(e)$ *Then Entferne* e *aus* E.

Lemma 4.17. *Sei* $(G = (V, E), c)$ *ein Netzwerk* $(n = |V|)$, Φ *ein Fluss auf* G, N_Φ *das Niveaunetzwerk dazu.*
Algorithmus 4.16 berechnet in Zeit $O(n^2)$ *einen Sperrfluss* Ψ *zu* N_Φ.

Beweis. Die Korrektheit des Algorithmus sieht man recht direkt: Weil wir in jeder Runde einen Knoten mit minimalem Potenzial wählen, ist das Potenzial p_v tatsächlich als Fluss zur Senke oder Quelle transportierbar. Offenbar sind für den gewählten Knoten v am Ende von Forward und Backward alle ausgehenden oder alle eingehenden Kanten saturiert. Darum kann v auch aus dem Netzwerk entfernt werden, es gibt keinen Weg, der ausschließlich

aus noch nicht saturierten Kanten besteht, der v benutzt. Wenn Quelle oder Senke so saturiert sind, ist offenbar ein Sperrfluss berechnet.
Bezüglich der Laufzeit beobachten wir zunächst, dass die initiale Berechnung des Potenzials aller Kanten in Zeit $O(n + e)$ möglich ist ($e = |E|$). Wir betrachten, was in Forward passiert, weil Backward offensichtlich analog funktioniert, brauchen wir das dann nicht mehr gesondert zu diskutieren. In Forward wird ein Knoten nur dann in die Queue aufgenommen, wenn sein aktueller Überschuss 0 ist. Weil nur für v selbst der Überschuss sinkt, kann kein Knoten in einem Forward-Durchlauf mehrfach in die Queue aufgenommen werden. Wir sehen uns also insgesamt in einem Forward-Durchlauf nur $\leq n$ Knoten an. Innerhalb des Forward-Durchlaufs gibt es noch eine Schleife über Kanten. Zentral ist, dass in der Schleife über die Kanten eine Kante entweder saturiert wird oder die Schleife endet: Wenn die betrachtete Kante nicht saturiert wird, kann das nur daran liegen, dass der Überschuss des Knotens „verbraucht“ ist. Dann endet die Schleife, die ja Überschuss$[v] > 0$ voraussetzt. Saturierte Kanten werden aus dem Netzwerk entfernt, darum sehen wir uns jede Kante höchstens zweimal an. Nach jedem Aufruf von Forward wird der aktuelle Knoten v entfernt, es kann also nur $\leq n$ Aufrufe von Forward geben. Jeder Forward-Aufruf kommt mit Zeit $O(n + \#\text{entfernte Kanten})$ aus, alle $\leq n$ Forward-Aufrufe haben dann insgesamt Laufzeit $O(n^2 + \#\text{entfernte Kanten})$. Weil nicht mehr als $2e$ Kanten entfernt werden können, haben wir insgesamt Laufzeit $O(n \cdot n + e) = O(n^2)$ wie behauptet. □

Mit der so verbesserten Sperrflussberechnung erhalten wir einen schnelleren Flussalgorithmus. Im Vergleich zum Algorithmus von Dinic (Algorithmus 4.13) unterscheidet sich nur die Sperrflussberechnung. Der hier als Algorithmus 4.18 folgende Algorithmus wird auch Algorithmus von Malhotra, Pramodh Kumar und Maheshwari genannt.

Algorithmus 4.18.
1. $\Phi := 0$
2. *Repeat*
3. *Berechne das Niveaunetzwerk* N_Φ.
4. *Berechne einen Sperrfluss* Ψ *mit Algorithmus 4.16.*
5. $\Phi := \Phi + \Psi$
6. *Until* $\Psi = 0$
7. *Ausgabe* Φ

Theorem 4.19. *Sei* $(G = (V, E), c)$ *ein Netzwerk. Der Algorithmus von Malhotra, Pramodh Kumar und Maheshwari (Algorithmus 4.18) berechnet zu* G *einen maximalen Fluss in Zeit* $O(n^3)$*, dabei sind* $|V| = n$ *und* $|E| = e$.

Beweis. Wir hatten im Beweis von Theorem 4.14 für den Algorithmus von Dinic (Algorithmus 4.13) gezeigt, dass es nur $O(n)$ Sperrflussberechnungen gibt. Das gilt natürlich weiterhin. Zusammen mit der auf $O(n^2)$ verbesserten Laufzeit für die Sperrflussberechnung (Lemma 4.17) ergibt sich also eine Gesamtlaufzeit von $O(n^3)$. □

4.4 Flussalgorithmen nach Goldberg und Tarjan

Allen drei bisher betrachteten Flussalgorithmen ist gemeinsam, dass sie rundenorientiert vorgehen: Sie starten mit dem leeren Fluss und verbessern den Fluss in jeder Runde. Der Algorithmus von Ford und Fulkerson (Algorithmus 4.4) macht das denkbar naiv, indem er einfach irgendeinen flussvergrößernden Weg im Restgraphen sucht. Der Algorithmus von Dinic (Algorithmus 4.13) führt den Begriff des Sperrflusses im Niveaunetzwerk ein und berechnet einigermaßen naiv einen solchen Sperrfluss. Der Algorithmus von Malhotra, Pramodh Kumar und Maheshwari (Algorithmus 4.18) schließlich berechnet die Sperrflüsse effizienter, indem er Fluss von einem Knoten aus nach vorne Richtung Senke und zurück Richtung Quelle treibt. Das rundenortientierte Vorgehen dieser Flussalgorithmen kann man durchaus vergleichen mit dem Vorgehen der Matchingalgorithmen (Algorithmus von Hopcroft und Karp (Algorithmus 3.12) für bipartite Graphen bzw. Algorithmus von Micali und Vazirani (Algorithmus 3.15) für allgemeine Graphen), die mit dem leeren Matching starten und in jeder Runde eine möglichst große (im Sinne von „eine nicht mehr erweiterbare“) Menge knotendisjunkter kürzester M-verbessernder Pfade berechnen. Es lag darum durchaus nahe, auch beim Flussproblem so vorzugehen. Es ist aber keineswegs so, dass a priori klar ist, dass ein rundenorientiertes Vorgehen der einzig mögliche Weg oder auch nur besonders effizient ist. Tatsächlich werden wir jetzt eine Klasse von Algorithmen diskutieren, die gänzlich anders vorgehen. Man darf sich im Prinzip davon inspirieren lassen, wie bei der Sperrflussberechnung im Algorithmus von Malhotra, Pramodh Kumar und Maheshwari (Algorithmus 4.16) der Fluss von einem Knoten aus als Überschuss auf andere Knoten verteilt wird, bis er entweder die Senke oder die Quelle erreicht hat.
Solange im Netzwerk ein Knoten positiven Überschuss hat, haben wir keinen gültigen Fluss im Sinne von Definition 4.1: Knoten mit positivem Überschuss verletzen die Kirchhoff-Regel. Wir führen jetzt einen Begriff ein, mit dem wir diesem Umstand Rechnung tragen.

Definition 4.20. *Sei $(G = (V, E), c)$ ein Netzwerk. Ein* Präfluss Φ *ist eine Abbildung $\Phi\colon E \to \mathbb{R}_0^+$, für die $\forall e \in E\colon \Phi(e) \leq c(e)$ und $\forall v \in V \setminus$*

$\{Q\}$: $e(v) \geq 0$ *gilt, dabei ist*

$$e(v) := \sum_{e=(\cdot,v)\in E} \Phi(e) - \sum_{e=(v,\cdot)\in E} \Phi(e)$$

der Überschuss *am Knoten* v.
Ein Knoten $v \in V \setminus \{Q, S\}$ *mit positivem Überschuss* ($e(v) > 0$) *heißt* aktiv.

Die Definition des Präflusses hier unterscheidet sich von der Definition eines Flusses in Definition 4.1 durch die Lockerung der Kirchhoff-Regel: Es darf mehr in einen Knoten hineinfließen als wieder herausfließt. In umgekehrter Richtung gestatten wir das nicht, ein Knoten darf also auch weiterhin nicht selbst Fluss produzieren – das gilt aber natürlich nicht für die Quelle Q.
In den drei betrachteten Flussalgorithmen haben wir immer vom Begriff des Restgraphen Gebrauch gemacht, entweder explizit oder implizit durch die Verwendung des Niveaunetzwerkes, das in seiner Definition ja wesentlich vom Restgraphen abhängt. Müssen wir auf diesen zentralen Begriff hier verzichten? Wir sehen direkt, dass das zum Glück nicht der Fall ist. Wenn in Definition 4.2 das das Wort „Fluss“ durch das Wort „Präfluss“ ersetzen, so erhalten wir weiterhin eine sinnvolle Definition, die für Präflüsse, die sogar Flüsse sind, natürlich mit der ursprünglichen Definition zusammenfällt. Wir können also beim Entwurf neuer Algorithmen, die auf der Idee des Präflusses basieren, weiterhin auf Restgraphen zurückgreifen.
Natürlich sind wir nicht wirklich an Präflüssen interessiert, wir wollen weiterhin das Flussproblem lösen, also einen maximalen Fluss berechnen. Daraus folgt, dass wir daran interessiert sind, aus einem Präfluss einen Fluss zu machen. Dazu müssen alle aktiven Knoten ihren Fluss abgeben können. Wir überzeugen uns jetzt zunächst, dass das auf jeden Fall zumindest möglich ist, wenn wir auch nicht unbedingt den Fluss bis zur Senke bringen können, was uns natürlich am liebsten ist.

Lemma 4.21. *Sei* $(G = (V, E), c)$ *ein Netzwerk,* Φ *ein Präfluss auf* G, $v \in V$ *aktiv. In* Rest_Φ *gibt es einen gerichteten Weg von* v *nach* Q.

Beweis. Wir definieren die Relation $\rightsquigarrow_\Phi$, dabei gelte $x \rightsquigarrow_\Phi y$ genau dann, wenn es einen gerichteten Weg in Rest_Φ von x nach y gibt. Wir benutzen diese Relation $\rightsquigarrow_\Phi$ und unseren Knoten v mit positivem Überschuss, um die Knotenmenge V zu partitionieren. Wir fassen die von v erreichbaren Knoten in der Menge V^* zusammen, definieren also $V^* := \{w \in V \mid v \rightsquigarrow_\Phi w\}$. Die übrigen Knoten sammeln wir in der Menge $\overline{V^*}$, es ist also $\overline{V^*} = V \setminus V^*$.
Wir wollen zeigen, dass $Q \in V^*$ gilt und führen dazu einen Beweis durch Widerspruch. Sei also $Q \in \overline{V^*}$. Wir betrachten die Summe der Überschüsse

der Knoten aus V^*, also $\sum_{w \in V^*} e(w)$. Natürlich gilt $\sum_{w \in V^*} e(w) \geq 0$, da ja sogar summandenweise $e(w) \geq 0$ für alle $w \in V \setminus \{Q\}$ gilt und wir $Q \in V^*$ gemäß Annahme ausschließen können. Wir benutzen jetzt den gleichen Trick wie beim Beweis des Max-Flow-Min-Cut-Theorems und spalten diese Summe *kantenweise* auf. Damit ganz klar ist, wie wir die Summe über die Überschüsse der Knoten nach den beteiligten Kanten aufspalten wollen, erinnern wir uns an die Definition des Überschusses und schreiben die Summe der Überschüsse etwas ausführlicher auf:

$$\sum_{w \in V^*} e(w) = \sum_{w \in V^*} \left(\sum_{e=(\cdot,w) \in E} \Phi(e) - \sum_{e=(w,\cdot) \in E} \Phi(e) \right). \tag{5}$$

Wir partitionieren die Kanten in die drei Mengen $E \cap \left(V^* \times \overline{V^*}\right)$, $E \cap \left(\overline{V^*} \times V^*\right)$ und $E \cap (V^* \times V^*)$. Wir erkennen auch hier, dass die Kanten, die ganz in V^* verlaufen, also Kanten $e \in E \cap (V^* \times V^*)$, mit ihrem Fluss $\Phi(e)$ einmal positiv und einmal negativ und somit in der Summe 0 beitragen. Kanten $e \in E \cap \left(\overline{V^*} \times V^*\right)$ tauchen mit ihrem Fluss $\Phi(e)$ ausschließlich positiv in der ersten Summe auf, analog tauchen Kanten $e \in E \cap \left(V^* \times \overline{V^*}\right)$ ausschließlich negativ in der zweiten Summe auf. Wir können also äquivalent für Gleichung (5) auch

$$\sum_{w \in V^*} e(w) = \sum_{e \in E \cap \left(\overline{V^*} \times V^*\right)} \Phi(e) - \sum_{e \in E \cap \left(V^* \times \overline{V^*}\right)} \Phi(e)$$

schreiben.

Betrachten wir eine Kante $e = (v^*, v') \in E \cap \left(V^* \times \overline{V^*}\right)$ aus der zweiten (der negativen) Summe. Gemäß Definition von V^* und $\overline{V^*}$ gibt es einen Weg von v zu v^* in Rest_Φ, aber keinen Weg von v zu v' in Rest_Φ. Es kann also insbesondere nicht die Kante (v^*, v') in Rest_Φ geben, sonst wäre v' ja über v^* und diese Kante doch von v aus erreichbar. Es gibt diese Kante aber in G, sonst hätten wir sie gar nicht betrachtet. Eine Kante e gehört zu G, aber nicht zu Rest_Φ, wenn ihre Restkapazität 0 ist, wir haben also $\Phi(e) = c(e)$ für alle diese Kanten aus der zweiten Summe.

Betrachten wir nun eine Kante $e = (v', v^*) \in E \cap \left(\overline{V^*} \cap V^*\right)$ aus der ersten (der positiven) Summe. Gemäß Definition von V^* und $\overline{V^*}$ gibt es einen Weg von v zu v^* in Rest_Φ, aber keinen Weg von v zu v' in Rest_Φ. Wir können jetzt daraus schließen, dass es die Kante (v^*, v') (also $\text{rev}(e)$) in Rest_Φ nicht gibt, weil es sonst wie oben doch einen Weg von v^* zu v' gäbe. Es gibt die Kante (v', v^*) in G, sonst hätten wir die Kante e gar nicht als Bestandteil der Summe gehabt. Können wir etwas über ihren Fluss aussagen? Wir sehen,

dass $\Phi(e) = 0$ gelten muss: Hätten wir positiven Fluss auf (v', v^*), so gäbe es in Rest_Φ die Rückwärtskante $\text{rev}(e)$, die es aber, wie wir uns gerade überzeugt haben, nicht gibt.
Mit diesen Überlegungen bekommen wir jetzt also äquivalent für Gleichung (5)

$$\sum_{w \in V^*} e(w) = \sum_{e \in E \cap \left(\overline{V^*} \times V^*\right)} 0 - \sum_{e \in E \cap \left(V^* \times \overline{V^*}\right)} c(e) = - \sum_{e \in E \cap \left(V^* \times \overline{V^*}\right)} c(e) \leq 0$$

und haben jetzt einerseits $\sum_{w \in V^*} e(w) \leq 0$ wie gerade gesehen und andererseits natürlich wie oben erläutert $\sum_{w \in V^*} e(w) \geq 0$. Wir haben damit insgesamt $\sum_{w \in V^*} e(w) = 0$ bewiesen. Weil $e(w) \geq 0$ für alle $w \in V^*$ gilt, muss $e(w) = 0$ für alle $w \in V^*$ gelten, sonst könnte die Summe ja nicht 0 sein. Das ist aber ein Widerspruch zur Voraussetzung $e(v) > 0$. Also war die Annahme bezüglich Q falsch und $v \rightsquigarrow_\Phi Q$ folgt wie behauptet. □

Dass wir überschüssigen Fluss zurück zur Quelle treiben und somit garantiert immer aus einem Präfluss einen Fluss machen können, ist ja ganz schön. Aber überschüssigen Fluss nur zur Quelle zu treiben, bringt uns nicht wirklich weiter. Wir wollen ja einen möglichst großen Fluss berechnen. Dazu müssen wir Fluss möglichst zur Senke treiben. Wenn man an große Graphen denkt uns sich daran erinnert, dass ein Flussalgorithmus ja eine eher lokale Sicht auf den Graphen hat, wird klar, dass man dem Algorithmus irgendwie den Weg zur Senke weisen muss. Wir werden zu diesem Zweck eine Markierung der Knoten einfügen und grundsätzlich Fluss „nach unten“ fließen lassen: wir geben der Quelle einen großen Wert, der Senke einen kleinen und erlauben das Verschieben von Fluss über eine Kante (v, w), wenn v etwas höher als w liegt. Wir formalisieren diese Idee im Begrif der gültigen Knotenmarkierung.

Definition 4.22. *Sei $(G = (V, E), c)$ ein Netzwerk mit $|V| = n$, Φ ein Präfluss auf G, $\text{Rest}_\Phi = (V, E_\Phi, r_\Phi)$ der Restgraph dazu.*
Eine Funktion $d\colon V \to \mathbb{N}_0$ heißt gültige Knotenmarkierung*, wenn $d(Q) = n$, $d(S) = 0$ und für alle Kanten $(v, w) \in E_\Phi$ stets $d(v) \leq d(w) + 1$ gilt.*
Eine Kante $e = (v, w) \in \text{Rest}_\Phi$ heißt wählbar *unter der gültigen Knotenmarkierung d, wenn $d(v) = d(w) + 1$ gilt.*

Es ist überhaupt nicht klar, dass wir für ein beliebiges Netzwerk und einen beliebigen Präfluss immer auch eine gültige Knotenmarkierung finden. Wir werden darum als Erstes zeigen, dass das in der Tat nicht immer der Fall sein muss.

Lemma 4.23. *Sei $(G = (V, E), c)$ ein Netzwerk, Φ ein Präfluss auf G, $Rest_\Phi = (V, E_\Phi, r_\Phi)$ der Restgraph dazu, d eine gültige Knotenmarkierung für Φ.*

1. *Für alle Knoten $v \neq w \in V$ gilt, dass jeder Weg in $Rest_\Phi$ von v nach w mindestens Länge $d(v) - d(w)$ hat.*

2. *Es gibt in $Rest_\Phi$ keinen Weg von Q nach S.*

Beweis. Der Beweis der zweiten Teilaussage folgt ganz unmittelbar aus der ersten Teilaussage. Jeder kürzeste Weg von v nach w in Rest_Φ hat immer Länge $< n$, weil kürzeste Wege kreisfrei sind. Das gilt für beliebige Knoten v und w, insbesondere also für Q und S. In einer gültigen Knotenmarkierung gilt gemäß Definition 4.22 $d(Q) = n$ und $d(S) = 0$. Die erste Teilaussage liefert, dass jeder Weg von Q nach S mindestens Länge $d(Q) - d(S) = n - 0 = n$ hat. Also kann es keinen solchen Weg geben.
Sei $(v, v_1), (v_1, v_2), \ldots, (v_{l-1}, w)$ ein Weg der Länge l von v nach w in Rest_Φ, sei also $(v_{i-1}, v_i) \in E_\Phi$ für alle $i \in \{1, 2, \ldots, l\}$, dabei sei $v_0 = v$ und $v_l = w$. Weil d eine gültige Knotenmarkierung ist, gilt $d(v_0) \leq d(v_1) + 1$, $d(v_1) \leq d(v_2) + 1$, $\ldots$, $d(v_{l-1}) \leq d(v_l) + 1$. Es gilt also $d(v) = d(v_0) \leq d(v_l) + l = d(w) + l$. Es gilt also für die Weglänge $l \geq d(v) - d(w)$ wie behauptet. □

Die Ideen für einen Algorithmus, der das Flussproblem löst, liegen jetzt auf der Hand. Wir starten mit einem trivialen Präfluss, zu dem wir eine triviale zulässige Knotenmarkierung haben. Dann betrachten wir einen aktiven Knoten und versuchen, den Fluss Richtung Senke zu verschieben. Falls das nicht geht, werden wir Fluss von einem aktiven Knoten Richtung Quelle verschieben. Bei der Wahl der „Richtung“, in die wir Fluss verschieben, soll uns die Knotenmarkierung d helfen, die wir auch im Laufe des Algorithmus verändern werden. Dabei achten wir darauf, immer eine gültige Knotenmarkierung zu behalten. Wenn wir am Ende aus unserem Präfluss einen Fluss gemacht haben, haben wir sogar einen maximalen Fluss gefunden: Die gültige Knotenmarkierung garantiert, dass es im Restgraphen keinen Weg von der Quelle zur Senke gibt, der Fluss muss also maximal sein.
Wir konkretisieren diese Ideen und geben ein Algorithmengerüst an, das wir etwas ungenau trotzdem Algorithmus nennen werden. Es handelt sich eigentlich um ein Algorithmengerüst, weil es viele Wahlmöglichkeiten gibt und wir offen lassen, wie die Wahlen konkret durchzuführen sind. Wir werden im Anschluss beweisen, dass wir für jede beliebige Wahl in annehmbarer Zeit einen maximalen Fluss erhalten. Ob gewisse Wahlen vielleicht zu beweisbar besseren Laufzeitschranken führen, lassen wir hier offen.

Algorithmus 4.24 (**Algorithmus von Goldberg und Tarjan**).
1. *Für alle* $v \in V$
   $d(v) := 0$; $e(v) := 0$
2. $d(Q) := n$
3. $\Phi := 0$
4. *Für alle* $v \in V$ *mit* $e = (Q, v) \in E$
   $\Phi(e) := c(e)$; $e(v) := c(e)$
5. *While* $\exists v \in V$ *mit* $e(v) > 0$
6. *Führe eine anwendbare Basisoperation (Push oder Relabel) aus.*
7. *Ausgabe* Φ

Push($e = (v, w)$)
{$*$ *anwendbar, wenn* v *aktiv und* e *wählbar ist* $*$}
1. $\delta := \min\{e(v), r_\Phi(e)\}$
2. *If* $e \in E$
   Then $\Phi(e) := \Phi(e) + \delta$
   Else $\Phi(e) := \Phi(e) - \delta$ {$*$ *Rückwärtskante* $*$}
   $e(v) := e(v) - \delta$; $e(w) := e(w) + \delta$

Relabel(v)
{$*$ *anwendbar, wenn* v *aktiv und keine Kante* $(v, \cdot) \in E_\Phi$ *wählbar ist* $*$}
1. $d(v) := \min\{d(w) + 1 \mid (v, w) \in E_\Phi\}$

Wir bereiten den Beweis der Korrektheit von Algorithmus 4.24 vor, indem wir die beiden Basisoperationen Push und Relabel charakterisieren. Wir setzen dabei eine gültige Knotenmarkierung d voraus.

Lemma 4.25. *Sei* $(G = (V, E), c)$ *ein Netzwerk,* Φ *ein Präfluss auf* G, $Rest_\Phi = (V, E_\Phi, r_\Phi)$ *der Restgraph dazu,* d *eine gültige Knotenmarkierung für* Φ.
Wenn Relabel(v) *auf den Knoten* v *anwendbar ist, führt die Ausführung von Relabel*(v) *um eine Erhöhung von* $d(v)$ *um mindestens 1.*

Beweis. Weil v aktiv ist, gibt es in Rest_Φ einen Weg von v zu Q (Lemma 4.21). Also gibt es mindestens eine Kante $(v, w) \in E_\Phi$, so dass die Menge, über die das Minimum gebildet wird, nicht leer ist. Dass Relabel(v) anwendbar ist, impliziert, dass für alle Knoten $w \in V$ mit $(v, w) \in E_\Phi$ notwendig $d(v) \neq d(w) + 1$ gilt, sonst wäre eine solche Kante (v, w) wählbar. Es gilt $d(v) \leq d(w) + 1$, da d eine gültige Knotenmarkierung ist, daraus folgt $d(v) < d(w) + 1$ für alle diese Kanten $(v, w) \in E_\Phi$. Also ist $\min\{d(w) + 1 \mid (v, w) \in E_\Phi\} > d(v)$ und $d(v)$ wächst. Weil wir nur ganzzahlige d-Werte definieren, wächst $d(v)$ mindestens um 1. □

Definition 4.26. *Sei $(G = (V, E), c)$ ein Netzwerk, Φ ein Präfluss auf G, $Rest_\Phi = (V, E_\Phi, r_\Phi)$ der Restgraph dazu, d eine gültige Knotenmarkierung für Φ, $v \in V$ aktiv, $e = (v, w)$ wählbar.*
Die Operation Push(v, w) heißt saturierend*, wenn in der Operation $\delta = r_\Phi(e)$ gilt, andernfalls heißt sie* nichtsaturierend*.*

Wir sehen direkt, dass eine saturierende Push-Operation die betroffene Kante saturiert. Bevor wir uns jetzt um die Korrektheit von Algorithmus 4.24 kümmern, stellen wir zunächst sicher, dass wir es immer mit einer gültigen Knotenmarkierung zu tun haben.

Lemma 4.27. *Im Ablauf des Algorithmus von Goldberg und Tarjan (Algorithmus 4.24) ist d stets eine gültige Knotenmarkierung.*

Beweis. Es genügt offenbar zu zeigen, dass wir initial eine gültige Knotenmarkierung haben und jede Anwendung einer anwendbaren Basisoperation die Gültigkeit der Knotenmarkierung erhält.
Initial werden $d(Q) = n$ und $d(S) = 0$ korrekt gesetzt. Weil $d(v) = 0$ für alle $v \in V \setminus \{Q\}$ gilt, folgt $d(v) \leq d(w) + 1$ direkt für alle Knoten $v \neq Q$. Wir müssen also nur $d(Q) \leq d(w) + 1$ für alle Kanten $(Q, w) \in E_\Phi$ überprüfen. Weil initial $\Phi(e) = c(e)$ für alle Kanten $e = (Q, w) \in E$ gilt, sind alle diese Kanten saturiert und tauchen in Rest_Φ nicht auf. Wir haben also initial eine gültige Knotenmarkierung.
Wir betrachten eine anwendbare Operation Push(v, w). Wenn es sich um eine saturierende Push-Operation handelt, wird die Kante (v, w) anschließend aus Rest_Φ entfernt. Das schafft aber keine neuen Bedingungen, so dass die gültige Knotenmarkierung gültig bleibt. Es kann eine neue Kante (w, v) in Rest_Φ eingefügt werden, das passiert, wenn $\Phi((v, w)) = 0$ galt. Wir setzen voraus, dass Push(v, w) anwendbar ist, also ist v aktiv und (v, w) wählbar, es gilt also $d(v) = d(w) + 1$. Also ist $d(w) = d(v) - 1 \leq d(v) + 1$ und die Knotenmarkierung bleibt gültig.
Wir betrachten eine anwendbare Operation Relabel(v). Für alle Kanten (v, w) entstehen keine Probleme, da $d(v)$ ausdrücklich so gewählt wird, dass $d(v) \leq d(w) + 1$ für alle solche Kanten gilt. Wir müssen noch Kanten $(w, v) \in E_\Phi$ betrachten. Vor Anwendung von Relabel(v) gilt $d(w) \leq d(v) + 1$. Durch Anwendung von Relabel(v) wird $d(v)$ größer (Lemma 4.25), also bleibt die gültige Knotenmarkierung gültig. □

Dass d zu jedem Zeitpunkt gültig ist, sichert uns immerhin die partielle Korrektheit von Algorithmus 4.24. Wenn der Algorithmus terminiert, dann haben wir einen maximalen Fluss gefunden. Dass der Algorithmus tatsächlich terminiert, beweisen wir im Anschluss.

Lemma 4.28. *Wenn der Algorithmus von Goldberg und Tarjan (Algorithmus 4.24) stoppt, dann ist Φ ein maximaler Fluss.*

Beweis. Der Algorithmus stoppt, wenn es keinen aktiven Knoten mehr gibt. Gemäß Definition des Präflusses (Definition 4.20) ist der Präfluss Φ dann sogar ein Fluss. Die aktuelle Knotenmarkierung zu dem Zeitpunkt ist gültig (Lemma 4.27), darum ist der Fluss Φ maximal, da es im Restgraphen Rest_Φ keinen Weg von der Quelle zur Senke gibt (Lemma 4.23). □

Die reine Gültigkeit von d während des Ablaufs von Algorithmus 4.24 ist nur ein erster Schritt. Wir zeigen außerdem, dass d monoton wächst und nach oben beschränkt ist. Das ist ein wichtiger Schritt auf dem Weg des Nachweises, dass Algorithmus 4.24 terminiert.

Lemma 4.29. *Im Ablauf des Algorithmus von Goldberg und Tarjan (Algorithmus 4.24) gilt für alle Knoten $v \in V$, dass $d(v)$ nicht sinkt und $d(v) \leq 2n-1$ ist.*

Beweis. Wir beobachten, dass nur die Operation Relabel d-Werte verändert. Wir hatten uns schon davon überzeugt, dass dabei d-Werte nicht sinken (Lemma 4.25). Folglich kann $d(v)$ für keinen Knoten v sinken. Für die Quelle Q und die Senke S ist nichts weiter zu zeigen, da bei gültigen Knotenmarkierungen $d(Q) = n$ und $d(S) = 0$ gilt, so dass die Einhaltung der Grenze $2n-1$ gegeben ist. Betrachten wir also einen Knoten $v \in V \setminus \{Q, S\}$. Nur wenn Relabel($v$) anwendbar ist, kann $d(v)$ wachsen. Dann ist aber $e(v) > 0$ und gemäß Lemma 4.23 gibt es dann einen Weg von v zur Quelle Q in Rest_Φ. Ein kreisfreier v-Q-Weg hat Länge $\leq n-1$, sei v' der erste Knoten nach v auf einem solchen kreisfreien Weg. Der Weg von v' zu Q hat also Länge $\leq n-2$ und wir erinnern uns (Lemma 4.23), dass $d(v') - d(Q) \leq n-2$ gilt. Weil $d(Q) = n$ ist, folgt $d(v') \leq 2n-2$. Relabel(v) setzt nun $d(v) := \min\{d(w)+1 \mid (v,w) \in E_\Phi\} \leq 2n-2+1 = 2n-1$ und die Behauptung folgt. □

Wir wollen unser nun erworbenes Strukturwissen nutzen, um die Anzahl der Basisoperationen, die insgesamt ausgeführt werden, nach oben zu beschränken. Für Relabel ist das jetzt besonders einfach.

Lemma 4.30. *Im Ablauf des Algorithmus von Goldberg und Tarjan (Algorithmus 4.24) werden weniger als $2n^2$ Relabel-Operationen ausgeführt.*

Beweis. Initial sind alle Knotenmarkierungen mindestens 0, jede Relabel-Operation vergrößert eine Knotenmarkierung um mindestens 1 (Lemma 4.25).

Weil die Knotenmarkierung für jeden Knoten durch $2n-1$ nach oben beschränkt ist (Lemma 4.29), können je Knoten nicht mehr als $2n-1$ Relabel-Operationen durchgeführt werden. Da es n Knoten gibt, werden nicht mehr als $2n^2 - n < 2n^2$ Relabel-Operationen ausgeführt. □

Lemma 4.31. *Im Ablauf des Algorithmus von Goldberg und Tarjan (Algorithmus 4.24) werden weniger als $2n \cdot e$ saturierende Push-Operationen ausgeführt.*

Beweis. Wir schauen uns zwei Ausführungen von Push$((v,w))$ an. Dazwischen muss es eine Ausführung von Push$((w,v))$ gegeben haben, denn nach einem saturierenden Push verschwindet die betroffene Kante zunächst aus dem Restgraphen. Push$((x,y))$ ist aber nur anwendbar, wenn (x,y) wählbar ist, wenn also $d(x) = d(y) + 1$ gilt. Vor der ersten Ausführung von Push$((v,w))$ gilt also $d(v) = d(w) + 1$, vor der Ausführung von Push$((w,v))$ gilt $d(w) = d(v) + 1$, also muss $d(w)$ mindestens um 2 gewachsen sein, vor der zweiten Ausführung von Push$((v,w))$ schließlich gilt wieder $d(v) = d(w) + 1$, es muss also $d(v)$ mindestens um 2 gewachsen sein. Wir haben $d(v) + d(w) \geq 1$ vor dem ersten saturierenden Push$((v,w))$-Aufruf während des gesamten Ablaufs und $d(v) + d(w) \leq 4n - 3$ vor dem letzten saturierenden Push$((v,w))$-Aufruf des gesamten Ablaufs. Weil wie gesehen die Label jedesmal um mindestens je 2 wachsen müssen, gibt es insgesamt nicht mehr als $(4n-4)/4 = n - 1 < n$ saturierende Push$((v,w))$-Aufrufe. Weil es im Restgraphen nicht mehr als $2e$ Kanten gibt, gibt es insgesamt weniger als $2e \cdot n$ saturierende Push-Operationen. □

Lemma 4.32. *Im Ablauf des Algorithmus von Goldberg und Tarjan (Algorithmus 4.24) werden weniger als $4n^2 \cdot e$ nichtsaturierende Push-Operationen ausgeführt.*

Beweis. Der Beweis ist weniger einfach und wird nicht so direkt geführt wie die Beweis für die Anzahlen von Relabel-Operationen und saturierenden Push-Operationen. Wir beobachten den Ablauf des Algorithmus und definieren uns drei Funktionen P_1, P_2 und P_3, die jeweils numerisch den „Zustand“ des Algorithmus beschreiben. Wir fassen diese Informationen in einer Art Potenzialfunktion $P := P_1 + P_2 - P_3$ zusammen.
P_1 ist definiert als das $(2n-2)$-Fache der Anzahl bisher durchgeführter saturierender Push-Operationen. P_2 ist definiert als die Summe aller Knotenmarkierungen $d(v)$ für $v \in V$. P_3 ist schließlich definiert als die Summe alle Knotenmarkierungen $d(v)$ für aktive Knoten $v \in V$. Außerdem ist wie gesagt $P := P_1 + P_2 - P_3$.
Initial ist $P = P_2 = n$, da nur die Quelle von 0 verschiedene Knotenmarkierung $d(Q) = n$ hat und die Quelle nicht aktiv ist. Wir betrachten die

Auswirkungen der Anwendung einer Basisoperation auf P. Wir werden dabei zeigen, dass P monoton wächst und beschränkt ist.
Wir betrachten zunächst die Anwendung von Push$((v,w))$. Es gilt $d(v) = d(w)+1$, außerdem ist $e(v) > 0$ und v damit aktiv. Betrachten wir zunächst eine nichtsaturierende Push-Operation. Dann bleiben die Werte von P_1 und P_2 unverändert. Weil am Ende $e(v) = 0$ gilt, ist v nicht mehr aktiv und P_3 fällt um $d(v)$. Der Knoten w könnte bei der Operation aktiv werden, dann wächst P_3 um $d(w)$. Weil $d(v) = d(w)+1$ gilt, fällt insgesamt P_3 mindestens um 1 und P wächst mindestens um 1. Falls Push$((v,w))$ saturierend ist, wächst P_1 um $2n-2$ und P_2 bleibt unverändert. Der Knoten w kann bei der Operation aktiv werden, dann wächst P_3 um höchstens $2n-2$ (Lemma 4.29). Insgesamt kann in diesem Fall also P nicht fallen.
Betrachten wir jetzt die Anwendung von Relabel(v). Es wächst $d(v)$ um mindestens 1, das hat gleiche Auswirkungen auf P_2 und P_3, weil v aktiv ist, so dass insgesamt P unverändert bleibt.
Wir können P insgesamt nach oben beschränken. Am Ende des Algorithmus ist $P \leq (2n\cdot e)\cdot(2n-2)+n\cdot(2n-1) = 4n^2e+2n(n-2e)-n \leq 4n^2e+2n(n-2e)$. Wir dürfen voraussetzen, dass der Graph zusammenhängend ist, also gilt $e \geq n-1$ und $n-2e \leq 0$ folgt, weil $n \geq 2$ gilt. Wir können also insgesamt $P \leq 4n^2e$ festhalten.
Wir haben jetzt gezeigt, dass initial $P = n$ gilt, jede Basisoperation P nicht sinken lässt und jede nichtsaturierende Push-Operation P um mindestens 1 vergrößert. Folglich kann es insgesamt nur weniger als $4n^2e$ nichtsaturierende Push-Operationen geben. □

Nun haben wir schon eine ganze Reihe von Beobachtungen zu Algorithmus 4.24 zusammengetragen. Es lohnt sich, an dieser Stelle auch formal festzuhalten, was wir jetzt insgesamt bewiesen haben.

Theorem 4.33. *Der Algorithmus von Goldberg und Tarjan (Algorithmus 4.24) berechnet mit $O(n^2e)$ Basisoperationen, die in beliebiger Reihenfolge ausgeführt werden können, einen maximalen Fluss.*

Beweis. Lemma 4.28 garantiert, dass der berechnete Fluss am Ende maximal ist. Die obere Schranke für die Anzahl der ausgeführten Basisoperationen hatten wir in Lemma 4.30 für Relabel-Operationen, in Lemma 4.31 für saturierende Push-Operationen und in Lemma 4.32 für nichtsaturierende Push-Operationen festgehalten. □

Theorem 4.33 macht keine Aussagen über die Laufzeit. Dafür muss man sich noch überlegen, wie man möglichst effizient wählbare Operationen findet und für welche Operation man sich entscheidet. Wir definieren zu diesem Zweck

eine neue lokale Operation, die wir uns aus Push und Relabel definieren. Diese lokale Operation verlangt die Verwaltung eines Zeigers in jedem Knoten v, der auf ein Element der Adjazenzliste von v zeigt. Wir nennen dieses so ausgezeichnete Element die *aktive Kante* von v.

Algorithmus 4.34 (**Push/Relabel-Variante des Goldberg-Tarjan-Algorithmus**).
1. *Für alle* $v \in V$
   $d(v) := 0$; $e(v) := 0$
2. $d(Q) := n$
3. $\Phi := 0$
4. *Für alle* $v \in V$ *mit* $e = (Q, v) \in E$
   $\Phi(e) := c(e)$; $e(v) := c(e)$
5. *While* $\exists v \in V$ *mit* $e(v) > 0$
6. *Führe Push/Relabel*(v) *aus.*
7. *Ausgabe* Φ

Push/Relabel(v)
{∗ *anwendbar, wenn* v *aktiv ist* ∗}
1. *If aktive Kante* $e = (v, w)$ *wählbar*
2. *Then Push*(e)
3. *Else*
4. *Mache die nächste Kante aktiv.*
5. *If Listenende erreicht Then*
6. *Relabel*(v)
7. *Mache erste Kante der Adjazenzliste von* v *zur aktiven Kante.*

Wir machen uns zunächst klar, dass die Anwendung von Push/Relabel uns keine Schwierigkeiten bereitet: Die Anwendung von Relabel(v) ist nur zulässig, wenn keine Kante $(v, w) \in E_\Phi$ wählbar ist. Das wird hier nicht ausdrücklich geprüft, wir müssen uns also vergewissern, dass es sicher der Fall ist.

Lemma 4.35. *Wird im Ablauf von Algorithmus 4.34 Relabel*(v) *aufgerufen, so ist Relabel*(v) *auch anwendbar.*

Beweis. Damit Relabel(v) anwendbar ist, muss v aktiv sein ($e(v) > 0$) und es darf keine Kante $(v, w) \in E_\Phi$ wählbar sein ($\forall (v, w) \in E_\Phi : d(v) < d(w)+1$). Bei Aufruf von Push/Relabel(v) ist v aktiv, Relabel(v) wird nur ausgeführt, wenn keine andere lokale Operation ausgeführt wurde, also ist v weiterhin aktiv. Wir führen Relabel(v) nur aus, wenn sich vorher die aktive Kante (v, w) als nicht wählbar erwiesen hat. Wenn wir zeigen, dass $e = (v, w)$ auch zeitlich zwischen dem letzten Push(e)-Aufruf und dem jetzt anstehenden Relabel(v)-Aufruf nicht wählbar wird, ist der Beweis geführt.

Relabel-Operationen mit anderen Knoten $v' \neq v$ sind unkritisch, weil durch sie höchstens Kanten $(v', \cdot)$ wählbar werden können. Push-Operationen verändern die d-Werte gar nicht, so dass sie vorhandene Kanten, die nicht wählbar sind, auch nicht wählbar machen können. Allerdings können Push-Operationen neue Kanten in den Restgraphen einfügen. Wir müssen uns davon überzeugen, dass dabei nicht eine wählbare Kante (v, w) entstehen kann. Falls Push(e') für eine Kante $e' = (x, y)$ eine neue Kante im E_Φ erzeugt, kann es sich nur um die Kante $(y, x) = \text{rev}(e)$ handeln. Weil Push(e') voraussetzt, dass e' wählbar ist, muss $d(x) = d(y) + 1$ gelten. Dann ist aber die Kante rev(e') nicht wählbar. □

Wir können jetzt verhältnismäßig leicht eine konkrete Laufzeitschranke für Algorithmus 4.34 nachweisen. Wir präsentieren zunächst das Ergebnis und seinen Beweis, bevor wir mögliche Verbesserungen diskutieren.

Theorem 4.36. *Die Push/Relabel-Variante des Goldberg-Tarjan-Algorithmus (Algorithmus 4.34) berechnet in Zeit $O(n^2 \cdot e) = O(n^4)$ einen maximalen Fluss.*

Beweis. Wir können die Initialisierung in Zeit $O(n + e)$ durchführen. Die aktiven Knoten können wir in einem Stack oder einer Queue verwalten, so dass der Zugriff jeweils in konstanter Zeit möglich ist. Es gelten weiterhin die in den Lemmata 4.30–4.32 bewiesenen oberen Schranken für die Anzahl von Basisoperationen. Wir können jede Push-Operation in konstanter Zeit durchführen, also kommen wir mit Zeit $O(ne)$ für alle saturierenden und Zeit $O(n^2e)$ für alle nichtsaturierenden Push-Operationen aus.
Die Ausführung von Relabel kann länger dauern, weil wir das Minimum berechnen müssen; die Zeit ist linear in der Anzahl dabei betrachteter Kanten. Es gibt aber insgesamt nur $O(n^2)$ Relabel-Operationen, in der jeder nicht mehr als e Kanten betrachtet werden, so dass wir insgesamt mit Zeit $O(n^2e)$ für die Relabel-Operationen auskommen. Für das zu beweisende Resultat reicht diese Aussage über Relabel aus. Man kann aber auch leicht eine noch wesentlich bessere obere Schranke zeigen: Wir wissen, dass jede Relabel-Operation die Knotenmarkierung vergrößert (Lemma 4.25) und die Knotenmarkierungen durch $2n-1$ nach oben beschränkt sind (Lemma 4.29). Darum kann jede Kante nur an $O(n)$ Relabel-Operationen beteiligt sein. Wir können darum die Gesamtzeit für alle $O(n^2)$ Relabel-Operationen auch durch $O(ne)$ nach oben abschätzen.
Was wir bis jetzt noch nicht berücksichtigt haben, ist die Zeit für das Durchlaufen der Adjazenzlisten in Push/Relabel, ohne dass eine Basisoperation ausgeführt wird. Allerdings muss vor dem zweiten Berühren einer Kante in dieser Schleife eine Relabel-Operation ausgeführt worden sein, das erfolgt ja

jeweils am Ende der Adjazenzliste. Weil es nur $O(n^2)$ Relabel-Aufrufe gibt, ist der Aufwand für das Durchlaufen der Adjazenzlisten auch durch $O(n^2)$ nach oben beschränkt. Wir haben also insgesamt Zeit $O(n^2e)$ wie behauptet. □

Theorem 4.36 ist als Resultat ernüchternd: Wir haben mit großem Aufwand einen neuen Algorithmus entwickelt und sind dabei einer ganz neuen algorithmischen Idee gefolgt. Erreicht haben wir dabei aber nur eine obere Schranke für die Laufzeit, wie wir sie schon für den Algorithmus von Dinic (Algorithmus 4.13) hatten, haben also nicht einmal die Laufzeitschranke $O(n^3)$ des Algorithmus von Malhotra, Pramodh Kumar und Maheshwari (Algorithmus 4.18) erreicht. Das ist aber durchaus möglich und erfordert kaum Mehraufwand.

Algorithmus 4.37 (**FIFO-Variante des Goldberg-Tarjan-Algorithmus**).
1. *Für alle* $v \in V$
   $d(v) := 0$; $e(v) := 0$
2. $d(Q) := n$
3. $\Phi := 0$; $Qu := \emptyset$
4. *Für alle* $v \in V$ *mit* $e = (Q, v) \in E$
   $\Phi(e) := c(e)$; $e(v) := c(e)$; *Qu.Enqueue*(v)
5. *While* $Q \neq \emptyset$
6. $v :=$ *Qu.Dequeue*()
7. *Repeat*
8. *Push/Relabel*(v), *füge dabei aktiv werdende Knoten in Qu ein.*
9. *Until* $e(v) = 0$ *oder Relabel*(v) *aufgerufen wurde.*
10. *If* $e(v) > 0$ *Then Qu.Enqueue*(v)
11. *Ausgabe* Φ

Theorem 4.38. *Die FIFO-Variante des Goldberg-Tarjan-Algorithmus (Algorithmus 4.37) berechnet in Zeit* $O(n^3)$ *einen maximalen Fluss.*

Beweis. Die FIFO-Variante unterscheidet sich von Algorithmus 4.34 nur in der Reihenfolge, in der die aktiven Knoten betrachtet werden. Wir wissen darum, dass abgesehen von den nichtsaturierenden Push-Operationen alle Operationen mit Zeit $O(n \cdot e) = O(n^3)$ auskommen.
Für die Analyse definieren wir sogenannte Durchläufe. Der erste Durchlauf umfasst alle Schritte, die für die initial aktiven Knoten durchgeführt werden. Der i-te Durchlauf umfasst dann alle Schritte, die für die aktiven Knoten durchgeführt werden, die im Durchlauf davor, also im $(i-1)$-ten Durchlauf, in die Queue eingefügt worden.
Wir stellen fest, dass es in einem Durchlauf für jeden Knoten v nur eine nichtsaturierende Push-Operation geben kann. Nach einem nichtsaturierenden Push ist nämlich $e(v) = 0$ und der Durchlauf endet für diesen Knoten.

Es genügt also nachzuweisen, dass die Anzahl der Durchläufe durch $O(n^2)$ nach oben beschränken ist.
Wir betrachten wieder eine Art Potenzialfunktion P, die wir diesmal als $P_1 - P_2$ definieren, dabei ist P_1 das Doppelte der Summe alle Knotenmarkierungen $d(v)$ für $v \in V$ und P_2 die maximale Knotenmarkierung eines aktiven Knotens. Initial haben wir offenbar $P = 2n$ und am Ende $P \leq 4n^2$, da die Knotenmarkierungen durch $2n$ nach oben beschränkt sind.
Wir wollen zeigen, dass P in einem Durchgang um mindestens 1 wächst; dann folgt schon die Behauptung. Dazu betrachten wir wieder die möglichen Basisoperationen. Die Ausführung von Relabel(v) vergrößert $d(v)$ um $h \geq 1$, so dass P_1 um $2h$ wächst. P_2 kann durch diese Änderung von $d(v)$ höchstens um h wachsen, so dass insgesamt P in diesem Fall um mindestens h wächst. Push-Operationen ändern P_1 gar nicht, da sie Knotenmarkierungen nicht ändern. Sie können aber P_2 ändern, weil sie Knoten aktivieren und deaktivieren können. Durch eine Deaktivierung kann P_2 nur kleiner werden, so dass P höchstens wachsen kann. Bei einer Aktivierung müssen wir eine Spur genauer hinsehen. Beim Aufruf von Push(e) mit $e = (v, w)$ ist v aktiv, so dass $P_2 \geq d(v)$ gilt. Außerdem ist (v, w) wählbar, so dass $d(v) = d(w) + 1$ gilt. Wenn w aktiv wird, so wächst P_2 nicht, da $d(w) < d(v)$ gilt.
Wir sehen, dass Push-Operationen P jedenfalls nicht kleiner machen und Relabel-Operationen P um mindestens 1 vergrößern. Wenn es in einem Durchlauf eine Relabel-Operation gibt, ist für ausreichendes Wachstum von P gesorgt. Andernfalls wurde die Repeat-Until-Schleife in diesem Durchlauf für jeden Knoten durch $e(v) = 0$ abgebrochen. Dann kann die Queue nur Knoten enthalten, die in diesem Durchlauf neu eingefügt wurden. Das sind Knoten w, für die es einen Push-Aufruf mit der Kante (v, w) gegeben hat. Dazu muss (v, w) wählbar gewesen sein, es galt also $d(v) = d(w) + 1$. Es sind also ausschließlich Knoten aktiv geworden, deren d-Wert um mindestens 1 kleiner ist als der aktuelle maximale d-Wert eines aktiven Knotens. Folglich muss in diesem Durchlauf P_2 um mindestens 1 sinken. Also ist auch in diesem Durchlauf P um mindestens 1 gewachsen und die Behauptung folgt. □

Man kann die Laufzeit noch weiter verbessern, indem man bei der Wahl der aktiven Knoten noch geschickter vorgeht. Wir wollen das hier aber nicht weiter vertiefen und beenden hiermit unsere Überlegungen zum Flussproblem.

5 Amortisierte Analyse

Als wir in der Einleitung diskutiert hatten, warum man an der Vorlesung „Effiziente Algorithmen“ interessiert sein sollte, hatten wir als einen Grund genannt, dass man in der Praxis mit Problemen konfrontiert werden kann, für die keine effizienten Algorithmen bekannt sind und man selber einen effizienten Algorithmus entwerfen muss. Vor allem bei Flussalgorithmen nach Goldberg und Tarjan (Algorithmus 4.24) haben wir gesehen, dass es sehr schwierig sein kann, einen Algorithmus überhaupt als effizient nachzuweisen. Wir mussten uns dort viel Mühe geben und in der Analyse sehr geschickt vorgehen. Weil das häufig so ist, lohnt es sich, etwas Abstand zu nehmen und sich nach Analyse*methoden* umzusehen, die in verschiedenen Situationen anwendbar sind und gute Ergebnisse liefern.

Wir betrachten exemplarisch eine Situation, in der wir eine Datenstruktur und die Operationen auf dieser Datenstruktur entworfen haben und jetzt die Worst-Case-Rechenzeit analysieren wollen. Nehmen wir an, dass wir in unserer Datenstruktur n Daten speichern. Dann werden wir in der Regel die Worst-Case-Rechenzeit für die Anwendung einer Operation als eine Funktion $t\colon \mathbb{N} \to \mathbb{N}$ angeben wollen: $t(n)$ gibt die Worst-Case-Rechenzeit der Operation an, wenn sich aktuell in der Datenstruktur n Daten befinden. Im Laufe eines Algorithmus werden wir die Operation sicher nicht nur einmal ausführen, nehmen wir an, dass es insgesamt m Ausführungen gibt. Offensichtlich bekommen wir eine obere Schranke für die gesamte Worst-Case-Rechenzeit der Folge von m Operationen durch $O(m \cdot n)$. Wir könnten uns aber auch mehr Mühe geben und die Worst-Case-Rechenzeit für die gesamte Folge von Operationen analysieren. Dann werden wir die Worst-Case-Rechenzeit als eine Funktion $t'\colon \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ angeben: $t'(n, m)$ gibt die Worst-Case-Rechenzeit einer Folge von m Operationen an, die auf einer Datenstruktur mit höchstens n Elementen operiert. Es ist natürlich denkbar, dass $t'(n, m)$ viel kleiner ist als $t(n)$. Das ist der Fall, wenn die Worst-Case-Rechenzeit einer Operation zwar groß ist, es aber in einer längeren Folge von Operationen nicht viele Operationen mit so großer Rechenzeit geben kann. In solchen Fällen sind wir zur Abschätzung der Gesamtlaufzeit natürlich viel mehr an t' als an t interessiert. Wir werden darum hier jetzt Methoden besprechen, mit denen man solche Analysen systematisch durchführen kann. Wir besprechen die Methoden, die wir kennenlernen wollen, am gleichen, sehr einfachen Beispiel. Wir nennen diese Art der Analyse *amortisierte Analyse*, weil manche Menschen die durchschnittliche Rechenzeit einer Operation (also $t'(n, m)/m$) die amortisierte Laufzeit einer Rechenoperation nennen.

5.1 Amortisierte Analyse mit der Potenzialmethode

Wir führen zunächst unser Beispiel ein. Wir wollen einen Binärzähler verwalten, der zu einer Zahl $z \in \mathbb{N}_0$ ihre Repräsentation $(z_{l-1}z_{l-2}\cdots z_0)_2$ in Standardbinärcodierung verwaltet, es soll also $z = \sum_{i=0}^{l-1} 2^i z_i$ gelten. Wir benutzen als Datenstruktur eine lineare Liste, die mit z_0 startet und im Listenelement zusätzlich zu z_i einen Zeiger auf das Listenelement mit z_{i+1} verwaltet, wenn das existiert (für jedes $i \geq 0$). Wir betrachten nur zwei verschiedene Operationen: Die Operation SetzeNull() initialisiert einen neuen Binärzähler für $z = 0$. Die Operation PlusEins() erhöht den aktuellen Wert des Zählers von z auf $z + 1$. Wir interessieren uns für die Worst-Case-Rechenzeit $t(l)$ einer PlusEins-Operation und die amortisierte Rechenzeit von m PlusEins-Operationen und einer SetzeNull-Operation.

SetzeNull() erzeugt eine neue Liste, die nur aus $z_0 = 0$ besteht und hat darum immer Rechenzeit $O(1)$, interessant ist nur die PlusEins-Operation. Wir wissen, dass $l = \lfloor \log_2 z \rfloor + 1$ gilt. Beim Schritt von $z = 2^k - 1$ auf 2^k müssen alle k Bits der Binärdarstellung von z von 1 auf 0 geändert und ein neues Listenelement für z_k angefügt werden. Die Worst-Case-Rechenzeit beträgt also $t(l) = \Theta(l)$. Wir bekommen also für eine Folge bestehend aus einer SetzeNull-Operation und m PlusEins-Operationen eine obere Schranke von $O(m \cdot l)$. Die Schranke ist aber sehr pessimistisch: den Schritt von $2^k - 1$ auf 2^k für irgendein $k \in \mathbb{N}$ kann es ja nicht so oft geben. Ist aber zum Beispiel $z_0 = 0$, so funktioniert PlusEins() in Zeit $O(1)$. Wir wollen darum versuchen, eine bessere Gesamtrechenzeit nachzuweisen und kommen auf dieses Beispiel zurück, wenn wir mit der Potenzialmethode eine erste allgemeine Methode zur amortisierten Analyse kennengelernt haben.

Wir beschreiben die *Potenzialmethode* für eine Datenstruktur D recht abstrakt. Kern der Methode ist die Definition einer *Potenzialfunktion* Φ, die von der Menge aller möglichen Instanzen unserer Datenstruktur $\mathcal{D}$ in die Menge der nichtnegativen reellen Zahlen abbildet. In gewisser Weise beschreibt also diese Potenzialfunktion $\Phi\colon \mathcal{D} \to \mathbb{R}_0^+$ den aktuellen Zustand unserer Datenstruktur D numerisch als $\Phi(E)$. Wir fordern dabei, dass für die initiale Datenstruktur D_0 stets $\Phi(D_0) = 0$ gilt.

Wir definieren die *amortisierte Rechenzeit* einer Operation O_i, die unsere Datenstruktur vom Zustand D_{i-1} in den Zustand D_i überführt, als $T_i + \Phi(D_i) - \Phi(D_{i-1})$, dabei ist T_i die tatsächliche Laufzeit der Operation O_i.

Wir betrachten eine Folge von Operationen $O_1, O_2, \ldots, O_m$ und dazu die Summe ihrer amortisierten Rechenzeiten, dann vereinfachen wir so weit wie

möglich.

$$\begin{aligned}
& \sum_{i=1}^{m} T_i + \Phi(D_i) - \Phi(D_{i-1}) \\
= & \left(\sum_{i=1}^{m} T_i\right) + \left(\sum_{i=1}^{m} \Phi(D_i)\right) - \left(\sum_{i=1}^{m} \Phi(D_{i-1})\right) \\
= & \left(\sum_{i=1}^{m} T_i\right) + \Phi(D_m) + \left(\sum_{i=1}^{m-1} \Phi(D_i)\right) - \left(\sum_{i=1}^{m-1} \Phi(D_i)\right) - \Phi(D_0) \\
= & \left(\sum_{i=1}^{m} T_i\right) + \Phi(D_m) \geq \sum_{i=1}^{m} T_i
\end{aligned}$$

Die Summe der amortisierten Rechenzeiten ist also eine obere Schranke für die Summe der tatsächlichen Rechenzeiten. Andererseits bekommen wir auch eine obere Schranke für die tatsächliche Rechenzeit, wenn wir eine obere Schranke $\tilde{T}$ für $T_i + \Phi(D_i) - \Phi(D_{i-1})$ bestimmen und $m \cdot \tilde{T}$ betrachten. Natürlich muss die Potenzialfunktion Φ so geschickt gewählt werden, dass wir für $T_i + \Phi(D_i) - \Phi(D_{i-1})$ eine möglichst kleine obere Schranke angeben können.

Die Idee der Potenzialmethode lässt sich auch anschaulich gut beschreiben: Wir weisen der Datenstruktur einen „Wert“ (ihr Potenzial) so zu, dass eine zeitaufwendige Operation nur durchgeführt werden kann, wenn das Potenzial der Datenstruktur groß ist. Wir hatten von der Potenzialmethode (allerdings in abgewandelter Form) Gebrauch gemacht, als wir bei der Analyse des Algorithmus von Goldberg und Tarjan beim Zählen der Anzahl nichtsaturierender Push-Operationen eine Potenzialfunktion eingeführt hatten, von der wir nachgewiesen haben, dass sie nicht sinkt, bei nichtsaturierenden Push-Operationen wächst und insgesamt nach oben beschränkt ist.

Wir wenden die Potenzialmethode hier konkret auf das Beispiel des Binärzählers an. Wir definieren als Potenzialfunktion die Anzahl der Einsbits im Binärzähler, also $\Phi(z) = \sum_{i=0}^{l-1} z_i$. Wir bestimmen $T_i + \Phi(D_i) - \Phi(D_{i-1})$, indem wir eine Fallunterscheidung nach z_0 durchführen. Ist $z_0 = 0$ (also z gerade), so muss für PlusEins() nur z_0 von 0 auf 1 geändert werden. Wir nehmen mal an, dass das in einem Rechenschritt geht, also $T_i = 1$ in diesem Fall gilt. Die Anzahl der Einsbits erhöht sich in diesem Fall offenbar genau um 1, wir haben also $\Phi(D_i) = \Phi(D_{i-1}) + 1$ und insgesamt ergibt sich $T_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + \Phi(D_{i-1}) + 1 - \Phi(D_{i-1}) = 2$ als amortisierte Rechenzeit einer solchen Operation. Im anderen Fall ist z ungerade und es gibt eine Zahl $k \in \mathbb{N}$, so dass die hinteren k Ziffern der Binärdarstellung von z Einsen

sind, es gilt also $z_0 = z_1 = \cdots = z_{k-1} = 1$ und $z_k \neq 1$. Dieser Fall schließt den Fall $k = l$ ein. Zur Durchführung von PlusEins() müssen diese hinteren k Bits von 1 auf 0 geändert werden, außerdem muss das folgende Bit entweder von 1 auf 0 geändert oder neu erzeugt werden. Wir nehmen an, dass sich dies in Zeit $k + 1$ bewältigen lässt. Das neue Potenzial ergibt sich in diesem Fall als $\Phi(D_i) = \Phi(D_{i-1}) - k + 1$, weil wir k Einsbits in Nullbits geändert und ein Einsbit hinzugefügt haben. Als amortisierte Rechenzeit dieser Operation erhalten wir also $T_i + \Phi(D_i) - \Phi(D_{i-1}) = k + 1 + \Phi(D_{i-1}) - k + 1 - \Phi(D_{i-1}) = 2$. Wir haben also in jedem Fall amortisierte Rechenzeit 2 und dürfen folgern, dass die Rechenzeit einer Folge bestehend aus einer SetzeNull-Operation und m PlusEins-Operationen durch $2m + 1$ nach oben beschränkt ist. Wir erinnern uns daran, dass wir oben eine obere Schranke von $O(m \cdot l)$ hatten und tatsächlich erheblich besser abschätzen konnten. Die Schranke $2m + 1$ ist offenbar sogar asymptotisch exakt, denn m Operationen erfordern auf jeden Fall Zeit $\Omega(m)$.

5.2 Amortisierte Analyse durch Kostenverteilung

Die Idee der amortisierten Analyse durch Kostenverteilung besteht darin, dass wir die tatsächliche Rechenzeit einer Operation nicht unbedingt dieser Operation anlasten. Wir erlauben uns, die Rechenzeit gedanklich auf von der Rechenoperation berührten Teile der Datenstruktur oder allgemein beliebigen Kostenträgern zu verteilen. Wenn wir nicht alle Kosten einer Operation so verteilen, dann verbleiben bei der Operation selbst die Restkosten und auch die Operation selbst ist dann ein Kostenträger. Wir können nun die Gesamtrechenzeit bestimmen, indem wir über die Restrechenzeiten der verschiedenen Kostenträger addieren.
Auch diese Analysemethode hatten wir schon einmal implizit verwendet: In Lemma 4.17 hatten wir eine obere Schranke für die Dauer eine Sperrflussberechnung mit Forward-Backward-Propagation bewiesen, indem wir die Kosten der While-Schleife in Forward und Backward jeweils nicht der Operation zugeordnet hatten, sondern die betroffenen Kanten als Kostenträger ausgewählt haben.
Wir beschreiben das Vorgehen wiederum konkret am Beispiel des Binärzählers. Wir verteilen die Kosten einer PlusEins-Operation an die von ihr berührten Stellen. Wir hatten uns im Abschnitt 5.1 darauf geeinigt, die Kosten einer PlusEins-Operation, die k Stellen berührt, mit k anzugeben. Es verbleiben also bei dieser Kostenverteilung keine Restkosten bei den PlusEins-Operationen. Kommen wir zur Gesamtkostenbestimmung. Wir beobachten, dass die Ziffer z_i ($i \in \mathbb{N}_0$) nur in jeder (2^{i+1})-ten PlusEins-Operation berührt wird. Bei m PlusEins-Operationen verursacht also die Stelle z_0 genau Kos-

ten m, die Stelle z_1 Kosten $\lceil m/2 \rceil$, die Stelle z_2 Kosten $\lceil m/4 \rceil$, die Stelle z_i Kosten $\lceil m/2^i \rceil$. Wir erhalten also Gesamtkosten

$$1 + \sum_{i=0}^{l-1} \left\lceil \frac{m}{2^i} \right\rceil < 1 + l + m \cdot \sum_{i=0}^{l-1} 2^{-i} < 2m + l + 1$$

und erkennen, dass $l = O(\log m)$ gilt. Wir haben also Gesamtkosten $O(m)$, was wiederum asymptotisch optimal ist.

5.3 Amortisierte Analyse mit der Kontenmethode

Amortisierte Analyse mit der Kontenmethode vereint in gewisser Weise die Idee der Potenzialmethode und der Kostenverteilung. Wie bei der Kostenverteilung beziehen wir die von einer Operation berührten Teile der Datenstruktur in unsere Überlegung ein. Wie bei der Potenzialmethode wollen wir dabei ausdrücken, dass unsere Datenstruktur durch manche Operationen in „wertvollere Zustände“ kommen kann, die dann teurere Operationen nach sich ziehen können. Wir modifizieren die Kostenberechnung einer einzelnen Operation auf zwei Arten. Zum einen erlauben wir einer Operation, „Konten“ ein „Guthaben“ zuzuweisen: Ein Konto ist dabei Teil einer Datenstruktur oder auch eine andere Entität, es entspricht in gewisser Weise den Kostenträgern bei der amortisierten Analyse durch Kostenverteilung. Ein Guthaben $g \in \mathbb{N}$ symbolisiert eine Anzahl von g Rechenschritten. Durch das Einzahlen eines solchen Guthabens g auf ein Konto vergrößert sich die amortisierte Rechenzeit der Operation von T_i auf $T_i + g$. Die andere Modifikation erlaubt es Operationen, von Knoten mit positivem Guthaben Beträge abzuheben: Wird von einem Konto mit Guthaben g ein Betrag $g' \in \mathbb{N}$ mit $g' \leq g$ abgehoben, so reduziert sich die amortisierte Rechenzeit der Operation von T_i um g' auf $T_i - g$. Weil wir niemals mehr von einem Konto abbuchen, als zuvor eingezahlt wurde, ist die Summe der amortisierten Rechenzeiten eine obere Schranke für die tatsächliche Rechenzeit.
Wir wenden die Methode wieder auf das Beispiel des Binärzählers an. Als Konten verwenden wir die Stellen des Binärzählers und erinnern uns daran, dass die reale Rechenzeit T_i jeweils der Anzahl der von der Operation Op_i berührten stellen entspricht. Für jede Stelle, die von 0 auf 1 gesetzt oder neu mit Wert 1 geschaffen wird, zahlen wir 1 auf das entsprechende Konto ein. Für jede Stelle, die von 1 auf 0 gesetzt wird, buchen wir 1 von dem entsprechenden Konto ab. Wir wissen, dass wir beim Binärzähler nur Stellen auf 0 setzen, die vorher auf 1 gesetzt wurden. Wir sind also sicher, nicht von leeren

Konten abzubuchen, und unser Vorgehen ist korrekt. Weil jede PlusEins-Operation nur eine einzige Stelle von 0 auf 1 ändert, erhalten wir genau für diese Stelle jetzt Kosten 2 statt bisher 1. Weil wir für jede andere Stelle 1 vom entsprechenden Konto abbuchen, ergeben sich nun Kosten 0 für diese Stellen. Damit haben wir für jede PlusEins-Operation amortisierte Kosten 2 und erhalten $2m+1$ als obere Schranke für die Gesamtrechenzeit einer Folge bestehend aus einer SetzeNull-Operation und m PlusEins-Operationen, was wieder asymptotisch exakt ist.

5.4 Union-Find-Datenstrukturen mit schnellen Unions und Pfadkompression

Wir wollen das Kapitel „Amortisierte Analyse“ noch durch die Besprechung eines nichttrivialen Beispiels anreichern und werden uns dabei gleichzeitig um ein „loses Ende“ kümmern, das noch aus der Vorlesung „Datenstrukturen, Algorithmen und Programmierung 2 (DAP 2)“ stammt. Wir erinnern uns darum hier an die Implementierung von Union-Find mit schnellen Unions und Pfadkompression. Man braucht sich keine Sorgen zu machen, wenn man davon bisher nichts gehört hat, wir wollen kurz darstellen, worum es geht.
Wir haben die n Zahlen $\{1, 2, \ldots, n\}$ und anfangs n Mengen, wobei die i-te Menge genau das Element i enthält. Wir können darum zunächst eine Menge mit einem in ihr enthaltenen Element identifizieren, das wir Repräsentanten der Menge nennen Auf diesen Mengen erklären wir zwei Operationen: Die Operation Find(i) liefert uns zum Element $i \in \{1, 2, \ldots, n\}$ den Repräsentanten j der Menge, in der sich i befindet. Wir verlangen nicht, das wir irgendeinen Einfluss darauf haben, durch welches Element eine Menge repräsentiert wird. Natürlich muss der Repräsentant selbst Element der durch ihn repräsentierten Menge sein, außerdem müssen alle Find-Aufrufe von Elementen dieser Menge den gleichen Repräsentanten liefern. Die Operation Union(i, j) setzt voraus, dass i und j jeweils Repräsentanten ihrer Mengen sind. Union(i, j) vereinigt die beiden Mengen, die durch i und j repräsentiert werden und liefert als Ergebnis den Repräsentanten der neue Menge zurück. Wir implementieren diese Datenstruktur wie folgt: Wir verwalten für jede Menge einen wurzelgerichteten Baum, dabei realisieren wir die wurzelgerichteten Bäume alle in einem gemeinsamen Array A der Größe n und setzen initial $A[i] = i$ für alle $i \in \{1, 2, \ldots, n\}$. In $A[i]$ speichern wir den Elter von Element i, man erkennt eine Wurzel daran, dass $A[i] = i$ gilt. Zusätzlich speichern wir zu jedem Repräsentanten einer Menge die Größe der Menge, die er repräsentiert. Wir können dazu ein zweites Array G der Größe n ver-

wenden und initial $G[i] = 1$ für alle $i \in \{1, 2, \ldots, n\}$ setzen. Wir halten den Inhalt von G nicht für alle Positionen aktuell, sondern nur für die aktuellen Wurzeln.
Einen Aufruf von Union(i, j) können wir nun in konstanter Zeit realisieren: Im ersten Schritt bestimmen wir die kleinere Menge, sei etwa $G[i] \leq G[j]$. Gilt $G[i] = G[j]$ benutzen wir das Wurzelelement als Entscheidungskriterium, falls also $G[i] = G[j]$ gilt, sei $i < j$. Nun vereinigen wir die Menge, indem wir die kleinere Menge (also die durch i repräsentierte Menge) „in die größere Menge einhängen“, konkret machen wir durch $A[i] := j$ das Element j zum Elter von i. Anschließend aktualisieren wir noch die Größe der neuen vergrößerten Menge durch $G[j] := G[j] + G[i]$. Dieses Vorgehen ist insgesamt korrekt, weil wir voraussetzen, dass i und j die Repräsentanten ihrer Mengen sind.
Einen Aufruf von Find(i) können wir realisieren, indem wir mit $k := A[i]$ starten und fortlaufend $k := A[k]$ setzen, bis $k = A[k]$ gilt. Die dafür benötigte Zeit ist offenbar proportional zur Länge des Weges, den wir im wurzelgerichteten Baum gehen und beschränkt durch die Tiefe des Baumes mit Repräsentanten Find(i). Um folgende Find-Aufrufe zu beschleunigen, durchlaufen wir, nachdem wir den Repräsentanten der Menge, die Wurzel $w \in \{1, 2, \ldots, n\}$, gefunden haben, die gleiche Folge $k, A[k], A[A[k]], \ldots$ noch einmal und ersetzen auf dem Weg $A[k]$ jeweils durch w. Wir machen also jeden Knoten auf dem Weg zu einem direkten Nachfolger der Wurzel. Man nennt dieses Vorgehen *Pfadkompression*. Offensichtlich ändert es asymptotisch nicht die Laufzeit von Find, außerdem bleibt offenbar die Worst-Case-Rechenzeit eines einzelnen Find-Befehls unverändert. Erst wenn man eine Folge von Find-Operationen betrachtet, kann man hoffen durch Pfadkompression zu sparen.
Man macht sich leicht klar, dass man durch eine Folge von Union-Befehlen Bäume logarithmischer Tiefe $\Theta(\log n)$ erzeugen kann, so dass eine Folge von $n-1$ Union und m Find-Befehlen *ohne Pfadkompression* Zeit $\Omega(n + m \log n)$ kosten kann. Man kann auch leicht zeigen, dass man mit dieser Zeit auskommt und wir ohne Pfadkompression auf Zeit $\Theta(n + m \log n)$ kommen. Weil m viel größer als n sein kann, wäre es schön, schnellere Find-Befehle zu haben, was die Einführung von Pfadkompression motiviert. Weil, wie wir schon argumentierten, die Worst-Case-Zeit eines Find-Befehles weiter $\Theta(\log n)$ ist, haben wir ein schönes Anwendungsfeld für amortisierte Analyse gefunden.
Wir wollen mit dem zu beweisenden Resultat beginnen und definieren zunächst zwei Funktionen, die wir brauchen, um das Ergebnis formulieren zu können. Wir werden eine „fast lineare“ Laufzeit nachweisen und definieren dazu eine „fast konstante“ Funktion.

Definition 5.1. *Die Funktion* $Z \colon \mathbb{N}_0 \to \mathbb{N}$ *ist rekursiv definiert durch* $Z(0) := 1$ *und* $Z(i) := 2^{Z(i-1)}$ *für* $i \in \mathbb{N}$.

Die Funktion $\log^*\colon \mathbb{N} \to \mathbb{N}_0$ *ist definiert durch* $\log^* n := \min\{k \mid Z(k) \geq n\}$.

Offensichtlich wächst $\log^* n$ unbeschränkt, es gilt $\lim_{n\to\infty} \log^* n \to \infty$, dieses Wachstum ist aber sehr langsam: für $n < 2^{65536}$ ist $\log^* n \leq 5$, für praktisch relevante n (praktisch relevant in dem Sinn, dass wir Eingaben der Größe n erwarten können), ist also $\log^* n$ durch 5 nach oben beschränkt. In diesem sehr praktisch orientierten und umgangssprachlichen Sinn ist $\log^* n$ „fast konstant".

Theorem 5.2. *Mit der Datenstruktur mit wurzelgerichteten Bäumen und Pfadkompression kann eine Folge von bis zu* $n-1$ *Union-Befehlen und* m *Find-Befehlen in Zeit* $O((n+m)\log^* n)$ *ausgeführt werden.*

Wir werden vor dem Beweis von Theorem 5.2 einige Vorarbeit leisten. Wir interessieren uns natürlich in erster Linie für die Weglängen, die bei Find-Befehlen bewältigt werden müssen, also die Länge von Wegen von Knoten zur zugehörigen Wurzel. Wir werden zunächst die maximale Weglänge mit der Anzahl der Elemente einer Menge in Beziehung setzen. Alle folgenden Aussagen beziehen sich ausschließlich auf solche wurzelgerichtete Bäume, wie sie durch Anwendung von Union- und Find-Befehlen erzeugt werden können.

Lemma 5.3. *Betrachte wurzelgerichtete Bäume, wie sie durch Anwendung von Union- und Find-Befehlen entstehen. Bezeichne die* Tiefe *h eines solchen Baumes die Länge eines längsten Weges von einem Blatt zur Wurzel. Ein Baum mit Tiefe h enthält mindestens 2^h Elemente.*

Beweis. Wir führen den Beweis mittels vollständiger Induktion über die Tiefe h. Für $h = 0$ besteht der Baum nur aus der Wurzel, enthält also $1 = 2^0 = 2^h$ Elemente und der Induktionsanfang ist gesichert. Für den Induktionsschluss betrachten wir einen Baum T mit Tiefe h und einer minimalen Anzahl von Elementen für diese Tiefe und nehmen an, dass die Aussage für alle Bäume mit Tiefe höchstens $h-1$ gilt. Weil $h > 0$ ist, muss T durch einen Union-Befehl entstanden sein, nehmen wir an, dass das Union(T_1, T_2) war und dass T_1 nicht mehr Elemente als T_2 enthält (andernfalls benennen wir T_1 und T_2 um). Weil wir bei einem Union-Befehl den Baum mit weniger Elementen in den Baum mit mehr Elementen hängen, muss die Tiefe von T_1 um mindestens 1 geringer sein als die Tiefe von T, die Tiefe von T_1 ist also durch $h-1$ nach oben beschränkt. Wenn die Tiefe von T_1 geringer ist als $h-1$, so ist die Tiefe von T gleich der Tiefe von T_2. Dann war also schon T_2 ein Baum der Tiefe h, außerdem hat T_2 offensichtlich weniger Knoten als T. Das ist aber ein Widerspruch zur Voraussetzung, dass T unter allen Bäumen mit Tiefe h minimale Anzahl von Knoten hat. Folglich hat T_1 tatsächlich genau Tiefe $h-1$. Gemäß Induktionsvoraussetzung enthält T_1 dann mindestens

2^{h-1} Knoten, T_2 ist nicht kleiner, enthält also auch mindestens 2^{h-1} Knoten, so dass T als disjunkte Vereinigung mindestens $2^{h-1} + 2^{h-1} = 2^h$ Knoten enthält. □

Die Anwendung eines Find-Befehls kann durch die Pfadkompression die Tiefe eines Baumes verkleinern, die Anzahl Elemente im Baum aber natürlich nicht ändern. Darum gilt Lemma 5.3 für die betrachtete Union-Find-Datenstruktur unabhängig davon, ob Pfadkompression eingesetzt wird oder nicht. Diese Überlegung bringt uns zur zentralen Idee der Analyse. Wir interessieren uns natürlich für die Union-Find-Datenstruktur mit Pfadkompression, wir werden aber zusätzlich gedanklich die Datenstruktur betrachten, die entsteht, wenn wir die gleiche Folge von Union- und Find-Befehlen anwenden, dabei aber auf die Anwendung von Pfadkompression verzichten.

Definition 5.4. *Wir betrachten eine Folge von ingesamt r Befehlen, davon bis zu $n-1$ Union- und genau m Find-Befehle. Dabei entstehen nacheinander die Datenstrukturen D_0^{mit}, D_1^{mit}, ..., D_r^{mit}. Es bezeichne D_0^{ohne}, D_1^{ohne}, ..., D_r^{ohne} die Folge von Datenstrukturen, die bei Anwendung der gleichen Befehlsfolge bei Verzicht auf Pfadkompression entsteht.*

Wir werden den Verlauf von D_i^{mit} und D_i^{ohne} aufmerksam verfolgen, dabei ist es hilfreich, sich einige allgemeine Eigenschaften von Union- und Find-Befehlen klar zu machen. Wir sehen unmittelbar ein, dass Find-Befehle in beiden Folgen weder die Anzahl von Elementen noch die Wurzel eines Baumes ändern. Außerdem hängt ein Union Bäume ineinander, wobei die Frage, welche Baum unter die Wurzel des anderen Baums gehängt wird, nur von der Anzahl der Elemente der Bäume abhängt. Wir halten die auf der Hand liegende Konsequenz als Lemma fest.

Lemma 5.5. *Betrachte die Folgen D_i^{mit} und D_i^{ohne} gemäß Definition 5.4. Für alle i gibt es zwischen D_i^{mit} und D_i^{ohne} eine bijektive Abbildung, die Bäume auf Bäume so abbildet, dass die enthaltenen Elemente und Wurzeln gleich sind.*

Beweis. Wir führen einen Induktionsbeweis über die Anzahl durchgeführter Operationen i. Für den Induktionsanfang ist $i = 0$ und die Aussage ist trivial. Für den Induktionsschluss nehmen wir an, dass die behauptete strukturelle Gleichheit nach i Operationen vorhanden ist und betrachten die nächste Operation. Wird eine Find-Operation durchgeführt, ändern sich weder die Elemente noch die Wurzel in den beiden betroffenen Bäumen, so dass die strukturelle Gleichheit natürlich erhalten bleibt. Bei einem Union werden zwei Mengen, bei denen gemäß Induktionsvoraussetzung die Wurzeln und Elemente gleich sind, vereinigt. Weil die Wurzeln gleich sind und die Anzahl

der Elemente gleich ist, wird in beiden Datenstrukturen die gleiche Entscheidung bezüglich der neuen Wurzel getroffen, so dass die Eigenschaft erhalten bleibt. □

Wir sehen also, dass in D_i^{mit} und D_i^{ohne} Bäume mit identischen Wurzeln und Elementen entstehen, allerdings kann sich das Aussehen der Bäume deutlich unterscheiden. Für eine Analyse brauchen wir mehr Informationen als nur über die Tiefe des tiefsten Blattes in einem Baum. Wir schärfen unseren Blick und führen dafür ein Maß für jeden Knoten ein.

Definition 5.6. *Betrachte die Folgen D_i^{mit} und D_i^{ohne} gemäß Definition 5.4. Für $v \in \{1, 2, \ldots, n\}$ ist der* Rang *Rang*$(v) := \max\{$*Weglänge von u nach v in D_i^{ohne} | Find*$(u) =$ *Find*(v) *und u Blatt*$\}$.
Für $k \in \mathbb{N}$ ist die Ranggruppe *R_k definiert durch $R_k := \{i \in \mathbb{N}_0 \mid Z(k-1) < i \leq Z(k)\}$, außerdem ist $R_0 := \{0, 1\}$.*

Die Rang eines Elements ist also *immer* in der Datenstruktur, die ohne Pfadkompression entsteht, definiert, auch wenn wir das Element in der Datenstruktur, die mit Pfadkompression entsteht, betrachten. Insbesondere ist es so, dass in der mit Pfadkompression entstehenden Datenstruktur die Ränge auf Wegen zur Wurzel strikt wachsen, eine Eigenschaft, die in der ohne Pfadkompression entstehenden Datenstruktur offenbar trivial erfüllt ist.

Lemma 5.7. *Betrachte die Folgen D_i^{mit} und D_i^{ohne} gemäß Definition 5.4. Für alle $v \in \{1, 2, \ldots, n\}$ und alle w, so dass v Elter von w in D_i^{mit} ist, gilt Rang$(v) >$ Rang(w).*

Beweis. Wir führen den Beweis mit vollständiger Induktion über die Anzahl durchgeführter Operationen i. Für den Induktionsanfang ist $i = 0$ und die Aussage trivial erfüllt, weil kein Element einen Elter hat. Für den Induktionsschluss nehmen wir an, dass die Aussage für D_i^{mit} gilt und betrachten D_{i+1}^{mit}. Wenn D_{i+1}^{mit} durch einen Find-Befehl entsteht, so bestehen alle Änderungen darin, dass einige Knoten ein neues Elter bekommen und zwar alle die Wurzel des entsprechenden Baumes. Weil für D_i^{mit} Ränge in Richtung Wurzel nur wachsen, kann diese Eigenschaft durch die von Find verursachten Änderungen nicht verletzt werden. Führt eine Union-Operation zu D_{i+1}^{mit}, so wird nur eine einzige Kante neu eingefügt, alle anderen Strukturen bleiben erhalten. Diese neue Kante verläuft zwischen einem Knoten v, der in D_i^{mit} noch Wurzel war, und einem Knoten w, der sowohl in D_i^{mit} als auch D_{i+1}^{mit} Wurzel ist. Wir wissen von Lemma 5.5, dass die Wurzeln in D_j^{mit} und D_j^{ohne} für alle j gleich sind. In der Datenstruktur mit Pfadkompression gilt die Rangwachstumseigenschaft trivial, also gilt sie auch für diese Kante und sie bleibt insgesamt erhalten. □

Wir können unsere bisherigen Erkenntnisse noch kombinieren, um Ränge und Anzahlen von Knoten in Beziehung zu setzen. Dann werden wir ausreichend vorbereitet sein, um das Ergebnis für die Gesamtlaufzeit beweisen zu können.

Lemma 5.8. $\forall k \in \mathbb{N}_0 \colon |\{v \in \{1, 2, \ldots, n\} \mid \mathit{Rang}(v) = k\}| \leq n/2^k$

Beweis. Wir betrachten D_i^{ohne} für irgendein i. Der Rang einer Wurzel v entspricht gerade der Tiefe des Baumes, also sind in jedem Baum mit Wurzel v mindestens $2^{\text{Rang}(v)}$ Elemente (Lemma 5.3). Ein Element mit Rang k kann weder Vorgänger noch Nachfolger mit Rang k geben. Wenn also m Elemente Rang k haben, so gibt es in den entsprechenden Teilbäumen mindestens $m \cdot 2^k$ Knoten. Folglich kann es nicht mehr als $n/2^k$ Elemente mit Rang k geben. □

Beweis von Theorem 5.2. Weil jeder Union-Befehl in Zeit $O(1)$ durchgeführt werden kann und es nicht mehr als $n - 1$ Union-Befehle geben kann, müssen wir nur die Zeit für die m Find-Befehle nach oben abschätzen. Wir verwenden dazu amortisierte Analyse mit Kostenverteilung.
Als Kostenträger definieren wir zum einen die Find-Befehle selbst, zum anderen jedes Element $i \in \{1, 2, \ldots, n\}$. Für die Kostenverteilung betrachten wir einen Find-Befehl, der k Kanten auf seinem Weg zur Wurzel betrachtet. Wir vereinbaren, dass dieser Find-Befehl (tatsächliche) Rechenzeit $k + 1$ hat. Die wirkliche Rechenzeit ist nicht mehr als einen konstanten Faktor größer: um Verwirrung zu vermeiden, halten wir noch einmal ausdrücklich fest, von welchen Zeiten hier die Rede ist. Die „wirkliche Rechenzeit“ ist die Ausführungszeit auf einem realen Rechner oder auch die Anzahl Takte in einem formalen Rechenmodell. Wir abstahieren davon uns zählen „tatsächliche Rechenzeit“ $k + 1$, die „tatsächliche Rechenzeit“ hatten wir in den vorderen Abschnitten mit T_i bezeichnet und von der amortisierten Rechenzeit unterschieden. Wir bestimmen $k + 1$ und nicht etwa k, weil sonst Find-Befehle, die direkt die Wurzel betreffen, unrealistisch mit Kosten 0 berücksichtigt würden.
Der betrachtete Find-Befehle Find(v) durchlaufe auf seinem Weg zur Wurzel die Knoten $v_k \to v_{k-1} \to \cdots \to v_1 \to v_0$, es sei also v_0 die Wurzel des Baumes und $v_k = v$. Von den zu verteilenden Gesamtkosten von $k + 1$ verteilen wir Kosten 1 pauschal auf den Find-Befehl, außerdem noch einmal Kosten 1 auf den Find-Befehl für die Kante $v_1 \to v_0$. Für alle übrigen Kanten $v_i \to v_{i-1}$ entscheiden wir die Frage der Kostenverteilung danach, ob die Ränge der Knoten v_i und v_{i-1} in der gleichen Ranggruppe liegen. Wenn die Ranggruppen der Ränge gleich sind, so verteilen wir die Kosten 1 für diese Kante an das Element v_i, andernfalls trägt diese Kosten 1 auch noch der Find-Befehl. Wir summieren jetzt die bei m Find-Befehlen verteilten Kosten getrennt nach Kosten, die wir auf Find-Befehle verteilt haben und Kosten, die wir auf

Elemente verteilt haben. Offensichtlich trägt jeder Find-Befehl, der direkt eine Wurzel betraf, Kosten 1, alle anderen Find-Befehle tragen Kosten $2 + l$, wenn l die Anzahl der Ranggruppenwechsel auf dem Weg vom Knoten zur Wurzel angibt. Es kann offenbar nicht mehr als $1+\log^* n$ Ranggruppenwechsel geben, weil nicht mehr als n Knoten zur Verfügung stehen. Also trägt jeder Find-Befehl höchstens Kosten $3 + \log^* n$ und wir haben für m Find-Befehle Gesamtkosten $O(m \log^* n)$. Dabei sind aber die auf die Elemente verteilten Kosten noch nicht berücksichtigt.
Wir beobachten, dass bei allen Find-Befehlen, bei denen für die Kante $v_i \to v_{i-1}$ das Element v_i Kosten 1 des Find-Befehls tragen musste, die Wurzel v_0, die Elter von v_i wird, nicht bereits vorher Elter von v_i war. Das liegt einfach daran, dass wir bei der Kostenverteilung die Kante $v_1 \to v_0$ als Sonderfall ausgenommen hatten und die Kosten für diese Kante immer an den Find-Befehl verteilt haben.
Gemäß der Rangwachstumseigenschaft (Lemma 5.7) bekommt v_i dann jedes Mal ein Elter mit echt größerem Rang. Wenn der Rang von v_i selbst zur Ranggruppe g gehört, wenn also $Z(g-1) < \text{Rang}(v_i) \leq Z(g)$ gilt, dann liegen die Ränge der Eltern von v_i auch jeweils in dieser Ranggruppe, wenn v_i Kosten 1 tragen muss. Andernfalls wären die Kosten für diese Kante an den Find-Befehl verteilt worden. Weil der Rang mit jedem Elternwechsel wächst, kann es nicht mehr als $Z(g) - Z(g-1) \leq Z(g)$ solche Find-Befehle mit Kosten 1 für v_i geben, spätestens dann gibt es beim Übergang von v_i zum Elter einen Ranggruppenwechsel, so dass ab dann Find-Befehle die Kosten für diese Kante tragen. Wir wissen (Lemma 5.8), dass nicht mehr als $n/2^r$ Elemente mit Rang r gibt. Also ist die Anzahl der Elemente in der Ranggruppe von v_i durch

$$\sum_{r=Z(g-1)+1}^{Z(g)} \frac{n}{2^r} < n \cdot \sum_{r>Z(g-1)} \frac{1}{2^r} = \frac{n}{2^{Z(g-1)+1}} \cdot \sum_{r\geq 0} \frac{1}{2^r} = \frac{n}{2^{Z(g-1)}} = \frac{n}{Z(g)}$$

nach oben beschränkt. Also haben wir für jedes Element mit Rang in Ranggruppe g höchstens Kosten $n/Z(g)$ für die Find-Befehle zu addieren. Wir addieren diese Kosten getrennt für jede Ranggruppe, in jeder Ranggruppe g' gibt es nicht mehr als Gesamtkosten $Z(g') \cdot (n/Z(g')) = n$. Weil es wie oben schon festgestellt nicht mehr als $1 + \log^* n$ Ranggruppen gibt, sind also die Gesamtkosten, die auf die Elemente verteilt wurden, durch $n \cdot (1 + \log^* n) = O(n \log^* n)$ nach oben beschränkt.
Wir addieren die Kosten $O(n)$ für die Union-Befehle, Kosten $O(m \log^* n)$, die bei der amortisierten Analyse auf die Find-Befehle verteilt wurden, und Kosten $O(n \log^* n)$, die bei der amortisierten Analyse auf die Elemente verteilt wurden. Wir erhalten wie behauptet Gesamtkosten $O((n + m) \log^* n)$. □

6 String Matching

Suchen gehört zu den grundlegendsten algorithmischen Problemen in der Informatik. Es gibt viele Varianten des Grundproblems, die durch die zu durchsuchenden Daten einerseits und die Suchmuster andererseits charakterisiert sind. So macht es natürlich einen Unterschied, ob man nach Zeichenketten oder regulären Ausdrücken sucht, ob man direkt suchen kann oder die zu durchsuchenden Datenmengen so groß sind, dass eine Vorabindizierung erforderlich sind. Wir werden uns hier nur mit dem allereinfachsten Suchproblem befassen, der Suche nach einem festen Suchmuster P in einem festen Text T. Schon diese sehr simple Variante, die man Textmustersuche oder auch String Matching nennt, stellt uns vor interessante und nichttriviale Probleme. Wir beginnen mit einer formalen Problemdefinition.

Definition 6.1. *Sei Σ ein endliches Alphabet, $x = x_1x_2\dots x_k \in \Sigma^k$ und $y = y_1y_2\dots y_l \in \Sigma^l$ Zeichenketten endlicher Länge über Σ. Für $1 \le i \le j \le k$ schreiben wir $x[i \mathinner{..} j]$ für die Zeichenkette $x_ix_{i+1}\cdots x_j$ und $x[i]$ für x_i.*
Wir schreiben $x \sqsubset y$, wenn $y[1 \mathinner{..} k] = x$ gilt, und nennen x ein Präfix *von y. Wir schreiben $x \sqsupset y$, wenn $y[(l-k+1) \mathinner{..} l] = x$ gilt, und nennen x ein* Suffix *von y.*
Beim String-Matching-Problem ist zu einem endlichen Alphabet Σ, einem Text $T \in \Sigma^*$ der Länge $|T| = n$ über Σ und einem Muster $P \in \Sigma^*$ der Länge $|P| = m$ über Σ die Menge $S = \{s_1, s_2, \dots, s_l\}$ aller Indizes zu berechnen, so dass $T[s_i \mathinner{..} (s_i + m - 1)] = P$ für alle $i \in \{1, 2, \dots, l\}$ gilt und für kein $s \notin S$ ebenfalls $T[s \mathinner{..} (s + m - 1)] = P$ gilt.

Es ist denkbar, nicht alle Vorkommen des Suchmusters P zu suchen, sondern nur nach dem ersten Vorkommen zu fragen. Obwohl diese Problemstellungen offensichtlich ähnlich sind, gibt es schon wichtige Unterschiede. Wenn man andere Texte über String Matching liest, sollte man darum als Erstes herausfinden, welche Variante denn eigentlich gelöst wird.
Das Problem lässt sich offenbar leicht lösen, indem man gedanklich das Muster „ganz links anlegt“, also das i-te Textzeichen mit dem i-ten Musterzeichen vergleicht für $i \in \{1, 2, \dots, m\}$. Findet man einen Unterschied, kann man den Vergleich beenden, „rückt das Muster eine Stelle nach rechts“, vergleicht dann also das $(i+1)$-te Textzeichen mit dem i-ten Suchmuster und immer so fort. Findet man für keines der m Zeichen einen Unterschied, so hat man das Suchmuster gefunden und gibt den entsprechenden Index aus. Wir beschreiben das Vorgehen noch einmal etwas formaler im folgenden Algorithmus.

Algorithmus 6.2 (**Naiver Algorithmus**).
1. *For* $i \in \{0, 1, 2, \ldots, n-m\}$
2. $\quad j := 1$*; While* $T[i+j] = P[j]$ *und* $j \leq m$
3. $\quad\quad j = j + 1$*; If* $j > m$ *Then Ausgabe* i

Theorem 6.3. *Der naive String-Matching-Algorithmus (Algorithmus 6.2) löst das String-Matching-Problem in Zeit* $\Theta((n-m+1) \cdot m)$.

Beweis. Für den Beweis der oberen Schranke beobachten wir, dass die äußere Schleife über $n-m+1$ verschiedene Werte läuft und die innere Schleife über $\leq m$ Werte läuft. Weil alle Befehle in Zeit $\Theta(1)$ ausgeführt werden können, bekommen wir $O((n-m+1) \cdot m)$ als obere Schranke.
Für die untere Schranke müssen wir lediglich die Anzahl der Schleifendurchläufe der inneren Schleife nach unten durch $\Omega(m)$ beschränken, dann folgt schon eine untere Schranke der Größe $\Omega((n-m+1) \cdot m)$. Wir betrachten dazu die Worst-Case-Eingabe $T = 0^n$, $P = 0^m$ und beobachten, dass die innere Schleife jedes Mal bis m läuft. □

Der naive Algorithmus ist offenbar nicht besonders schnell, für lange Suchmuster (im Extremfall $m = \Theta(n)$, zum Beispiel $m = n/4$) haben wir quadratische Laufzeit. Das gilt allerdings nur, wenn jeweils $\Theta(m)$ Vergleiche positiv ausgehen, beim ersten negativen Vergleich bricht die While-Schleife ja ab. Man kann sich darum überlegen, dass für Texte, die uniform zufällig gewählt werden, der naive Algorithmus gar nicht so schlecht ist. Aber in der Praxis sind Texte nicht rein zufällig und eine Funktion, die jeder noch so primitive Texteditor anbietet, sollte schneller erledigt werden als in quadratischer Zeit.

6.1 String Matching mit endlichen Automaten

Wenn man „von den Rändern“ absieht, vergleicht der naive Algorithmus (Algorithmus 6.2) jedes Zeichen des Textes mit jedem Zeichen des Suchmusters, was etwa $n \cdot m$ Vergleiche bedeutet. Das sieht nicht besonders effizient aus. Man könnte vielleicht auf die waghalsige Idee kommen, das Suchmuster bei einem Unterschied gleich um m Stellen weiterzuschieben, das geht aber nicht so einfach. Das Beispiel $T =$ OOOOHALLO und $P =$ OOOH zeigt, dass man dann Vorkommen verpassen kann. Andererseits arbeitet der Algorithmus offenbar fast völlig gedächtnislos: Das Ansehen eines Buchstaben $T[i]$ wird anschließend sofort wieder vergessen und nicht für zukünftige Vergleiche irgendwie ausgenutzt. Für schnellere Algorithmen wäre es sicher ratsam, hier zu sparen. Im Extremfall kann man sich wünschen, dass jedes Zeichen $T[i]$ nur einmal gelesen wird.

Gelegentlich ist es hilfreich, nicht alles Wissen aus dem Grundstudium zu vergessen. Hier ist es nützlich, sich an endliche Automaten zu erinnern, wie man sie in der Vorlesung „Grundbegriffe der Theoretischen Informatik (GTI)“ oder der Vorlesung „Theoretische Informatik für Studierender der Angewandten Informatik (TIfAI)“ kennengelernt hat. Ein deterministischer endlicher Automat (DFA) löst gemäß Definition jede Aufgabe in Linearzeit, weil er jedes Zeichen der Eingabe nur einmal ansehen darf und das auch noch in fester Reihenfolge von links nach rechts. Wir brauchen also nur einen DFA zu entwerfen, der das Problem des String Matching löst, dann können wir durch Simulation des DFA in einem Algorithmus direkt zu einem Linearzeitalgorithmus kommen. Wer sich vielleicht etwas zu gut an das Grundstudium erinnert, könnte etwas pessimistisch einwerfen, dass bekanntlich nicht alle Probleme regulär sind. Wir wollen uns von diesem Einwand in unserem ersten Enthusiasmus nicht bremsen lassen und verfolgen die Idee zunächst weiter.
Wir halten jetzt zunächst das Suchmuster P fest und betrachten nur noch den Text T als Eingabe des Problems. Man sollte der Fairness halber festhalten, dass das eine erhebliche Modifikation ist: Wenn P fest ist, so ist auch $m = |P|$ fest, also konstant, und auch der naive Algorithmus hat Laufzeit $\Theta(n)$. Unser Perspektivenwechsel macht aber trotzdem Sinn: Wir überlegen uns zunächst, wie wir für einen festes Suchmuster P einen DFA A_P entwickeln, der dann das Problem für beliebige Texte T in Linearzeit löst. Anschließend können wir einen Algorithmus beschreiben, der im ersten Schritt den DFA A_P berechnet und im zweiten Schritt den A_P auf den Eingabetext anwendet. Wir diskutieren das später, nachdem wir uns davon überzeugt haben, dass es uns überhaupt gelingt, einen solchen DFA A_P passend zu konstruieren. Jetzt halten wir zunächst fest, wie ein Algorithmus bei gegebenem DFA A_P in einem Text T suchen kann. Dabei gehen wir davon aus, dass der DFA $A_P = (Q, \Sigma, q_0, \delta, F)$ so definiert ist, dass $q \in F$ genau dann gilt, wenn gerade das komplette Suchmuster P gelesen worden ist.

Algorithmus 6.4 (**DFA-String-Matching**).
1. $q := q_0$
2. *For* $i \in \{1, 2, \ldots, n\}$
3. $\quad q := \delta(q, T[i])$
4. $\quad$ *If* $q \in F$ *Then Ausgabe* $i - m + 1$

Es ist offensichtlich, dass Algorithmus 6.4 Laufzeit $\Theta(n)$ hat und genau dann korrekt ist, wenn der DFA A_P korrekt ist. Wir machen uns also jetzt daran, einen solchen DFA zu konstruieren. Dabei wollen wir uns im Zustand merken, wie viele Zeichen des Suchmusters P aktuell mit dem Ende des gelesenen Textes übereinstimmen. Wir drücken das noch etwas genauer aus und benutzen die Begriffe Präfix und Suffix, die uns sehr präzise beschreiben lassen,

was wir meinen. Wir haben die Eingabe T bis zum i-ten Zeichen gelesen, wir betrachten als das Präfix $T[1 \,..\, i]$ von T. Nun wollen wir entscheiden, wie viele Zeichen von P mit dem Suffix dieses Präfixes $T[1 \,..\, i]$ übereinstimmen. Dabei lesen wir die Zeichen in P natürlich auch von links nach rechts, sind also an einem möglichst langen Präfix von P interessiert. Wir suchen also ein möglichst langes Präfix von P, das Suffix des aktuellen Präfixes $T[1 \,..\, l]$ von T ist. Wir können diese Beschreibung auch formal in einer Funktion fassen.

Definition 6.5. *Für ein festes Suchmuster $P \in \Sigma^m$ ist die* Suffixfunktion $\sigma_P \colon \Sigma^* \to \{0, 1, \ldots, m\}$ *durch*

$$\sigma_P(x) = \max\{l \mid P[1 \,..\, l] \sqsupset x\}$$

definiert.

Weil das leere Wort ε Suffix von jedem $x \in \Sigma^*$ ist, ist die Suffixfunktion wohldefiniert. Unser Ziel ist es jetzt, einen DFA A_P zu definieren, so dass für alle $i \in \{1, 2, \ldots, n\}$ stets $\delta(q_0, T[1 \,..\, i]) = \sigma_P(T[1 \,..\, i])$ gilt, der sich also in den Zuständen nur die Länge des Präfixes von P merkt, für den mit dem Suffix des aktuell gelesenen Präfixes des Textes Übereinstimmung herrscht. Dies ist erstaunlich einfach zu erreichen, wir halten das in der folgenden Definition fest.

Definition 6.6. *Zu einem Suchmuster $P \in \Sigma^m$ ist der DFA $A_P = (Q, \Sigma, q_0, \delta, F)$ definiert durch $Q := \{0, 1, \ldots, m\}$, $q_0 := 0$, $F := \{m\}$ und $\delta(q, a) := \sigma_P(P[1 \,..\, q]a)$.*

Dass Definition 6.6 so elegant und kurz ist, liegt natürlich an der passend definierten Suffixfunktion, von der wir uns noch keine Gedanken gemacht haben, wie wir sie konkret berechnen wollen. Wir verschieben das auch auf einen späteren Zeitpunkt und weisen zunächst nach, dass der in Definition 6.6 definierte DFA A_P tatsächlich geeignet ist, um gemeinsam mit Algorithmus 6.4 jedes Vorkommen von P in einem beliebigen Text T zu finden. Dazu beweisen wir zunächst zwei recht einfache Hilfsaussagen.

Lemma 6.7. $\forall P \in \Sigma^*, x \in \Sigma^*, a \in \Sigma \colon \sigma_P(xa) \leq \sigma_P(x) + 1$

Beweis. Für $\sigma_P(xa) = 0$ ist die Aussage trivial, weil $\sigma_P(x)$ nicht negativ sein kann. Wir dürfen also $\sigma(xa) > 0$ voraussetzen. Gemäß Definition von σ_P ist $P[1 \,..\, \sigma_P(xa)]$ Suffix von xa. Darum ist natürlich auch $P[1 \,..\, \sigma_P(xa) - 1]$ Suffix von x. Hier brauchen wir übrigens $\sigma_P(xa) > 0$, andernfalls wäre $\sigma_P(xa) - 1$ negativ. Gemäß Definition von σ_P folgt jetzt $\sigma_P(x) \geq \sigma_P(xa) - 1$. Wir formen äquivalent um und erhalten $\sigma_P(xa) \leq \sigma_P(x) + 1$. □

Lemma 6.8. $\forall P \in \Sigma^*, x \in \Sigma^*, a \in \Sigma\colon\ (\sigma_P(x) = q) \Rightarrow (\sigma_P(xa) = \sigma_P(P[1\ ..\ q]a))$

Beweis. Natürlich impliziert $\sigma_P(x) = q$ zunächst, dass $P[1\ ..\ q]$ Suffix von x ist. Also ist auch $P[1\ ..\ q]a$ Suffix von xa. Wir definieren $r := \sigma_P(xa)$ und haben $r \leq q + 1$ gemäß Lemma 6.7. Also ist $r = |P[1\ ..\ r]| \leq |P[1\ ..\ q]a| = q + 1$.
Wir haben $\sigma_P(xa) = r$ und wissen darum, dass $P[1\ ..\ r]$ Suffix von xa ist. Zusammen haben wir jetzt gezeigt, dass $P[1\ ..\ r]$ Suffix von $P[1\ ..\ q]a$ ist. Deshalb ist $\sigma_P(xa) = r \leq \sigma_P(P[1\ ..\ q]a)$.
Andererseits ist aber $\sigma_P(P[1\ ..\ q]a) \leq \sigma_P(xa)$, weil $P[1\ ..\ q]a$ Suffix von xa ist. Also haben wir zusammen $\sigma_P(xa) = \sigma_P(P[1\ ..\ q]a)$ wie behauptet. □

Wir haben jetzt das nötige Rüstzeug beisammen, um die Korrektheit des in Definition 6.6 definierten endlichen Automaten zu beweisen. Damit haben wir zumindest gezeigt, dass es beim String Matching ausreicht, sich jeden Buchstaben $T[i]$ nur einmal anzusehen.

Theorem 6.9. *Für alle Suchmuster $P \in \Sigma^*$ und alle Texte $T \in \Sigma^*$ gilt für den in Definition 6.6 definierten DFA A_P $\delta(q_0, T[1\ ..\ i]) = \sigma_P(T[1\ ..\ i])$ für alle $i \in \{0, 1, \ldots, |T|\}$.*

Beweis. Wir führen den Beweis mit vollständiger Induktion über i. Für den Induktionsanfang haben wir $i = 0$ und sehen, dass $\delta(q_0, \varepsilon) = q_0 = 0 = \sigma_P(\varepsilon)$ gilt. Für den Induktionsschluss dürfen wir voraussetzen, dass $\delta(q_0, T[1\ ..\ i]) = \sigma_P(T[1\ ..\ i])$ gilt. Wir betrachten den nächsten Buchstaben und definieren $a := T[i+1]$ und $q := \delta(q_0, T[1\ ..\ i])$. Damit haben wir $\delta(q_0, T[1\ ..\ (i+1)]) = \delta(q_0, T[1\ ..\ i]a)$, wir können die übliche Definition der Erweiterung von δ auf Zeichenketten benutzen und erhalten $\delta(q_0, T[1\ ..\ i]a) = \delta(\delta(q_0, T[1\ ..\ i]), a) = \delta(q, a)$, wobei die letzte Gleichung aus unserer Definition von q folgt. Gemäß Definition von δ haben wir $\delta(q, a) = \sigma_P(P[1\ ..\ q]a)$. Wir benutzen Lemma 6.8 und sehen, dass $\sigma_P(P[1\ ..\ q]a) = \sigma_P(T[1\ ..\ i]a)$ folgt. Wir ersetzen a durch $T[i+1]$ und haben insgesamt $\delta(q_0, T[1\ ..\ (i+1)]) = \sigma_P(T[1\ ..\ (i+1)])$ gezeigt. □

Wir haben jetzt also einen Plan für einen zweischrittigen Algorithmus zum String Matching: Im ersten Schritt wird der DFA A_P berechnet, im zweiten Schritt wird der DFA auf T mit Hilfe von Algorithmus 6.4 simuliert. Die Rechenzeit des ersten Schritts kennen wir noch nicht, wir wissen aber, dass sie nicht von T und damit auch nicht von n abhängt. Die Gesamtrechenzeit beträgt also $O(f(m)) + O(n)$ für eine Funktion f. Die Zeit für die Berechnung von A_P ist die Zeit fürs Preprocessing; man kann danach beliebig viele Texte nach dem Suchmuster P durchsuchen, ohne diesen Schritt noch einmal ausführen zu müssen. Wenn zum Beispiel viele Dateien durchsucht werden

sollen, ist ein Ansatz mit einem solchen Preprocessing also durchaus vernünftig. Asymptotisch schneller als der naive Algorithmus (Algorithmus 6.2) sind wir aber nur, wenn die noch unbekannte Funktion f nicht zu schnell wächst. Wir werden darum jetzt darüber nachdenken, wie wir den DFA A_P konkret berechnen können.
Wir müssen für alle Zustände $q \in Q = \{0, 1, \ldots, m\}$ und alle Buchstaben $a \in \Sigma$ die Zustandsüberführungsfunktion $\delta(q, a) = \sigma_P(P[1 \,..\, q]a)$ berechnen, dabei ist $\sigma_P(P[1 \,..\, q]a) = \max\{l \mid P[1 \,..\, l] \sqsupset P[1 \,..\, q]a\}$. Wir können offenbar analog zum naiven Algorithmus (Algorithmus 6.2) direkt vorgehen und alle möglichen Werte für l in absteigender Reihenfolge ausprobieren.

Algorithmus 6.10 (**Naive δ-Berechnung**).
1. *For* $q \in \{0, 1, 2, \ldots, m\}$
2. *For* $a \in \Sigma$
3. $l := \min\{m, q + 1\}$
4. *Repeat*
5. *If* $P[l] = a$ *Then* $\{*$ *Ist* $P[1 \,..\, (l-1)] \sqsupset P[1 \,..\, q]$*?* $*\}$
6. $i := l$*; istSuffix* $:=$ *true*
7. *While* $i > 1$ *und istSuffix* $=$ *true*
8. $i := i - 1$ *; If* $P[i] \neq P[q + i - (l - 1)]$ *Then istSuffix* $:=$ *false*
9. *Else istSuffix* $:=$ *false*
10. *If istSuffix* $=$ *false Then* $l := l - 1$
11. *Until* $l = 0$ *oder istSuffix* $=$ *true*
12. $\delta(q, a) := l$
13. *Ausgabe* δ

Theorem 6.11. *Für alle $P \in \Sigma^m$ berechnet Algorithmus 6.10 in Zeit $O(|\Sigma| \cdot m^3)$ die Funktion $\delta\colon Q \times \Sigma \to Q$ wie in Definition 6.6 festgelegt.*

Beweis. Für die Laufzeit beobachten wir, dass die Schleife über q genau $m + 1$ Werte und die Schleife über a genau $|\Sigma|$ Werte durchläuft. Das trägt einen Faktor $O(|\Sigma| \cdot m)$ bei. In der Repeat-Schleife nimmt l offenbar höchstens $m + 1$ verschiedene Werte an, außerdem wird die Schleife entweder beendet oder l sinkt um 1. Das bringt uns einen weiteren Faktor $O(m)$ ein. Die While-Schleife schließlich läuft über höchstens l Werte, mit $l \leq m$ haben wir also einen weiteren Faktor m. Da alle Befehle in konstanter Zeit durchführbar sind, ergibt sich zusammen $O(|\Sigma| \cdot m^3)$ als Laufzeitschranke.
Die Korrektheit ist offensichtlich. Für alle Werte von $q \in Q$ und $a \in \Sigma$ wird l zunächst auf den maximalen Wert für σ_P gesetzt (Zeile 3), dann wird in den Zeilen 5–9 getestet, ob $P[1 \,..\, l] \sqsupset P[1 \,..\, q]a$ gilt. Wenn das der Fall ist, wird $\delta(q, a)$ entsprechend gesetzt, andernfalls l um 1 gesenkt. Das Verfahren endet spätestens bei $l = 0$, das ist korrekt, weil das leere Wort ε mit Länge 0 auf jeden Fall Suffix ist. □

Wenn m lang ist, ist eine Laufzeit von $O(|\Sigma| \cdot m^3)$ natürlich völlig inakzeptabel. Für sehr lange Texte T und nicht zu lange Muster P kann das Verfahren aber wesentlich besser sein als der naive Algorithmus (Algorithmus 6.2): Für $m = n^{1/3}$ zum Beispiel erhalten wir für den naiven Algorithmus eine Laufzeit von $\Theta(n^{4/3})$, während DFA-Konstruktion in Zeit $O(|\Sigma|\, n)$ und anschließende Suche in Zeit $\Theta(n)$ uns eine Gesamtlaufzeit von $\Theta(n)$ bringt. Ob das DFA-basierte Verfahren auch praktisch schneller ist, hängt aber entscheidend von der Größe des Alphabets Σ ab. Suchen wir zum Beispiel über dem in der Praxis wichtigen Alphabet $\Sigma = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$, so ist der DFA-Ansatz schnell. Suchen wir in natürlichsprachlichen Texten und verwenden eine UTF-32-Codierung, verbietet sich der Aufbau des DFA ganz offensichtlich.

Man kann den DFA A_P noch wesentlich schneller konstruieren. Darum lohnt es sich auch nicht darüber nachzudenken, ob die obere Schranke aus Theorem 6.11 eigentlich asymptotisch exakt ist. Wir wollen das hier aber nicht vertiefen und uns stattdessen vom DFA-Ansatz zu einem schnelleren Algorithmus inspirieren lassen, der dem gleichen Ansatz folgt, aber nicht die Berechnung von δ erfordert.

6.2 Der Algorithmus von Knuth, Morris und Pratt

Für den DFA A_P haben wir $\delta(q, a) = \sigma_P(P[1 \,..\, q]a)$ für alle q und a berechnet und eingesehen, dass der Aufwand, für alle Buchstaben $a \in \Sigma$ solche Vorabberechnungen durchzuführen bei großen Alphabeten nicht praktikabel ist. Es wäre viel besser, wenn man sich darauf beschränken könnte, für jeden Zustand $q \in Q$ Informationen vorab zu berechnen, aus denen sich $\delta(q, a)$ oder ähnlich hilfreiche Informationen effizient berechnen lassen.

Nehmen wir an, dass wir $P[1 \,..\, q] \sqsupset T[1 \,..\, (s + q)]$ haben und feststellen, dass $P[q + 1] \neq T[s + q + 1]$ gilt. Der naive Algorithmus (Algorithmus 6.2) setzt in dieser Situation seinen Vergleich von T und P mit dem Vergleich von $T[1 \,..\, (s+1)]$ und $P[1]$ fort. Wir können das aber einsparen, denn wir haben die Buchstaben $T[s + 1 \,..\, (s + q)]$ schon mit P vergleichen und wissen, dass $T[s + 1 \,..\, (s + q)] = P[1 \,..\, q]$ gilt. Aus diesem Wissen kann man ableiten, mit welcher Position $s + s'$ der Vergleich von T und P fortgesetzt werden muss: Wir müssen nur herausfinden, was der größte Wert von q' ist, so dass $T[s' \,..\, (s' + q' - 1)] = P[1 \,..\, q']$ gilt für irgendeinen Wert von $s' \leq s + q$. Die zentrale Beobachtung an dieser Stelle ist, dass wir diese Information berechnen können, ohne T zu betrachten. Weil $T[s + 1 \,..\, (s + q)] = P[1 \,..\, q]$ gilt, finden wir alles Nötige auch in P. Damit können wir genau wie für

die Berechnung des DFA A_P diese Berechnung in einem Preprocessing ohne Kenntnis von T durchführen.

Definition 6.12. *Für ein festes Suchmuster $P \in \Sigma^m$ ist die* Präfixfunktion *$\pi_P \colon \{1, 2, \ldots, m\} \to \{0, 1, \ldots, m-1\}$ durch*

$$\pi_P(q) = \max\{l \mid l < q \text{ und } P[1 \,..\, l] \sqsupset P[1 \,..\, q]\}$$

definiert.

Wir bezeichnen also mit $\pi_P(q)$ die Länge des längsten Präfixes von P, das Suffix von $P[1 \,..\, q]$ ist. Wenn wir jetzt wissen, dass $T[s{+}1 \,..\, (s{+}q)] = P[1 \,..\, q]$ und $T[s{+}q{+}1] \neq P[q{+}1]$ gilt, können wir q durch $\pi_P(q)$ ersetzen und wissen, dass wiederum $T[(s{+}1) \,..\, (s{+}q)] = P[1 \,..\, q]$ gilt. Dabei bedeutet $q = 0$, dass wir noch von keinem Zeichen in P Übereinstimmung mit T sicher wissen und unser Vergleich tatsächlich mit $P[1]$ weitergehen muss.
Wir werden jetzt zunächst die Berechnung von π_P und die Anwendung zum String Matching in einem Algorithmus vorstellen. Danach kümmern wir uns um Laufzeit und Korrektheit des Verfahrens.

Algorithmus 6.13 (**Algorithmus von Knuth, Morris und Pratt**).
1. *Berechne π_P.*
2. $q := 0$ {$*$ *Länge des passenden Präfixes von P* $*$}
3. *For $i \in \{1, 2, \ldots, n\}$*
4. *While $q > 0$ und $P[q+1] \neq T[i]$*
5. $q := \pi_P[q]$
6. *If $P[q+1] = T[i]$ Then $q := q + 1$*
7. *If $q = m$ Then Ausgabe $i - m + 1$; $q := \pi_P[q]$*

Berechnung von π_P
1. $l := 0$
2. $\pi_P[1] := 0$
3. *For $q \in \{2, 3, \ldots, m\}$*
4. *While $l > 0$ und $P[l+1] \neq P[q]$*
5. $l := \pi_P[l]$
6. *If $P[l+1] = P[q]$ Then $l = l + 1$*
7. $\pi_P[q] := l$
8. *Ausgabe von π_P*

Lemma 6.14. *Die Berechnung von π_P im Algorithmus von Knuth, Morris und Pratt (Algorithmus 6.13) erfolgt in Zeit $\Theta(m)$.*

Beweis. Die untere Schranke $\Omega(m)$ ist offensichtlich, weil die For-Schleife $m-1$ Werte durchläuft. Wir müssen also nur die obere Schranke $O(m)$ nachweisen und benutzen dazu amortisierte Analyse mit der Potenzialmethode (vergleiche Abschnitt 5.1). Wir definieren als Potenzial Φ den aktuellen Wert von l. Offensichtlich ist l initial 0, so dass die initiale Belegung von Φ korrekt ist. Der Wert von l wird ausschließlich in Zeile 5 kleiner, wo $l = \pi_P[l]$ gesetzt wird. Weil $\pi_P[i] \geq 0$ für alle i gilt, wird Φ nie negativ und wir haben Φ insgesamt korrekt definiert.
Weil gemäß Definition 6.12 $\pi_P[i] < i$ gilt für alle i, wird l in einem Schritt entweder verkleinert (Zeile 5) oder höchstens um 1 vergrößert (Zeile 6). Damit ist klar, dass die Potenzialdifferenz in einem Schritt immer durch 1 nach oben beschränkt ist und die amortisierte Zeit für jede Zeile $O(1)$ bleibt. Die amortisierte Zeit für die Ausführung der While-Schleife (Zeilen 4–5) ist durch 0 nach oben beschränkt, weil die Zuweisung einen Rechenschritt erfordert und l dabei mindestens um 1 sinkt. Darum ist die amortisierte Zeit für die Anweisungen innerhalb der For-Schleife (Zeilen 4–7) durch $O(1)$ nach oben beschränkt und wir haben insgesamt Rechenzeit $O(m)$. □

Lemma 6.15. *Der Algorithmus von Knuth, Morris und Pratt (Algorithmus 6.13) hat Laufzeit $\Theta(n+m)$.*

Beweis. Die Zeit für die Berechnung von π_P beträgt $\Theta(m)$ (Lemma 6.14), so dass wir uns nur noch um den Hauptteil (Zeilen 2–7) kümmern müssen. Die untere Schranke $\Omega(n)$ ist wieder offensichtlich, weil die For-Schleife über n Werte läuft. Für die obere Schranke können wir ganz analog zum Beweis von Lemma 6.14 amortisierte Analyse mit einer Potenzialfunktion benutzen und als Potenzial den Wert von q benutzen. □

Der Algorithmus von Knuth, Morris und Pratt (Algorithmus 6.13) ist also ein Linearzeitalgorithmus, was schon ein Grund zur Zufriedenheit ist. Allerdings wäre schon ganz schön, wenn er auch noch unser Problem korrekt löste. Wir wollen uns davon als Nächstes überzeugen und beweisen dazu zunächst eine hilfreiche Aussage über die iterierte Anwendung der Präfixfunktion π_P.

Lemma 6.16. *Sei $P \in \Sigma^m$ ein Suchmuster. Wir definieren $\pi_P^{(0)}(q) := q$ und $\pi_P^{(i)}(q) := \pi_P\left(\pi_P^{(i-1)}(q)\right)$. Außerdem sei $\pi^*(q) := \bigcup_{i=1}^{\infty}\left\{\pi_P^{(i)}(q)\right\}$.*
$\forall q \in \{1,2,\ldots,m\}\colon \pi_P^*(q) = \{l \mid l < q \text{ und } P[1\,..\,l] \sqsupset P[1\,..\,q]\}$

Beweis. Wir beweisen die Gleichheit der Mengen $\pi_P^*(q)$ und $\{l \mid l < q$ und $P[1\,..\,l] \sqsupset P[1\,..\,q]\}$ in zwei Schritten. Zunächst zeigen wir $\pi_P^*(q) \subseteq \{l \mid l < q$ und $P[1\,..\,l] \sqsupset P[1\,..\,q]\}$.

Betrachte ein $l \in \pi_P^*(q)$. Gemäß Definition von $\pi_P^*(q)$ gibt es dann ein $i > 0$, so dass $\pi_P^{(i)}(q) = l$ gilt. Ist $i = 1$, gilt also $\pi_P(q) = l$, so folgt $P[1 \,..\, l] \sqsupset P[1 \,..\, q]$ aus der Definition von π_P. Andernfalls ist $\pi_P^{(i)}(q) = \pi_P\left(\pi_P^{(i-1)}(q)\right) = l$. Auf den ersten Blick scheint es hier schwierig zu werden, weil wir die Suche nach möglichst langen Suffixen nun von $P[1 \,..\, q]$ nach $P[1 \,..\, j]$ für $j = \pi_P^{(i-1)}(q) < q$ verlagern. Aber der erste Blick täuscht. Weil $P[1 \,..\, j] \sqsupset P[1 \,..\, q]$ gilt, ist natürlich $P[1 \,..\, j] = P[(q - j + 1) \,..\, q]$ und wir können die Suche gedanklich in $P[(q - j + 1) \,..\, q]$ fortsetzen, das ändert gar nichts im Vergleich zur Suche in $P[1 \,..\, j]$. Wir sehen, dass wir mit jedem i also immer kürzer werdende Suffixe betrachten, für die wir weiterhin nach möglichst langen Präfixen von P suchen, die jeweils Suffix sind. Darum ist klar, dass auf jeden Fall $P[1 \,..\, l] \sqsupset P[1 \,..\, q]$ gilt wie behauptet und wir haben $\pi_P^*(q) \subseteq \{l \mid l < q \text{ und } P[1 \,..\, l] \sqsupset P[1 \,..\, q]\}$ gezeigt.
Nun müssen wir noch $\{l \mid l < q \text{ und } P[1 \,..\, l] \sqsupset P[1 \,..\, q]\} \subseteq \pi_P^*(q)$ zeigen und führen dazu einen Beweis durch Widerspruch. Wir nehmen an, dass es ein $l < q$ mit $P[1 \,..\, l] \sqsupset P[1 \,..\, q]$ gilt, so dass $l \notin \pi_P^*(q)$ gilt. Sei l^* das größte l mit dieser Eigenschaft. Wir sehen, dass $\pi_P^{(i)}(q)$ monoton mit i fällt. Wir folgern, dass 0 auf jeden Fall zu $\pi_P^*(q)$ gehört und wir $l^* > 0$ annehmen können. Sei i^* das größte i, so dass $\pi_P^{(i)}(q) \geq l^*$ gilt. Weil $\pi_P^{(i^*)}(q) = l^*$ direkt zu $l^* \in \pi_P^*(q)$ und somit zum Widerspruch führt, haben wir $\pi_P^{(i^*)}(q) > l^*$. Ein solches i^* muss es geben, andernfalls wäre $l^* = \pi_P^{(1)}(q)$ und wir hätten wiederum einen Widerspruch zur Annahme, dass $l^* \notin \pi_P^*(q)$ gilt. Wir betrachten jetzt $\pi_P^{(i^*+1)}(q) = l'$. Wir haben $l' \leq l^*$, andernfalls ist i^* nicht maximal. Es kann nicht $l' = l^*$ gelten, andernfalls folgt $l^* \in \pi_P^*(q)$. Folglich gilt $l' < l^*$. Gemäß Definition von π_p wird aber immer das längste Präfix gewählt. Gilt $l' < l^*$, so ist l^* die Länge des längsten Präfixes und nicht l', ein Widerspruch. Wir haben also insgesamt $\pi_P^*(q) = \{l \mid l < q \text{ und } P[1 \,..\, l] \sqsupset P[1 \,..\, q]\}$ wie behauptet. □

Lemma 6.17. *Der Algorithmus von Knuth, Morris und Pratt (Algorithmus 6.13) berechnet für jedes Suchmuster $P \in \Sigma^*$ die Präfixfunktion σ_P korrekt.*

Beweis. Wir überzeugen uns zunächst, dass für alle $q \in \{1, 2, \ldots, m\}$ mit $\pi_P(q) > 0$

$$\pi_P(q) - 1 \in \pi_P^*(q - 1) \tag{6}$$

gilt: Aus $\pi_P(q) = r > 0$ folgt $r < q$ und $P[1 \,..\, r] \sqsupset P[1 \,..\, q]$. Wir lassen in beiden Zeichenketten den letzten Buchstaben fallen und erhalten $P[1 \,..\, r - 1] \sqsupset P[1 \,..\, q - 1]$, außerdem gilt natürlich $r - 1 < q - 1$. Folglich gilt $\pi_P(q - 1) = r - 1 \in \pi_P^*(q - 1)$ (Lemma 6.16).

Zentral bei der Berechnung von σ_P ist die Vergrößerung von l (Zeile 6 in der Berechnung von π_P in Algorithmus 6.13). Wir definieren dafür Mengen E_{q-1} für $q \in \{2, 3, \ldots, m\}$ durch

$$E_{q-1} := \{k \in \pi_P^*(q-1) \mid P[k+1] = P[q]\}\,.$$

Offenbar ist $E_{q-1} \subseteq \pi_P^*(q-1)$. Aus Lemma 6.16 folgt

$$E_{q-1} = \{k \mid k < q-1 \text{ und } P[1\,..\,k] \sqsupset P[1\,..\,q-1] \text{ und } P[k+1] = P[q]\}$$

und wir haben $E_{q-1} = \{k \mid k < q-1 \text{ und } P[1\,..\,k+1] \sqsupset P[1\,..\,q]\}$. Also enthält E_{q-1} die Werte $k < q-1$, so dass $P[1\,..\,k] \sqsupset P[1\,..\,(q-1)]$ und $P[1\,..\,(k+1)] \sqsupset P[1\,..\,q]$ gilt. Der Schluss von den längeren Zeichenketten zu den kürzeren ist natürlich trivial, man baut σ_P aber natürlich gerade in umgekehrter Richtung auf.
Wir behaupten, dass

$$\pi_P(q) = \begin{cases} 0 & \text{falls } E_{q-1} = \emptyset \\ 1 + \max\{k \mid k \in E_{q-1}\} & \text{sonst} \end{cases} \tag{7}$$

für alle $q \in \{2, 3, \ldots, m\}$ gilt. Betrachten wir zunächst den Fall, dass $E_{q-1} = \emptyset$ gilt. Dann gibt es kein k, so dass $P[1\,..\,(k+1)] \sqsupset P[1\,..\,q]$ gilt und $\pi_P(q) = 0$ folgt.
Sei nun $E_{q-1} \neq \emptyset$. Wir haben also $k+1 < q$ und $P[1\,..\,(k+1)] \sqsupset P[1\,..\,q]$ für alle $k \in E_{q-1}$. Also ist $\pi_P(q) \geq 1 + \max\{k \mid k \in E_{q-1}\}$ und insbesondere ist $\pi_P(q) > 0$. Sei $r = \pi_P(q) - 1$, also $\pi_P(q) = r + 1 \geq 1$. Offensichtlich gilt dann $P[r+1] = P[q]$. Aus Lemma 6.16 folgt $r \in \pi_P^*(q-1)$, also ist $r \in E_{q-1}$. Folglich gilt $r \leq \max\{k \mid k \in E_{q-1}\}$ und zusammen folgt $\pi_P(q) = 1 + \max\{k \mid k \in E_{q-1}\}$ wie behauptet.
Jetzt können wir die Korrektheit des Algorithmus gut nachweisen. Initial gilt $l = \pi_P(q-1)$ wegen der korrekten Initialisierung und spätere Schritte erhalten diese Eigenschaft. In den Zeilen 4–6 wird l so gesetzt, dass $l = \pi_P(q)$ gilt: Wir beginnen mit dem größten Wert für l und verkleinern l so weit wie nötig; Gleichung (7) garantiert, dass damit $\pi_P(q)$ korrekt berechnet wird. Die Schleife endet spätestens, wenn $l = 0$ gilt. In dem Fall können wir durch Vergleich von $P[1]$ und $P[q]$ entscheiden, ob $l = \pi_P(q) = 1$ gelten sollte. Für diesen und auch den Fall, dass die Schleife mit $l > 0$ beendet wird, wird l in Zeile 6 korrekt gesetzt. □

Theorem 6.18. *Der Algorithmus von Knuth, Morris und Pratt (Algorithmus 6.13) löst das String-Matching-Problem (Definition 6.1) in Zeit* $\Theta(n+m)$

Beweis. Die Aussage über die Laufzeit folgt aus Lemma 6.14 und Lemma 6.15, außerdem folgt aus Lemma 6.17, dass die Präfixfunktion π_P korrekt berechnet wird. Wir müssen also nur noch die Korrektheit der eigentlichen Suche im Text T nachweisen. Dazu zeigen, wir, dass Ausführung der Zeilen 4–6 die Eigenschaft $q = \delta(q, T[i])$ wie in Zeile 3 von Algorithmus 6.4 gewährleistet. Die Korrektheit folgt dann aus der Korrektheit von Algorithmus 6.4.
Es ist $\delta(q, T[i]) = \sigma_P(P[1 \,..\, q]T[i])$ gemäß Definition 6.6. Gilt $\sigma_P(P[1 \,..\, q]T[i]) = r$, so ist $P[1 \,..\, r] \sqsupset P[1 \,..\, q]T[i]$ gemäß Definition 6.5. Wir sehen direkt, dass dann auch $P[1 \,..\, (r-1)] \sqsupset P[1 \,..\, q]$ gilt und haben also $\delta(q, T[i]) - 1 \in \pi_P^*(q)$ oder $\delta(q, T[i]) = 0$. Wir haben beim Beweis von Lemma 6.17 gesehen, dass bei der Berechnung von π_P die Werte von $\pi_P^*(q)$ in absteigender Reihenfolge durchlaufen werden, das gilt analog auch für den Hauptalgorithmus. Der anschließende Vergleich in Zeile 6 setzt dann korrekt $q = \delta(q, T[i])$. □

6.3 Der Algorithmus von Boyer und Moore

Der Algorithmus von Knuth, Morris und Pratt (Algorithmus 6.13) ist mit seiner Laufzeit von $\Theta(m + n)$ asymptotisch sicher optimal: Im Zweifel wird man sich den ganzen Text und das ganze Suchmuster ansehen müssen. Er ist im Worst Case deutlich schneller als der naive Algorithmus (Algorithmus 6.2), wie hatten uns überlegt, dass er vor allem dann viel schneller ist, wenn im Suchmuster P viele Wiederholungen vorkommen: für $P = 0^m$ und $T = 0^n$ ist der naive Algorithmus besonders schlecht. Natürlich kommen solche Suchmuster und Texte bei der Suche in natürlichsprachlichen Texten nicht wirklich vor. Darum ist unklar, ob in praktischen Situationen der recht komplizierte Algorithmus von Knuth, Morris und Pratt wirklich viel schneller ist als der naive Algorithmus. Das motiviert die Suche nach praktisch schnelleren Algorithmen. Einen in der Praxis oft sehr effizienten Algorithmus wollen wir hier kennenlernen. Es gibt ihn in verschiedenen Varianten, gerne wird eine Version benutzt, die im günstigsten Fall mit Zeit $O(|\Sigma| + m + n/m)$ auskommt, für die allerdings wieder nur $O(|\Sigma| + n \cdot m)$ eine obere Schranke für die Laufzeit ist. Wir werden uns hier mit einer komplizierteren Fassung auseinandersetzen, die weiterhin im günstigsten Fall mit Zeit $O(|\Sigma| + m + n/m)$ auskommt, dabei aber asymptotisch optimal bleibt, insgesamt also mit Zeit $O(|\Sigma| + n + m)$ auskommt. Dass wir im günstigsten Fall mit Zeit $O(|\Sigma| + m + n/m)$ auskommen wollen, impliziert, dass wir

nicht mehr länger den Text T von links nach rechts durchlaufen können. Wir müssen auf das Lesen vieler Buchstaben von T ganz verzichten.
Als wir den naiven Algorithmus besprochen haben, haben wir uns klar gemacht, dass es Situationen gibt, in denen man das Suchmuster P um nicht mehr als eine Stelle nach rechts verschieben darf, wenn man nicht ein Vorkommen von P verpassen will. Das scheint zunächst die von uns anvisierte Verbesserung unmöglich zu machen. Es gibt aber einen einfachen Trick, der uns eine solche Verbesserung doch ermöglicht. Wir könnten den naiven Algorithmus geringfügig modifizieren, ohne seine Performanz zu beeinflussen: Algorithmus 6.2 vergleicht das Suchmuster P mit dem Text T ab Stelle i in der Reihenfolge $T[i+1] \stackrel{?}{=} P[1]$, $T[i+2] \stackrel{?}{=} P[2]$, …, $T[i+m] \stackrel{?}{=} P[m]$. Natürlich könnten wir die Reihenfolge ändern und in der Reihenfolge $T[i+m] \stackrel{?}{=} P[m]$, $T[i+m-1] \stackrel{?}{=} P[m-1]$, …, $T[i+1] \stackrel{?}{=} P[1]$ prüfen. Für den naiven Algorithmus macht das keinen Unterschied. Ein cleverer Algorithmus könnte sich aber fragen, ob das Zeichen $T[i+m]$ überhaupt in P vorkommt. Wenn das nicht der Fall ist, können wir i um m vergrößern und den Vergleich bei $T[(i+m)+m]$ fortsetzen: Jede kleinere Startposition s können wir ausschließen, weil $s < i+m$ bedeutet, dass das Suchmuster an den Positionen $s+1$ bis $s+m$ in T stehen müsste, weil aber $T[i+m]$ nicht in P vorkommt, muss $s+1 > i+m$ gelten, so dass $s > i+m-1$ folgt. Falls der Buchstabe $T[i+m]$ doch in P vorkommt, können wir trotzdem ausrechnen, um wie viel wir das Suchmuster nach rechts verschieben können, um unnötige Vergleiche zu sparen: Wir können das Suchmuster P so weit nach rechts verschieben, bis das am weitesten rechts gelegene Vorkommen des Buchstaben $T[i+m]$ in P mit Position $i+m$ übereinstimmt. Wir nennen diese Verschiebung Bad-Character-Verschiebung und halten ihre Definition auch formal fest. Dabei definieren wie die Bad-Character-Verschiebung so, dass um $i - \mathrm{bad}(a)$ verschoben wird, wenn beim Vergleich mit $P[i]$ der falsche Buchstabe $a \in \Sigma$ gelesen wurde.

Definition 6.19. *Für ein Suchmuster $P \in \Sigma^m$ ist die* Bad-Character-Verschiebung *bad*: $\Sigma \to \{0, 1, 2, \ldots, m\}$ *definiert durch*

$$bad(a) := \begin{cases} 0 & \text{falls } a \notin P \\ \max\{i \mid P[i] = a\} & \text{sonst} \end{cases}$$

für alle $a \in \Sigma$.

Wir sehen direkt, dass wir die Funktion bad in Zeit $\Theta(|\Sigma| + m)$ berechnen können, indem wir initial $b[a] := 0$ definieren für alle $a \in \Sigma$ und dann einmal das Suchmuster P aufsteigend (also von $P[1]$ nach $P[m]$) durchlaufen und

für jedes $i \in \{1, 2, \ldots, m\}$ einfach $b[P[i]] := i$ setzen. Am Ende haben wir dann $\text{bad}(a) = b[a]$.
Die Bad-Character-Verschiebung funktioniert hervorragend für Buchstaben, die nicht in P vorkommen. Für andere Buchstaben kann aber sogar eine Verschiebung nach links vorkommen, weil $i - \text{bad}(a)$ durchaus negativ werden kann. Wollte man nur die Bad-Character-Verschiebung benutzen, könnte man in diesen Fällen trotzdem um eine Stelle nach rechts verschieden. Das kann allerdings im Worst Case zu sehr schlechter Performanz, konkret zu Laufzeit $\Theta(n \cdot m)$ wie für den naiven Algorithmus, führen. Darum definieren wir noch eine zweite Verschiebung, die der Verschiebung mittels π_P aus dem Algorithmus von Knuth, Morris und Pratt durchaus ähnlich ist.
Wir nennen die zweite Verschiebung Good-Suffix-Verschiebung und definieren sie dreistufig in Bezug auf den aktuellen Vergleich, der an einem Buchstaben scheiterte. Zunächst definieren wir sie als kleinste Verschiebung von P, so dass das verschobene Suchmuster in allen in diesem Vergleich übereinstimmenden Buchstaben mit dem unverschobenen Suchmuster übereinstimmt und nicht mit ihm übereinstimmend bei dem Buchstaben, an dem der aktuelle Vergleich scheiterte. Wenn es keine solche Verschiebung gibt, so definieren wir sie als kleinste Verschiebung, die ein Präfix des verschobenen Suchmusters mit einem Suffix der im aktuellen Vergleich übereinstimmenden Buchstaben übereinstimmen lässt. Wenn auch das nicht möglich ist, definieren wir sie als m. Wir halten auch dies in einer formalen Definition fest.

Definition 6.20. *Für ein Suchmuster $P \in \Sigma^m$ ist die* Good-Suffix-Verschiebung *good*: $\{1, 2, \ldots, m\} \to \{1, 2, \ldots, m\}$ *definiert durch*

$$\mathit{good}(i) := \begin{cases} \min\{i - k \mid k < i, P[k] \neq P[i] \text{ und} \\ P[(k+1) \mathinner{..} m - i + k] = P[(i+1) \mathinner{..} m]\} & \text{falls so ein } k \text{ existiert} \\ \min\{m, m - k \mid P[1 \mathinner{..} k] \sqsupset P[1 \mathinner{..} m]\} & \text{sonst} \end{cases}$$

für alle $i \in \{1, 2, \ldots, m\}$.

Beide Verschiebungen sind zulässig, es ist darum zweckmäßig, die größere zu wählen, also das Muster um $\max\{i - \text{bad}(a), \text{good}(i)\}$ zu verschieben. Das löst auch das Problem der negativen Verschiebung bei der Bad-Character-Verschiebung.
Die Good-Suffix-Verschiebung lässt sich in Zeit $O(m)$ berechnen, wenn man im Prinzip wie beim Algorithmus von Knuth, Morris und Pratt (Algorithmus 6.13 vorgeht, allerdings jetzt von rechts nach links durch das Suchmuster schreitend. Wir geben einen Algorithmus zur Berechnung an.

Algorithmus 6.21 (**Algorithmus zur Good-Suffix-Berechnung**).

1. *For* $i \in \{1, 2, \ldots, m\}$ $good(i) := m$
2. $tmp(m) := 1$
3. $i := m$
4. *For* $k \in \{m-1, m-2, \ldots, 1\}$
5. *While* $P[i] \neq P[k]$
6. $good(i) := \min\{good(i), i-k\}$
7. *If* $i < m$ *Then* $i + tmp(i+1)$ *Else While-Schleife beenden.*
8. *If* $P[i] = P[k]$
9. *Then* $tmp(k) := i - k$; $i := i - 1$
10. *Else* $tmp(k) := i - k + 1$
11. $i := i + 1$; $i' := 1$
12. *While* $i \leq m$
13. *For* $j \in \{i', i'+1, \ldots, i-1\}$
14. $good(j) := \min\{good(j), i-1\}$
15. $i' := i$; $i := i + tmp(i)$

Lemma 6.22. *Für ein Suchmuster* $P \in \Sigma^*$ *können die Bad-Character-Verschiebung und die Good-Suffix-Verschiebung in Zeit* $O(|\Sigma| + m)$ *berechnet werden.*

Beweis. Dass die Bad-Character-Verschiebung in Zeit $O(|\Sigma| + m)$ berechnet werden kann, hatten wir direkt im Anschluss an ihre Definition (Definition 6.19) bemerkt. Es genügt also zu zeigen, dass Algorithmus 6.21 die Good-Suffix-Verschiebung korrekt in Zeit $O(m)$ berechnet. Der Beweis der Laufzeit erfolgt im Prinzip genau wie beim Algorithmus von Knuth, Morris und Pratt: In den Zeilen 3–10 ist nur die While-Schleife kritisch, in ihr wird i erhöht, das kann aber nur geschehen, wenn i vorher verkleinert wurde, was nur $O(m)$ mal passiert, so dass wir insgesamt Zeit $O(m)$ für diesen Teil haben. In zweiten Teil (Zeilen 11-15) wird i' mit i erhöht (Zeile 15), so dass auch hier Zeit $O(m)$ ausreicht.
Für die Korrektheit machen wir uns zunächst klar, dass tmp wie im Algorithmus die passenden Verschiebungen definiert, es ist also $\text{tmp}(i) = l(i) - i$, dabei ist $l(i)$ der kleinste Index größer als i, so dass $P[l(i) \,..\, m] \sqsubset P[i \,..\, m]$ gilt oder $m+1$, falls es kein solches Präfix gibt. Wir beobachten, dass $\text{tmp}(i)$ im ersten Teil (Zeilen 3–10) korrekt berechnet wird analog zum Algorithmus von Knuth, Morris und Pratt. Damit wird deutlich, dass wir im ersten Teil v_1 und im zweiten Teil v_2 korrekt berechnen. Da wir zur Initialisierung m benutzen und immer durch das Minimum des aktuellen Wertes und des neu errechneten Wertes ersetzen, wird good korrekt berechnet. □

Wir wollen jetzt den gesamten Algorithmus beschreiben und analysieren. Es wird allerdings einfacher sein, den Algorithmus für Suchmuster zu betrachten,

die nicht semizyklisch sind. Wir definieren zunächst formal, was wir damit meinen und halten zwei einfache Beobachtungen fest.

Definition 6.23. *Eine Zeichenkette $u \in \Sigma^*$ heißt* semizyklisch*, wenn es $w \in \Sigma^*$, $v \in \Sigma^+$ und $k \in \mathbb{N}$ gibt, so dass $u = wv^k$ ist, $v \neq w$ und $w \sqsupset v$ gilt. Falls sogar $w = \varepsilon$ gilt, heißt u* zyklisch*. Wir sagen, u ist semizyklisch (bzw. zyklisch) in Bezug auf v. Gilt $k > 1$, so heißt u* echt semizyklisch *(bzw.* echt zyklisch*) in Bezug auf v.*
Eine Zeichenkette $v \in \Sigma^*$ heißt echte zyklische Verschiebung *einer Zeichenkette $w \in \Sigma^*$, wenn es $x, y \in \Sigma^+$ gibt, so dass $w = xy$ und $v = yx$ gilt.*

Lemma 6.24. *Seien $x, y \in \Sigma^+$. Falls $xy = yx$ gilt, so gibt es ein $z \in \Sigma^+$, so dass x und y beide zyklisch in z sind.*

Beweis. Wir führen den Beweis mit vollständiger Induktion über $|x| + |y|$, beobachten aber zunächst, dass das Ergebnis für $|x| = |y|$ direkt folgt. Wir behaupten nämlich, dass in diesem Fall $x = y$ gilt. Das ist leicht einzusehen: $xy = yx$ impliziert, dass die ersten beiden Buchstaben in x und y gleich sind. Wir entfernen diese beiden Buchstaben und nennen den Rest x' und y'. Es gilt nun $x'y' = y'x'$ und wir können die Argumentation fortsetzen, bis $|x| = |y| = 1$ gilt. Die Aussage folgt also induktiv für alle gleich langen x und y. Für $x = y$ ist aber $z = x = y$ passend, x und y sind dann beide (nicht echt) zyklisch in Bezug auf z. Sei nun $|x| \neq |y|$, durch Umbenennung können wir $|x| < |y|$ sicherstellen. Aus $xy = yx$ folgt $x \sqsubset y$ und wir können $y = xy'$ mit $y' \in \Sigma^+$ schreiben. Wir erhalten also $xxy' = xy'x$ und können $xy' = y'x$ schließen. Weil $|y'| < |y|$ gilt, erhalten wir dank Induktionsvoraussetzung ein z', so dass $x = z'^i$ und $y' = z'^j$ mit $i, j \in \mathbb{N}$ gilt. Wir haben also $y = xy' = z'^i z'^j = z'^{i+j}$ wie behauptet. □

Lemma 6.25. *Ist $v \in \Sigma^+$ eine echte zyklische Verschiebung von $w \in \Sigma^+$ und gilt $v = w$, so ist v zyklisch.*

Beweis. Weil v eine echte zyklische Verschiebung von w ist, gibt es $x, y \in \Sigma^+$, so dass $v = xy$ und $w = yx$ gilt. Wir können also statt $v = w$ auch $xy = yx$ schreiben und Lemma 6.24 anwenden. Es gibt also ein $z \in \Sigma^+$, so dass $x = z^i$ und $y = z^j$ gilt, damit ist also $v = z^{i+j}$. □

Wir beschreiben jetzt den Algorithmus von Boyer und Moore und präsentieren dann die Laufzeit für Suchmuster, die nicht echt semizyklisch sind. Die Verallgemeinerung für beliebige Suchmuster diskutieren wir im Anschluss.

Algorithmus 6.26 (**Algorithmus von Boyer und Moore**)**.**
1. $i := 1$ {∗ *Position von* $P[1]$ *in* T ∗}

2. *While* $i \leq n - m + 1$
3. $\quad j := m$ {$*$ *nächster Vergleichsversuch* $*$}
4. $\quad$ *While* $P[j] = T[i + j - 1]$ *und* $j \geq 1$
5. $\quad\quad j := j - 1$
6. $\quad$ *If* $j > 0$ {$*$ $j > 0 \Leftrightarrow$ *keine Übereinstimmung* $*$}
7. $\quad$ *Then* $i := i + \max\{j - \mathit{bad}(T[i + j - 1]), \mathit{good}(j)\}$
8. $\quad$ *Else Ausgabe* i *;* $i := i + \mathit{good}(1)$

Die Verschiebung in Zeile 8 setzt voraus, dass P nicht echt semizyklisch ist, andernfalls könnten wir ein Vorkommen verpassen. Wir können das modifizieren, indem wir nach einer Ausgabe immer $i := i + 1$ setzen. Dann finden wir das erste Vorkommen von P immer noch im Worst Case in Zeit $O(|\Sigma|+m+n)$, brauchen aber im Worst Case für das Finden alle Vorkommen Zeit $\Omega(|\Sigma| + m \cdot n)$, was offenbar unbefriedigend ist. Wir diskutieren darum später, wie wir mit echt semizyklischen Suchmustern umgehen wollen.

Theorem 6.27. *Für nicht echt semizyklische Suchmuster $P \in \Sigma^*$ löst der Algorithmus von Boyer und Moore (Algorithmus 6.26) das String-Matching-Problem in Zeit $O(|\Sigma| + m + n)$.*

Beweis. Die Korrektheit folgt unmittelbar aus der Korrektheit der Verschiebungen. Wir müssen also nur die obere Schranke für die Laufzeit beweisen. Die Zeit $O(|\Sigma| + m)$ brauchen wir für das Preprocessing, also die Berechnung der beiden Verschiebungsfunktionen. Für den Vergleich mit dem Text werden wir eine obere Schranke von $O(n)$ nachweisen. Das ist nicht ganz so einfach, wie es auf den ersten Blick scheinen mag. Dadurch, dass wir von rechts nach links vergleichen und mitunter P nur wenig nach rechts verschieben, kann ein einzelner Buchstabe in T mehrmals Gegenstand eines Vergleiches sein, darum sieht man nicht unmittelbar, dass im Worst Case nicht doch $\Omega(m \cdot n)$ Vergleiche notwendig sind.
Wir führen eine Beweis mit amortisierter Analyse mit der Potenzialmethode. Wir definieren als Potenzial Φ $3s+u$, dabei bezeichnet u die Anzahl der Buchstaben in T, die wir uns noch gar nicht angesehen haben und s die Anzahl der Positionen, über die wir P noch nicht hinweggeschoben haben. Initial ist also $\Phi = 3n + n = 4n$, was von 0 verschieden ist. Wir kümmern uns um diesen klaren Verstoß gegen unsere Anforderungen an Potenzialfunktionen (vergleiche Abschnitt 5.1) am Ende dieses Beweises.
Wir nehmen für die Analyse der amortisierten Rechenzeit zunächst an, dass nur good verwendet wird und überlegen uns danach, was die zusätzliche Verwendung von bad ändert.

Wir betrachten den aktuellen nicht zum Erfolg führenden Vergleichsversuch und nehmen an, dass t Buchstaben in T betrachtet wurden und dieser Ausschnitt mit T' bezeichnet ist. Die Verschiebung, die durch $P[j] \neq T[i+j]$ verursacht wird, betrage genau s. Wir betrachten das Suffix u der Länge s des Suchmusters. Nehmen wir an, dass $u = v^k$ mit $k \in \mathbb{N}$ für ein nicht echt zyklisches $v \in \Sigma^+$ gilt. Wenn $t \leq 3s$ gilt, sind die amortisierten Kosten sicher nicht positiv. Wir müssen uns also nur um den Fall $t > 3s$ kümmern. In diesem Fall muss T' semizyklisch in u sein, also auch semizyklisch in v. Wir wollen jetzt zeigen, dass vor dem aktuellen Vergleichsversuch in T' nur die am weitesten links stehenden $|v|-1$ und die am weitesten rechts stehenden $2|v|-1$ (aber nicht das am weitesten rechts stehende) Zeichen gelesen worden sind.

Wir beobachten zunächst, dass in einem früheren Versuch das rechteste Zeichen in P nicht über dem rechtesten Zeichen in v positioniert gewesen sein kann für jedes Vorkommen von v in T'. Nehmen wir an, das sei doch der Fall. Dann hätte es links vom linkesten Zeichen in T' eine Nichtübereinstimmung gegeben und wir hätten P um s Stellen nach rechts verschoben wie bei diesem Versuch, was aber in einem früheren Versuch nicht auch an gleicher Stelle passiert sein kann. Jede kürzere Verschiebung um ein Vielfaches von $|v|$ ist nicht möglich, weil dann der gleiche Buchstabe in die Position der Nichtübereinstimmung käme. Kürzere Verschiebungen um andere Längen brächten eine echte zyklische Verschiebung von v in Übereinstimmung mit v, gemäß Lemma 6.25 ist dann v selbst zyklisch, was wir ausgeschlossen hatten.

Wir sehen, dass in einem früheren Versuch nicht mehr als $|v|$ Vergleiche stattgefunden haben können und der letzte Vergleich eine Nichtübereinstimmung gewesen sein muss, andernfalls folgt wie eben mit Lemma 6.25, das v selbst zyklisch ist.

Wir behaupten außerdem, dass in einem früheren Versuch das rechteste Zeichen in P entweder über einem Zeichen aus dem am weitesten rechts gelegenen Vorkommen von v in T' positioniert war oder mit einem der ersten $|v|-1$ Zeichen in T'. Nehmen wir an, dass das nicht der Fall ist, und betrachten $|v|$ Vergleiche. Wäre das rechteste Zeichen in P anders positioniert, so fänden alle diese Vergleiche in T' statt. Wir haben uns schon überlegt, dass es dann spätestens beim $|v|$-ten Vergleich eine Nichtübereinstimmung mit anschließender Verschiebung gibt. Entweder wird bei dieser Verschiebung oder einer späteren Verschiebung in dieser Folge der rechteste Buchstabe von P über dem rechtesten Buchstaben von v positioniert, was aber nicht passieren kann, wie wir uns überlegt hatten.

Damit haben wir jetzt insgesamt gezeigt, dass höchstens $3|v|-3$ Zeichen in T' schon vorher gelesen worden sind. Wir haben $|v| \leq s$, also sind vorher höchs-

tens $3s - 3$ Zeichen in T' gelesen worden, so dass wir im aktuellen Versuch mindestens $t - 3s + 3$ Zeichen neu lesen. Das ergibt eine Potenzialdifferenz von mindestens $-t + 3s - 3$ durch das erstmalige Lesen der Buchstaben, $-3s$ durch die Verschiebung um s und reale Kosten t im aktuellen Versuch, so dass die amortisierten Kosten nicht positiv sind.
Man überzeugt sich leicht, dass die Bad-Character-Verschiebung die amortisierten Kosten nicht vergrößern kann, weil sie nur dann zum Tragen kommt, wenn sie eine größere Verschiebung verursacht als die Good-Suffix-Verschiebung, was die Potenzialdifferenz nur kleiner machen kann.
Es bleibt noch das Problem des initialen Potenzials. Wir erinnern uns (Abschnitt 5.1), dass

$$T_{\text{amortisiert}} = T_{\text{real}} + \Phi(D_m) - \Phi(D_0)$$

gilt, wobei $T_{\text{amortisiert}}$ die Summe der amortisierten Rechenzeiten, T_{real} die Summe der realen Rechenzeiten, $\Phi(D_0)$ das initiale Potenzial und $\Phi(D_m)$ das Potenzial am Ende der Rechnung bezeichnet. Wir erhalten also eine obere Schranke für die Gesamtrechenzeit, indem wir die amortisierte Rechenzeit noch um das initiale Potenzial vergrößern. Das gilt, weil die Potenzialfunktion keine negativen Werte annehmen kann. Da das initiale Potenzial hier $4n$ beträgt, haben wir $O(n)$ als obere Schrank für die Gesamtrechenzeit. □

Wenn man genau hinsieht, haben wir eine obere Schranke von $4n$ Vergleichen bewiesen. Es ist interessant zu beobachten, dass diese Schranke sogar gilt, wenn man auf die Bad-Character-Verschiebung verzichtet. Es ist aber gerade diese Verschiebung, die dem Algorithmus von Boyer und Moore in günstigen Fällen sublineare Laufzeit erlaubt, so dass wir in Implementierungen nicht auf sie verzichten wollen. Wenn man sich mehr Mühe gibt, kann man übrigens auch $3n$ als obere Schranke für die benötigte Anzahl Vergleiche zeigen. Außerdem kann man nachweisen, dass $3n - o(n)$ Vergleiche erforderlich sein können. Es gibt auch noch etwas aufwendigere Implementierungen, die mit $2n$ Vergleichen auskommen. Wir wollen das hier und jetzt aber nicht weiter vertiefen und uns nur noch überlegen, wie wir mit echt semizyklischen Suchmustern umgehen können.
Wir betrachten also ein Suchmuster $P \in \Sigma^+$ mit $P = wv^i$, dabei sind $w, v \in \Sigma^*$, $i \in \mathbb{N} \setminus \{1\}$, $w \sqsupset v$ und $|w| < |v|$. Wir suchen nun nicht direkt nach $P = wv^i$, stattdessen suchen wir nach wv. Wenn wir nun i Vorkommen von wv gefunden haben, deren Anfangspositionen jeweils Abstand $|v|$ haben, so haben wir eine Instanz von $wv^i = P$ gefunden. Offenbar wird die Anzahl der Vergleiche nicht verändert, die Länge des tatsächlich benutzten Muster ist nun kleiner, was die Laufzeit für den günstigsten Fall vergrößert. Allerdings

treten zumindest in natürlichsprachlichen Texten solche semizyklischen Suchmuster eher selten auf, so dass das keine wesentliche Einschränkung darstellt. Wir halten dieses allgemeine Ergebnis zum Schluss des Abschnitts noch als Theorem fest.

Theorem 6.28. *Der Algorithmus von Boyer und Moore (Algorithmus 6.26) löst das String-Matching-Problem (Definition 6.1) in Zeit* $O(|\Sigma| + m + n)$.

7 Hidden-Markow-Modelle

Man muss zugeben, dass die Überschrift dieses Kapitels bewusst etwas irreführend gewählt wurde: Es geht uns nur in sehr begrenztem Umfang tatsächlich um Hidden-Markow-Modelle, wir werden dieses interessante Thema nur benutzen, um dynamische Programmierung zu wiederholen, die auch in späteren Abschnitten noch eine Rolle spielen wird. Dynamische Programmierung ist schon im Grundstudium intensiv besprochen und benutzt worden: In der Vorlesung DAP 2 sind Algorithmen für das Rucksackproblem, die Berechnung optimaler statischer binärer Suchbäume, die Berechnung kürzester Wege zwischen allen Paaren von Knoten eines gewichteten Graphen sowie die Berechnung eines global optimalen Alignments gemäß der Methode der dynamischen Programmierung entworfen worden. In den Vorlesungen GTI und TIfAI kommt sowohl beim CYK-Algorithmus für das Wortproblem bei kontextfreien Sprachen als auch bei der Transformation eines deterministischen endlichen Automaten in einen regulären Ausdruck dynamische Programmierung zum Einsatz. Wir dürfen hier also beruhigt die Methode als bekannt voraussetzen und uns darauf beschränken, sie ausschließlich am konkreten Beispiel zu wiederholen.

Man kann zeitdiskrete *Markow-Ketten* als eine Form randomisierter endlicher Automaten beschreiben: Wir haben eine endliche Zustandsmenge Q, sind zu jedem Zeitpunkt in einem Zustand und stellen uns einen diskreten Takt vor, so dass wir in jedem Schritt einen neuen Zustand wählen, der nicht notwendig vom aktuellen Zustand verschieden ist. Endliche Automaten machen diese Zustandswechsel vom aktuellen Zustand und dem aktuellen Eingabesymbol abhängig. Markow-Ketten machen diese Zustandswechsel vom aktuellen Zustand und dem Ausgang eines Zufallsexperiments abhängig. Für jeden Zustand $i \in Q$ gibt es für jeden Zustand $j \in Q$ eine Wahrscheinlichkeit $a_{i,j}$ (also $0 \leq a_{i,j} \leq 1$), die angibt, mit welcher Wahrscheinlichkeit vom aktuellen Zustand i in den Zustand j gewechselt wird. Diese $a_{i,\cdot}$ definieren für den Zustand i eine Wahrscheinlichkeitsverteilung, es gilt also $\sum_{j \in Q} a_{i,j} = 1$ für jeden Zustand $i \in Q$. Das Besondere an Markow-Ketten ist, dass diese *Transitionswahrscheinlichkeiten* $a_{i,j}$ das Verhalten der Markow-Kette alleine beschreiben, es spielt also keine Rolle, wie die früheren Zufallsexperimente ausgegangen sind. Wenn wir eine Folge von Zuständen $q_0, q_1, \ldots, q_t$ betrachten, so gilt

$$\text{Prob}\,(q_{t+1} = i \mid q_0, q_1, \ldots, q_t) = \text{Prob}\,(q_{t+1} = i \mid q_t) = a_{q_t,i}$$

für alle solche Zustandsfolgen und Zustände $i \in Q$. Man nennt diese Gleichheit auch die *Markow-Eigenschaft*.
Hidden-Markow-Modelle können grob als eine Markow-Kette mit Ausgabe beschrieben werden: Wir haben eine endliche Zustandsmenge Q und Transitionswahrscheinlichkeiten $a_{i,j}$, zusätzlich haben wir ein endliches Ausgabealphabet Σ und Ausgabewahrscheinlichkeiten $e_i(b)$, die für jeden Buchstaben $b \in \Sigma$ die Wahrscheinlichkeit ausgeben, mit der b beim Erreichen des Zustandes $i \in Q$ ausgegeben wird. Die Idee ist, dass die Ausgabe beobachtet werden kann, während die Zustandswechsel verborgen sind. Wir halten die zentralen Komponenten eines Hidden-Markow-Modells (HMM) auch formal fest.

Definition 7.1. *Ein* Hidden-Markow-Modell (HMM) *ist definiert durch eine Markow-Kette mit endlicher Zustandsmenge* $Q = \{0, 1, \ldots, |Q| - 1\}$ *und Transitionswahrscheinlichkeiten* $a_{i,j}$ *mit* $\sum_{j \in Q} a_{i,j} = 1$ *für alle* $i \in Q$ *und ein endliches Ausgabealphabet* Σ *und Ausgabewahrscheinlichkeiten* $e_i(b)$ *mit* $\sum_{b \in \Sigma} e_i(b) = 1$ *für alle* $i \in Q$. *Der Anfangszustand ist* $\pi_0 = 0$.

Hidden-Markow-Modelle werden in vielen verschiedenen Bereichen eingesetzt, *die* klassische Anwendung findet man in der Erkennung gesprochener Sprache. Dort interpretiert man die aufgenommenen Signale als beobachtbare Ausgabe, die durchlaufene Zustandsfolge stellt die gesprochenen Teilwörter dar und man möchte auf Basis der Signale auf die Folge von Teilwörtern zurückschließen.
Im Zusammenhang mit Hidden-Markow-Modellen steht man vor drei Problemen: Beim *Decodierungsproblem* möchte man zu einer gegebenen Ausgabefolge eine wahrscheinlichste Zustandsfolge berechnen, also eine Zustandsfolge, die maximale Wahrscheinlichkeit hat, während der Ausgabe durchlaufen worden zu sein. Beim *Auswertungsproblem* möchte man zu einer Ausgabefolge bestimmen, mit welcher Wahrscheinlichkeit sie von einem gegebenen Hidden-Markow-Modell ausgegeben wird. Beim *Lernproblem* schließlich möchten wir ein passendes Hidden-Markow-Modell bestimmen, also zu gewünschten Ausgabefolgen Transitionswahrscheinlichkeiten $a_{i,j}$ und Ausgabewahrscheinlichkeiten $e_i(b)$ so bestimmen, dass die Erzeugungswahrscheinlichkeiten maximiert werden. Für das Decodierungsproblem und das Auswertungsproblem werden wir effiziente Algorithmen angeben, das Lernproblem hingegen ist schwierig. Wir werden vollständigkeithalber abschließend kurz eine bekannte Heuristik für dieses Problem besprechen.
Wir beginnen mit dem Decodierungsproblem. Gegeben ist also eine Ausgabefolge $x = (x_1, x_2, \ldots, x_n) \in \Sigma^n$, gesucht ist eine Zustandsfolge $\pi = (\pi_1, \pi_2, \ldots, \pi_n) \in Q^n$, so dass die bedingte Wahrscheinlichkeit, die Zustandsfolge π zu durchlaufen bei gegebener Ausgabefolge x maximal ist. Um uns

die Notation zu erleichtern, schreiben wir $[x]_a^b$, wenn wir $(x_a, x_{a+1}, \ldots, x_b)$ meinen, es ist also in dieser Notation $x = [x]_1^n$. Wir betrachten jetzt ein gegebenes Hidden-Markow-Modell für n Schritte, dabei durchläuft das HMM die Zustandsfolge $q = (q_1, q_2, \ldots, q_n)$ und erzeugt die Ausgabefolge $\sigma = (\sigma_1, \sigma_2, \ldots, \sigma_n)$. Wir suchen also eine Zustandsfolge π, so dass die Wahrscheinlichkeit $\text{Prob}\,(q = \pi \mid \sigma = x)$ maximal wird.
Wir erinnern uns jetzt an die Definition der bedingten Wahrscheinlichkeit $\text{Prob}\,(A \mid B) = \text{Prob}\,(A \wedge B)\,/\text{Prob}\,(B)$. Wir haben folglich

$$\text{Prob}\,(q = \pi \mid \sigma = x) = \frac{\text{Prob}\,(q = \pi \wedge \sigma = x)}{\text{Prob}\,(\sigma = x)}$$

und sehen, dass die Zustandsfolge π keinen Einfluss auf $\text{Prob}\,(\sigma = x)$ hat. Weil uns nicht die Wahrscheinlichkeit sondern die Folge π selbst interessiert, genügt es also, $\text{Prob}\,(q = \pi \wedge \sigma = x)$ zu maximieren.
Es ist leicht einzusehen, dass

$$\text{Prob}\,(q = \pi \wedge \sigma = x) = \prod_{i=1}^{n} a_{\pi_{i-1},\pi_i} \cdot e_{\pi_i}(x_i)$$

gilt: Wir hatten festgelegt, dass das HMM im Zustand $\pi_0 = 0$ startet, um die Zustandsfolge $\pi_1, \pi_2, \ldots, \pi_n$ zu durchlaufen, müssen jeweils die passenden Zustandswechsel erfolgen, die Einzelwahrscheinlichkeiten sind direkt als Transitionswahrscheinlichkeiten gegeben, wegen der Markow-Eigenschaft multiplizieren sich die Wahrscheinlichkeiten. Bei gegebenem Zustand π_i wird das für x passende Zeichen x_i mit Ausgabewahrscheinlichkeit $e_{\pi_i}(x_i)$ erzeugt, auch hier multiplizieren sich die Wahrscheinlichkeiten wegen der Markow-Eigenschaft. Transitionswahrscheinlichkeiten, Ausgabewahrscheinlichkeiten und Ausgabefolge stehen fest, wir müssen noch die Zustandsfolge π so wählen, dass die Wahrscheinlichkeit maximal wird. Wir überlegen uns jetzt, dass man das schrittweise machen kann.
Wir definieren

$$v_k(i) := \max\left\{\text{Prob}\left([q]_1^i = [\pi]_1^i \wedge [\sigma]_1^i = [x]_1^i\right) \mid [x]_1^i \in \Sigma^i, [\pi]_1^i \in Q^i, \pi_i = k\right\}$$

und sehen, dass wir eine Folge π suchen, die $\max\{v_k(n) \mid k \in Q\}$ maximiert. Wir überlegen uns zunächst, wie wir $v_k(i)$ berechnen können.
Für $i = 0$ ist klar, dass

$$v_k(0) = \begin{cases} 1 & \text{falls } k = 0 \\ 0 & \text{sonst} \end{cases}$$

gilt, weil unser HMM deterministisch im Zustand 0 startet. Für $i > 0$ beobachten wir, dass

$$v_k(i) = e_k(x_i) \cdot \max \{v_j(i-1) \cdot a_{j,k} \mid j \in Q\}$$

gilt wegen der Markow-Eigenschaft. Wenn wir $v_j(i-1)$ für alle $j \in Q$ als bekannt voraussetzen, können wir $v_k(i)$ in Zeit $O(|Q|)$ berechnen. Weil $v_k(0)$ bekannt ist, können wir die $v_k(i)$ in Reihenfolge aufsteigender Werte für i berechnen und uns die Ergebnisse in einer Tabelle merken. Die Tabelle hat Größe $|Q| \cdot n$, jeden Tabelleneintrag berechnen wir in Zeit $O(|Q|)$, wir kommen also insgesamt mit Zeit $O\left(|Q|^2 \cdot n\right)$ und Platz $O(|Q| \cdot n)$ aus. Damit haben wir alle $v_k(i)$, woran wir ursprünglich gar nicht interessiert waren. Es ist aber leicht einzusehen, dass wir auch eine passende Zustandsfolge mitberechnen können, wenn wir uns bei der Berechnung des Maximums jeweils merken, für welchen Zustand das Maximum angenommen wurde. Wir können das Decodierungsproblem also leicht lösen, der Algorithmus, der offenbar dem Paradigma der dynamischen Programmierung folgt, ist als *Viterbi-Algorithmus* bekannt. Wir halten das Ergebnis noch einmal ausdrücklich fest.

Theorem 7.2. *Der Viterbi-Algorithmus löst das Decodierungsproblem für ein HMM und eine Ausgabefolge der Länge n in Zeit $O\left(|Q|^2 \cdot n\right)$ auf Platz $O(|Q| \cdot n)$.*

Kommen wir jetzt zum Auswertungsproblem. Wir wollen zu einer HMM und einer festen Ausgabefolge $x = (x_1, x_2, \ldots, x_n) \in \Sigma^n$ bestimmen, mit welcher Wahrscheinlichkeit x erzeugt wird, wir sind also an $\text{Prob}(\sigma = x)$ interessiert. Wir machen zunächst die triviale und etwas willkürlich anmutende Beobachtung, dass

$$\text{Prob}(\sigma = x) = \sum_{k \in Q} \text{Prob}(q_n = k \wedge \sigma = x)$$

gilt. Wir definieren

$$f_k(i) := \text{Prob}\left(q_i = k \wedge [\sigma]_1^i = [x]_1^i\right)$$

und sehen, dass wir jetzt $\text{Prob}(\sigma = x) = \sum_{k \in Q} f_k(n)$ schreiben können. Diese Darstellung motiviert unsere triviale und willkürliche Beobachtung von oben: Es bietet sich an, die $f_k(i)$ mittels dynamischer Programmierung zu berechnen und so das Auswertungsproblem zu lösen.
Wir beginnen mit den einfachen Randfällen. Wir haben

$$f_k(0) = \begin{cases} 1 & \text{falls } k = 0, \\ 0 & \text{sonst,} \end{cases}$$

weil das HMM deterministisch im Zustand 0 startet. Für $i \geq 0$ haben wir

$$f_k(i+1) = \text{Prob}\left(q_{i+1} = k \wedge [\sigma]_1^{i+1} = [x]_1^{i+1}\right)$$

gemäß Definition der $f_k(i)$ und

$$\begin{aligned}&\text{Prob}\left(q_{i+1} = k \wedge [\sigma]_1^{i+1} = [x]_1^{i+1}\right)\\ &\qquad = \sum_{j \in Q} \text{Prob}\left(q_i = j \wedge q_{i+1} = k \wedge [\sigma]_1^{i+1} = [x]_1^{i+1}\right)\end{aligned}$$

analog zu unserer Beobachtung oben. Aus der Definition der bedingten Wahrscheinlichkeit $\text{Prob}(A \mid B) = \text{Prob}(A \wedge B)/\text{Prob}(B)$ können wir natürlich

$$\text{Prob}(A \wedge B) = \text{Prob}(A \mid B) \cdot \text{Prob}(B) \tag{8}$$

H ableiten. Wir spalten $[\sigma]_1^{i+1} = [x]_1^{i+1}$ auf in $[\sigma]_1^i = [x]_1^i \wedge \sigma_{i+1} = x_{i+1}$, was offensichtlich nichts ändert. Wir wenden Gleichung (8) mit $A = \left[q_{i+1} = k \wedge \sigma_{i+1} = x_{i+1}\right]$ und $B = \left[q_i = j \wedge [\sigma]_1^i = [x]_1^i\right]$ an, was

$$\begin{aligned}&\sum_{j \in Q} \text{Prob}\left(q_i = j \wedge q_{i+1} = k \wedge [\sigma]_1^{i+1} = [x]_1^{i+1}\right)\\ &\qquad = \sum_{j \in Q} \text{Prob}\left(q_{i+1} = k \wedge \sigma_{i+1} = x_{i+1} \mid q_i = j \wedge [\sigma]_1^i = [x]_1^i\right)\\ &\qquad\qquad \cdot \text{Prob}\left(q_i = j \wedge [\sigma]_1^i = [x]_1^i\right)\end{aligned}$$

liefert. Das ist hilfreich, weil wir $f_j(i)$ erkennen und darum jetzt insgesamt

$$\begin{aligned}&f_k(i+1)\\ &\qquad = \sum_{j \in Q} \text{Prob}\left(q_{i+1} = k \wedge \sigma_{i+1} = x_{i+1} \mid q_i = j \wedge [\sigma]_1^i = [x]_1^i\right)\\ &\qquad\qquad \cdot f_j(i)\end{aligned}$$

schreiben können. Wir erinnern uns daran, dass bei jedem Zustandswechsel die Wahrscheinlichkeiten nur vom aktuellen Zustand abhängen und die Wahrscheinlichkeiten bei der Ausgabe ebenfalls nur vom aktuellen Zustand abhängen. Darum ist für die Wahrscheinlichkeit von $q_{i+1} = k \wedge \sigma_{i+1} = x_{i+1}$ nur der Zustand q_i ausschlaggebend, die Ausgaben $\sigma_1, \ldots, \sigma_i$ haben keinen Einfluss darauf. Wenn wir das berücksichtigen, kommen wir zu

$$f_k(i+1) = \sum_{j \in Q} \text{Prob}(q_{i+1} = k \wedge \sigma_{i+1} = x_{i+1} \mid q_i = j) \cdot f_j(i)$$

als etwas vereinfachter Gleichung. Wir benutzen jetzt noch einmal Gleichung (8), diesmal mit $A = \left[\sigma_{i+1} = x_{i+1}\right]$ und $B = \left[q_{i+1} = k\right]$, so dass wir

$$\begin{aligned}&\sum_{j\in Q} \text{Prob}\left(q_{i+1} = k \wedge \sigma_{i+1} = x_{i+1} \mid q_i = j\right) \cdot f_j(i)\\ &= \sum_{j\in Q} \text{Prob}\left(\sigma_{i+1} = x_{i+1} \mid q_{i+1} = k \wedge q_i = j\right) \cdot \text{Prob}\left(q_{i+1} = k \mid q_i = j\right) \cdot f_j(i)\end{aligned}$$

erhalten. Wegen der Markow-Eigenschaft gilt

$$\text{Prob}\left(\sigma_{i+1} = x_{i+1} \mid q_{i+1} = k \wedge q_i = j\right) = \text{Prob}\left(\sigma_{i+1} = x_{i+1} \mid q_{i+1} = k\right)$$

und wir sehen, dass diese Wahrscheinlichkeit von j nicht abhängt, also als konstanter Faktor aus der Summe gezogen werden kann. Wir haben inzwischen schon

$$f_k(i+1) = \text{Prob}\left(\sigma_{i+1} = x_{i+1} \mid q_{i+1} = k\right) \sum_{j\in Q} \text{Prob}\left(q_{i+1} = k \mid q_i = j\right) \cdot f_j(i)$$

hergeleitet und sind jetzt am Ziel: Wenn der aktuelle Zustand bekannt ist, kennen wir die Wahrscheinlichkeit, einen bestimmten Buchstaben auszugeben, als Ausgabewahrscheinlichkeit, so dass $\text{Prob}\left(\sigma_{i+1} = x_{i+1} \mid q_{i+1} = k\right) = e_k(x_{i+1})$ gilt. Die Wahrscheinlichkeit für einen Zustandswechsel bei gegebenem aktuellen Zustand kennen wir als Transitionswahrscheinlichkeit, es gilt also $\text{Prob}\left(q_{i+1} = k \mid q_i = j\right) = a_{j,k}$. Wir fassen insgesamt zu

$$f_k(i+1) = e_k(x_{i+1}) \cdot \sum_{j\in Q} a_{j,k} \cdot f_j(i)$$

zusammen und sehen, dass wir eine Tabelle mit $|Q| \cdot n$ Einträgen bestimmen können, wobei jeder Eintrag in Zeit $O(|Q|)$ berechnet werden kann. Weil wir die $f_k(i)$ wie üblich in aufsteigender Reihenfolge berechnen, nennt man diesen Algorithmus, der auch dem Paradigma der dynamischen Programmierung folgt, *Forward-Algorithmus*.

Theorem 7.3. *Der Forward-Algorithmus löst das Auswertungsproblem für ein HMM und eine Ausgabefolge der Länge n in Zeit $O\left(|Q|^2 \cdot n\right)$ auf Platz $O(|Q| \cdot n)$.*

Mit dem Viterbi-Algorithmus haben wir ja einen effizienten Algorithmus, der uns zu einer Ausgabefolge x eine Zustandsfolge sucht, die größte Wahrscheinlichkeit hat, x zu erzeugen. Angenommen, wir wollen für einen Zeitpunkt t

mit $0 < t \leq n$ einen Zustand $q \in Q$ kennen, der mit größter Wahrscheinlichkeit der aktuelle Zustand ist. Können wir uns die Zustandsfolge, die uns der Viterbi-Algorithmus liefert, an der t-ten Stelle ansehen und das Ergebnis auslesen? Das ist leider nicht der Fall. Wir fragen jetzt nicht mehr nach einer Zustandsfolge, sondern nach einem einzelnen Zustand, der bei gegebener Ausgabefolge maximale Wahrscheinlichkeit hat. Natürlich können wir für eine Folge von Zeitpunkten $t \in \{1, 2, \ldots, n\}$ jeweils einen solchen wahrscheinlichsten Zustand kennen wollen und diese Zustände zu einer Zustandsfolge zusammensetzen. Das unterscheidet sich erheblich von der Ausgabe des Viterbi-Algorithmus: Separat betrachtet haben vielleicht die Zustände i und j bei gegebener Ausgabefolge x maximale Wahrscheinlichkeit, aktuelle Zustände zum Zeitpunkt t und Zeitpunkt $t+1$ zu sein; es ist aber durchaus möglich, dass $a_{i,j} = 0$ gilt, so dass vollständig ausgeschlossen ist, dass beides gleichzeitig in einem Durchlauf zutrifft und wir darum in der Ausgabe des Viterbi-Algorithmus i und j mit Sicherheit nicht bei Position t und $t+1$ finden.
Beschreiben wir noch einmal formal, was wir wollen. Zu einem gegebenen HMM, einer Ausgabefolge $x = (x_1, x_2, \ldots, x_n) \in \Sigma^n$ und einem Zeitpunkt t mit $0 < t \leq n$ suchen wir einen Zustand $\pi_t \in Q$, so dass $\text{Prob}\,(q_t = \pi_t \mid \sigma = x)$ maximal ist.
Für die Lösung dieses Problems erinnern wir uns wieder zunächst an die Definition der bedingten Wahrscheinlichkeit und sehen, dass

$$\text{Prob}\,(q_t = \pi_t \mid \sigma = x) = \frac{\text{Prob}\,(q_t = \pi_t \wedge \sigma = x)}{\text{Prob}\,(\sigma = x)}$$

gilt. Den Nenner, $\text{Prob}\,(\sigma = x)$, haben wir schon beim Auswertungsproblem berechnet, wir müssen also uns also nur noch um $\text{Prob}\,(q_t = \pi_t \wedge \sigma = x)$ kümmern. Weil der Zeitpunkt t offenkundig eine Art Zäsur darstellt, liegt es nahe, die Ausgabefolge x an diesem Zeitpunkt t zu zerlegen. Wir erhalten dann

$$\begin{aligned}&\text{Prob}\,(q_t = \pi_t \wedge \sigma = x)\\ &\qquad = \text{Prob}\left([\sigma]_{t+1}^n = [x]_{t+1}^n \wedge q_t = \pi_t \wedge [\sigma]_1^t = [x]_1^t\right)\end{aligned}$$

und benutzen Gleichung (8) mit $A = \left[\,[\sigma]_{t+1}^n = [x]_{t+1}^n\,\right]$ und $B = \left[q_t = \pi_t \wedge [\sigma]_1^t = [x]_1^t\,\right]$, so dass wir

$$\begin{aligned}&\text{Prob}\left([\sigma]_{t+1}^n = [x]_{t+1}^n \wedge q_t = \pi_t \wedge [\sigma]_1^t = [x]_1^t\right)\\ &\qquad = \text{Prob}\left([\sigma]_{t+1}^n = [x]_{t+1}^n \mid q_t = \pi_t \wedge [\sigma]_1^t = [x]_1^t\right)\\ &\qquad\qquad \cdot \text{Prob}\left(q_t = \pi_t \wedge [\sigma]_1^t = [x]_1^t\right)\end{aligned}$$

erhalten. In $\text{Prob}\left(q_t = \pi_t \wedge [\sigma]_1^t = [x]_1^t\right)$ erkennen wir $f_{\pi_t}(t)$, ein Blick auf

$$\text{Prob}\left([\sigma]_{t+1}^n = [x]_{t+1}^n \mid q_t = \pi_t \wedge [\sigma]_1^t = [x]_1^t\right)$$

lässt uns erkennen, dass die betrachtete Wahrscheinlichkeit vom Zustand zum Zeitpunkt t und nicht zusätzlich der Ausgabefolge bis zu diesem Zeitpunkt abhängt. Das vereinfacht die Angelegenheit erheblich, wir haben jetzt insgesamt

$$\text{Prob}\left(q_t = \pi_t \mid \sigma = x\right) = \text{Prob}\left([\sigma]_{t+1}^n = [x]_{t+1}^n \mid q_t = \pi_t\right) \cdot f_{\pi_t}(t)$$

nachgewiesen, dabei können wir die $f_{\pi_t}(t)$ berechnen, das haben wir ja für das Auswertungsproblem gemacht. Wir erleichtern uns die Notation, indem wir

$$b_k(i) := \text{Prob}\left([\sigma]_{i+1}^n = [x]_{i+1}^n \mid q_i = k\right)$$

definieren, damit haben wir

$$\text{Prob}\left(q_t = \pi_t \mid \sigma = x\right) = b_{\pi_t}(t) \cdot f_{\pi_t}(t)$$

und wollen die $b_k(i)$ berechnen. Wir gehen dafür ganz analog zu unserem Vorgehen bei der Herleitung des Forward-Algorithmus vor, schreiben also zunächst

$$\begin{aligned} b_k(i) = \text{Prob}\left([\sigma]_{i+1}^n = [x]_{i+1}^n \mid q_i = k\right) \\ = \sum_{j \in Q} \text{Prob}\left(q_{i+1} = j \wedge [\sigma]_{i+1}^n = [x]_{i+1}^n \mid q_i = k\right) \end{aligned}$$

und benutzen Gleichung (8) mit $A = \left[(\sigma_{i+1}, \ldots, \sigma_n) = (x_{i+1}, \ldots, x_n)\right]$ und $B = \left[q_{i+1} = j\right]$, was uns zu

$$\begin{aligned} \sum_{j \in Q} \text{Prob}\left(q_{i+1} = j \wedge [\sigma]_{i+1}^n = [x]_{i+1}^n \mid q_i = k\right) \\ = \sum_{j \in Q} \text{Prob}\left([\sigma]_{i+1}^n = [x]_{i+1}^n \mid q_{i+1} = j \wedge q_i = k\right) \\ \cdot \text{Prob}\left(q_{i+1} = j \mid q_i = k\right) \end{aligned}$$

führt. Wir erkennen die Transitionswahrscheinlichkeit und die stochastische Unabhängigkeit vom vorletzten Zustand, so dass wir zu

$$b_k(i) = \sum_{j \in Q} a_{k,j} \cdot \text{Prob}\left([\sigma]_{i+1}^n = [x]_{i+1}^n \mid q_{i+1} = j\right)$$

vereinfachen können. Wir spalten das erste uns interessierende Zeichen der Ausgabe x_{i+1} ab, so dass wir $[\sigma]_{i+1}^n = [x]_{i+1}^n$ durch $[\sigma]_{i+2}^n = [x]_{i+2}^n \wedge \sigma_{i+1} = x_{i+1}$ ersetzen und wenden wieder Gleichung (8) an, diesmal mit $A = \left[[\sigma]_{i+2}^n = [x]_{i+2}^n \right]$ und $B = \left[\sigma_{i+1} = x_{i+1}\right]$. Wir dürfen also

$$\begin{aligned} &b_k(i) \\ &\quad = \sum_{j \in Q} a_{k,j} \cdot \text{Prob}\left((\sigma_{i+2}, \ldots, \sigma_n) = [x]_{i+2}^n \mid \sigma_{i+1} = x_{i+1} \wedge q_{i+1} = j\right) \\ &\qquad\qquad \cdot \text{Prob}\left(\sigma_{i+1} = x_{i+1} \mid q_{i+1} = j\right) \end{aligned}$$

schreiben und erkennen in $\text{Prob}\left(\sigma_{i+1} = x_{i+1} \mid q_{i+1} = j\right)$ die Ausgabewahrscheinlichkeit $e_j(x_{i+1})$. Außerdem beobachten wir, dass die Wahrscheinlichkeitsverteilung der Ausgaben zu den Zeitpunkten zwischen $i+2$ und n (jeweils einschließlich) bei bekanntem Zustand zum Zeitpunkt $i+1$ vom zum Zeitpunkt $i+1$ ausgegebenen Symbol unabhängig ist. Wir haben also

$$\begin{aligned} &\text{Prob}\left([\sigma]_{i+2}^n = [x]_{i+2}^n \mid \sigma_{i+1} = x_{i+1} \wedge q_{i+1} = j\right) \\ &\qquad\qquad = \text{Prob}\left([\sigma]_{i+2}^n = [x]_{i+2}^n \mid q_{i+1} = j\right) \end{aligned}$$

und erkennen $b_j(i+1)$. Damit haben wir jetzt also insgesamt

$$b_k(i) = \sum_{j \in Q} a_{k,j} \cdot b_j(i+1) \cdot e_j(x_{i+1})$$

nachgewiesen. Wir können wieder dynamische Programmierung benutzen, um alle $b_k(i)$ zu berechnen, dafür müssen wir in Reihenfolge absteigender Werte von i vorgehen und brauchen $b_k(n)$ als „Randfall“. Wir haben $b_k(n)$ als bedingte Wahrscheinlichkeit für die Ausgaben vom Zeitpunkt $n+1$ bis n, es geht also um eine leere Folge von Ausgaben, darum gilt $b_k(n) = 1$. Weil wir diesmal im Vergleich zum Forward-Algorithmus in umgekehrter Reihenfolge vorgehen, spricht man hier vom *Backward-Algorithmus*.

Theorem 7.4. *Mit dem Backward-Algorithmus kann man für ein HMM und eine Ausgabefolge der Länge n für jeden Zeitpunkt $0 < t \leq n$ einen wahrscheinlichsten Zustand berechnen in Zeit $O\left(|Q|^2 \cdot n\right)$ auf Platz $O(|Q| \cdot n)$.*

Bei gegebenem HMM können wir also sowohl das Auswertungsproblem als auch das Decodierungsproblem effizient lösen, wir kommen mit Zeit $O\left(n \cdot |Q|^2\right)$ und Platz $O\left(n \cdot |Q|\right)$ aus. Es bleibt aber noch das Lernproblem. Wir zerlegen dieses Problem in zwei Teilprobleme, es ist die Größe der Zustandsmenge $|Q|$ festzulegen, außerdem müssen Wahrscheinlichkeiten $a_{i,j}$ und $e_i(b)$

geeignet gewählt werden. Über die Wahl von $|Q|$ werden wir uns hier keine Gedanken machen, für die Bestimmung der Wahrscheinlichkeiten werden wir uns mit einer einfachen Heuristik zufriedengeben. Die Heuristik ist als *Baum-Welch-Algorithmus* bekannt. Wir nehmen an, dass wir ein HMM M haben, so dass wir eine Ausgabefolge $x \in \Sigma^n$ mit Wahrscheinlichkeit $\text{Prob}(\sigma = x \mid M)$ erhalten. Anfangs kann man mit einem grundsätzlich beliebig gewählten HMM M starten, man wird in der Praxis versuchen, eine heuristisch gute erste Wahl zu treffen. Wir hatten bisher das HMM als gegeben angenommen und darum kurz $\text{Prob}(\sigma = x)$ geschrieben. Weil wir jetzt ja gerade das HMM modifizieren wollen, müssen wir etwas präziser sein und die Abhängigkeit der Wahrscheinlichkeit vom gewählten HMM explizit machen. Ziel des Baum-Welch-Algorithmus ist es, ein HMM M' zu finden, so dass $\text{Prob}(x \mid M') > \text{Prob}(x \mid M)$ gilt.
Für ein festes HMM M sei

$$p_{i,j}(t) := \text{Prob}(q_t = i \wedge q_{t+1} = j \mid \sigma = x)$$

die bedingte Wahrscheinlichkeit dafür, dass bei bekannter Ausgabefolge x der $(t+1)$-te Zustandswechsel vom Zustand i in den Zustand j führt. Wir können diese Wahrscheinlichkeiten $p_{i,j}(t)$ dafür benutzen, um zu schätzen, wie häufig ein Zustandsübergang von i nach j vorkommt, was uns einen Hinweis gibt, wie die Transitionswahrscheinlichkeit $a_{i,j}$ gesetzt werden sollte. Wir überlegen uns darum jetzt, wie wir $p_{i,j}(t)$ berechnen können.

Lemma 7.5. *Für ein festes HMM M, jede Ausgabefolge $x \in \Sigma^n$, alle Zustände $i, j \in Q$ und alle Zeitpunkte $t < n$ gilt*

$$p_{i,j}(t) = \frac{f_i(t) \cdot a_{i,j} \cdot e_j(x_{t+1}) \cdot b_j(t+1)}{\mathit{Prob}(\sigma = x)}.$$

Beweis. Gemäß Definition von $p_{i,j}(t)$ haben wir

$$p_{i,j}(t) = \text{Prob}(q_t = i \wedge q_{t+1} = j \mid \sigma = x),$$

so dass

$$p_{i,j}(t) = \frac{\text{Prob}(q_t = i \wedge q_{t+1} = j \wedge \sigma = x)}{\text{Prob}(\sigma = x)}$$

aus der Definition der bedingten Wahrscheinlichkeit folgt. Wir müssen also nur noch

$$\text{Prob}(q_t = i \wedge q_{t+1} = j \wedge \sigma = x) = f_i(t) \cdot a_{i,j} \cdot e_j(x_{t+1}) \cdot b_j(t+1)$$

nachweisen. Uns wieder am Zeitpunkt t als Zäsur orientierend spalten wir die Ausgabefolge x entsprechend auf, erhalten

$$\begin{aligned}&\text{Prob}\,(q_t = i \wedge q_{t+1} = j \wedge \sigma = x)\\ &\qquad = \text{Prob}\left(q_{t+1} = j \wedge [\sigma]_{t+1}^n = [x]_{t+1}^n \wedge q_t = i \wedge [\sigma]_1^t = [x]_1^t\right)\end{aligned}$$

und benutzen wieder Gleichung (8), diesmal mit $A = \big[q_{t+1} = j \wedge (\sigma_{t+1}, \ldots, \sigma_n) = (x_{t+1}, \ldots, x_n)\big]$ und $B = \big[q_t = i \wedge (\sigma_1, \ldots, \sigma_t) = (x_1, \ldots, x_t)\big]$. Wir bekommen so

$$\begin{aligned}&\text{Prob}\left(q_{t+1} = j \wedge [\sigma]_{t+1}^n = [x]_{t+1}^n \wedge q_t = i \wedge [\sigma]_1^t = [x]_1^t\right)\\ &\qquad = \text{Prob}\left(q_{t+1} = j \wedge [\sigma]_{t+1}^n = [x]_{t+1}^n \mid q_t = i \wedge [\sigma]_1^t = [x]_1^t\right)\\ &\qquad\qquad \cdot \text{Prob}\left(q_t = i \wedge [\sigma]_1^t = [x]_1^t\right)\end{aligned}$$

und erkennen in $\text{Prob}\left(q_t = i \wedge [\sigma]_1^t = [x]_1^t\right)$ natürlich $f_i(t)$ wieder. Es genügt also,

$$\begin{aligned}&\text{Prob}\left(q_{t+1} = j \wedge [\sigma]_{t+1}^n = [x]_{t+1}^n \mid q_t = i \wedge [\sigma]_1^t = [x]_1^t\right)\\ &\qquad\qquad = a_{i,j} \cdot e_j(x_{t+1}) \cdot b_j(t+1)\end{aligned}$$

nachzuweisen. Wir spalten die Ausgabe zum Zeitpunkt $t+1$ ab und wenden wieder Gleichung (8) an, diesmal mit $A = \big[(\sigma_{t+2}, \ldots, \sigma_n) = (x_{t+2}, \ldots, x_n)\big]$ und $B = \big[\sigma_{t+1} = x_{t+1} \wedge q_{t+1} = j\big]$. Damit haben wir

$$\begin{aligned}&\text{Prob}\left(q_{t+1} = j \wedge [\sigma]_{t+1}^n = [x]_{t+1}^n \mid q_t = i \wedge [\sigma]_1^t = [x]_1^t\right)\\ &\qquad = \text{Prob}((\sigma_{t+2}, \ldots, \sigma_n) = [x]_{t+2}^n \mid q_t = i \wedge [\sigma]_1^t = [x]_1^t\\ &\qquad\qquad \wedge \sigma_{t+1} = x_{t+1} \wedge q_{t+1} = j)\\ &\qquad\qquad \cdot \text{Prob}\left(\sigma_{t+1} = x_{t+1} \wedge q_{t+1} = j \mid q_t = i \wedge [\sigma]_1^t = [x]_1^t\right)\end{aligned}$$

und können zunächst direkt zu

$$\begin{aligned}&\text{Prob}\left(q_{t+1} = j \wedge [\sigma]_{t+1}^n = [x]_{t+1}^n \mid q_t = i \wedge [\sigma]_1^t = [x]_1^t\right)\\ &\qquad = \text{Prob}\left([\sigma]_{t+2}^n = [x]_{t+2}^n \mid q_{t+1} = j\right)\\ &\qquad\qquad \cdot \text{Prob}\left(\sigma_{t+1} = x_{t+1} \wedge q_{t+1} = j \mid q_t = i\right)\end{aligned}$$

vereinfachen. Wir erkennen in $\text{Prob}\left([\sigma]_{t+2}^n = [x]_{t+2}^n \mid q_{t+1} = j\right)$ jetzt $b_j(t+1)$ wieder, wir müssen also nur noch

$$\text{Prob}\left(\sigma_{t+1} = x_{t+1} \wedge q_{t+1} = j \mid q_t = i\right) = a_{i,j} \cdot e_j(x_{t+1})$$

nachweisen. Gleichung (8) mit $A = \left[\sigma_{t+1} = x_{t+1}\right]$ und $B = \left[q_{t+1} = j\right]$ liefert

$$\begin{aligned}&\text{Prob}\left(\sigma_{t+1} = x_{t+1} \wedge q_{t+1} = j \mid q_t = i\right)\\ &\qquad = \text{Prob}\left(\sigma_{t+1} = x_{t+1} \mid q_{t+1} = j \wedge q_t = i\right) \cdot \text{Prob}\left(q_{t+1} = j \mid q_t = i\right)\end{aligned}$$

und wir erkennen die Transitionswahrscheinlichkeit $a_{i,j}$. Die Markow-Eigenschaft liefert

$$\text{Prob}\left(\sigma_{t+1} = x_{t+1} \mid q_{t+1} = j \wedge q_t = i\right) = \text{Prob}\left(\sigma_{t+1} = x_{t+1} \mid q_{t+1} = j\right),$$

so dass wir die Ausgabewahrscheinlichkeit $e_j(x_{t+1})$ erkennen und der Beweis vollständig ist. □

Wir hatten uns schon überlegt, wie wir $b_i(k)$ und $f_i(k)$ mit dem Backward- und Forward-Algorithmus berechnen können, so dass wir algorithmisch hier vor keine neuen Schwierigkeiten gestellt werden. Wir hatten eingangs argumentiert, dass wir die erwartete Anzahl Zustandsübergänge von i nach j schätzen wollen. Sei $T_{i,j}$ diese Anzahl, wir wollen also $\text{E}\left(T_{i,j}\right)$ berechnen. Wenn wir

$$T_{i,j}(t) = \begin{cases} 1 & \text{falls im } t\text{-ten Schritt Übergang } i \rightsquigarrow j \\ 0 & \text{sonst} \end{cases}$$

betrachten, sehen wir, dass $T_{i,j} = \sum_{t=1}^{n-1} T_{i,j}(t)$ gilt und wir haben

$$\begin{aligned}\text{E}\left(T_{i,j}\right) &= \text{E}\left(\sum_{t=1}^{n-1} T_{i,j}(t)\right) = \sum_{t=1}^{n-1} \text{E}\left(T_{i,j}(t)\right)\\ &= \sum_{t=1}^{n-1} \text{Prob}\left(T_{i,j}(t) = 1\right) = \sum_{t=1}^{n-1} p_{i,j}(t).\end{aligned}$$

Es erscheint intuitiv sinnvoll, in der HMM M' die Transitionswahrscheinlichkeit $a'_{i,j}$ proportional zur erwarteten Anzahl von Übergängen zu setzen. Wir können das machen, wenn wir nicht nur wissen, wie oft wir einen Übergang von i zu j erwarten, sondern zusätzlich angeben können, wie oft wir erwarten, den Zustand i zu erreichen. Wir nennen die Wahrscheinlichkeit, im t-ten Schritt im Zustand i zu sein $\gamma_i(t)$ und haben $\gamma_i(t) = \sum_{j \in Q} p_{i,j}(t)$, so dass wir also auch $\gamma_i(t)$ ohne Schwierigkeiten ausrechnen können. Wir definieren analog zu $T_{i,j}$ die Zufallsvariable T_i, die angibt, wie oft der Zustand i bei Ausgabe von x erreicht wird. Analog zu $\text{E}\left(T_{i,j}\right)$ erhalten wir

$$\text{E}\left(T_i\right) = \sum_{t=1}^{n-1} \gamma_i(t).$$

Wir tragen das jetzt zusammen zum Algorithmus von Baum-Welch. Wir starten mit beliebigen Wahrscheinlichkeiten $a_{i,j}$ und $e_i(b)$ und setzen

$$a'_{i,j} := \frac{\mathrm{E}\,(T_{i,j})}{\mathrm{E}\,(T_i)} = \frac{\sum\limits_{t=1}^{n-1} p_{i,j}(t)}{\sum\limits_{t=1}^{n-1} \gamma_i(t)}$$

sowie

$$e'_i(b) := \frac{\sum\limits_{t=1}^{n-1} \gamma_i(t) \cdot \mathbb{1}_{x_t=b}}{\mathrm{E}\,(T_i)} = \frac{\sum\limits_{t=1}^{n-1} \gamma_i(t) \cdot \mathbb{1}_{x_t=b}}{\sum\limits_{t=1}^{n-1} \gamma_i(t)}.$$

Das iteriert man so lange, bis die Wahrscheinlichkeiten sich nicht mehr oder nur noch unwesentlich ändern. Natürlich wird man sich mit dem Fall, dass $\mathrm{E}\,(T_i) = 0$ gilt, besonders beschäftigen müssen: Er ist eventuell ein Hinweis darauf, dass $|Q|$ zu groß gewählt wurde. Außerdem wird man vermeiden wollen, Wahrscheinlichkeiten 0 zu setzen, so dass man in solchen Fällen Zähler und Nenner jeweils gleichartig vergrößern wird. Wir wollen uns hier nicht überlegen, dass dieses Vorgehen tatsächlich $\mathrm{Prob}\,(\sigma = x \mid M') \geq \mathrm{Prob}\,(\sigma = x \mid M)$ garantiert und unseren kleinen Ausflug zu den HMM an dieser Stelle beenden.

Teil II

Approximationsalgorithmen

8 Ein echt polynomielles Approximationsschema

Der zweite Teil dieses Skripts ist approximativen Lösungen gewidmet, wobei wir den Begriff „approximativ“ weit fassen wollen. Häufig wird man mit approximativen Lösungen zufrieden sein, wenn die Berechnung einer optimalen Lösung zu zeitaufwendig ist. Dabei kann die Grenze zwischen „schnell genug“ und „zu langsam“ der Grenze zwischen „polynomiell“ und „superpolynomiell“ entsprechen, das muss aber nicht unbedingt der Fall sein. Bevor wir uns ein erstes konkretes Beispiel ansehen, wollen wir kurz wiederholen, was wir unter einer Approximation verstehen und wie wir Approximationsalgorithmen klassifizieren.

8.1 Approximationsalgorithmen

Ein *Optimierungsproblem* ist definiert durch eine Menge von Instanzen I, eine Funktion S, die jeder Instanz $w \in I$ eine Menge zulässiger Lösungen zuordnet, einer Funktion v, die jeder zulässigen Lösung $s \in S(w)$ einen Wert $\in \mathbb{R}^+$ zuordnet und der Angabe, ob zu minimieren oder maximieren ist. Zu einer Instanz $w \in I$ sei $\mathrm{OPT}(w)$ der Wert einer optimalen Lösung, also je nach Optimierungsziel das Minimum oder Maximum von $\{v(s) \mid s \in S(w)\}$. Wir wissen schon aus GTI bzw. TIfAI, dass wir jedem Optimierungsproblem ein Entscheidungsproblem zuordnen können, indem wir als Eingabe des Entscheidungsproblems Paare aus einer Instanz $w \in I$ und einer Zahl k definieren, es ist dann zu entscheiden, ob $\mathrm{OPT}(w) \leq k$ (für Minimierungsprobleme) bzw. $\mathrm{OPT}(w) \geq k$ (für Maximierungsprobleme) gilt. Wir interessieren uns in der Regel nur für solche Optimierungsprobleme, bei denen sowohl I als auch $S(w)$ für alle $w \in I$ zu P gehören und v in polynomieller Zeit auswertbar ist, bei denen es also nicht zu schwierig ist zu entscheiden, ob wir eine legale Instanz w vor uns haben, ob wir zu einer solchen Instanz $w \in I$ eine zulässige Lösung s vor uns haben und welchen Wert $v(s)$ sie eine zulässige Lösung hat. Die Menge der Optimierungsprobleme, die diesen Anforderungen genügen und für die das zugehörige Entscheidungsproblem zu NP gehört, nennen wir $\mathcal{NPO}$. Wir betrachten ausschließlich Optimierungsprobleme aus $\mathcal{NPO}$.
Haben wir zu einem Optimierungsproblem eine zulässige Lösung $s \in S(w)$ zu einer Instanz $w \in I$, so nennen wir

$$r := \max\left\{\frac{v(s)}{\mathrm{OPT}(w)}, \frac{\mathrm{OPT}(w)}{v(s)}\right\}$$

Version vom 21. Oktober 2008, 8:11 Uhr

die *Güte* der Lösung s für w. Offensichtlich ist die Güte $r \geq 1$ und wir interessieren uns für Algorithmen, die möglichst kleine Güten garantieren. Die *Approximationsgüte* r_A eines Algorithmus A ist das Infimum über alle r', so dass A für Eingaben $w \in I$ Lösungen $A(w) \in S(w)$ mit Güte höchstens r' liefert. Wir nennen solch einen Algorithmus A dann einen r_A-*Approximationsalgorithmus* oder auch eine r_A-*Approximation*. Einen Algorithmus A, der für jedes $r = 1 + \varepsilon > 1$ eine Lösung mit Güte höchstens r für jede Eingabe $w \in I$ garantiert und Rechenzeit polynomiell in der Länge der Eingabe hat, nennen wir ein *polynomielles Approximationsschema* (PTAS). Einen Algorithmus A, der für jedes $r = 1 + \varepsilon > 1$ eine Lösung mit Güte höchstens r für jede Eingabe $w \in I$ garantiert und Rechenzeit polynomiell in der Länge der Eingabe und ε^{-1} hat, nennen wir ein *echt polynomielles Approximationsschema* (FPTAS).
Ein Optimierungsproblem, das eine r-Approximation mit konstantem $r \geq 1$ hat, heißt *approximierbar*. Die Menge aller approximierbaren Optimierungsprobleme bezeichnen wir mit $\mathcal{APX}$. Die Menge aller Optimierungsprobleme, die ein PTAS haben, nennen wir $\mathcal{PTAS}$. Die Menge aller Optimierungsprobleme, die sogar ein FPTAS haben, nennen wir $\mathcal{FPTAS}$. Die Menge aller Optimierungsprobleme, deren zugehöriges Entscheidungsproblem zu P gehört, nennen wir $\mathcal{PO}$. Es ist nicht schwer zu sehen, dass

$$\mathcal{PO} \subseteq \mathcal{FPTAS} \subseteq \mathcal{PTAS} \subseteq \mathcal{APX} \subseteq \mathcal{NPO}$$

gilt. Sollte P = NP gelten, so ist natürlich $\mathcal{PO} = \mathcal{NPO}$ und alle diese Klassen fallen zusammen. Sollte aber P $\neq$ NP gelten, so kann man zeigen, dass diese fünf Klassen alle echt verschieden sind, so dass es auch komplexitätstheoretisch zur Einordnung in eine Komplexitätsklasse sinnvoll ist, für ein Optimierungsproblem nach möglichst guten Approximationsalgorithmen zu suchen. Wir werden hier nicht beweisen, dass die Klassen unter der Annahme P $\neq$ NP alle verschieden sind, und verweisen das in eine Vorlesung, deren Schwerpunkt die Komplexitätstheorie ist.

8.2 Ein echt polynomielles Approximationsschema für das Rucksackproblem

Das Rucksackproblem (KP) kennen wir schon aus GTI bzw. TIfAI, Eingabe sind Gewichtswerte $g_1, g_2, \ldots, g_n \in \mathbb{N}$, Nutzenwerte $v_1, v_2, \ldots, v_n \in \mathbb{N}$ und eine Gewichtsschranke $G \in \mathbb{N}$, Ausgabe ist eine Bepackung $B \subseteq \{1, 2, \ldots, n\}$, so dass $\sum_{i \in B} g_i \leq G$ gilt und $\sum_{i \in B} v_i$ maximal wird. Wir dürfen natürlich $g_i \leq$

G für alle $i \in \{1, 2, \ldots, n\}$ voraussetzen. Wir wissen ebenfalls schon aus GTI bzw. TIfAI, dass die Entscheidungsvariante des Rucksackproblems NP-vollständig ist, also gilt KP $\in \mathcal{NPO}$ und aus P $\neq$ NP folgt KP $\notin \mathcal{PO}$. Wir wollen hier zeigen, dass das Rucksackproblem aber sehr gut approximierbar ist, wir werden ein echt polynomielles Approximationsschema für das Rucksackproblem besprechen.
Es gibt für das Rucksackproblem pseudopolynomielle Algorithmen, also Algorithmen, die das Problem optimal lösen und dabei eine Laufzeit haben, die polynomiell ist in der Länge der Eingabe und der Größe der größten vorkommenden Zahl. Tatsächlich gibt es zwei solche Algorithmen, die beide nach dem Prinzip der dynamischen Programmierung funktionieren: Beide bauen Tabellen auf, in denen einerseits die Menge der betrachteten Elemente schrittweise erweitert wird. Einer der beiden Algorithmus konstruiert Teilprobleme, indem andererseits das zulässige Gesamtgewicht schrittweise vergrößert wird, so dass man jeweils den maximalen Nutzen speichert, der mit den betrachteten Elementen und dem aktuellen zulässigen Gesamtgewicht erzielbar ist. Der andere Algorithmus konstruiert Teilprobleme, indem andererseits der zu erzielende Gesamtnutzen schrittweise vergrößert wird, so kann man jeweils das minimale Gewicht speichert, das ausreicht, um mit den betrachteten Elementen den gewünschten Nutzen zu erzielen. Wir werden diesen zweiten Algorithmus jetzt zunächst etwas genauer diskutieren.

Theorem 8.1. *Es gibt einen Algorithmus für das Rucksackproblem, der in Zeit* $O\left(n^2 \cdot \max\{v_1, \ldots, v_n\}\right)$ *eine optimale Lösung berechnet.*

Beweis. Für $i \in \{0, 1, \ldots, n\}$ und $V \in \left\{0, 1, \ldots, \sum_{i=1}^{n} v_i\right\}$ definieren wir $M_{i,V}$ als das minimale Gewicht, mit dem Nutzen V erzielt werden kann durch eine Auswahl der Objekte $\{1, 2, \ldots, i\}$, formal ist also

$$M_{i,V} = \min\left\{\sum_{j \in B} g_j \mid B \subseteq \{1, 2, \ldots, i\}, \sum_{j \in B} v_j = V\right\}$$

wenn es eine solche Auswahl B gibt, andernfalls ist $M_{i,V} = \infty$.
Wir haben $M_{0,0} = 0$, weil sich Nutzen 0 natürlich ohne jedes Element erzielen lässt, außerdem gilt $M_{0,V} = \infty$ für $V > 0$, da sich ohne Objekte kein positiver Nutzen erzielen lässt. Außerdem ist $M_{i,V} = \infty$ für $V < 0$ und jedes $i \in \{0, 1, \ldots, n\}$, weil sich negativer Nutzen gar nicht erzielen lässt.
Wir können mit den ersten i Objekten Gesamtnutzen V erzielen, indem wir entweder mit einer Auswahl der ersten $i-1$ Objekte Gesamtnutzen V erzielen und das i-te Objekt nicht benutzen oder das i-te Objekt benutzen und dann

mit einer Auswahl der ersten $i-1$ Objekte Gesamtnutzen $V - v_i$ erzielen. Darum ist

$$M_{i,V} = \min\{M_{i-1,V}, M_{i-1,V-v_i} + g_i\},$$

und wir können offenbar alle $M_{i,V}$ in Zeit $O(n \cdot V_{\max})$ bestimmen mit $i \in \{0, 1, \ldots, n\}$ und $V \in \{0, 1, \ldots, V_{\max}\}$. Das gilt für jeden Wert von $V_{\max} \in \mathbb{N}$; natürlich möchten wir $V_{\max}$ so klein wie möglich wählen, damit die Laufzeit möglichst klein wird, andererseits muss $V_{\max}$ groß genug gewählt sein, so dass wir nicht die optimale Lösung „übersehen". Weil natürlich nicht mehr als Nutzen $\sum_{i=1}^{n} v_i$ erzielbar ist, können wir $V_{\max} = \sum_{i=1}^{n} v_i \leq n \cdot \max\{v_1, v_2, \ldots, v_n\}$ wählen. Den Wert einer optimalen Bepackung erhalten wir als größtes V, so dass $M_{n,V} \leq G$ ist. Wir können nicht nur den Wert bestimmen, sondern auch die Bepackung selbst mit gleichem Aufwand berechnen, wenn wir zusätzlich zu jedem $M_{i,V}$ speichern, welches Element des Minimums den Wert bestimmt. □

Die Idee für einen Approximationsalgorithmus für das Rucksackproblem ist schlicht die Nutzung dieses pseudopolynomiellen Algorithmus. Der pseudopolynomielle Algorithmus hat ja polynomielle Laufzeit und berechnet sogar eine optimale Lösung, wenn alle Nutzenwerte polynomiell in der Länge der Eingabe sind. Wenn das nicht der Fall ist, können wir die Nutzenwerte alle durch k dividieren und abrunden; bei passender Wahl von k sind dann alle modifizierten Nutzenwerte polynomiell beschränkt und wir erhalten eine optimale Lösung für das modifizierte Problem in Polynomialzeit. Weil wir die Gewichtswerte nicht geändert haben, ist diese Lösung jedenfalls zulässig für die ursprüngliche Instanz. Das ist der Grund, aus dem der andere pseudopolynomielle Algorithmus, der eine Tabelle über die Gewichtswerte aufbaut, ungeeignet ist: Wenn wir die Gewichtswerte verändern, können zulässige Lösungen für das modifizierte Problem unzulässig für das ursprüngliche Problem werden und umgekehrt. Bei unserem Algorithmus kann das nicht passieren und wir müssen nur noch k passend wählen und nachrechnen, dass wir tatsächlich eine Lösung mit gewünschter Güte erhalten.

Theorem 8.2. *$KP \in \mathcal{FPTAS}$*

Beweis. Wir betrachten eine Instanz des Rucksackproblems mit Gewichtswerten $g_1, g_2, \ldots, g_n$, Nutzenwerten $v_1, v_2, \ldots, v_n$ sowie Gewichtsschranke G und wollen dazu in Polynomialzeit eine Lösung mit Güte $\leq 1 + \varepsilon$ berechnen, dabei ist $\varepsilon > 0$ eine Konstante.
Wir definieren

$$k := \frac{\varepsilon \cdot \max\{v_1, v_2, \ldots, v_n\}}{(1+\varepsilon) \cdot n}$$

und eine neue Instanz des Rucksackproblems mit Gewichtswerten $g_1, g_2, \ldots, g_n$, Nutzenwerten $\tilde{v}_1, \tilde{v}_2, \ldots, \tilde{v}_n$ sowie Gewichtsschranke G, dabei ist

$$\tilde{v}_i := \left\lfloor \frac{v_i}{k} \right\rfloor$$

für alle $i \in \{1, 2, \ldots, n\}$. Wir nennen diese neue Instanz auch modifizierte Instanz.
Wir beobachten zunächst, dass jede Bepackung $B \subseteq \{1, 2, \ldots, n\}$ genau dann zulässig für ursprüngliche Instanz ist, wenn sie zulässig für die modifizierte Instanz ist, weil Zulässigkeit nur anhand der Gewichtswerte entschieden wird, die für beide Instanzen identisch sind. Außerdem ist natürlich $k > 0$, weil $\max\{v_1, v_2, \ldots, v_n\} \geq 1$ gilt und $\varepsilon > 0$ ist.
Wir benutzen den pseudopolynomiellen Algorithmus aus dem Beweis von Theorem 8.1 und sehen, dass seine Laufzeit durch

$$\begin{aligned}
& O\left(n^2 \cdot \max\left\{\tilde{v}_1, \tilde{v}_2, \ldots, \tilde{v}_n\right\}\right) \\
= \; & O\left(n^2 \cdot \max\left\{\left\lfloor \frac{v_1}{k} \right\rfloor, \left\lfloor \frac{v_2}{k} \right\rfloor, \ldots, \left\lfloor \frac{v_n}{k} \right\rfloor\right\}\right) \\
= \; & O\left(n^2 \cdot \max\left\{\left\lfloor \frac{(1+\varepsilon) \cdot n \cdot v_1}{\varepsilon \cdot \max\{v_1, v_2, \ldots, v_n\}} \right\rfloor, \ldots, \left\lfloor \frac{(1+\varepsilon) \cdot n \cdot v_n}{\varepsilon \cdot \max\{v_1, v_2, \ldots, v_n\}} \right\rfloor\right\}\right) \\
= \; & O\left(n^2 \cdot \left\lfloor \frac{(1+\varepsilon) \cdot n}{\varepsilon} \right\rfloor\right) = O\left(\frac{n^3}{\varepsilon} + n^3\right)
\end{aligned}$$

nach oben beschränkt ist. Die Laufzeit ist also polynomiell in der Eingabelänge und $1/\varepsilon$.
Wir behaupten, dass die von diesem Algorithmus für das modifizierte Problem berechnete Bepackung B Güte $\leq 1 + \varepsilon$ für das ursprüngliche Problem hat. Dazu betrachten wir eine optimale Bepackung B' für das ursprüngliche Problem. Weil beide Bepackungen für beide Probleme zulässig sind, können wir beide Bepackungen für beide Probleme betrachten.
Es gilt

$$\sum_{i \in B} \tilde{v}_i \geq \sum_{i \in B'} \tilde{v}_i,$$

weil B eine optimale Bepackung für das modifizierte Problem ist. Die optimale Lösung für das ursprüngliche Problem B' ist zwar zulässig für das modifizierte Problem, kann aber nicht besser sein.
Weil $\tilde{v}_i = \lfloor v_i/k \rfloor$ gilt, folgt

$$\frac{v_i}{k} - 1 < \tilde{v}_i \leq \frac{v_i}{k}$$

für alle $i \in \{1, 2, \ldots, n\}$. Wir benutzen diese Beziehung, wenn wir jetzt abschätzen, welche Güte die Lösung B für das ursprüngliche Problem erreicht.

Dabei bezeichne OPT den Wert einer optimalen Lösung für das ursprüngliche Problem, also $\mathrm{OPT} = \sum_{i \in B'} v_i$. Wir haben

$$\begin{aligned}
\sum_{i \in B} v_i &\geq k \cdot \sum_{i \in B} \tilde{v}_i \geq k \cdot \sum_{i \in B'} \tilde{v}_i \\
&\geq k \cdot \sum_{i \in B'} \left(\frac{v_i}{k} - 1 \right) = \left(\sum_{i \in B'} v_i \right) - k \cdot |B'| \\
&= \mathrm{OPT} - k \cdot |B'| \geq \mathrm{OPT} - \frac{\varepsilon \cdot \max\{v_1, v_2, \ldots, v_n\}}{(1+\varepsilon) \cdot n} \cdot n \\
&= \mathrm{OPT} - \frac{\varepsilon}{1+\varepsilon} \cdot \max\{v_1, v_2, \ldots, v_n\} \geq \mathrm{OPT} - \frac{\varepsilon}{1+\varepsilon} \cdot \mathrm{OPT} \\
&= \mathrm{OPT} \cdot \left(1 - \frac{\varepsilon}{1+\varepsilon} \right) = \mathrm{OPT} \cdot \frac{1}{1+\varepsilon}
\end{aligned}$$

und sehen, dass

$$\frac{\mathrm{OPT}}{\sum_{i \in B} v_i} \leq 1 + \varepsilon$$

gilt und wir tatsächlich ein FPTAS angegeben haben. □

9 Das Traveling-Salesperson-Problem

Mit dem Rucksackproblem haben wir uns schon ein Problem angeschaut, das wir aus früheren Veranstaltungen kannten und ausgehend von diesem Vorwissen einen interessanten neuen Algorithmus konstruiert, ein echt polynomielles Approximationsschema in diesem Fall. Wir wollen jetzt als nächstes Problem das Traveling-Salesperson-Problem (TSP) betrachten, das ebenfalls zumindest aus GTI bzw. TIfAI bereits bekannt ist. Besonders interessant am TSP sind seine Varianten, die verschieden stark eingeschränkten Versionen, die uns die Konstruktion verschieden guter Approximationsalgorithmen erlauben.

Beim TSP besteht eine Probleminstanz, also eine gültige Eingabe, aus einer $n \times n$-Matrix $D = (d_{i,j})$ mit $d_{i,j} \in \mathbb{N}_0 \cup \{\infty\}$. Zulässige Lösung ist jede Permutation π von $\{1, 2, \ldots, n\}$, der Wert einer solchen Permutation ist $v(\pi) = \sum_{i=0}^{n-1} d_{\pi(i+1),\pi(((i+1) \bmod n)+1)}$ und wir möchten minimieren. Gesucht ist also eine einfacher Kreis in einem durch D definierten gewichteten gerichteten Graphen über n Knoten, der minimale Länge hat, wobei die Länge in den Kantengewichten gemessen wird.

Man kann sich leicht davon überzeugen, dass das TSP nicht approximierbar ist: Man kann eine Instanz des Hamiltonkreisproblems in eine TSP-Instanz „übersetzen“, bei der Kanten des ursprünglichen Graphen den Wert 1 und fehlende Kanten des ursprünglichen Graphen in der TSP-Instanz Kanten mit passend gewähltem sehr großen Gewicht werden. Eine c-Approximation für das TSP mit konstantem c lässt uns dann erkennen, ob mindestens eine von diesen „schweren“ Kanten gewählt werden muss, was uns sagt, ob der ursprüngliche Graph einen Hamiltonkreis enthielt.

Die Nichtapproximierbarkeit des TSP motiviert die Betrachtung von eingeschränkten Varianten. Wir interessieren uns hier für zwei Varianten. Beim *metrischen TSP* ist der Graph ungerichtet ($d_{i,j} = d_{j,i}$ für alle $i, j \in \{1, 2, \ldots, n\}$) und die Dreiecksungleichung ist erfüllt ($d_{i,k} \leq d_{i,j} + d_{j,k}$ für alle $i, j, k \in \{1, 2, \ldots, n\}$). Beim *euklidischen TSP* ist die Eingabe gegeben durch n Punkte in der Ebene, das Kantengewicht $d_{i,j}$ ist gegeben als euklidischer Abstand zwischen den Punkten i und j. Diese Variante entspricht nicht mehr unserer ursprünglichen Definition: Die Kantengewichte sind keine natürlichen Zahlen mehr, wir müssen reelle Zahlen zulassen. Das ist eine nicht ganz unproblematische Erweiterung, wir werden das dadurch entstehende Problem aber weitgehend ignorieren und davon ausgehen, dass wir weiterhin in konstanter Zeit die üblichen Operationen mit Kantengewichten

durchführen können. Wir tragen zu unserer Rechtfertigung vor, dass die Problematik hier weniger akut ist, weil wir uns zumindest auf rationale Zahlen zurückziehen könnten, wenn wir wollten, weil wir nur an Approximationen interessiert sind und nicht an exakten Lösungen: Bei gegebener Approximationsgüte $r = 1 + \varepsilon$ könnten wir alle Distanzen in einem Vorverarbeitungsschritt so runden, dass eine $(1 + \varepsilon/2)$-Approximation für die so modifizierte Instanz eine $(1+\varepsilon)$-Approximation für die ursprüngliche Instanz garantierte. Wir wollen das hier aber nicht vertiefen und etwas naiv einfach mit reellen Zahlen rechnen.

9.1 Das metrische TSP

Instanzen des metrischen TSP haben im Vergleich zum allgemeinen TSP die garantierte Eigenschaft, dass wir Kanten in beliebiger Richtung durchlaufen dürfen, ohne dass sich dadurch die Weglänge ändert, außerdem können wir in Algorithmen als Zwischenschritt ruhig „Rundreisen“ zulassen, die Knoten mehrfach besuchen: Wir können aus einer solchen „Reise“ eine Permutation gewinnen, indem wir der „Reise“ folgen und Knotenbesuche auslassen, wenn ein Knoten mehrfach besucht werden soll. Wenn also die Reise die Knoten i, j, k in dieser Reihenfolge besucht (also die Kantengewichte $d_{i,j}$ und $d_{j,k}$ zum Gesamtwert beitragen), der Knoten j aber schon vorher besucht wurde, so können wir vom Knoten i direkt zum Knoten k wechselt. Das entspricht dem Austauschen der Kanten (i, j) und (j, k) durch die Kante (i, k), so dass wir die Summe der Kantengewichte um $d_{i,j} + d_{j,k}$ verkleinern und gleichzeitig um $d_{i,k}$ vergrößern. Das funktioniert natürlich auch beim allgemeinen TSP, allerdings kann dort durch diese Änderung die Gesamtlänge kräftig wachsen. Beim metrischen TSP kann das nicht passieren, weil die Dreiecksungleichung $d_{i,j} + d_{j,k} \geq d_{i,k}$ garantiert, so dass die Gesamtlänge nicht größer werden kann und vielleicht sogar kleiner wird.

Es stellt sich die Frage, wie wir überhaupt einen Weg über alle Knoten finden, von dem wir glauben können, dass er kurz ist. Die Antwort finden wir, indem wir uns eine optimale Rundreise (also ein optimales π) vorstellen und darin Objekte finden, die wir besser berechnen können. Eine optimale Rundreise ist ein Kreis minimaler Länge über alle Knoten. Wenn wir eine beliebige Kante aus diesem Kreis entfernen, so haben wir einen Pfad im Graphen, der jeden Knoten berührt, also einen Spannbaum. Der Spannbaum ist insofern „entartet“, als jeder Knoten Grad 1 oder 2 hat. Wir erinnern uns daran, dass wir minimale Spannbäume (MST) effizient berechnen können, der Algorithmus von Prim erledigt das in Zeit $O(n^2)$. Wir könnten also damit beginnen,

einen minimalen Spannbaum zu berechnen. Die Gewichtssumme der Kanten dieses Spannbaums ist sicher nicht größer als die Gewichtssumme der Kanten des Spannbaums, den wir durch Entfernen einer Kante aus der optimalen Rundreise gewonnen haben. Wenn wir aus unserem minimalen Spannbaum eine Rundreise gewinnen könnten, ohne all zu viele Kanten mit all zu großem Gewicht hinzunehmen zu müssen, hätten wir sicher eine gute Approximation. Wir wollen uns überlegen, wie das sehr einfach geschehen kann, führen aber vorher den Begriff des Eulerkreises ein, der bei der formalen Beschreibung hilfreich sein wird. Außerdem brauchen wir noch den Begriff des Multigraphen. Den Begriff des Kreises lassen wir in der folgenden Definition offen. Wir werden je nach Kontext einen Kreis als eine geschlossene Kantenfolge oder eine geschlossene Knotenfolge beschreiben, ohne dass es dadurch zu Unklarheiten kommt.

Definition 9.1. *Ein ungerichteter* Multigraph $G = (V, E)$ *ist definiert durch eine endliche Knotenmenge* V *und eine endliche Multimenge über* $\{\{v, w \mid v \neq w \in V\}$.
Sei $G = (V, E)$ *ein ungerichteter Multigraph. Ein* Eulerkreis *ist ein Kreis in dem Graphen, der jede Kante genau einmal enthält.*

Ein Multigraph unterscheidet sich von einem gewöhnlichen Graphen darin, dass er Kanten mehrfach enthalten darf. Natürlich ist eine Menge eine spezielle Multimenge, so dass wir Graphen als Spezialfall von Multigraphen erkennen. Definition 9.1 bestimmt also den Begriff des Eulerkreises auch für gewöhnliche Graphen. Bevor wir uns ansehen, warum uns Multigraphen überhaupt interessieren, halten wir kurz fest, wann Multigraphen überhaupt Eulerkreise enthalten.

Lemma 9.2. *Ein zusammenhängender Multigraph* $G = (V, E)$ *enthält genau dann einen Eulerkreis, wenn alle Knoten geraden Grad haben.*

Beweis. Nehmen wir zunächst an, dass wir einen Eulerkreis in einem zusammenhängenden Multigraphen haben. Wir wählen einen Startknoten und durchlaufen den Eulerkreis einmal, dabei benutzen wir keine Kante mehrfach gemäß Definition des Eulerkreises. Wir bestimmen den Grad eines Knotens, indem wir zählen, wie oft wir ihn während dieses Durchlaufs betreten und verlassen. Weil jeder Knoten gleich oft betreten und verlassen wird, muss jeder Knoten geraden Grad haben.
Nehmen wir nun an, dass wir einen zusammenhängenden Multigraphen $G = (V, E)$ haben, in dem alle Knoten geraden Grad haben. Wir beweisen, dass G dann auch einen Eulerkreis enthält durch vollständige Induktion über die Anzahl der Kanten $m = |E|$. Für den Induktionsanfang ist $m = 2$, bei

$m = 1$ haben nicht alle Knoten geraden Grad, bei $m = 0$ ist der Graph entweder trivial oder unzusammenhängend. Weil alle Knoten geraden Grad haben, muss der Graph bis auf Umbenennung der Knoten $G = (V, E)$ mit $V = \{1, 2\}$ und $E = \{\{1, 2\}, \{1, 2\}\}$ entsprechen. Wir sehen, dass G den Eulerkreis $(1, 2, 1)$ enthält.
Für den Induktionsschluss betrachten wir den Graphen G und konstruieren darin einen einfachen Kreis. Dazu starten wir in einem beliebigen Knoten v und besuchen über eine noch nicht benutzte Kante einen noch nicht besuchten Knoten. Eine noch nicht benutzte Kante muss es geben, weil jeder Knoten geraden Grad hat. Wenn es keinen unbesuchten Knoten mehr gibt, wählen wir eine unbenutzte Kante zu einem besuchten Knoten und haben damit einen einfachen Kreis. Dieser Kreis muss allerdings nicht unbedingt den Knoten v enthalten, es kann einen mit v startenden einfachen Pfad geben, der zum Kreis gehört. Wir betrachten jetzt nur den Kreis, sei K die Menge seiner Kanten. Wir betrachten den Graphen $G' = (V, E \setminus K)$. Weil jede Knoten bezogen auf Kanten K entweder Grad 0 oder Grad 2 hat, hat jeder Knoten in G' geraden Grad. Allerdings muss G' nicht zusammenhängend sein. Für jede Zusammenhangskomponente von G' gilt aber die Induktionsvoraussetzung, weil wir ja mindestens zwei Kanten aus E entfernt haben. Wir haben also eine Kollektion von Eulerkreisen, in jeder Zusammenhangskomponente eine. Jeder dieser Eulerkreise wird von Kanten aus K berührt, darum erhalten wir einen einzigen Eulerkreis für G, indem wir in diese Eulerkreise K einfügen. □

Wir merken an, dass man Eulerkreise in Zeit $O(|V| + |E|)$ konstruieren kann, wollen uns hier aber nicht im Detail überlegen, wie das geht. Wir haben jetzt aber alles zusammen, um einen einfachen Algorithmus für das metrische TSP zu diskutieren.

Algorithmus 9.3 (**Spannbaumalgorithmus für das metrische TSP**).
1. *Berechnen einen MST $T = (V, E_T)$ auf G.*
2. *Erzeuge einen Multigraphen $G' = (V, E')$ mit $E' = \{e, e \mid e \in E_T\}$.*
3. *Berechne einen Eulerkreis K auf G'.*
4. *Berechne π, indem K durchlaufen und mehrfaches Vorkommen von Knoten entfernt wird.*
5. *Ausgabe π.*

Theorem 9.4. *Algorithmus 9.3 berechnet in Zeit $O(n^2)$ eine Lösung für das metrische TSP mit Güte ≤ 2.*

Beweis. Wir sehen, dass der Algorithmus jedenfalls eine Permutation π berechnet: Der einzige Punkt, über den man zweifeln kann, ist die Berechnung des Eulerkreises K. Die ist, wie wir aus Lemma 9.2 wissen, möglich, wenn

G' zusammenhängend ist und alle Knoten geraden Grad haben. Weil T ein Spannbaum ist, ist G' natürlich zusammenhängend. Weil jede Kante aus E_T doppelt in E' auftaucht, hat natürlich auch jeder Knoten geraden Grad.
Die Laufzeit wird dominiert durch die Berechnung des minimalen Spannbaums in Zeile 1. Wie erwähnt funktioniert das mit dem Algorithmus von Prim in Zeit $O(n^2)$. Die Konstruktion des Eulerkreises geht wie erwähnt in Zeit $O(n)$, alle anderen Schritte sind offenbar auch in Zeit $O(n)$ durchführbar.
Uns fehlt noch der Nachweis der Güte, die garantiert werden kann. Bezeichne OPT den Wert einer optimalen Lösung, bezeichne $v(T)$ die Kantensumme des minimalen Spannbaums, also $v(T) = \sum_{\{i,j\} \in E_T} d_{i,j}$. Wir hatten schon oben diskutiert, dass $v(T) \leq \text{OPT}$ gilt, weil jede Rundreise auch n verschiedene Spannbäume enthält, die alle nicht billiger sein können als der minimale Spannbaum T. Der Eulerkreis K enthält genau alle Kanten des Multigraphen G', die Summe der Kantengewichte aller im Eulerkreis K vorkommenden Kanten ist also gerade $2v(T)$. Durch das Entfernen von mehrfach vorkommenden Knoten können die Gesamtkosten nur kleiner werden, weil die Dreiecksungleichung gilt, wie wir uns auch schon überlegt hatten.
Wir haben also insgesamt

$$v(\pi) = \sum_{i=0}^{n-1} d_{\pi((i \bmod n)+1),\pi(((i+1) \bmod n)+1)} \leq 2v(T) \leq 2 \cdot \text{OPT}$$

und sehen, dass $v(\pi)/\text{OPT} \leq 2$ gilt. □

Eine ziemlich einfache 2-Approximation ist natürlich ein schönes Ergebnis, das immerhin metrisches TSP $\in \mathcal{APX}$ nachweist. Es stellt sich trotzdem die Frage, ob wir nicht noch bessere Ergebnisse erzielen können.
Schauen wir uns Algorithmus 9.3 noch einmal an. Der Spannbaum T hat Kosten, die nicht oberhalb der Kosten einer optimalen Tour liegen können. Wir haben an dieser Stelle also sicher noch nichts verloren. Im zweiten Schritt verdoppeln wir den Spannbaum, damit alle Knoten geraden Grad haben. Dieser Schritt bringt uns im Vergleich zur optimalen Tour einen Faktor 2 ein. Die Berechnung des Eulerkreises ändert an diesen Kosten nichts, der folgende Schritt kann die Tour kürzer machen, es ist aber nur schwer abschätzbar, was man an der Stelle garantieren kann. Wir sollten uns also überlegen, ob wir nicht auf die Verdopplung des Spannbaumes verzichten können. Sie ist ja an den Knoten, die in T ohnehin geraden Grad haben, völlig unnötig. Wir sollten uns also auf die Hinzunahme von Kanten beschränken für Knoten, die in T ungeraden Grad haben. Wenn wir dafür Kanten mit kleinem Gewicht wählen, können wir hoffentlich eine bessere Schranke für die Güte der berechneten Permutation nachweisen. Wie man das auf geschickte Weise

macht, sieht man im folgenden Algorithmus von Christofides, der zwischen den Knoten ungeraden Grades in T ein kostenminimales perfektes Matching berechnet. Ein Matching auf k Knoten heißt perfekt, wenn es $k/2$ Kanten umfasst. Als Kosten eines Matchings M bezeichnen wir die Summe der Kantengewichte $\sum_{\{i,j\}\in M} d_{i,j}$. Ein kostenminimales perfektes Matching ist also ein perfektes Matching, so dass kein anderes perfektes Matching kleinere Kosten hat.

Algorithmus 9.5 (**Algorithmus von Christofides**).
1. *Berechne einen MST $T = (V, E_T)$ auf G.*
2. *Berechne die Menge V' der Knoten mit ungeradem Grad in T.*
3. *Berechne ein kostenminimales perfektes Matching M auf den Knoten in V'.*
4. *Erzeuge den Multigraphen $G' = (V, E')$ mit $E' = E_T \cup M$.*
5. *Berechne einen Eulerkreis K auf G'.*
6. *Berechne π, indem K durchlaufen und mehrfaches Vorkommen von Knoten entfernt wird.*
7. *Ausgabe π*

Theorem 9.6. *Der Algorithmus von Christofides (Algorithmus 9.5) ist eine* $(3/2)$*-Approximation für das metrische TSP.*

Beweis. Wir bemerken zunächst, dass man kostenminimale perfekte Matchings auf n Knoten in Zeit $O(n^3)$ berechnen kann und dieser Schritt die Laufzeit des Algorithmus dominiert. Man kann also insgesamt mit Zeit $O(n^3)$ auskommen, so dass die Laufzeit jedenfalls polynomiell ist.
Kritisch für die Korrektheit sind die Berechnung des perfekten Matchings M und des Eulerkreises K. Für das Matching müssen wir zeigen, dass $|V'|$ gerade ist. Das ist aber leicht einzusehen. Wir betrachten dazu die Summe alle Grade der Knoten in T. Diese Summe ist natürlich gerade (wie bei allen Graphen), da jede Kante insgesamt 2 zu dieser Summe beiträgt. Die Summe der Grade aller Knoten mit geradem Grad ist natürlich ebenfalls gerade. Also muss, weil die Gesamtsumme gerade ist, die Summe der Grade aller Knoten mit ungeradem Grad ebenfalls gerade sein, das ist aber nur möglich, wenn wir gerade viele ungerade Knotengrade addieren. Also ist $|V'|$ wie behauptet gerade und wir können M berechnen. Über die Existenz der Kanten müssen wir uns keine Gedanken machen, der Graph ist ja vollständig. Für den Eulerkreis müssen wir uns überlegen, dass alle Knoten in G' geraden Grad haben. Das ist aber offensichtlich: Für die Knoten, die schon in T geraden Grad hatten, haben wir nichts geändert, für die Knoten, die in T ungeraden Grad haben, fügen wir in G' jeweils genau eine Kante hinzu, so dass ihr Grad gerade ist. Der Algorithmus ist also jedenfalls korrekt und berechnet eine Permutation π.

Wir sehen sofort, dass für die Kosten der Permutation π

$$v(\pi) \leq v(T) + v(M) \leq \mathrm{OPT} + v(M)$$

gilt, wenn v wie üblich die Summe der Gewichte der beteiligten Kanten bezeichnet. Wir müssen nachweisen, dass $v(\pi) \leq (3/2)\mathrm{OPT}$ gilt, es genügt also zu zeigen, dass $v(M) \leq \mathrm{OPT}/2$ gilt. Wir betrachten eine optimale Tour, die natürlich jeden Knoten des Graphen enthält, also auch die Knoten aus V'. Wir benennen die Knoten aus V' so, dass sie in der Reihenfolge $v_1, v_2, \ldots, v_{|V'|}$ in dieser Tour vorkommen. Nun betrachten wir den Kreis $C = v_1, v_2, \ldots, v_{|V'|}, v_1$. Die Summe der Kantengewichte ist durch OPT nach oben beschränkt, da der Kreis durch Entfernen von Knoten aus der optimalen Tour entsteht und durch solche „Abkürzungen“ das Gewicht nicht steigen kann, weil die Dreiecksungleichung gilt. Wir beobachten, dass dieser Kreis zwei disjunkte perfekte Matchings enthält, zum einen das Matching $M_1 = \{\{v_{2i+1}, v_{2i+2}\} \mid i \in \{0, 1, \ldots \lfloor |V'|/2 \rfloor - 1\}\}$, zum anderen das Matching M_2, das alle die Kanten aus C enthält, die nicht zu M_1 gehören. Wir betrachten die Summe der Kantengewichte in M_1 und M_2, also $v(M_1)$ und $v(M_2)$. Natürlich gilt $\min\{v(M_1), v(M_2)\} \leq \mathrm{OPT}/2$, weil $v(M_1) + v(M_2) \leq \mathrm{OPT}$ gilt: Die Kanten aus M_1 und M_2 ergeben zusammen genau C, die Kosten von C sind aber wie oben diskutiert durch OPT nach oben beschränkt. Beide Matchings, M_1 und M_2, sind perfekte Matchings auf den Knoten in V' und können aus diesem Grund nicht kleinere Kosten als M haben, da M ein kostenminimales perfektes Matching auf den Knoten in V' gilt. Also

$$v(M) \leq \min\{v(M_1), v(M_2)\} \leq \frac{\mathrm{OPT}}{2}$$

und $v(\pi)/\mathrm{OPT} \leq 3/2$ folgt. □

Es ist bekannt, dass das metrische TSP unter der Voraussetzung, dass P $\neq$ NP gilt, kein polynomielles Approximationsschema hat. Es ist aber offen, ob es noch bessere Approximationen konstanter Güte als den Algorithmus von Christofides gibt. Man könnte sogar spekulieren, dass die Abschätzung 3/2 für die Güte zu pessimistisch ist. Das ist aber nicht der Fall, es gibt Graphen, für die der Algorithmus von Christofides tatsächlich so schlechte Touren liefert. Wenn wir also an wesentlich besseren Approximationsalgorithmen interessiert sind, müssen wir uns noch weiter eingeschränkte Varianten des TSP ansehen.

9.2 Das euklidische TSP

Wir sehen uns jetzt das euklidische TSP in der Ebene an. Man kann die Ergebnisse dieses Abschnittes auch in höherdimensionale Räume verallgemeinern, darauf verzichten wir hier aber. Zunächst einmal ist klar, dass das euklidische TSP tatsächlich noch eingeschränkter ist, als das metrische TSP: Die euklidische Norm definiert eine Metrik, Entfernungen sind symmetrisch und es gilt die Dreiecksungleichung. Weil aber nur Kantengewichte vorkommen, die sich als euklidische Abstände von Punkten in der Ebene ergeben, sind wir noch eingeschränkter bei der Wahl der Distanzmatrix. Man darf also zumindest hoffen, dass man hier noch bessere Approximationsalgorithmen findet.

Wir werden hier tatsächlich ein polynomielles Approximationsschema für das euklidische TSP vorstellen. Der Algorithmus ist einigermaßen komplex, er kann grob in drei Schritten beschrieben werden. Im ersten Schritt wird die Eingabe, die ja aus n Punkten im $\mathbb{R}^2$ besteht, diskretisiert und normiert. Danach wird das Problem auf geschickte Weise geteilt, diese Teilung geschieht randomisiert und ist mit einer nicht zu kleinen Wahrscheinlichkeit erfolgreich, was bedeutet, dass sie zu einer Teilung mit günstigen Eigenschaften führt. Wenn dies der Fall ist, dann wird im dritten und letzten Schritt eine Tour nach dem Prinzip der dynamischen Programmierung berechnet, wobei die Teilprobleme durch die Teilung des vorangegangenen Schritts definiert sind. Wir haben jetzt also eine Art randomisiertes Approximationsschema mit Fehlschlagswahrscheinlichkeit, also etwas, wovon wir gar nicht klar definiert haben, was wir erwarten. Wir werden das hier auch nicht vertiefen, stattdessen überlegen wir uns zum Schluss, wie wir ausgehend von diesem randomisierten Algorithmus ein deterministisches polynomielles Approximationsschema für das euklidische TSP erhalten.

Seien $\varepsilon > 0$ und eine Instanz des euklidischen TSP im $\mathbb{R}^2$ gegeben, zu der wir eine $(1+\varepsilon)$-Approximation berechnen wollen. Sei OPT die Länge einer optimalen Tour für diese Instanz, wir wollen also eine Tour mit Länge $\leq (1+\varepsilon)\mathrm{OPT}$ berechnen.

Wir betrachten das kleinste achsenparallele Quadrat, das alle Punkte der Eingabe einschließt. Sei die Seitenlänge dieses Quadrates L. Wir beobachten schon an dieser Stelle, dass L natürlich eine untere Schranke für die Länge einer optimalen Tour ist, weil es mindestens zwei Punkte gibt, die Abstand mindestens L haben. Natürlich können wir dieses Quadrat gedanklich so verschieben, dass seine linke untere Ecke der Ursprung des Koordinatensystems ist. Das ändert offensichtlich nichts wesentlich. Wir legen nun gedanklich über dieses Quadrat ein quadratisches Gitter mit Abstand $(\varepsilon L)/(8n)$ zwischen zwei

Gitterlinien. Jetzt verschieben wir jeden Punkt der ursprünglichen Eingabe so, dass er auf den nächstgelegenen Gitterpunkt zu liegen kommt. Wenn dabei Punkte deckungsgleich werden, betrachten wir sie zukünftig als einen Punkt. Für jeden Punkt können wir durch diese Verschiebung um weniger als $(\varepsilon L)/(8n)$ die Länge einer optimalen Tour im Vergleich zur ursprünglichen Eingabe offensichtlich um nicht mehr als $2(\varepsilon L)/(8n)$ verlängern. Darum weichen wir insgesamt von der Länge einer optimalen Tour um nicht mehr als $n \cdot 2(\varepsilon L)/(8n) = \varepsilon L/4 \leq (\varepsilon/4) \cdot \mathrm{OPT}$ ab. Wenn wir also nun für diese modifizierte Instanz eine $(1 + (3/4)\varepsilon)$-Approximation berechnen, haben wir für die ursprüngliche Instanz wie gewünscht eine $(1 + \varepsilon)$-Approximation berechnet.
Nach der Verschiebung haben alle Punkte Koordinaten $(i \cdot (\varepsilon L)/(8n), j \cdot (\varepsilon L)/(8n))$ mit $i, j \in \mathbb{N}_0$. Wir dividieren nun alle Distanzen durch $(\varepsilon L)/(64n)$. Danach sind alle Koordinaten ganzzahlig und zwei Punkte haben einen Mindestabstand von 8. Außerdem ist $L/((\varepsilon L)/(64n)) = (64/\varepsilon)n = O(n)$, die Seitenlänge des umschließenden Quadrates ist also $O(n)$. Nach dieser Transformation können wir das Gitter, das wir für die Transformation benutzt haben, vergessen.
Im nächsten Schritt wollen wir uns um die Zerlegung in Teilprobleme kümmern. Dazu schauen wir uns zunächst eine *Zergliederung* des Quadrats mit Seitenlänge L an. Wir nehmen aus Gründen der Bequemlichkeit an, dass L eine Zweierpotenz ist. Wir betrachten das ganze Quadrat und teilen es in vier gleich große Quadrate auf. Wir stellen uns dabei das ganze Quadrat als Wurzel eines 4-ären Baumes vor, die als Kinder die vier kleineren Quadrate hat. Wir setzen diese Unterteilung rekursiv fort, bis wir bei Quadraten der Größe 1 angekommen sind. Kein Quadrat kann mehr als einen Punkt unserer TSP-Instanz enthalten, weil der Mindestabstand zwischen zwei Punkten wie oben diskutiert 8 beträgt. Der entstandene Baum hat offenbar Tiefe $O(\log L)$ und Größe $O(L^2)$. Natürlich sind viele Quadrate in dieser Zergliederung leer. Wir betrachten darum lieber eine adaptive Zergliederung durch einen *Quadtree*. Wir konstruieren den Quadtree genau wie die Zergliederung, allerdings stoppen wir die rekursive Aufteilung, wenn ein Quadrat weniger als zwei Punkte unserer TSP-Eingabe enthält. Natürlich hat auch der Quadtree Tiefe $O(\log L)$, er hat aber nur noch $O(n)$ Blätter, so dass seine Größe durch $O(n \log L)$ beschränkt ist.
Wir möchten die Aufteilung des Quadrats durch den Quadtree als Strukturierung für die dynamische Programmierung benutzen. Wir werden dazu das Problem weiter strukturieren, indem wir keine beliebigen Verbindungen zwischen Punkten mehr zulassen, sondern für jedes Quadrat sogenannte *Portale* definieren und nur noch Verbindungen über diese Portale erlauben. Leider könnten die Punkte unserer TSP-Instanz so ungünstig verteilt sein, dass wir keine ausreichend gute Approximation finden. Wir können uns aber gegen

solche Worst-Case-Instanzen wappnen, indem wir nicht einen Quadtree wie eben beschrieben konstruieren, sondern die konkrete Anordnung der Quadrate randomisieren. Natürlich kann dann immer noch der gleiche Worst Case eintreten, wir werden aber zeigen, dass das mit nicht zu großer Wahrscheinlichkeit passieren wird, wobei die Wahrscheinlichkeit über unsere zufällige Anordnung der Quadtree-Quadrate berechnet wird; wir machen also keine Annahme über die Eingabe, wie das bei Average-Case-Analysen geschieht.

Für die konkrete Definition des Zufallsexperiments kehren wir zunächst zur Zergliederung zurück. Wir wählen zwei Zahlen $a, b \in \{0, 1, \ldots, L-1\}$ gemäß Gleichverteilung zufällig. Die (a, b)-*Verschiebung* der oben beschriebenen Zergliederung entsteht, indem wir die x-Koordinaten aller Linien um a und die y-Koordinaten um b verschieben, dabei wird diese Verschiebung mod L durchgeführt, so dass einige Quadrate in bis zu vier Teile geteilt werden, die natürlich keine Quadrate mehr sein müssen. Wir werden in der Beschreibung später diese Teilung berücksichtigen, sprechen bei einem derart geteilten Quadrat aber weiterhin von *einem* Quadrat. Wir verschieben natürlich nur die Quadrate und nicht die Punkte unserer TSP-Instanz; andernfalls ergäbe sich ja keine wesentliche Änderung. Die (a, b)-Verschiebung des Quadtrees erhalten wir aus der (a, b)-Verschiebung der Zergliederung auf die gleiche Art, wie wir den Quadtree aus der Zergliederung erhalten haben.

Mit dieser Strukturierung des Problems im Hintergrund können wir nun einige Begriffe definieren, die genauer festlegen, wie wir vorgehen wollen. Zum einen werden wir die schon oben erwähnten Portale genau beschreiben. Zum anderen werden wir den Begriff des TSP-Pfades einführen, der sich von der TSP-Tour darin unterscheidet, dass er nicht nur direkte Verbindungen zwischen Punkten unserer TSP-Instanz wählt; das liegt daran, dass er Quadrate nur durch Portale betritt und verlässt, was zu Umwegen führen kann. Schließlich werden wir formal fassen, wann ein TSP-Pfad für unsere Approximation günstige Eigenschaften hat. Wir nehmen im Folgenden an, dass kein Punkt unserer TSP-Instanz auf einer Kante liegt, andernfalls skalieren wir alles mit dem Faktor 2, so dass Kanten gerade Koordinaten bekommen und Punkte ungerade.

Definition 9.7. *Betrachte eine wie beschrieben modifizierte euklidische TSP-Instanz mit Eingabepunkten $x_1, x_2, \ldots, x_n$, einen (a, b)-verschobenen Quadtree dazu und zwei Zahlen $m, r \in \mathbb{N}$.*

Eine m-regülare Menge von Portalen *ist eine Menge von Punkten, davon liegt auf jeder Ecke eines Quadrats jeweils einer und m weitere Punkte liegen auf jeder Kante eines Quadrats äquidistant verteilt.*

Ein TSP-Pfad *besucht jeden Eingabepunkt $x_1, x_2, \ldots, x_n$ und kreuzt Quadrate ausschließlich an Portalen, dabei dürfen Portale auch mehrfach benutzt werden.*
Ein TSP-Pfad heißt (m, r)-leicht *für den (a, b)-verschobenen Quadtree, wenn jede Kante jedes Quadrats höchstens r-mal gekreuzt wird.*

Natürlich kann der kürzeste Weg von x_i zu x_j über ein Portal länger sein als die direkte Verbindung von x_i und x_j. Wir sagen dann, dass der Weg am Portal gebogen ist. Wir berechnen eine TSP-Tour mit solchen gebogenen Wegen, vor der eigentlichen Ausgabe werden dann aber natürlich alle Wege begradigt, indem die Portale aus der Tour entfernt werden und zwei Punkte unserer TSP-Instanz jeweils direkt verbunden werden. Offensichtlich kann das die Tour nicht länger machen.
Dass der Aufwand der Modifikation der ursprünglichen Instanz und die Berechnung der zufälligen (a, b)-Verschiebung sich lohnt, zeigt das folgende Theorem, das für die Korrektheit des Algorithmus zentral ist. Wir stellen es zunächst nur vor und besprechen seine Anwendung, den Beweis heben wir uns für später auf.

Theorem 9.8 (Strukturtheorem)**.** *Sei $\varepsilon > 0$ konstant, sei eine euklidische TSP-Instanz $x_1, x_2, \ldots, x_n$ so gegeben, dass alle Punkte ganzzahlige Koordinaten haben und der kleinste Abstand zwischen zwei Punkten 8 ist, sei L die Seitenlänge des kleinsten einschließenden Quadrats, sei OPT die Länge einer optimalen Tour zu dieser Instanz, seien $a, b \in \{0, 1, \ldots, L-1\}$ zufällig gemäß Gleichverteilung gewählt.*
Es gibt $m = O(\varepsilon^{-1} \log L)$ und $r = O(\varepsilon^{-1})$, so dass es mit Wahrscheinlichkeit mindestens $1/2$ einen für die (a, b)-Verschiebung (m, r)-leichten TSP-Pfad mit Länge $\leq (1 + \varepsilon)OPT$ gibt.

Damit ist der Algorithmus, den wir verwenden wollen, klar: Wir beginnen mit der Vorverarbeitung der Instanz wie beschrieben. Dann berechnen wir einen zufällig (a, b)-verschobenen Quadtree, anschließend wählen wir m und r passend und berechnen mit dynamischer Programmierung einen optimalen (m, r)-leichten TSP-Pfad. Dank des Strukturtheorems (Theorem 9.8) wissen wir, dass wir dann mit Wahrscheinlichkeit mindestens $1/2$ eine $(1 + \varepsilon)$-Approximation einer optimalen Tour berechnet haben.
Die dynamische Programmierung benutzt die Quadrate des (a, b)-verschobenen Quadtrees als Teilprobleme. Jedes Teilproblem entspricht also einem Knoten des Quadtrees, die Wurzel repräsentiert das Gesamtproblem. Wir wollen die Probleme bottom-up lösen, betrachten die Situation aber jetzt zunächst top-down, um besser zu verstehen, wie wir die Tour konstruieren wollen. Wir betrachten also einen Knoten des Quadtrees, ein Quadrat Q, das

einige (mehr als einen, sonst sind wir an einem Blatt, das wollen wir später besprechen) Punkte unserer TSP-Instanz betrachtet. Wir interessieren uns für einen optimalen (m, r)-leichten TSP-Pfad in Q. Dieser Pfad kreuzt die vier Kanten von Q insgesamt $2p$-mal (ungerade Anzahlen sind offensichtlich nicht möglich, wir konstruieren ja einen Kreis), dabei ist $2p \leq 4r$, sonst wäre der Pfad nicht (m, r)-leicht. Die Portale seien so benannt, dass der Pfad in dieser Reihenfolge die Portale $a_1, a_2, \ldots, a_{2p}$ benutzt. Der Pfad P zerfällt so auf natürliche Weise in p Teilpfade, wobei der i-te Teilpfad die Portale a_{2i-1} und a_{2i} miteinander verbindet. Diese p Teilpfade müssen gemeinsam (m, r)-leicht sein und alle Punkte in Q besuchen. Wir erhalten einen optimalen Pfad P, indem wir unter allen Teilpfaden mit diesen Eigenschaften p optimale Teilpfade aussuchen. Damit ist der Ansatz für die dynamische Programmierung gefunden.

Wir haben also ein nichtleeres Quadrat Q im (a, b)-verschobenen Quadtree, für jede Kante von Q eine Multimenge (Portale dürfen mehrfach benutzt werden) von $\leq r$ Portalen, insgesamt $2p \leq 4r$ Portale und eine Paarung der Portale (a_1, a_2), (a_3, a_4), $\ldots$, (a_{2p-1}, a_{2p}). Dazu wollen wir eine kostenminimale Folge von p Pfaden berechnen, wobei der i-te Pfad die Portale a_{2i-i} und a_{2i} verbindet, die Folge insgesamt (m, r)-leicht ist und zusammen alle Punkte aus Q besucht. Wie entscheiden wir denn, welche Portale und welche Paarungen die richtigen sind? Kann man das vorher wissen? Weil wir das nicht wissen, werden wir zur denkbar einfachsten (aber aufwendigen) Lösung greifen und alle Möglichkeiten ausprobieren. Es gibt in Q insgesamt $m + 4$ Portale, es gibt also höchstens $(m+5)^{4r}$ Möglichkeiten, eine Multimenge der Größe $\leq 4r$ auszuwählen. Dann müssen wir die $\leq 4r$ Portale paarweise zuordnen, dazu gibt es höchstens $(4r)!$ Möglichkeiten. Es gibt also in einem Quadrat Q höchstens $(m + 5)^{4r} \cdot (4r)!$ Teilprobleme, die wir lösen müssen. Machen wir uns jetzt also an die Lösung eines Teilproblems.

Wir haben $\leq 4r$ gepaarte Portale festgelegt und sollen die Pfade bestimmen. Wenn unser Quadrat ein Blatt ist, kann es nur einen oder gar keinen Punkt unserer TSP-Instanz enthalten. Wenn es keinen Punkt enthält, gibt es nur eine Lösung, wenn es einen Punkt enthält, kann dieser Punkt in jedem der $\leq 2r$ Pfade untergebracht werden. Wir müssen in diesem Fall also nur $\leq 2r$ Möglichkeiten ausprobieren, können das Problem an einem Blatt also in jedem Fall optimal in Zeit $O(r)$ lösen. Ist das Quadrat Q kein Blatt, so hat es gemäß Konstruktion genau vier Kinder, seien das die Quadrate Q_1, Q_2, Q_3 und Q_4. Wir benutzen dynamische Programmierung, für diese Quadrate sind alle möglichen Teilprobleme schon gelöst. Wir gehen jetzt alle Möglichkeiten durch, wie ein (m, r)-leichter Pfad die Kanten der vier Quadrate Q_1, Q_2, Q_3 und Q_4 kreuzen kann. Wir haben dabei $(m + 5)^{4r}$ Möglichkeiten, die Portale festzulegen, $(4r)^{4r}$ Möglichkeiten, diese Portale den Pfaden zuzuordnen und

$(4r)!$ Möglichkeiten, die Portale zu paaren. Für jedes Teilproblem lesen wir die optimalen Kosten ab und bekommen unsere Kosten als Summe der vier Teilkosten. Die minimalen Kosten sind die Lösung unseres Problems.
Wir haben jetzt in jedem Knoten des Baumes einen Rechenaufwand, der durch

$$O\big((m+5)^{8r} \cdot (4r)^{4r} \cdot ((4r)!)^2\big)$$

nach oben beschränkt ist. Wir erinnern uns daran, dass uns das Strukturtheorem (Theorem 9.8) $r = O(1/\varepsilon)$ und $m = O((\log L)/\varepsilon) = O((\log n)/\varepsilon)$ verspricht. Wir setzen das ein und erhalten

$$O\big((m+5)^{8r} \cdot (4r)^{4r} \cdot ((4r)!)^2\big) = O\Big((\log n)^{O\left(\varepsilon^{-1}\right)}\Big)$$

als obere Schranke des Rechenaufwandes je Quadrat des Quadtrees. Der Quadtree hat Tiefe $O(\log L) = O(\log n)$ und $O(n)$ Blätter, so dass wir insgesamt $O(n \log n)$ Knoten im Quadtree haben. Damit hat dieser randomisierte Algorithmus eine Gesamtrechenzeit von $O\Big(n\,(\log n)^{O\left(\varepsilon^{-1}\right)}\Big)$ und eine Erfolgswahrscheinlichkeit von mindestens $1/2$. Im Misserfolgsfall bekommen wir auch eine Tour, sie ist allerdings keine $(1+\varepsilon)$-Approximation. Leider kann man einer Ausgabe nicht ansehen, ob sie eine $(1+\varepsilon)$-Approximation ist, und der Algorithmus liefert dazu auch keine Hinweise. Natürlich kann man den Algorithmus k-mal laufen lassen und die beste Lösung ausgeben, dann hat man mit Wahrscheinlichkeit mindestens $1 - 2^{-k}$ eine $(1+\varepsilon)$-Approximation berechnet. Wir werden uns zum Abschluss noch überlegen, wie wir aus diesem randomisierten Algorithmus auch einen deterministischen Algorithmus gewinnen können, der garantiert eine $(1+\varepsilon)$-Approximation berechnet; dieser Algorithmus wird allerdings erheblich langsamer sein.
Es gibt noch zwei konkrete Probleme: Wir wissen nicht, welche Werte für m und r geeignet sind, brauchen aber natürlich konkrete Zahlen für unseren Algorithmus. Außerdem hängt die Approximationsgüte von der Korrektheit des Strukturtheorems (Theorem 9.8) ab, das wir bisher noch nicht bewiesen haben. Darum werden wir uns jetzt kümmern, im Laufe des leider recht aufwendigen Beweises werden wir dann auch geeignete Werte von m und r bestimmen können.
Im Zentrum des Strukturtheorems (Theorem 9.8) stehen (m, r)-leichte Pfade, dabei steckt in m eine Aussage über die Anzahl der Portale, die zur Verfügung stehen und r ist eine obere Schranke für die Anzahl der Schnitte des Pfades mit einer einzelnen Quadratseite. Wir wollen eine $(1+\varepsilon)$-Approximation schaffen mit solchen (m, r)-leichten Pfaden; natürlich ist das umso einfacher, je größer m und r sind, da m und r unsere Freiheiten bei der Pfadwahl beschränken. Wir wollen aber zeigen, dass wir r sogar unabhängig von n (in

diesem Sinn also „konstant“) wählen können. Dazu zeigen wir jetzt in einem ersten Schritt, dass wir einen TSP-Pfad, der eine einzelne Quadratseite oft schneidet, in einen anderen TSP-Pfad transformieren können, der weniger Schnitte hat und dabei nur beschränkt länger wird. Wir erreichen das durch geschicktes Zerschneiden und Zusammenkleben des Pfades, darum nennen wir die Aussage Patchinglemma. Wir abstrahieren in dieser Aussage etwas von unserem konkreten Problem und sprechen von einem Liniensegment, dass von einem geschlossenen Pfad geschnitten wird.

Lemma 9.9 (**Patchinglemma**). *Sei S ein Liniensegment der Länge s, sei P ein geschlossener Pfad, der S mindestens dreimal schneidet.*
Es gibt einen geschlossenen Pfad P', der S höchstens zweimal schneidet und als Erweiterung von P durch einige Linienabschnitte von S beschrieben werden kann, wobei die Gesamtlänge der Linienabschnitte von S durch $2s$ nach oben beschränkt ist.

Beweis. Wir betrachten den geschlossenen Pfad P und seine Schnittpunkte mit S, seien das M_1, M_2, …, M_t, dabei ist $t \geq 3$. Der Pfad P zerfällt an diesen Schnittpunkten in Pfadteile, wir nennen die Pfadteile P_1, P_2, …, P_t. Wir können die Situation modellieren als einen Graphen $G = (V, E)$ mit $V = \{M_1', M_1'', M_2', M_2'', \ldots, M_t', M_t''\}$, dabei stehen M_i' und M_i'' für den Schnittpunkt M_i, wir stellen uns M_i' auf der einen Seite des Liniensegments S und M_i'' auf der anderen Seite von S vor. Wir stellen uns vor, dass S senkrecht von oben nach unten verläuft und nennen die Seite, auf der sich alle Knoten M_i' befinden, links, entsprechend heißt die Seite, auf der sich alle Knoten M_i'' befinden, rechts. Wir verbinden diese Knotenpaare mit Kanten, es ist also $\{M_i', M_i''\} \in E$ für alle $i \in \{1, 2, \ldots, t\}$. Außerdem gibt es eine Kante $\{M_i', M_j'\}$, wenn es einen Pfadteil zwischen M_i und M_j gibt, der vollständig links von S verläuft, entsprechend gibt es eine Kante $\{M_i'', M_j''\}$, wenn es einen Pfadteil zwischen $\{M_i, M_j\}$ gibt, der vollständig rechts von S verläuft. Schließlich gibt es eine Kante $\{M_i', M_j''\}$, wenn es einen Pfadteil von M_i zu M_j gibt, der M_i von links und M_j von rechts erreicht; dieser Pfadteil hat keinen weiteren Schnittpunkt mit S, andernfalls zerfiele er ja in zwei Pfadteile, er führt also außen um S herum. Weitere Kanten enthält E nicht.
Weil P ein geschlossener Pfad ist, ist der Graph G zusammenhängend. Außerdem beobachten wir, dass G das Liniensegment S genau t-mal schneidet und einen Eulerkreis enthält. Wir werden jetzt G so verändern, dass wir nur Kanten $\{M_i', M_i''\}$ entfernen, nur Kanten $\{M_i', M_{i+1}'\}$ und $\{M_i'', M_{i+1}''\}$ einfügen, der modifizierte Graph G' zusammenhängend bleibt und weiterhin einen Eulerkreis enthält. Damit ist klar, dass G' weiterhin ein geschlossener Pfad ist, er alle Pfadteile aus P enthält und zusätzlich nur Linienabschnitte von S enthält. Wir überzeugen uns dann noch davon, dass G' höchstens zweimal S

schneidet und die eingefügten Linienabschnitte zusammen höchstens Länge s haben. Dann ist der Beweis geführt.
Wir erzeugen G' aus G wie folgt. Für alle $i \in \{1, 3, 5, \ldots, 2\lceil t/2 \rceil - 3\}$ entfernen wir die Kanten $\{M'_i, M''_i\}$ und $\{M'_{i+1}, M''_{i+1}\}$, außerdem fügen wir die Kanten $\{M'_i, M'_{i+1}\}$, $\{M''_i, M''_{i+1}\}$ ein. Dadurch kann der Graph in zwei Zusammenhangskomponenten zerfallen. Falls das passiert, stellen wir den Zusammenhang wieder her, indem wir entweder die Kanten $\{M'_{i+1}, M'_{i+2}\}$ und $\{M'_{i+1}, M'_{i+2}\}$ oder die Kanten $\{M''_{i+1}, M''_{i+2}\}$ und $\{M''_{i+1}, M''_{i+2}\}$ einfügen, wir fügen also in dem Fall eine Kante doppelt ein und erzeugen damit einen Multigraphen.
Wir beobachten zunächst, dass offensichtlich die Anzahl der Schnittpunkte auf ≤ 2 fällt: Für jedes ungerade $i \leq 2\lceil t/2 \rceil - 3$ entfernen wir die Kanten $\{M'_i, M''_i\}$ und $\{M'_{i+1}, M''_{i+1}\}$, die Schnittpunkte symbolisieren. Es bleiben also höchstens die Schnittpunkte M_{t-1} und M_t übrig.
Wir beobachten außerdem, dass alle Knoten geraden Graph behalten: Die Knoten M'_i, M''_i, M'_{i+1} und M''_{i+1} verlieren alle jeweils eine Kante und erhalten alle jeweils eine Kante hinzu, was ihren Grad unverändert lässt. Das zweifache Einfügen einer Kante vergrößert den Grad der beteiligten Knoten um 2, so dass alle Knotengrade gerade bleiben.
Schließlich beobachten wir noch, dass der Graph zusammenhängend bleibt: Das Entfernen einer einzelnen Kante aus dem Graphen kann den Zusammenhang nicht zerstören, da der Graph einen Eulerkreis enthält. Wir müssen also nur noch das Entfernen der Kante $\{M'_{i+1}, M''_{i+1}\}$ betrachten. Wenn aus einem zusammenhängenden Graphen eine Kante entfernt wird, so können dabei offensichtlich höchstens zwei Zusammenhangskomponenten entstehen. Wir nennen die Zusammenhangskomponente, zu der ein Knoten v gehört, $Z(v)$. Wenn $Z(M'_{i+1}) = Z(M''_{i+1})$ gilt, ist der Graph nach Entfernen beider Kanten noch zusammenhängend und es ist nichts zu zeigen. Nehmen wir also an, dass $Z(M'_{i+1}) \neq Z(M''_{i+1})$ gilt. Wir betrachten jetzt die Zusammenhangskomponenten $Z(M'_{i+2})$ und $Z(M''_{i+2})$. Weil $i \leq t + 2$ gilt, gibt es diese Zusammenhangskomponenten jedenfalls. Nehmen wir zunächst an, dass $Z(M'_{i+1}) = Z(M'_{i+2})$ und $Z(M''_{i+1}) = Z(M''_{i+2})$ gilt. Es gibt die Kante $\{M'_{i+2}, M''_{i+2}\}$, so dass wir von M'_{i+1} über M'_{i+2} und M''_{i+2} schließlich zu M''_{i+1} kommen. Das ist ein Widerspruch zur Annahme, dass $Z(M'_{i+1}) \neq Z(M''_{i+1})$ gilt. Wir haben also entweder $Z(M'_{i+1}) \neq Z(M'_{i+2})$ oder $Z(M''_{i+1}) \neq Z(M''_{i+2})$. Durch das Einfügen der Doppelkante verringern wir also die Anzahl der Zusammenhangskomponenten um 1. Weil die Anzahl der Zusammenhangskomponenten vorher 2 betrug, ist der Graph also tatsächlich weiterhin zusammenhängend.
Wir müssen uns nur noch davon überzeugen, dass die Länge aller eingefügten Linienabschnitte von S zusammen $2s$ nicht übersteigt. Für jeden betrachte-

ten Wert von i fügen wir zwei oder vier Kanten ein: Wir fügen auf jeden Fall die Kanten $\{M'_i, M'_{i+1}\}$ und $\{M''_i, M''_{i+1}\}$ ein, das entspricht dem zweifachen Einfügen des Linienabschnitts zwischen M_i und M_{i+1}. Außerdem fügen wir eventuell entweder die Kante $\{M'_{i+1}, M'_{i+2}\}$ oder die Kante $\{M'_{i+1}, M'_{i+2}\}$ doppelt ein, das entspricht dem zweifachen Einfügen des Linienabschnitts zwischen M_{i+1} und M_{i+2}. Weil i von Runde zu Runde in Zweierschritten wächst, wird also jeder Linienabschnitt höchstens zweimal eingefügt, so dass die Gesamtlänge der eingefügten Linienabschnitte durch $2s$ nach oben beschränkt ist. □

Wir werden beim Beweis des Strukturtheorems (Theorem 9.8) amortisierte Analyse mit Kostenverteilung verwenden und dabei Kosten auf Gitterlinien umverteilen und zwar anteilig für jedes benutzte Portal. Darum ist es hilfreich, eine Aussage über die Anzahl der Schnittpunkte zu haben, die eine optimale Tour π mit den Gitterlinien haben kann. Eine solche Aussage werden wir jetzt in Form eines technischen Lemmas vorab festhalten und beweisen.

Lemma 9.10. *Sei eine TSP-Instanz so gegeben, dass alle Punkte ganzzahlige Koordinaten haben und der minimale Abstand zwischen zwei Punkten vier beträgt. Sei π eine optimale Tour für diese TSP-Instanz mit Länge OPT, sei g eine Gitterlinie des Einheitsgitters, gebe $\tau(\pi, g)$ die Anzahl der Schnittpunkte zwischen π und g an.*

$$\sum_{\text{Gitterlinie } g} \tau(\pi, l) \leq 2OPT$$

Beweis. Wir betrachten eine Kante e aus der optimalen Tour π, sei s die Länge von e. Wir betrachten die Projektionen von e auf die x- und y-Achse, seien die Länge dieser Projektionen u und v. Es gilt nun $s^2 = u^2 + v^2$, weil e und die beiden Projektionen natürlich ein rechtwinkliges Dreieck bilden. Die Anzahl der Schnitte von e mit vertikalen und horizontalen Gitterlinien ist natürlich durch $(u + 1) + (v + 1) = u + v + 2$ nach oben beschränkt. Wir haben $(u - v)^2 \geq 0$, so dass $u^2 + v^2 \geq 2uv$ folgt. Wir haben damit $2(u^2 + v^2) \geq (u + v)^2$ und können $u + v \leq \sqrt{2(u^2 + v^2)}$ folgern. Damit haben wir jetzt gezeigt, dass die Anzahl der Schnitte von e mit vertikalen und horizontalen Gitterlinien durch $\sqrt{2s^2} + 2 = s \cdot \sqrt{2} + 2$ nach oben beschränkt ist. Wir haben $2s \geq s \cdot \sqrt{2} + 2$ für $s \geq 2/\left(2 - \sqrt{2}\right)$, so dass $s > 3{,}42$ sicher ausreicht. Weil wir in den Voraussetzungen haben, dass der Minimalabstand zweier Punkte 4 ist, können wir jetzt also für jede Kante der optimalen Tour die Anzahl der Schnittpunkte mit horizontalen und vertikalen Gitterlinien durch das Doppelte ihrer Länge nach oben abschätzen. Summation über alle Kanten der Tour führt uns direkt zur Aussage das Lemmas. □

Wir haben jetzt alle Aussagen zusammen, um das Strukturtheorem (Theorem 9.8) zu beweisen. Wir werden dabei über Wahrscheinlichkeiten und Erwartungswerte sprechen, dabei ist der Wahrscheinlichkeitsraum durch die zufällige Wahl der (a, b)-Verschiebung definiert. Die Beweisstrategie ist recht naheliegend: Wir gehen von einer optimalen TSP-Tour π aus und beschreiben, wie wir aus π einen (m, r)-leichten TSP-Pfad konstruieren. Dabei werden wir die Größe der Längenzunahme durch amortisierte Analyse beschränken, indem wir die Mehrkosten auf Gitterlinien, in denen wir Portale benutzen, verteilen.

Beweis des Strukturtheorems (Theorem 9.8). Wir nehmen an, dass die Seitenlänge des kleinsten einschließenden Quadrats L eine Zweierpotenz ist, also $L = 2^l$ gilt mit $l \in \mathbb{N}$, andernfalls müssen wir das Quadrat höchstens verdoppeln. Wir betrachten die zufällige (a, b)-Verschiebung des Quadtrees und betrachten dazu die entsprechend verschobene Zergliederung, die wir so weit führen, bis die Länge der kürzesten Quadratseiten 1 beträgt. Der Baum hat also Tiefe $\log L = l$, in diesem Baum ordnen wir jedem Knoten ein Level zu, die Wurzel habe Level 0, die Blätter entsprechend Level l. Weil jeder Knoten einem Quadrat und die Quadratseiten Gitterlinien entsprechen, können wir jetzt auch Gitterlinien ein Level i zuordnen, offensichtlich hat eine Gitterlinie auf Level i Länge $L/2^i$. Das Level einer Gitterlinie ist natürlich nicht eindeutig, eine Gitterlinie kann ja zu mehreren Quadraten gehören. Es gibt aber für jede Gitterlinie ein eindeutiges höchstes Level, wir nennen dies das maximale Level der Gitterlinie, das maximale Level der Gitterlinie g ist also $\min\{i \mid g \text{ ist Gitterlinie auf Level } i\}$. Vertikale Gitterlinien auf Level i haben Koordinaten $(a+p\cdot L/2^i) \bmod L$, dabei ist $p \in \{0, 1, \ldots, 2^i-1\}$, entsprechend haben horizontale Gitterlinien auf Level i Koordinaten $(b + p \cdot L/2^i) \bmod L$, dabei ist ebenfalls $p \in \{0, 1, \ldots, 2^i - 1\}$.
Weil wir die Level unabhängig von der TSP-Instanz bis zum Einheitsgitter definiert haben, hängt es nur von der Verschiebung (a, b) ab, welches maximale Level eine Gitterlinie hat. Wir betrachten eine vertikale Gitterlinie g und fragen uns, wie groß die Wahrscheinlichkeit ist, dass g maximales Level i hat. Es gibt offensichtlich nur jeweils genau einen Wert für a, so dass g maximales Level 0 bzw. 1 hat. Danach verdoppelt sich die Anzahl der möglichen Wahlen, so dass wir insgesamt

$$\text{Prob}\,(g \text{ hat maximales Level } i) = \begin{cases} \frac{2^{i-1}}{L} & \text{falls } i \geq 1 \\ \frac{1}{L} & \text{falls } i = 0 \end{cases}$$

haben. Für vertikale Linien gilt das natürlich ebenfalls.
Wir definieren jetzt $s := \lceil 16/\varepsilon \rceil$, $r := s + 4$, $m := 2s \log L$. Kern des Beweises ist eine Funktion $\texttt{Modify}(g, i, b)$, die für die vertikale Gitterlinie g, die

maximales Level i hat, bei vertikaler Verschiebung b die optimale Tour π so modifiziert, dass sie am Ende aller Modifikationen (m, r)-leicht ist. Zentrales Instrument in dieser Funktion ist die Verkleinerung der Anzahl der Schnittpunkte, die wir wie im Beweis des Patchinglemmas (Lemma 9.9) ausgeführt durchführen. Es gibt natürlich auch eine entsprechende Funktion für horizontale Gitterlinien, für die wir die horizontale Verschiebung a berücksichtigen und die ganz analog funktioniert.

`Modify`(g, i, b)
1. Für alle $j \in \{l, l-1, \ldots, i\}$
2. Für alle $p \in \{0, 1, \ldots, 2^j - 1\}$
3. If π und g haben zwischen $\left(b + p \cdot \frac{L}{2^j}\right) \bmod L$ und $\left(b + (p+1) \cdot \frac{L}{2^j}\right) \bmod L$ mehr als s Schnittpunkte
4. Then Verkleinere die Anzahl der Schnittpunkte auf ≤ 4.

Die Verkleinerung der Anzahl der Schnittpunkte auf ≤ 4 in Zeile 4 ist vielleicht etwas überraschend, da das Patchinglemma uns ja eine Verkleinerung der Anzahl der Schnittpunkte auf höchstens 2 erlaubt. Wir müssen uns aber daran erinnern, dass wir ja Quadrate durch die Verschiebung teilen können, für diese geteilten Quadrate führen wir die Konstruktion des Patchinglemmas getrennt in den Teilen durch, so dass sich die Anzahl der Schnittpunkte insgesamt auf höchstens 4 verringert.

Dass der entstehende TSP-Pfad (m, r)-leicht ist, ist offensichtlich. Wir müssen aber zeigen, dass `Modify` die Kosten der Tour nicht zu sehr vergrößert. Das machen wir nur für Gitterlinien mit maximalem Level > 0, weil Level 0 ja für das umschließende Quadrat steht, dessen Linien nicht geschnitten werden.

Es bezeichne $c_{g,j}(b)$ die Anzahl der Abschnitte der Gitterlinie g, für die wir das Patchinglemma (Lemma 9.9) auf Level j angewendet haben. Initial haben wir die Notation aus Lemma 9.10 benutzend insgesamt $\tau(\pi, g)$ Schnittpunkte. Jede Anwendung des Patchinglemmas verkleinert die Anzahl der Schnittpunkte von mindestens $s+1$ auf höchstens 4, die Anzahl der Schnittpunkte sinkt also um mindestens $s-3$. Darum gilt

$$\sum_{j=i}^{\log L} c_{g,j}(b) \leq \frac{\tau(\pi, g)}{s-3}$$

und wir können die Kostenzunahme abschätzen. Bei der Anwendung des Patchinglemmas führen wir höchstens neue Segmente der Länge $2s$ ein, wenn s die Länge des geschnittenen Liniensegments bezeichnet, wir können also die

Kostenzunahme durch `Modify`(g, i, b) insgesamt durch

$$\sum_{j=i}^{\log L} c_{g,j}(b) \cdot 2 \cdot \frac{L}{2^j}$$

nach oben beschränken. Wir ordnen diese Kosten der betroffenen Gitterlinie g zu.
Wie groß sind nun die Gesamtkosten, die wir der Gitterlinie g für die Anwendung des Patchinglemmas zuordnen? Das hängt offensichtlich vom maximalen Level i von g ab, das hängt seinerseits von der zufällig gewählten Verschiebung ab. Wir können also die *erwarteten Gesamtkosten*, die der Gitterlinie g durch Anwendung des Patchinglemmas entstehen, abschätzen. Wir erhalten

$$\begin{aligned}
\mathrm{E}\,(\text{Kosten für } g) &= \sum_{i=1}^{\log L} \mathrm{Prob}\,(g \text{ hat max. Level } i) \cdot \text{Gesamtkosten}(\texttt{Modify}(g, i, b)) \\
&\leq \sum_{i=1}^{\log L} \frac{2^{i-1}}{L} \cdot \sum_{j=i}^{\log L} c_{g,j}(b) \cdot 2 \cdot \frac{L}{2^j} \\
&= \sum_{i=1}^{\log L} \sum_{j=i}^{\log L} 2^i \cdot \frac{c_{g,j}(b)}{2^j} = \sum_{j=1}^{\log L} \sum_{i=1}^{j} 2^i \cdot \frac{c_{g,j}(b)}{2^j} \\
&= \sum_{j=1}^{\log L} \frac{c_{g,j}(b)}{2^j} \cdot \sum_{i=1}^{j} 2^i = \sum_{j=1}^{\log L} \frac{c_{g,j}(b)}{2^j} \cdot \left(2^{j+1} - 2\right) \\
&\leq \sum_{j=1}^{\log L} 2c_{g,j}(b) \leq 2\frac{\tau(\pi, g)}{s-3}
\end{aligned}$$

als obere Schranke. Damit haben wir die Kosten für die Anwendung des Patchinglemmas, das die Anzahl der Schnittpunkte reduziert, auf die Gitterlinien verteilt und je Gitterlinie abgeschätzt. Was wir bis hier noch nicht berücksichtigt haben, sind die Mehrkosten, die dadurch entstehen, dass wir die Gitterlinien nicht mehr an beliebigen Stellen kreuzen dürfen, wir sind ja auf Portalbenutzung festgelegt. Wir haben auf jeder Quadratseite m gleichmäßig verteilte Portale, der Abstand zum nächsten Portal ist also durch $2L/(2^i \cdot m)$ nach oben beschränkt, die Kosten hängen also wiederum vom Level der Quadratseite ab und sind somit Zufallsvariablen, die von der Verschiebung abhängen. Für die Gitterlinie g ergeben sich damit *erwartete Mehrkosten* durch

Portalbenutzung als

$$\begin{aligned} & \sum_{i=1}^{\log L} \frac{2^{i-1}}{L} \cdot \frac{2L}{2^i \cdot m} \cdot \tau(\pi, g) \\ = & \sum_{i=1}^{\log L} \frac{\tau(\pi, g)}{m} = \frac{\tau(\pi, g) \log L}{m}, \end{aligned}$$

und wir erinnern uns daran, dass wir $m = 2s \log L$ gewählt hatten, so dass wir die Mehrkosten durch Portalbenutzung an Gitterlinie g nach oben durch $\tau(\pi, g)/(2s) < \tau(\pi, g)/s$ abschätzen können. Wir haben jetzt also insgesamt für Gitterlinie g die erwarteten Gesamtmehrkosten durch Modifikation der optimalen Tour nach oben durch

$$2\frac{\tau(\pi, g)}{s-3} + \frac{\tau(\pi, g)}{s} < \frac{4\tau(\pi, g)}{s}$$

abschätzen, weil s nicht all zu klein sein kann. Falls ε so groß ist, dass diese Abschätzung nicht stimmt, können wir den Algorithmus von Christofides (Algorithmus 9.5) benutzen.
Wir erhalten die erwarteten Mehrkosten für alle Modifikationen, indem wir über alle Gitterlinien summieren. Das liefert

$$\begin{aligned} \mathrm{E}\,(\text{Mehrkosten}) &\leq \sum_{g \text{ vertikal}} \frac{4\tau(\pi, g)}{s} + \sum_{g \text{ horizontal}} \frac{4\tau(\pi, g)}{s} \\ &= \frac{4}{s} \cdot \left(\sum_{g \text{ vertikal}} \tau(\pi, g) + \sum_{g \text{ horizontal}} \tau(\pi, g) \right) \\ &\leq \frac{8 \cdot \mathrm{OPT}}{s} \leq \frac{\varepsilon}{2} \cdot \mathrm{OPT} \end{aligned}$$

als obere Schranke für die erwarteten Mehrkosten, dabei folgt die vorletzte Ungleichung durch Anwendung von Lemma 9.10. Eine Anwendung der Markow-Ungleichung liefert, dass die Wahrscheinlichkeit, dass die Mehrkosten größer als

$$2 \cdot \frac{\varepsilon}{2} \cdot \mathrm{OPT} = \varepsilon \cdot \mathrm{OPT}$$

sind, durch 1/2 nach oben beschränkt ist. Damit ist das Strukturtheorem (Theorem 9.8) bewiesen. □

Wir hatten eingangs versprochen, dass wir aus dem randomisierten Algorithmus noch einen deterministischen Algorithmus, also ein PTAS, machen wollen. Das ist hier tatsächlich trivial möglich. Wir wählen die Verschiebung

$a, b \in \{0, 1, \ldots, L-1\}$ und haben $L = O(n)$. Es gibt also insgesamt nur $O(n^2)$ viele Möglichkeiten, a und b zu wählen. Wir verzichten auf die zufällige Wahl von a und b und probieren stattdessen alle $O(n^2)$ verschiedenen Wahlen aus. Weil die Wahrscheinlichkeit, dass eine Wahl günstig ist, wie wir eben gesehen haben, durch $1/2$ nach unten beschränkt ist, muss mindestens die Hälfte der Wahlen von a und b günstig sein, also zu $(1+\varepsilon)$-Approximationen führen. Wir haben also am Ende mit Sicherheit eine $(1+\varepsilon)$-Approximation gefunden. Das erste PTAS für das euklidische TSP wurde 1996 von Arora vorgestellt, die Darstellung hier folgt im Wesentlichen der Darstellung einer verbesserten Version, die 1998 im Journal of the ACM ebenfalls von Arora vorgestellt wurde. Wir halten die Ergebnisse, die wir jetzt insgesamt bewiesen haben, noch einmal formal fest.

Theorem 9.11. *Der randomisierte Algorithmus von Arora hat Laufzeit* $O\left(n\,(\log n)^{O\left(\varepsilon^{-1}\right)}\right)$*, er berechnet für jede Instanz des euklidischen TSP in der Ebene mit* n *Punkten für jedes* $\varepsilon > 0$ *mit Wahrscheinlichkeit mindestens* $1/2$ *eine* $(1+\varepsilon)$*-Approximation.*
Der deterministische Algorithmus von Arora ist ein PTAS für das TSP und hat Laufzeit $O\left(n^3\,(\log n)^{O\left(\varepsilon^{-1}\right)}\right)$*.*
Es gilt euklidisches TSP $\in \mathcal{PTAS}$*.*

□

Mehr als ein polynomielles Approximationsschema kann man für das euklidische TSP nicht erwarten: Es ist bekannt, dass es ein echt polynomielles Approximationsschema für das euklidische TSP nur gibt, wenn P = NP gilt. Der Algorithmus von Arora war in Bezug auf die Erkenntnisse über die Approximierbarkeit von TSP-Problemen zweifellos ein Durchbruch. Besondere praktische Relevanz hat er aber nicht, die in der O-Notation steckenden Konstanten sind einfach zu groß. Man darf noch anmerken, dass der Algorithmus sich vom euklidischen TSP in der Ebene auch aufs euklidische TSP im $\mathbb{R}^d$ verallgemeinern lässt und für konstante d weiterhin polynomielle Laufzeit hat. Die randomisierte Variante kommt dann mit Laufzeit $O\left(n\,(\log n)^{O\left(\sqrt{d}\cdot\varepsilon^{-1}\right)^{d-1}}\right)$ aus, das Derandomisieren (das Ausprobieren alle möglichen Verschiebungen) führt zu einem zusätzlichen Faktor $O\left(n^d\right)$ in der Laufzeit.

10 Erfüllbarkeitsprobleme

Ein weiteres Problem, das schon intensiv in GTI bzw. TIfAI behandelt wurde und das wir uns hier unter dem Aspekt der Optimierung noch einmal ansehen wollen, ist das Erfüllbarkeitsproblem SAT. Die Kapitelüberschrift spricht bewusst von Erfüllbarkeitsproblemen, weil wir uns verschiedene Varianten ansehen wollen. Schon in GTI bzw. TIfAI wurde ja zwischen dem allgemeinen Erfüllbarkeitsproblem SAT und einer eingeschränkten Variante, bei der alle Klauseln höchstens drei Literale enthalten dürfen, unterschieden. Wir werden hier auch auf diese Art Varianten des Erfüllbarkeitsproblems unterscheiden, dabei werden wir aber noch deutlich genauer sein. Im Kontext der Approximation spielt es zum Beispiel eine erhebliche Rolle, ob Klauseln drei Literale oder höchstens drei Literale enthalten, es ist auch nicht egal, ob in einer Klausel gleiche Literale wiederholt werden dürfen oder nicht. Ob die von uns jeweils aktuell betrachtete Variante NP-vollständig ist oder nicht, werden wir zitieren und hier selbst keinen derartigen Beweis führen. Es geht uns hier vor allem um Algorithmen und nicht so sehr um Komplexitätstheorie.

Nach dem Rucksackproblem (Kapitel 8) und dem Traveling-Salesperson-Problem (Kapitel 9) ist das Erfüllbarkeitsproblem das dritte Beispiel für ein Problem aus $\mathcal{NPO}$, dessen zugehöriges Entscheidungsproblem schon in GTI bzw. TIfAI diskutiert wurde, mit dem wir uns hier auseinandersetzen. Man braucht keine Angst zu haben, dass das hier eine endlose und langweilige Liste wird. Wir besprechen in diesen drei Kapiteln auch jeweils ganz andere Aspekte der Optimierung von Problem aus $\mathcal{NPO}$. Beim Rucksackproblem ging es uns um ein echt polynomielles Approximationsschema, beim Traveling-Salesperson-Problem einerseits um eine mit verhältnismäßig einfachen Mitteln erzielbare (3/2)-Approximation und andererseits um ein polynomielles Approximationsschema, das aus einem im Grunde simplen dynamischen Programmierungsansatz besteht. Randomisierung hat beim Algorithmus von Arora eher am Rande eine Rolle gespielt, es war insbesondere nicht schwierig, der Verwendung des Zufalls zu entgehen, indem alle möglichen Ausgänge des einzigen Zufallsexperiments des randomisierten Algorithmus deterministisch durchprobiert werden. In diesem Kapitel werden wir stärker Gebrauch von Randomisierung machen und uns überlegen, ob man auch dann systematisch auf den Gebrauch von Zufallsexperimenten verzichten kann, wenn die Anzahl möglicher Ausgänge der Zufallsexperimente zu groß ist, um vollständiges Ausprobieren zuzulassen. Bevor wir das am

Beispiel des Erfüllbarkeitsproblems machen, wollen wir einen ganz kurzen wiederholenden Ausflug in die Analyse randomisierter Algorithmen machen.

10.1 Analyse randomisierter Algorithmen

Wir erinnern uns Turingmaschinen als elementares Rechenmodell. Bei einer deterministischen Turingmaschine steht bei gegebener Eingabe der Rechenweg fest, die Anzahl der Rechenschritte entspricht der Länge dieses Rechenweges. Bei einer randomisierten Turingmaschine ist der Rechenweg bei gegebener Eingabe noch vom Ausgang von Zufallsexperimenten abhängig, man kann bei fester Eingabe alle möglichen Rechenwege als einen Baum beschreiben, in dem die Startkonfiguration die Wurzel ist, jeder Rechenschritt zu einem Nachfolger des aktuellen Knotens führt und die Kante mit der Wahrscheinlichkeit beschriftet ist, mit der dieser Rechenschritt durchgeführt wird. Der erwartete (oder auch durchschnittliche) Rechenzeit einer solchen randomisierten Turingmaschine auf dieser Eingabe ist dann die durchschnittliche Tiefe des Baumes, wobei aber die einzelnen Tiefen jeweils mit ihren Wahrscheinlichkeiten, die sich als Produkt der Kantenbeschriftungen auf den Wegen ergeben, gewichtet werden. Randomisierten Turingmaschinen erlaubt man in der Regel nur faire Münzwürfe, so dass alle Kantengewichte entweder $1/2$ oder 1 sind. Wir werden hier aber im Grunde beliebige Wahrscheinlichkeiten zulassen und erinnern uns daran, dass das in der Regel unproblematisch ist, weil wir „vernünftige“ Wahrscheinlichkeitsverteilungen durch polynomiell viele faire Münzwürfe ausreichend gut approximieren können. Weil das Thema in GTI bzw. TIfAI angesprochen wurde, wollen wir diese Diskussion hier nicht weiter vertiefen.
Wir schauen uns ein simples und sinnloses Beispiel an, um uns zu versichern, dass wir auch dann die erwartete Rechenzeit bestimmen können, wenn der Algorithmus anscheinend endlos lange laufen könnte. Außerdem können wir unsere Notation für Zufallsexperimente kennenlernen. Sei für den folgenden Algorithmus p mit $0 < p < 1$ eine feste Wahrscheinlichkeit.

1. Mit Wahrscheinlichkeit p Setze $z := 0$
   Sonst setze $z := 1$.
2. If $z = 0$ Then Weiter bei 1.

Wir legen fest, dass wir eine Ausführung von Zeile 1 und Zeile 2 als einen Rechenschritt zählen wollen. Wer zwei Rechenschritte vernünftiger findet, kann ja am Ende unser Ergebnis noch verdoppeln. Was ist mit dieser Festlegung die erwartete Rechenzeit, also die erwartete Anzahl von Rechenschritten?

Das hängt offensichtlich vom Ausgang der Zufallsexperimente in Zeile 1 ab. Mit Wahrscheinlichkeit p wird in Zeile 2 noch ein Durchlauf gestartet, mit der Gegenwahrscheinlichkeit $1-p$ war der aktuelle Durchlauf der letzte. Wir können uns das wie bei einer randomisierten Turingmaschine als Baum vorstellen, es gibt eine Wurzel mit zwei ausgehenden Kanten, eine ist mit p und die andere mit $1-p$ beschriftet. Die mit $1-p$ beschriftete Kante führt zu einem Blatt, die andere Kante zu einem Knoten, der die gleichen Eigenschaften wie die Wurzel hat. Wir können die erwartete Rechenzeit ausrechnen, indem wir für jedes Blatt im Baum seine Tiefe mit der Wahrscheinlichkeit gewichten, dass dieses Blatt erreicht wird. Das Blatt mit Tiefe 1 erreichen wir offenbar mit Wahrscheinlichkeit $1-p$, das Blatt mit Tiefe 2 mit Wahrscheinlichkeit $p \cdot (1-p)$, allgemein das Blatt mit Tiefe d mit Wahrscheinlichkeit $p^{d-1} \cdot (1-p)$. Wenn die Zufallsvariable T die tatsächliche Rechenzeit angibt, erhalten wir also

$$\begin{aligned}
\mathrm{E}(T) &= \sum_{d=1}^{\infty} d \cdot p^{d-1} \cdot (1-p) = (1-p) \cdot \sum_{d=1}^{\infty} d \cdot p^{d-1} \\
&= (1-p) \cdot \left(\sum_{d=1}^{\infty} \sum_{i=d}^{\infty} p^{i-1} \right) = (1-p) \cdot \left(\sum_{d=1}^{\infty} \sum_{i=d-1}^{\infty} p^{i} \right) \\
&= (1-p) \cdot \left(\sum_{d=1}^{\infty} \left(\sum_{i=0}^{\infty} p^i - \sum_{i=0}^{d-2} p^i \right) \right) \\
&= (1-p) \cdot \left(\sum_{d=1}^{\infty} \left(\frac{1}{1-p} - \frac{1-p^{d-1}}{1-p} \right) \right) \\
&= \sum_{d=1}^{\infty} p^{d-1} = \sum_{d=0}^{\infty} p^d = \frac{1}{1-p}
\end{aligned}$$

als erwartete Rechenzeit. Wir werfen gedanklich noch einen Blick auf den Berechnungsbaum und sehen, dass wir natürlich auch anders hätten summieren können. Statt jedes Blatt mit seiner Tiefe und seiner Wahrscheinlichkeit zu gewichten, hätten wir auch jeden Knoten im Baum gewichtet mit seiner Wahrscheinlichkeit zählen können. Dieser Ansatz führt zu

$$\begin{aligned}
\mathrm{E}(T) &= \sum_{d=1}^{\infty} \left((1-p) \cdot p^{d-1} + p^d \right) = \sum_{d=1}^{\infty} \left(p^{d-1} - p^d + p^d \right) \\
&= \sum_{d=0}^{\infty} p^d = \frac{1}{1-p}
\end{aligned}$$

als erwarteter Rechenzeit. Das Ergebnis ist natürlich das gleiche, die Rechnung in diesem Fall etwas einfacher. Allerdings war ohnehin klar, dass bei

diesem simplen Algorithmus die erwartete Rechenzeit der Wartezeit auf einen Erfolg bei Erfolgswahrscheinlichkeit $1-p$ entsprach, die erwartete Rechenzeit also geometrisch verteilt mit Parameter $1-p$ ist, so dass sich $1/(1-p)$ als Erwartungswert ergibt.
Nach diesem ganz offen gesprochen trivialen Beispiel wollen wir uns noch ein etwas weniger triviales (aber immer noch sehr simples) Beispiel anschauen, diesmal an einem mehr oder minder praktischem Problem. Nehmen wir an, wir wollen aus der Menge $\{1, 2, \ldots, 49\}$ „rein zufällig“ eine sechselementige Teilmenge ziehen, sind also daran interessiert, aus allen $\binom{49}{6}$ sechselementigen Teilmengen von $\{1, 2, \ldots, 49\}$ eine zufällig gemäß Gleichverteilung auszuwählen.
Es gibt sicher verschiedene Arten, dieses Problem zu lösen. Betrachten wir zunächst die direkte Implementierung der folgenden Idee. Weil wir die sechs Zahlen gleichverteilt auswählen, gehört die Zahl 1 mit Wahrscheinlichkeit $6/49$ zu unserer Auswahl. Abhängig davon, ob die Zahl 1 gewählt wird oder nicht, können wir sagen, mit welcher Wahrscheinlichkeit die Zahl 2 zu unserer Auswahl gehört. Wird die Zahl 1 nicht gewählt, so gehört die Zahl 2 mit Wahrscheinlichkeit $6/48$ zu unserer Auswahl, andernfalls gehört sie mit Wahrscheinlichkeit $5/48$ zu unserer Auswahl. Diese Argumentation können wir bis zum Ende fortsetzen und auch direkt algorithmisch umsetzen.

1. $n := 49$; $k := 6$; $i := 1$
2. Repeat
3. Mit Wahrscheinlichkeit k/n
   Gib i aus; $k := k - 1$
4. $i := i + 1$; $n := n - 1$
5. Until $k = 0$

Weil i anfangs gleich 1 gesetzt und in jedem Schleifendurchlauf um 1 erhöht wird, sei $i - 1$ unser Maß für die Anzahl der Rechenschritte. Wir definieren also T als Wert von $i-1$ am Ende des Algorithmus und interessieren uns für die erwartete Rechenzeit $\mathrm{E}(T)$. Natürlich gilt

$$\mathrm{E}(T) = \sum_{t=0}^{\infty} t \cdot \mathrm{Prob}(T = t),$$

aber es scheint schwierig zu sein, $\mathrm{Prob}(T = t)$ abzuschätzen. Wenn wir uns daran erinnern, was der Algorithmus macht, ist es aber nicht so schwierig. Der Algorithmus gibt gleichverteilt eine sechselementige Teilmenge von $\{1, 2, \ldots, 49\}$ aus, dabei erfolgt die Ausgabe sortiert und der Wert von i am Ende ist gerade um 1 größer als die größte in der Auswahl vorkommende

Zahl Wenn also M die größte in der Auswahl vorkommende Zahl bezeichnet, so haben wir $T = i - 1 = M$ und wir können

$$\begin{aligned} \mathrm{E}(T) &= \sum_{m=6}^{49} m \cdot \mathrm{Prob}(M = m) \\ &= \sum_{m=6}^{49} m \cdot \frac{\binom{m-1}{5}}{\binom{49}{6}} \\ &= \frac{300}{7} \approx 42{,}857 \end{aligned}$$

als erwartete Rechenzeit bestimmen. Wir können natürlich beliebige Zahlen $n, k \in \mathbb{N}$ mit $k \leq n$ wählen. Wenn n sehr groß ist und die Durchführung von Zufallsexperimenten eine teure Rechenoperation, wünscht man sich vielleicht ein Verfahren, dass etwas sparsamer mit Zufallsexperimenten umgeht. Dazu schlagen wir folgenden Algorithmus vor, den wir wieder konkret für $n = 49$ und $k = 6$ beschreiben. Die Verallgemeinerung für beliebige Werte von n und k ist wieder offensichtlich.

```
1.  Für i ∈ {1, 2, ..., 49} setze a[i] := 0.
2.  Für i ∈ {1, 2, ..., 6}
3.      Repeat
4.          Wähle z ∈ {1, 2, ..., 49} gemäß Gleichverteilung zufällig.
5.      Until a[z] = 0
6.      a[z] := 1
7.  Für i ∈ {1, 2, ..., 49}
8.      If a[i] = 1 Then Ausgabe i
```

Der Algorithmus ist auch offensichtlich korrekt. Die Rechenzeit ist hier offenbar durch $\Omega(n)$ nach unten beschränkt, so dass wir im Vergleich zum ersten Algorithmus an Rechenzeit nicht gewinnen können. Wie sieht es mit der Anzahl Z von Zufallsexperimenten aus, die in Zeile 4 durchgeführt werden? Bei der Durchführung des Zufallsexperiments wird uniform zufällig eine Zahl aus $\{1, 2, \ldots, 49\}$ gezogen, dabei sind aber schon $i-1$ Zahlen nicht mehr zulässig und führen zu einer Wiederholung des Zufallsexperiments. Wir wissen, dass die erwartete Anzahl Wiederholungen für jeden Wert von i geometrisch verteilt ist mit Parameter $(49 - (i-1))/49$, so dass sich $49/(49 - (i-1))$ als erwartete Anzahl von Zufallsexperimenten für i ergibt. Wir erhalten $\mathrm{E}(Z)$

durch Addition, also ist

$$\begin{aligned} \mathrm{E}(Z) &= \sum_{i=1}^{6} \frac{49}{49-(i-1)} \\ &= 1 + \frac{49}{48} + \frac{49}{47} + \frac{49}{46} + \frac{49}{45} + \frac{49}{44} \approx 6{,}33 \end{aligned}$$

die erwartete Anzahl von Zufallsexperimenten, was offensichtlich erheblich weniger ist. Vor allem für große Werte von n und kleine Werte von k ist der Unterschied bedeutsam.
Aus Gründen der Vollständigkeit wollen wir noch anmerken, dass man das Problem auch mit einer festen Laufzeit und genau 6 Zufallsexperimenten lösen kann, was offenbar beiden Methoden vorzuziehen ist.

1. Für $i \in \{1, 2, \ldots, 49\}$ setze $a[i] := i$.
2. Für $i \in \{1, 2, \ldots, 6\}$
3. Wähle $z \in \{i, i+1, \ldots, 49\}$ gemäß Gleichverteilung zufällig.
4. Vertausche $a[i]$ und $a[z]$.
7. Für $i \in \{1, 2, \ldots, 6\}$
8. Ausgabe $a[i]$

Der Algorithmus implementiert offensichtlich wieder die Idee, die schon unser erster Algorithmus implementiert. Er spart sich aber den aufwendigen Durchlauf durch alle Zahlen mit jeweiligem Zufallsexperiment und wählt stattdessen in jeder Runde aus den verbleibenden Zahlen eine uniform zufällig. Nach diesem hoffentlich amüsanten Ausflug in die Analyse randomisierter Algorithmen wollen wir uns aber jetzt wieder dem Thema Approximation zuwenden und uns das Erfüllbarkeitsproblem ansehen.

10.2 Approximation von MAX-*k*-SAT und MAXSAT

Beim Erfüllbarkeitsproblem SAT besteht die Eingabe aus m Klauseln c_1, c_2, …, c_m über den Variablen x_1, x_2, …, x_n, eine Klausel c_i ist dabei eine Disjunktion von Literalen über den Variablen x_1, x_2, …, x_n. Beispiele für Klauseln sind also $\overline{x_2} \vee x_4 \vee x_6 \vee \overline{x_8}$ oder auch $\overline{x_3}$. Die Anzahl der Literale in einer Klausel nennen wir ihre Länge, die erste Beispielklausel hat also Länge 4, die zweite Beispielklausel Länge 1. Zulässige Lösungen einer solchen SAT-Instanz sind Belegungen $b \in \{0,1\}^n$ der n Variablen. Eine solche Belegung b erfüllt eine Klausel, wenn eine in der Klausel negiert vorkommende Variable mit 0 oder eine nichtnegiert vorkommende Variable mit 1 belegt

ist. Der Wert einer Belegung $v(b)$ ist die Anzahl der durch diese Belegung erfüllten Klauseln. Das Erfüllbarkeitsproblem ist ein Maximierungsproblem, zur Unterscheidung vom zugehörigen Entscheidungsproblem, das häufig SAT genannt wird, nennen wir das Problem MAXSAT.

Wir wollen die Eingaben für MAXSAT noch etwas stärker strukturieren. Wir nennen eine Klausel reduziert, wenn in ihr zu jeder Variable x_i höchstens ein Literal vorhanden ist. Die Klauseln $x_1 \vee x_2 \vee x_2$ und $x_1 \vee x_3 \vee \overline{x_1}$ sind also beide nicht reduziert. Wir nennen eine MAXSAT-Instanz reduziert, wenn alle Klauseln reduziert sind und werden im Folgenden ausschließlich über reduzierte MAXSAT-Instanzen sprechen. Außerdem definieren wir das Problem MAX-k-SAT, bei dem zulässige Eingaben reduzierte MAXSAT-Instanzen sind, in denen jede Klausur *genau* Länge k hat. Wir erinnern daran, dass das zugehörige Entscheidungsproblem zu MAX-k-SAT für jedes $k \in \mathbb{N} \setminus \{1\}$ NP-vollständig ist, MAX-k-SAT also für jedes $k \in \mathbb{N} \setminus \{1\}$ NP-schwierig ist.

Wir sehen uns jetzt als Erstes einen sehr einfachen randomisierten Algorithmus an, der für große k gute approximative Lösungen verspricht. Dabei werden wir verstehen, dass das Problem für große k leichter approximierbar wird.

Algorithmus 10.1.
1. *Für* $i \in \{1, 2, \ldots, n\}$
2. *Mit Wahrscheinlichkeit* $1/2$ *setze* $b[i] := 0$, *sonst setze* $b[i] := 1$.
3. *Ausgabe* b

Theorem 10.2. *Algorithmus 10.1 hat Laufzeit* $\Theta(n)$ *und erfüllt im Durchschnitt* $\left(1 - 2^{-k}\right) \cdot m$ *aller Klauseln einer reduzierten MAX-k-SAT-Instanz mit* m *Klauseln.*

Beweis. Die Aussage über die Laufzeit ist offensichtlich. Für die erwartete Anzahl erfüllter Klauseln definieren wir die Zufallsvariable X, sie gebe die Anzahl der durch b erfüllten Klauseln an. Außerdem definieren wir m 0-1-wertige Zufallsvariablen $X_1, X_2, \ldots, X_m$ (man sagt auch *Indikatorvariablen*), dabei nimmt X_i den Wert 1 an, wenn die Belegung b die Klausel i-te Klausel erfüllt, und andernfalls den Wert 0. Natürlich gilt

$$X = \sum_{i=1}^{m} X_i$$

und damit auch

$$\mathrm{E}(X) = \mathrm{E}\left(\sum_{i=1}^{m} X_i\right) = \sum_{i=1}^{m} \mathrm{E}(X_i) = \sum_{i=1}^{m} \mathrm{Prob}(X_i = 1)$$
$$= \sum_{i=1}^{m} \mathrm{Prob}(b \text{ erfüllt } i\text{-te Klausel}).$$

Betrachten wir eine Klausel, zum Beispiel die erste. Weil wir eine reduzierte MAX-k-SAT-Instanz betrachten, enthält sie Literale über k verschiedene Variable. Wir betrachten alle 2^k verschiedene Belegungen der k betroffenen Variablen und sehen, dass genau eine Belegung die Klausel nicht erfüllt und genau $2^k - 1$ Belegungen die Klausel erfüllen. Die Wahrscheinlichkeit, dass die uniform zufällige gewählte Belegung b die Klausel erfüllt, beträgt also $(2^k - 1)/2^k$. Also ist

$$\mathrm{Prob}(b \text{ erfüllt } i\text{-te Klausel}) = \frac{2^k - 1}{2^k} = 1 - \frac{1}{2^k}$$

für alle m Klauseln und der Beweis ist geführt. □

Wir sehen, dass es ziemlich schwierig ist, eine MAX-k-SAT-Klausel nicht zu erfüllen, wenn k nicht sehr klein ist. Wenn wir aus dem simplen Algorithmus 10.1 einen Approximationsalgorithmus im Sinne unserer Definition machen wollen, müssen wir eine obere Schranke für die Abweichung von der maximal erfüllbaren Anzahl Klauseln garantieren. Das fällt uns leicht, wenn wir aus dem randomisierten Algorithmus einen deterministischen Algorithmus machen können. Die systematische Umwandlung eines randomisierten Algorithmus in einem deterministischen Algorithmus nennt man *Derandomisierung*. Die triviale Derandomisierung, die wir beim Algorithmus von Arora angewendet haben, funktioniert hier leider nicht: Wir können nicht in Polynomialzeit alle 2^n verschiedenen Belegungen ausprobieren. Wir werden darum anders vorgehen.
Wir erinnern uns daran, wie wir in Algorithmus 10.1 die zufällige Belegung $b \in \{0,1\}^n$ gewählt haben: Das geschah bitweise von links nach rechts. Wir betrachten die Wahrscheinlichkeit $p_i := \mathrm{Prob}(b[i] = 1)$ und sehen, dass es für diese Wahrscheinlichkeit zwei wesentlich verschiedene Fälle gibt. So lange wir b an der Stelle i noch nicht gesetzt haben, ist $p_i = \mathrm{Prob}(b[i] = 1) = 1/2$. Nachdem wir b an der Stelle i gemäß Ausgang des zugehörigen Zufallsexperiments auf 0 oder 1 gesetzt haben, ist $p_i = \mathrm{Prob}(b[i] = 1) = 1$, wenn wir uns für eben für $b[i] = 1$ entschieden haben, andernfalls ist $p_i = \mathrm{Prob}(b[i] = 1) = 0$. Wir haben also zu jeder Zeit $p_i \in \{0, 1/2, 1\}$.

Der Wert von X_i, der Indikatorvariable, die angibt, ob die i-te Klausel erfüllt ist, hängt ausschließlich von der Belegung der Variablen ab, die in der i-ten Klausel c_i vorkommen, wir definieren darum

$$I_i^+ := \{j \in \{1, 2, \ldots, n\} \mid x_j \text{ kommt in } c_i \text{ vor}\}$$

und entsprechend

$$I_i^- := \{j \in \{1, 2, \ldots, n\} \mid \overline{x_j} \text{ kommt in } c_i \text{ vor}\}.$$

Mit diesen beiden Indexmengen können wir jetzt $\mathrm{E}(X_i)$ bequem aufschreiben, es ist

$$C_i(p_1, p_2, \ldots, p_n) := \mathrm{E}(X_i) = 1 - \prod_{j \in I_i^+} (1 - p_j) \cdot \prod_{j \in I_i^-} p_j$$

und diese Gleichheit gilt für beliebige Werte von p_j, insbesondere also auch, wenn schon einige Variablen fest belegt wurden und entsprechend $p_j \in \{0, 1\}$ gilt.
Weil wir an der Gesamtheit aller Klauseln interessiert sind, definieren wir noch

$$C(p_1, p_2, \ldots, p_n) := \mathrm{E}(X) = \sum_{i=1}^{m} C_i(p_1, p_2, \ldots, p_n)$$

und bemerken, dass sowohl C als auch alle C_i für jede *einzelne* Variable p_j lineare Funktionen in p_j sind. Wir wollen C maximieren und erinnern uns daran, dass man die Extrema linearer Funktionen an den Rändern ihres Definitionsbereichs findet. Wir haben natürlich $0 \leq p_j \leq 1$ für alle j, also wird das Maximum für $p_j = 0$ oder $p_j = 1$ angenommen, dabei sind die anderen p_i fest. Damit haben wir alle Ideen für einen deterministischen Algorithmus zusammen.

Algorithmus 10.3.
1. *Für* $i \in \{1, 2, \ldots, n\}$ *setze* $p_i := 1/2$.
2. *Für* $i \in \{1, 2, \ldots, n\}$
3. *If* $C(p_1, p_2, \ldots, p_{i-1}, 0, p_{i+1}, \ldots, p_n) > C(p_1, p_2, \ldots, p_{i-1}, 1, p_{i+1}, \ldots, p_n)$
4. *Then* $p_i := 0$ *Else* $p_i := 1$.
5. *Für* $i \in \{1, 2, \ldots, n\}$ *setze* $b[i] := p_i$
6. *Ausgabe* b

Theorem 10.4. *Algorithmus 10.3 berechnet zu einer reduzierten MAX-k-SAT-Instanz mit m Klauseln über n Variablen in Zeit $\Theta(n \cdot m)$ eine Belegung, die mindestens $\left(1 - 2^{-k}\right) \cdot m$ Klauseln erfüllt.*
Algorithmus 10.3 ist eine $\left(1 + \frac{1}{2^k - 1}\right)$-Approximation für MAX-k-SAT.

Beweis. Wir belegen initial alle p_i mit $1/2$, das geht ebenso wie das abschließende Übertragen in b und die Ausgabe von b in Zeit $\Theta(n)$. Außerdem halten wir $C_j = 1 - 2^{-k}$ und $C = m \cdot \left(1 - 2^{-k}\right)$ fest, was in Zeit $\Theta(m)$ geht. Wir wollen in Zeile 3 in konstanter Zeit nachschlagen können, ob die Variable x_i in der Klausel c_j vorkommt, dazu legen wir für jede Klausel ein Array über den Variablen an, die Initialisierung braucht also insgesamt Zeit $\Theta(m \cdot n)$. Dann durchlaufen wir mit i (Zeile 2) n Werte, für die Ausführung von Zeile 3 durchlaufen wir alle m Klauseln und passen die C_j und C entsprechend an, das geht für jede Klausel in konstanter Zeit. Wir kommen also insgesamt mit Zeit $\Theta(n \cdot m)$ aus, damit ist die Laufzeit jedenfalls polynomiell.
Für die Approximationsgüte beobachten wir, dass wir initial $C = m \cdot \left(1 - 2^{-k}\right)$ haben, was ausreichend gut ist. Wir halten alle p_j fest und betrachten nur p_i für das aktuelle i als variabel, damit ist C eine lineare Funktion in p_i. Wir setzen $p_i \in \{0, 1\}$, also auf den Wert am Rand des Definitionsbereichs, der C maximal werden lässt. Dadurch kann C nicht kleiner werden. Es gilt also auch am Ende des Algorithmus $C \geq m \cdot \left(1 - 2^{-k}\right)$ und die erste Behauptung ist bewiesen. Für die zweite Aussage müssen wir nur beobachten, dass $\mathrm{OPT} \leq m$ gilt. □

Wir werden noch einen anderen randomisierten Algorithmus für MAX-k-SAT besprechen, müssen dazu aber etwas ausholen. Wir werden zwei andere recht allgemeine Optimierungsprobleme vorstellen, bei denen wir uns zunächst keine Gedanken darüber machen, wie wir sie lösen können. Soweit ein effizienter Algorithmus für die Lösung verfügbar ist, werden wir seine Existenz lediglich zitieren und ihn als eine Art Black Box benutzen. Wir werden uns wesentlich später noch einmal damit auseinandersetzen, wie man solche Probleme lösen kann (siehe Abschnitt 14.1).
Bei der *ganzzahligen linearen Programmierung* besteht eine Eingabe aus einer linearen Zielfunktion über den Variablen $x_1, x_2, \ldots, x_n$, die zu maximieren ist, und insgesamt m linearen Ungleichungen über diesen Variablen. Zulässige Lösungen sind Belegungen $x_1, x_2, \ldots, x_n \in \mathbb{Z}$, so dass alle m Ungleichungen erfüllt sind, die Zielfunktion ist wie gesagt zu maximieren. Wir legen fest, dass alle vorkommenden Koeffizienten ganzzahlig sein müssen, so dass wir nicht in Schwierigkeiten mit reellen Zahlen geraten und sehen damit, dass das Problem jedenfalls zu $\mathcal{NPO}$ gehört.
Wir können MAX-k-SAT als eine Instanz dieses Problems formulieren, wir sagen, dass man MAX-k-SAT als ganzzahliges lineares Programm beschreiben kann. Weil wir x_i, n und m schon bei der Definition von MAX-k-SAT „verbraucht“ haben, werden wir im Vergleich zur allgemeinen Problembeschreibung der ganzzahligen linearen Programmierung die Variablen anders benennen. Für jede Variable x_i in der MAX-k-SAT-Instanz definieren wir

eine Variable y_i, die auch die Rolle von x_i übernehmen soll. Für jede Klausel c_j definieren wir eine Variable z_j, die den Wert 1 annehmen soll, wenn c_j erfüllt ist. Damit können wir $\sum_{j=1}^{m} z_j$ als Zielfunktion festlegen. Wir legen für alle z_j und alle y_i die Ungleichungen $z_j \leq 1$ bzw. $y_i \leq 1$ und $z_j \geq 0$ bzw. $y_i \geq 0$ fest, damit nur Belegungen aus $\{0, 1\}$ zulässig sind. Jetzt müssen wir noch eine Beziehung zwischen den y_i und den z_j so herstellen, wie das für die MAX-k-SAT-Instanz die Klauseln machen; bis jetzt könnten wir ja einfach alle $z_j = 1$ und alle y_i beliebig setzen. Für eine Klausel c_j mit den passend definierten Mengen I_j^+ und I_j^- definieren wir die Ungleichung

$$\sum_{i \in I_j^+} y_i + \sum_{i \in I_j^-} 1 - y_i \geq z_j.$$

Damit haben wir jetzt insgesamt $2n + 3m$ Ungleichungen über $n + m$ Variablen definiert und unser lineares Programm vollständig. Ein Blick auf die zur Klausel c_j definierte Ungleichung lässt uns erkennen, dass wir Belegungen der x_i- und y_i-Variablen direkt übernehmen können und bei maximal möglicher Wahl der z_j-Variablen als Wert der Zielfunktion die Anzahl der erfüllten Klauseln bekommen: Eine Klausel ist nicht erfüllt, wenn alle positiv vorkommenden Variablen mit 0 und alle negativ vorkommenden Variablen mit 1 belegt sind. Genau dann nimmt die linke Seite der Ungleichung den Wert 0 an, wir müssen also $z_j = 0$ setzen. Andernfalls wird die linke Seite mindestens 1 und wir können $z_j = 1$ setzen. Wir haben also MAX-k-SAT polynomiell auf das Problem der ganzzahligen linearen Programmierung reduziert. Weil MAX-k-SAT NP-schwierig ist, können wir nicht erwarten, dass es für die ganzzahlige lineare Programmierung Polynomialzeitalgorithmen gibt. Wir haben also zunächst noch nichts gewonnen.

Ein in zumindest formaler Weise der ganzzahligen linearen Programmierung ähnliches Problem ist das Problem der linearen Programmierung. Die Definition ist fast gleich, wir lassen aber nun als zulässige Belegungen x_1, x_2, …, $x_n \in \mathbb{R}$ zu, die alle m Ungleichungen gleichzeitig erfüllen. Wir lassen also die Bedingung der Ganzzahligkeit fallen, man spricht darum im Vergleich zur ganzzahligen linearen Programmierung von einer *Relaxierung*. Wir können jetzt unser Formulierung einer MAX-k-SAT-Instanz als ganzzahliges lineares Programm unverändert als Formulierung als lineares Programm übernehmen. Wie erhalten als Ergebnis dann eine Belegung y_1, y_2, …, y_m, z_1, z_2, …, $z_m \in [0; 1]$, so dass $\sum_{j=1}^{m} z_j$ maximal ist. Natürlich lassen sich solche Belegungen nicht so gut interpretieren und schon gar nicht direkt in eine Belegung für die MAX-k-SAT-Instanz zurückübersetzen. Wo liegt also der Sinn?

Zum einen halten wir fest, dass es für die lineare Programmierung Polynomialzeitalgorithmen gibt, die eine optimale Lösung berechnen. Wir können also für das relaxierte Problem deterministisch in Polynomialzeit eine optimale Belegung berechnen. Jetzt werden etwas kreativ aus dieser Belegung eine Belegung für die MAX-k-SAT-Instanz machen. Die Idee ist naheliegend: Wenn eine y_i-Variable mit einem kleinen Wert (nahe 0) belegt ist, dann ist es vermutlich günstig, die korrespondierende x_i-Variable mit 0 zu belegen. Analog dazu ist es vermutlich günstig, die x_i-Variable mit 1 zu belegen, wenn y_i einen großen Wert (nahe 1) hat. Wir konkretisieren diese Idee im folgenden randomisierten Algorithmus. Die dabei verwendete Technik bezeichnet man als *randomisiertes Runden*.

Algorithmus 10.5.
1. *Formuliere zur MAX-k-SAT-Instanz das lineare Programm.*
2. *Berechne eine optimale Lösung* $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_n, \hat{z}_1, \hat{z}_2, \ldots, \hat{z}_m$ *dazu.*
3. *Für* $i \in \{1, 2, \ldots, n\}$
4. *Mit Wahrscheinlichkeit* $\hat{y}_i$ *setze* $b[i] := 1$ *sonst setze* $b[i] := 0$.
5. *Ausgabe* b

Theorem 10.6. *Algorithmus 10.5 berechnet zu einer reduzierten MAX-k-SAT-Instanz mit m-Klauseln über n Variablen in Polynomialzeit eine Belegung, die jede einzelne Klausel* c_j *mit k Literalen wird mit Wahrscheinlichkeit mindestens*

$$\left(1 - \left(1 - \frac{1}{k}\right)^k\right) \cdot \hat{z}_j$$

erfüllt.
Im Erwartungswert werden insgesamt mindestens $\left(1 - (1 - 1/k)^k\right) \cdot OPT$ *Klauseln erfüllt, wenn OPT die maximal gleichzeitig erfüllbare Anzahl der Klauseln angibt.*
Algorithmus 10.5 berechnet zu einer MAXSAT-Instanz mit m-Klauseln über n Variablen in Polynomialzeit eine Belegung, die im Erwartungswert mindestens $(1 - e^{-1}) \cdot OPT$ *Klauseln erfüllt, wenn OPT die maximal gleichzeitig erfüllbare Anzahl der Klauseln angibt.*

Vor dem Beweis des Theorems ist es nützlich, zwei technische Hilfsaussagen festzuhalten, die uns bei den Abschätzungen helfen. Wir machen das zuerst, damit wir uns später ganz auf die wesentliche Idee im Beweis von Theorem 10.6 konzentrieren können und nicht von Details der Rechnungen abgelenkt werden. Beide Aussagen erfordern etwas Analysis für ihren Beweis, wir ersparen uns das an dieser Stelle und notieren die Resultate ohne Beweis.

Lemma 10.7.

$$\forall a_1, a_2, \ldots, a_k \in \mathbb{R}_0^+ \colon \prod_{i=1}^{k} a_i \leq \left(\frac{\sum_{i=1}^{k} a_i}{k} \right)^k$$

Lemma 10.8.

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

Beweis von Theorem 10.6. Wir betrachten eine Klausel c_j und nehmen an, dass $c_j = x_1 \vee x_2 \vee \cdots \vee x_k$ gilt. Sollten andere Variablen vorkommen, können wir umbenennen, sollte eine Variable x_i negiert vorkommen, können wir im Folgenden $1 - \hat{y}_i$ durch $\hat{y}_i$ ersetzen und entsprechend $\hat{y}_i$ durch $1 - \hat{y}_i$, ohne dass sich etwas wesentlich ändert.
Wir fragen uns, mit welcher Wahrscheinlichkeit die zufällig gewählte Belegung b diese Klausel c_j erfüllt. Dank unserer Annahmen haben wir

$$\text{Prob}\,(c_j \text{ nicht erfüllt}) = \prod_{i=1}^{k} (1 - \hat{y}_i)$$

und entsprechend

$$\text{Prob}\,(c_j \text{ erfüllt}) = 1 - \prod_{i=1}^{k} (1 - \hat{y}_i).$$

Das lineare Programm garantiert uns, dass

$$\sum_{i=1}^{k} \hat{y}_i \geq \hat{z}_j$$

gilt, so dass

$$\sum_{i=1}^{k} (1 - \hat{y}_i) \leq k - \hat{z}_j$$

folgt. Wir wenden Lemma 10.7 an und haben

$$\begin{aligned} \prod_{i=1}^{k} (1 - \hat{y}_i) &\leq \left(\frac{\sum_{i=1}^{k} (1 - \hat{y}_i)}{k} \right)^k \\ &\leq \left(\frac{k - \hat{z}_j}{k} \right)^k = \left(1 - \frac{\hat{z}_j}{k}\right)^k, \end{aligned}$$

woraus offensichtlich

$$\text{Prob}\,(c_j \text{ erfüllt}) = 1 - \prod_{i=1}^{k} (1 - \hat{y}_i) \geq 1 - \left(1 - \frac{\hat{z}_j}{k}\right)^k \geq 1 - \left(1 - \frac{1}{k}\right)^k \cdot \hat{z}_j$$

folgt, wobei wir für die letzte Ungleichung Lemma 10.8 benutzt haben. Damit ist die erste Teilaussage bewiesen.
Für den Beweis der beiden anderen Teilaussagen bemerken wir zunächst, dass $1 - (1 - 1/k)^k \geq 1 - e^{-1}$ gilt. Wir haben

$$\begin{aligned}
\text{E}\,(\#\text{erfüllte Klauseln}) &= \sum_{j=1}^{m} \text{Prob}\,(c_j \text{ erfüllt}) \geq \sum_{j=1}^{m} 1 - \left(1 - \frac{1}{k}\right)^k \cdot \hat{z}_j \\
&\geq \sum_{j=1}^{m} \left(1 - \left(1 - \frac{1}{k}\right)^k\right) \cdot \hat{z}_j \\
&\geq \sum_{j=1}^{m} \left(1 - e^{-1}\right) \cdot \hat{z}_j = \left(1 - e^{-1}\right) \cdot \sum_{j=1}^{m} \hat{z}_j
\end{aligned}$$

und erinnern uns daran, dass das lineare Programm eine Relaxierung des ganzzahligen linearen Programms ist, das die MAX-k-SAT-Instanz exakt beschreibt. Die Relaxierung ist sicher nicht schwerer zu maximieren, so dass $\sum_{j=1}^{m} \hat{z}_j \geq \text{OPT}$ gilt, wenn OPT die maximale Anzahl gleichzeitig erfüllbarer Klauseln angibt. Wir haben also insgesamt

$$\text{E}\,(\#\text{erfüllte Klauseln}) \geq \left(1 - e^{-1}\right) \cdot \text{OPT}$$

wie behauptet. Lassen wir die Abschätzung durch $1 - e^{-1}$ aus, erhalten wir die zweite Teilaussage. □

Wir haben jetzt zwei verschiedene randomisierte Algorithmen für MAX-k-SAT, der eine erfüllt eine Klausel c_j mit Wahrscheinlichkeit $1-2^{-k}$, der andere erfüllt eine Klausel c_j mit Wahrscheinlichkeit mindestens $\left(1 - (1 - 1/k)^k\right) \cdot \hat{z}_j$. Es ist interessant zu beobachten, dass die erste Wahrscheinlichkeit streng monoton mit k wächst, die andere Wahrscheinlichkeit streng monoton mit k fällt. Algorithmus 10.1 ist also gut für lange Klauseln, Algorithmus 10.5 ist gut für kurze Klauseln. Es liegt darum durchaus nahe, die beiden Algorithmen miteinander zu kombinieren und zu hoffen, dass man so insgesamt einen besseren MAXSAT-Algorithmus bekommt.

Algorithmus 10.9.

1. *Berechne Belegung a_1 mit Algorithmus 10.1.*
2. *Berechne Belegung a_2 mit Algorithmus 10.5.*
3. *Gib die Belegung aus, die mehr Klauseln erfüllt.*

Theorem 10.10. *Algorithmus 10.9 berechnet zu einer MAXSAT-Instanz, in der höchstens OPT Klauseln gleichzeitig erfüllt werden können, in Polynomialzeit eine Belegung, in der im Erwartungswert mindestens* $(3/4) \cdot OPT$ *Klauseln gleichzeitig erfüllt sind.*

Beweis. Sei A_1 die Anzahl der Klauseln, die mit Belegung a_1 erfüllt sind, sei A_2 die Anzahl der Klauseln, die mit Belegung a_2 erfüllt sind. Algorithmus 10.9 erfüllt $\max\{A_1, A_2\}$ viele Klauseln und wir wollen zeigen, dass $\mathrm{E}(\max\{A_1, A_2\}) \geq (3/4) \cdot \mathrm{OPT}$ gilt.
Natürlich gilt $\max\{A_1, A_2\} \geq (A_1 + A_2)/2$ für beliebige natürliche Zahlen A_1 und A_2. Es genügt also, $\mathrm{E}((A_1 + A_2)/2) \geq (3/4) \cdot \mathrm{OPT}$ zu beweisen.
Wir betrachten eine Klausel c_j mit Länge l_j. Wir wissen, dass Belegung a_1 die Klausel c_j mit Wahrscheinlichkeit $1-2^{-l_j}$ erfüllt und dass Belegung a_2 die Klausel c_j mit Wahrscheinlichkeit mindestens $1-(1-1/l_j)^{l_j}$ erfüllt. Darum ist

$$\mathrm{E}(A_1) = \sum_{j=1}^{m} \left(1 - 2^{-l_j}\right) \geq \sum_{j=1}^{m} \left(1 - 2^{-l_j}\right) \cdot \hat{z}_j$$

und

$$\mathrm{E}(A_2) \geq \sum_{j=1}^{m} \left(1 - (1 - 1/l_j)^{l_j}\right) \cdot \hat{z}_j.$$

Wir haben also

$$\begin{aligned} \mathrm{E}\left(\frac{A_1 + A_2}{2}\right) &= \frac{1}{2} \cdot (\mathrm{E}(A_1) + \mathrm{E}(A_2)) \\ &\geq \frac{1}{2} \cdot \left(\sum_{j=1}^{m} \left(\left(1 - 2^{-l_j}\right) + \left(1 - (1 - 1/l_j)^{l_j}\right)\right) \cdot \hat{z}_j\right), \end{aligned}$$

und es genügt,

$$\left(1 - 2^{-l_j}\right) + \left(1 - (1 - 1/l_j)^{l_j}\right) \geq \frac{3}{2}$$

nachzuweisen. Das Wachstumsverhalten hatten wir uns schon vorab überlegt, wir begnügen uns darum mit einem Blick auf die folgende Tabelle, in der wir die Einträge nach der dritten Nachkommastelle runden. Weil $1-(1-1/l_j)^{l_j}$ gegen $1 - e^{-1} > 0{,}63$ konvergiert und $1 - 2^{-l_j}$ schon für $l_j \geq 4$ groß genug ist, folgt damit die Behauptung.

l_j	$1 - 2^{-l_j}$	$1 - (1 - 1/l_j)^{l_j}$	$\left(1 - 2^{-l_j}\right) + \left(1 - (1 - 1/l_j)^{l_j}\right)$
1	0,5	1,0	1,5
2	0,75	0,75	1,5
3	0,875	0,704	1,579
4	0,938	0,684	1,622
5	0,969	0,672	1,641

□

Wir haben also nicht nur zwei randomisierte Algorithmen, von denen der eine gut für MAX-k-SAT-Instanzen mit großem k und der andere gut für MAX-k-SAT-Instanzen mit kleinem k funktioniert. Die einfache Kombination beider Algorithmen liefert insgesamt einen randomisierten Algorithmus, der gut für beliebige MAXSAT-Instanzen funktioniert, in denen die Länge der Klauseln beliebig gemischt sein darf.

10.3 Erfüllbarkeitsprobleme exakt lösen

Weil die MAXSAT-Probleme, mit denen wir uns beschäftigt haben, NP-schwierig sind, haben wir uns mit Approximationen und zum Teil sogar mit randomisierten Approximationen zufriedengegeben. Wir haben auch einen randomisierten Algorithmus, der in Polynomialzeit im Erwartungswert mindestens eine (4/3)-Approximation berechnet, gefunden. Es sind Situationen denkbar, in denen eine (4/3)-Approximation nicht ausreicht. Wir brauchen uns leider nicht auf die Suche nach einem PTAS zu machen, wenn P $\neq$ NP gilt, gibt es kein PTAS für MAXSAT.
Natürlich können wir einfach alle 2^n Belegungen ausprobieren und die beste aussuchen. Für nicht zu große Werte von n mag das ein akzeptables Vorgehen sein. Aber schon für nur moderat große Werte von n (zum Beispiel $n = 40$) wird das vollständige Ausprobieren sehr langsam. Wir werden uns hier auf das bekannte Entscheidungsproblem 3-SAT beschränken, wir haben also eine Eingabe wie für MAX-3-SAT und wollen entscheiden, ob alle m Klauseln gleichzeitig erfüllbar sind. Wir werden einen randomisierten Algorithmus erarbeiten, der für erfüllbare 3-SAT-Instanzen in erwarteter exponentieller Zeit eine erfüllende Belegung ausgibt. Die wesentliche Verbesserung gegenüber dem naiven Ausprobieren wird sein, dass wir eine erwartete Laufzeit von $O(\text{poly}(n) \cdot b^n)$ nachweisen werden für ein Polynom p und eine Konstante $b \in]1;2[$, die erheblich kleiner als 2 sein wird. Zum Abschluss dieses Kapitels werden wir beispielhaft durchrechnen und erkennen, dass der dadurch erzielte Rechenzeitgewinn erheblich ist.
Um uns mit der Technik, die wir verwenden wollen, vertraut zu machen, beginnen mit mit dem Entscheidungsproblem 2-SAT, bei dem also die Klausellänge 2 ist. Wir wollen einen randomisierten Algorithmus finden, der zu einer erfüllbaren 2-SAT-Instanz eine erfüllende Belegung ausgibt. Der Algorithmus sollte zumindest erwartete polynomielle Laufzeit haben, es gilt ja bekanntlich 2-SAT $\in$ P, das Problem ist also sogar deterministisch in Polynomialzeit lösbar. Man kann sogar deterministisch in Polynomialzeit entscheiden, ob es eine erfüllende Belegung gibt; unser Algorithmus wird für

unerfüllbare Instanzen keine hilfreichen Informationen ausgeben. Es bleibt offen, ob es keine erfüllende Belegung gibt oder mit der Zeit nicht doch noch eine gefunden wird.
Bevor wir uns an den Entwurf eines randomisierten Algorithmus für 2-SAT machen, werden wir uns einen Zufallsprozess, den man Random Walk nennt, ansehen und die erwartete Zeit, bis er anhält, analysieren. Wir werden für die Analyse den Begriff der *Stoppzeit* benötigen; anschaulich gesprochen ist eine Zeit T eine Stoppzeit, wenn sie nicht von zukünftigen Ereignissen abhängt, wenn also das Wissen über alle Ereignisse bis zum Zeitpunkt t ausreicht, um zu entscheiden, ob $T \leq t$ gilt. Wir fassen den Begriff auch formal.

Definition 10.11. *Eine Zufallsvariable* $T\colon \Omega \to \mathbb{N}_0 \cup \{\infty\}$ *heißt* Stoppzeit *eines Zufallsprozesses* X_0, X_1, X_2, ..., *wenn für alle* $i \in \mathbb{N}_0$ *das Ereignis* $T = n$ *basierend auf* X_0, X_1, ..., X_n *ausgedrückt werden kann.*

Kommen wir jetzt zum Zufallsprozess $X_0, X_1, X_2, \ldots$, den wir uns näher ansehen können. Wir haben $X_i \in \mathbb{Z}$ für alle $i \in \mathbb{N}_0$, setzen $X_0 := a$ für ein festes $a \in \{0, 1, \ldots, n\}$, haben $X_{t+1} \in \{X_t - 1, X_t + 1\}$ mit $\text{Prob}\,(X_{t+1} = X_t - 1) = \text{Prob}\,(X_{t-1} = X_t + 1) = 1/2$ für alle $t \in \mathbb{N}_0$. Man kann den Zufallsprozess X_0, X_1, X_2, ... als zufällige Bewegung eines Partikels auf dem Zahlenstrahl auffassen: Man startet deterministisch bei a und bewegt sich dann in jedem Schritt mit gleicher Wahrscheinlichkeit $1/2$ um 1 in Richtung größerer oder kleinerer Zahlen. Weil man sich mit gleicher Wahrscheinlichkeit in beide Richtungen bewegt, spricht man auch von einem *fairen Random Walk*. Offensichtlich ist der Prozess als Markow-Kette beschreibbar: Es hängt nur vom aktuellen Zustand X_t ab, wie die Wahrscheinlichkeitsverteilung zum Zeitpunkt $t+1$ aussieht. Wir interessieren uns für den ersten Zeitpunkt, zu dem wir entweder $-n$ oder n erreichen, also für die Zufallsvariable $T := \min\{t \mid |X_t| = n\}$ gilt. Wir sehen direkt, dass T eine Stoppzeit ist.
Wir wollen die erwartete Stoppzeit $\text{E}\,(T)$ bestimmen. Dabei wird uns ein zentrales Resultat über Martingale helfen, das wir an dieser Stelle nur zitieren und nicht selbst herleiten wollen. Zunächst definieren wir formal, wann ein Zufallsprozess ein Martingal ist.

Definition 10.12. *Ein Zufallsprozess* Y_0, Y_1, Y_2, ... *mit* $Y_i \in \mathbb{R}$ *für alle* $i \in \mathbb{N}_0$ *heißt* Martingal *in Bezug auf einen Zufallsprozess* X_0, X_1, X_2, ..., *wenn für alle* $n \in \mathbb{N}_0$ *die folgenden drei Aussagen gelten.*

1. Y_n *ist eine Funktion von* X_0, X_1, ..., X_n.
2. $E(|Y_n|) < \infty$ *oder* $Y_n \geq 0$
3. $E(Y_{n+1} \mid X_0, X_1, \ldots, X_n) = Y_n$

Ein Martingal ist also ein Zufallsprozess (Y_i), der als Funktion eines anderen Zufallsprozesses (X_i) beschrieben werden kann, dabei kann Y_n von der gesamten Historie von (X_i) bis zum Zeitpunkt n abhängen, der Zufallsprozess (Y_i) muss also insbesondere *nicht* der Markow-Eigenschaft genügen. Wir verlangen, dass der Zufallsprozess (Y_i) nicht beliebig in Richtung negativer Zahlen „wegläuft“. Die zentrale Eigenschaft ist aber die dritte: Im Durchschnitt ändert sich der Wert von Y_i in einem Schritt gar nicht. Wer reitet, kennt ein Martingal vielleicht als einen beim Springreiten benutzten Zügel. Das ist nicht unpassend, dank dieser dritten Eigenschaft ist ein Martingal ein „gezügelter“ Zufallsprozess. Wir interessieren uns für Martingale, weil sie sich in vielen Zufallsprozessen wiederfinden lassen und es eine sehr hilfreiche Aussage gibt, mit der man Stoppzeiten berechnen kann.

Theorem 10.13 (**Das Optional-Stopping-Theorem**). *Sei $(Y_i)_{i\geq 0}$ ein Martingal in Bezug auf X_0, X_1, X_2, ..., sei T eine Stoppzeit von X_0, X_1, X_2, ...*
Wenn es ein $k \in \mathbb{N}_0$ gibt, so dass fast sicher $T \leq k$ gilt oder $T < \infty$ gilt und es ein $k \in \mathbb{N}_0$ gibt, so dass $|Y_t| \leq k$ für alle $t < T$ fast sicher gilt, dann ist $E(Y_T) = E(Y_0)$.

Wir zitieren das Optional-Stopping-Theorem wie gesagt ohne Beweis. Wenn wir die eher technischen Einschränkungen außer Acht lassen, sagt es schlicht aus, dass sich der Zufallsprozess (Y_i), der sich ja im Erwartungswert in einem Schritt gar nicht bewegt, weil er ein Martingal ist, sich in der gesamten Zeit bis T im Erwartungswert gar nicht bewegt. Die Aussage ist also nicht so sehr überraschend. Wie hilfreich sie ist, werden wir jetzt bei der Bestimmung der Stoppzeit T unseres fairen Random Walk feststellen.

Theorem 10.14. *Betrachte $(X_i)_{i\in\mathbb{N}_0}$ mit $X_0 := a$ für ein festes $a \in \{0, 1, \ldots, n\}$, $X_{t+1} \in \{X_t - 1, X_t + 1\}$ mit*

$$\mathit{Prob}\,(X_{t+1} = X_t - 1) = \mathit{Prob}\,(X_{t+1} = X_t + 1) = 1/2$$

für alle $t \in \mathbb{N}_0$, $T := \min\{t \mid |X_t| = n\}$. Dann ist $E(T) = (n - a) \cdot (n + a)$.

Beweis. Betrachte $(Y_i)_{i\in\mathbb{N}_0}$ mit $Y_n := X_n^2 - n$. Wir behaupten zunächst, dass $(Y_i)_{i\in\mathbb{N}_0}$ ein Martingal in Bezug auf $(X_i)_{i\in\mathbb{N}_0}$ ist. Dass Y_n eine Funktion von $X_0, X_1, \ldots, X_n$ ist, ist offenbar wahr. Wir haben außerdem $|Y_n| \leq X_n^2 + n \leq (a + n)^2 + n < \infty$. Es bleibt nur noch die zentrale Martingaleigenschaft

nachzurechnen.

$$\begin{aligned}
\mathrm{E}\left(Y_{n+1} \mid X_0, X_1, \ldots, X_n\right) &= \mathrm{E}\left(X_{n+1}^2 - n - 1 \mid X_0, X_1, \ldots, X_n\right) \\
&= \mathrm{E}\left(X_{n+1}^2 \mid X_n\right) - n - 1 \\
&= \frac{1}{2} \cdot (X_n + 1)^2 + \frac{1}{2} \cdot (X_n - 1)^2 - n - 1 \\
&= X_n^2 + 1 - n - 1 = X_n^2 - n = Y_n
\end{aligned}$$

Wir wollen das Optional-Stopping-Theorem anwenden uns überzeugen uns zunächst davon, dass das möglich ist. Wir haben $T < \infty$, weil wir immer mit einer Folge von höchstens n Bewegungen in die gleiche Richtung eines der beiden Enden $-n$ oder n erreichen. Eine solche Schrittfolge hat Wahrscheinlichkeit 2^{-n}, also ist

$$\mathrm{Prob}\left(T > t\right) < \left(1 - 2^{-n}\right)^{\lfloor t/n \rfloor}$$

für alle $t \in \mathbb{N}_0$. Außerdem haben wir uns schon oben überlegt, dass $|Y_t| \leq n^2 + n + a$ gilt, so dass wir mit $k := n^2 + n + a$ die Voraussetzungen für die Anwendungen des Optional-Stopping-Theorems erfüllt haben.
Es gilt also einerseits

$$\mathrm{E}\left(Y_T\right) = \mathrm{E}\left(Y_0\right) = \mathrm{E}\left(X_0^2 - 0\right) = \mathrm{E}\left(a^2\right) = a^2$$

und andererseits

$$\mathrm{E}\left(Y_T\right) = \mathrm{E}\left(X_T^2 - T\right) = \mathrm{E}\left(X_T^2\right) - \mathrm{E}\left(T\right) = \mathrm{E}\left(n^2\right) - \mathrm{E}\left(T\right) = n^2 - \mathrm{E}\left(T\right),$$

so dass $a^2 = n^2 - \mathrm{E}\left(T\right)$ folgt und wir schließlich

$$\mathrm{E}\left(T\right) = n^2 - a^2 = (n - a) \cdot (n + a)$$

haben. □

Bevor wir fortfahren, schauen wir uns noch einen sehr ähnlichen fairen Random Walk an. Sei $X_0 := i$ mit $i \in \{0, 1, \ldots, n\}$, $X_{t+1} \in \{X_t - 1, X_t + 1\}$ mit

$$\mathrm{Prob}\left(X_{t+1} = X_t - 1\right) = \begin{cases} \frac{1}{2} & \text{für } 0 < X_t < n \\ 1 & \text{für } X_t = n \\ 0 & \text{sonst} \end{cases}$$

und

$$\mathrm{Prob}\left(X_{t+1} = X_t + 1\right) = \begin{cases} \frac{1}{2} & \text{für } 0 < X_t < n \\ 1 & \text{für } X_t = 0 \\ 0 & \text{sonst,} \end{cases}$$

sei $T := \min\{t \mid X_t = 0\}$. Der Random Walk kann also über 0 und n nicht hinauslaufen, die beiden Grenzen nennt man *reflektierend*. Wir interessieren uns für den ersten Zeitpunkt, zu dem die Grenze 0 erreicht wird. Wir brauchen zum Glück zur Bestimmung von $\mathrm{E}(T)$ nicht neu zu rechnen. Scharfes Hinsehen offenbart, dass dieser Random Walk zu dem in Theorem 10.14 diskutierten Random Walk äquivalent ist. Gedanklich können wir den Zahlenstrahl zwischen $-n$ und n genau bei 0 falten und übereinanderlegen. Dann entspricht die 0 gerade n und n gerade der 0. Der Startzustand a übersetzt sich in den Startzustand $i = n - a$, so dass sich $\mathrm{E}(T) = i \cdot (2n - i)$ ergibt.
Nach dieser ganzen Vorarbeit kommen wir jetzt zu einem erstaunlich einfachen Algorithmus für 2-SAT, in dem wir den fairen Random Walk ein Stück weit wiedererkennen werden. Der Algorithmus erhält eine 2-SAT-Instanz über n Variablen und einen Parameter $T \in \mathbb{N} \cup \{\infty\}$ als Eingabe.

Algorithmus 10.15.
1. *Für alle* $i \in \{1, 2, \ldots, n\}$ *setze* $b[i] := 0$.
2. *Für alle* $t \in \{1, 2, \ldots, T\}$
3. *Falls* b *alle Klauseln erfüllt, gib* b *aus. STOP.*
4. *Wähle eine Klausel* c_j*, die unter* b *nicht erfüllt ist.*
5. *Wähle uniform zufällig ein Literal aus* c_j*.*
6. *Invertiere die zu diesem Literal gehörige Stelle in* b*.*
7. *Ausgabe „Es gibt vermutlich keine erfüllende Belegung.“*

Theorem 10.16. *Für eine erfüllbare 2-SAT-Instanz über* n *Variablen liefert Algorithmus 10.15 mit* $T = \infty$ *im Durchschnitt nach höchstens* n^2 *Schleifendurchläufen eine erfüllende Belegung.*

Beweis. Weil die 2-SAT-Instanz erfüllbar ist, gibt es eine erfüllende Belegung b^*. Wir bezeichnen mit $d(b, b^*) := \sum_{i=1}^{n} |b[i] - b^*[i]|$ den Hammingabstand von b und b^*. Natürlich gilt $0 \leq d(b, b^*) \leq n$ und $d(b, b^*) = 0$ impliziert, dass $b = b^*$ gilt und b erfüllend ist.
Die zentrale Beobachtung ist, dass sich $d(b, b^*)$ im Verlauf des Algorithmus nicht ungünstiger als der Random Walk auf $\{0, 1, \ldots, n\}$ verhält: Wenn wir eine unter b nicht erfüllte Klausel c_j haben, können in b^* nicht beide Variablen, die in c_j vorkommen, so belegt sein wie in b. Andernfalls wäre c_j auch unter b^* nicht erfüllt und b^* nicht erfüllend. Wenn wir also uniform zufällig eine Variable auswählen und invertieren, so verringern wir $d(b, b^*)$ mit Wahrscheinlichkeit mindestens $1/2$. Es ist möglich, dass beide Variablen in b^* anders als in b belegt sind und wir den Hammingabstand sogar mit Wahrscheinlichkeit 1 verringern. Das meinten wir, als wir „nicht ungünstiger“ sagten. Wenn der initiale Hammingabstand i beträgt, so ist die erwartete Dauer

des Random Walks, bis erstmals $d(b, b^*) = 0$ gilt, durch $i \cdot (2n - i)$ nach oben beschränkt. Weil $0 \leq i \leq n$ gilt, haben wir wie behauptet n^2 als obere Schranke für die erwartete Länge des Random Walks. □

Korollar 10.17. *Für jedes (nicht notwendig konstante) ε mit $0 < \varepsilon < 1$ kann man durch unabhängige Wiederholung von Algorithmus 10.15 einen Algorithmus erhalten, der mit $O(-\log(\varepsilon) \cdot n^2)$ Runden von Algorithmus 10.15 auskommt und für eine erfüllbare 2-SAT-Instanz über n Variablen nur mit Wahrscheinlichkeit höchstens ε die Ausgabe „Es gibt vermutlich keine erfüllende Belegung“ erzeugt.*

Beweis. Die erwartete Anzahl an Runden, bis Algorithmus 10.15 eine erfüllende Belegung ausgibt, ist durch n^2 nach oben beschränkt. Anwendung der Markow-Ungleichung ergibt, dass Algorithmus 10.16 mit $T = 2n^2$ mit Wahrscheinlichkeit mindestens $1/2$ eine erfüllende Belegung ausgibt. Wir wiederholen Algorithmus 10.15 mit $T = 2n^2$ unabhängig $\lceil -\log \varepsilon \rceil$-mal. Die Wahrscheinlichkeit, nie eine erfüllende Belegung zu finden, ist dann durch

$$2^{-\lceil -\log \varepsilon \rceil} \leq \varepsilon$$

nach oben beschränkt. □

Wir wollen diese Idee jetzt auf 3-SAT übertragen und können dazu sogar Algorithmus 10.15 unverändert übernehmen. Das Ergebnis, das wir damit erzielen, unterscheidet sich aber natürlich.

Theorem 10.18. *Für eine 3-SAT-Instanz mit erfüllender Belegung b^* findet Algorithmus 10.15 mit $T = 3n + 1$ mit Wahrscheinlichkeit mindestens*

$$\frac{1}{3n+1} \cdot \left(\frac{1}{2}\right)^{d(0^n, b^*)}$$

eine erfüllende Belegung.

Beweis. Wir beobachten, dass die Wahrscheinlichkeit, den Hammingabstand zur erfüllenden Belegung b^* in einem Schritt um 1 zu verkleinern, durch $1/3$ nach unten beschränkt ist und entsprechend die Wahrscheinlichkeit, den Hammingabstand zur erfüllenden Belegung b^* um 1 zu vergrößern, durch $2/3$ nach oben beschränkt ist. Es bezeichne $S^+(t)$ die Anzahl der Schritte unter den ersten t Schritten, die den Hammingabstand zwischen b und b^* vergrößern. Entsprechend bezeichne $S^-(t)$ die Anzahl der Schritte unter den

ersten t Schritten, die den Hammingabstand zwischen b und b^* verkleinern. Mit dieser Notation haben wir:

$$\begin{aligned}
& \text{Prob}\,(\text{finde erfüllende Belegung in } 3n+1 \text{ Schritten}) \\
\geq\ & \text{Prob}\,(\text{finde erfüllende Belegung } b^* \text{ in } 3n+1 \text{ Schritten}) \\
\geq\ & \text{Prob}\,(\text{finde erfüllende Belegung in } 3d\,(0^n, b^*) \text{ Schritten}) \\
\geq\ & \text{Prob}\left(\left(S^-\left(3d\,(0^n,b^*)\right) \geq 2d\,(0^n,b^*)\right) \wedge \left(S^+\left(3d\,(0^n,b^*)\right) \leq d\,(0^n,b^*)\right)\right) \\
=\ & \text{Prob}\left(S^-\left(3d\,(0^n,b^*)\right) \geq 2d\,(0^n,b^*)\right)
\end{aligned}$$

Wir erinnern uns daran, dass wir in jedem Schritt den Hammingabstand mit Wahrscheinlichkeit mindestens $1/3$ verkleinern. Wir erleichtern uns die Notation etwas und schreiben $d^* := d\,(0^n, b^*)$. Damit ist

$$\text{Prob}\left(S^-\left(3d\,(0^n,b^*)\right) \geq 2d^*\right) \geq \binom{3d^*}{2d^*} \left(\frac{1}{3}\right)^{2d^*} \left(\frac{2}{3}\right)^{d^*},$$

und wir wollen jetzt $\binom{3d^*}{2d^*}$ möglichst genau abschätzen. Wir schieben darum an dieser Stelle eine technische Hilfsaussage ein, die vielleicht auch anderswo zur Abschätzung von Binomialkoeffizienten hilfreich sein kann.

Definition 10.19. *Für $\alpha \in [0;1]$ heißt*

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

die Entropie *von α. Es ist $H(0) := H(1) := 0$.*

Lemma 10.20.

$$\forall n \in \mathbb{N}\colon \forall k \in \{1,2,\ldots,n\}\colon \frac{1}{n+1} \cdot 2^{H(k/n)\cdot n} \leq \binom{n}{k} \leq 2^{H(k/n)\cdot n}$$

Beweis. Wir betrachten n unabhängige Münzwürfe mit einer Münze, die bei jedem Wurf mit Wahrscheinlichkeit p Kopf zeigt. Bei diesen n Münzwürfen zeigt die Münze K-mal Kopf. Natürlich ist K binomialverteilt mit Parametern n und p, es ist also $\text{Prob}\,(K=k) = \binom{n}{k} p^k (1-p)^{n-k} \leq 1$. Wir wählen $p := k/n$ und haben

$$\begin{aligned}
& \text{Prob}\,(K=k) = \tbinom{n}{k} \left(\tfrac{k}{n}\right)^k \left(1-\tfrac{k}{n}\right)^{n-k} && \leq\ 1 \\
\Leftrightarrow\quad & \tbinom{n}{k}^{1/n} \left(\tfrac{k}{n}\right)^{k/n} \left(1-\tfrac{k}{n}\right)^{(n-k)/n} && \leq\ 1 \\
\Leftrightarrow\quad & \left(\tfrac{k}{n}\right)^{k/n} \left(1-\tfrac{k}{n}\right)^{(n-k)/n} && \leq\ \tbinom{n}{k}^{-1/n}.
\end{aligned}$$

Wir erinnern uns an die Definition der Entropie und sehen, dass

$$2^{-H(\alpha)} = \alpha^\alpha \cdot (1-\alpha)^{1-\alpha}$$

gilt. Wir nutzen das aus und bekommen

$$\begin{aligned}
& \left(\tfrac{k}{n}\right)^{k/n}\left(1-\tfrac{k}{n}\right)^{(n-k)/n} \leq \binom{n}{k}^{-1/n}. \\
\Leftrightarrow \quad & 2^{-H(k/n)} \leq \binom{n}{k}^{-1/n} \\
\Leftrightarrow \quad & \binom{n}{k}^{1/n} \leq 2^{H(k/n)} \\
\Leftrightarrow \quad & \binom{n}{k} \leq 2^{H(k/n)\cdot n}
\end{aligned}$$

wie behauptet. Für den Beweis des zweiten Teils der Ungleichung holen wir etwas aus. Wir wollen in einem ersten Schritt zeigen, dass

$$\mathrm{Prob}\,(K=k) = \max\{\mathrm{Prob}\,(K=i) \mid i \in \{0,1,\ldots,n\}\}$$

bei $p = k/n$ gilt. Wir erinnern daran, dass $\mathrm{E}\,(K) = n \cdot p = k$ ist. Wir behaupten also, dass bei der Binomialverteilung der Zentralterm maximale Wahrscheinlichkeit hat. Man rechnet das einfach nach: Wir haben

$$\begin{aligned}
\frac{\mathrm{Prob}\,(K=i)}{\mathrm{Prob}\,(K=i-1)} = \frac{\binom{n}{i}\left(\frac{k}{n}\right)^{i}\left(1-\frac{k}{n}\right)^{n-i}}{\binom{n}{i-1}\left(\frac{k}{n}\right)^{i-1}\left(1-\frac{k}{n}\right)^{n-i+1}} &= \frac{(n-i+1)\cdot\frac{k}{n}}{i\cdot\frac{n-k}{n}} \\
&= 1 + \frac{k(n+1)-in}{in-ik}
\end{aligned}$$

und sehen, dass $\frac{\mathrm{Prob}(K=i)}{\mathrm{Prob}(K=i-1)} < 1$ für $i > (k/n)\cdot(n+1)$ gilt, entsprechend gilt $\frac{\mathrm{Prob}(K=i)}{\mathrm{Prob}(K=i-1)} > 1$ für $i < (k/n)\cdot(n+1)$. Natürlich ist $k+1 > k\cdot(n+1)/n > k$, so dass wie behauptet $\mathrm{Prob}\,(K=k)$ für $k=i$ maximal wird.
Weil

$$\sum_{i=0}^{n}\binom{n}{i}\left(\frac{k}{n}\right)^{i}\left(1-\frac{k}{n}\right)^{n-i} = 1$$

gilt und der Summand für $i=k$ maximal ist, haben wir

$$\binom{n}{i}\left(\frac{k}{n}\right)^{k}\left(1-\frac{k}{n}\right)^{n-k} \geq \frac{1}{n+1}$$

und es folgt

$$\frac{1}{n+1}\cdot 2^{H(k/n)\cdot n} \geq \binom{n}{k}$$

analog zu unserer Rechnung vorhin. □

Wir setzen den Beweis von Theorem 10.18 fort. Wir hatten

$$\begin{aligned}
&\mathrm{Prob}\,(\text{finde erfüllende Belegung in } 3n+1 \text{ Schritten}) \\
&\qquad \geq \mathrm{Prob}\left(S^{-}\left(3d\left(0^n, b^*\right)\right) \geq 2d^*\right) \geq \binom{3d^*}{2d^*}\left(\frac{1}{3}\right)^{2d^*}\left(\frac{2}{3}\right)^{d^*}
\end{aligned}$$

und können jetzt Lemma 10.20 benutzen, um $\binom{3d^*}{2d^*}$ abzuschätzen.

$$\begin{aligned}
& \text{Prob}\left(\text{finde erfüllende Belegung in } 3n+1 \text{ Schritten}\right) \\
\geq\ & \text{Prob}\left(S^-\left(3d\left(0^n, b^*\right)\right) \geq 2d^*\right) \\
\geq\ & \binom{3d^*}{2d^*}\left(\frac{1}{3}\right)^{2d^*}\left(\frac{2}{3}\right)^{d^*} \\
\geq\ & \frac{1}{3d^*+1}\cdot 2^{H(2/3)\cdot 3d^*}\cdot\left(\frac{1}{3}\right)^{2d^*}\cdot\left(\frac{2}{3}\right)^{d^*} \\
=\ & \frac{1}{3d^*+1}\cdot 2^{H(2/3)\cdot 3d^*}\cdot\left(\left(\frac{1}{3}\right)^{1/3}\cdot\left(\frac{2}{3}\right)^{2/3}\right)^{3d^*}\cdot\left(\frac{1}{2}\right)^{d^*} \\
=\ & \frac{1}{3d^*+1}\cdot 2^{H(2/3)\cdot 3d^*}\cdot\left(2^{-H(2/3)}\right)^{3d^*}\cdot\left(\frac{1}{2}\right)^{d^*} \\
=\ & \frac{1}{3d^*+1}\cdot\left(\frac{1}{2}\right)^{d^*} \geq \frac{1}{3n+1}\cdot\left(\frac{1}{2}\right)^{d^*}
\end{aligned}$$

□

Die Wahrscheinlichkeit in Theorem 10.18 hängt schon von d^*, dem initialen Abstand zwischen der Belegung b und einer erfüllenden Belegung b^* ab. Wenn wir diesen initialen Abstand nur mit n abschätzen können, haben wir noch gar nichts gewonnen. Allerdings haben wir jetzt auch b initial fest als 0^n gewählt, was denkbar naiv ist. Wir können uns leicht im Worst-Case-Verhalten verbessern, wenn wir bei der initialen Wahl der Belegung etwas geschickter vorgehen.
Wir definieren nun eine Funktion `RandomWalk`(T, b), die genau wie Algorithmus 10.15 funktioniert, allerdings zusätzlich zur Schrittzahl T die initiale Belegung b als Parameter erhält, Zeile 1 von Algorithmus 10.15 entfällt also. Wir nutzen das zur Beschreibung einer einfachen Variante, die schon wesentlich besser als 2^n ist. Wir übergeben wieder einen Parameter T, der die Anzahl der Runden angibt.

Algorithmus 10.21.
1. *Für alle* $i \in \{1, 2, \ldots, T\}$
2. `RandomWalk`$(3n+1, 0^n)$
3. `RandomWalk`$(3n+1, 1^n)$
4. *If keine erfüllende Belegung gefunden*
Then Ausgabe „Es gibt vermutlich keine erfüllende Belegung.“

Theorem 10.22. *Für eine erfüllbare 3-SAT-Instanz über n Variablen liefert Algorithmus 10.21 mit $T = \infty$ im Erwartungswert in Zeit $O(p(n) \cdot 1{,}42^n)$ eine erfüllende Belegung.*

Beweis. In einem der beiden Aufrufe von `RandomWalk` ist der initiale Hammingabstand $d^* \leq n/2$. Wir finden darum mit Wahrscheinlichkeit

$$\geq \frac{1}{3n+1} \cdot \left(\frac{1}{2}\right)^{n/2} = \frac{1}{3n+1} \cdot \left(\frac{1}{\sqrt{2}}\right)^{n} \geq \frac{1}{3n+1} \cdot 1{,}42^{-n}$$

eine erfüllende Belegung, so dass die erwartete Anzahl von Aufrufen durch $(3n+1) \cdot 1{,}42^n$ nach oben beschränkt ist. Weil jeder Aufruf polynomielle Rechenzeit hat, ergibt sich die behauptete Schranke für die Gesamtrechenzeit. □

Die Verbesserung von 2^n auf $1{,}42^n$ ist schon erheblich: So haben wir zum Beispiel für $n = 25$, dass $2^n = 33\,554\,432$ gilt, was erheblich größer ist als $1{,}42^n \approx 6\,415$. Wir können die Basis des Exponenten sogar leicht auf $4/3$ drücken und merken an, dass $(4/3)^n \approx 1\,329$ gilt für $n = 25$.

Algorithmus 10.23.
1. *Für alle* $i \in \{1, 2, \ldots, T\}$
2. *Wähle* $b \in \{0,1\}^n$ *uniform zufällig.*
3. `RandomWalk`$(3n+1, b)$
4. *If keine erfüllende Belegung gefunden*
Then Ausgabe „Es gibt vermutlich keine erfüllende Belegung.“

Theorem 10.24. *Für eine erfüllbare 3-SAT-Instanz über n Variablen liefert Algorithmus 10.23 mit $T = \infty$ im Erwartungswert in Zeit $O(p(n) \cdot (4/3)^n)$ eine erfüllende Belegung.*

Beweis. Sei A das Ereignis, dass Algorithmus 10.23 mit einem Aufruf von `RandomWalk` eine erfüllende Belegung findet. Es gilt

$$\begin{aligned}
& \text{Prob}\,(A) \\
= & \sum_{b \in \{0,1\}^n} \text{Prob}\,(b \text{ gewählt}) \cdot \text{Prob}\,(\texttt{RandomWalk}(3n+1, b) \text{ findet erf. Belegung}) \\
\geq & \sum_{b \in \{0,1\}^n} \text{Prob}\,(b \text{ gewählt}) \cdot \frac{1}{3n+1} \cdot \left(\frac{1}{2}\right)^{d(b,b^*)} \\
= & \frac{1}{3n+1} \cdot \sum_{b \in \{0,1\}^n} \text{Prob}\,(b \text{ gewählt}) \cdot \left(\frac{1}{2}\right)^{d(b,b^*)} \\
= & \frac{1}{3n+1} \cdot \text{E}\left(\left(\frac{1}{2}\right)^{d(b,b^*)}\right),
\end{aligned}$$

und wir definieren Zufallsvariablen $X_i \in \{0,1\}$, die den Wert 1 genau dann annehmen, wenn $b[i] \neq b^*[i]$ gilt. Damit haben wir

$$\begin{aligned}
\text{Prob}\,(A) &= \frac{1}{3n+1} \cdot \text{E}\left(\left(\frac{1}{2}\right)^{d(b,b^*)}\right) \\
&= \frac{1}{3n+1} \cdot \text{E}\left(\left(\frac{1}{2}\right)^{\sum_{i=1}^{n} X_i}\right) \\
&= \frac{1}{3n+1} \cdot \text{E}\left(\prod_{i=1}^{n}\left(\frac{1}{2}\right)^{X_i}\right) \\
&= \frac{1}{3n+1} \cdot \prod_{i=1}^{n} \text{E}\left(\left(\frac{1}{2}\right)^{X_i}\right) \\
&= \frac{1}{3n+1} \cdot \prod_{i=1}^{n}\left(\text{Prob}\,(X_i = 0) + \text{Prob}\,(X_i = 1) \cdot \frac{1}{2}\right) \\
&= \frac{1}{3n+1} \cdot \prod_{i=1}^{n}\left(\frac{1}{2} + \frac{1}{4}\right) \\
&= \frac{1}{3n+1} \cdot \left(\frac{3}{4}\right)^{n}
\end{aligned}$$

wie behauptet. □

Wir haben die Basis des Exponenten jetzt auf $4/3 \approx 1{,}33333$ verkleinert. Wir werden uns jetzt noch einmal erheblich Mühe geben, um zu einer weiteren Verbesserung zu kommen. Dazu werden wir anders als bisher die initiale Belegung nicht mehr unabhängig von der gegebenen 3-SAT-Instanz wählen, sondern uns bemühen, Strukturen der aktuellen 3-SAT-Instanz auszunutzen, um eine bessere initiale Belegung zu finden. Der Ansatz klingt sinnvoll und ist sicher auch nicht unvernünftig, wir bemerken trotzdem vorab, dass wir damit die Basis nur auf 1,33026 werden senken können. Zum Vergleich sei angemerkt, dass für $n = 25$ nun $1{,}33026^n \approx 1\,254$ gilt. Für größere Werte von n werden die Unterschiede etwas eindrucksvoller, ob das in der Praxis ein Gewinn in der Rechenzeit ist, ist von vielen Faktoren abhängig und vermutlich nur empirisch zu beantworten.
Wir diskutieren die Ideen zur Verbesserung der Wahl der initialen Belegung wieder am Beispiel von Klauseln, die keine negativen Literale enthalten. Es ist wieder nicht schwierig zu sehen, wie sich die Ideen auf beliebige Klauseln übertragen lassen.

Wir betrachten eine Klausel $c_j = x_1 \vee x_2 \vee x_3$. Wenn wir die initiale Belegung zufällig gleichverteilt wählen, wie das Algorithmus 10.23 macht, wählen wir mit Wahrscheinlichkeit $1/8$ $b[1] = b[2] = b[3] = 0$ und die Klausel c_j ist nicht erfüllt. Wenn wir uns die Klauseln vorher ansehen, können wir das natürlich vermeiden, indem wir die Wahrscheinlichkeitsverteilung für die initiale Belegung ändern. Das ginge prima, wenn die Klauseln unabhängig wären. Leider sind sie das aber im Allgemeinen nicht: Verschiedene Klauseln können gleiche Variablen enthalten und das sowohl negativ als auch positiv. Was für einige Klauseln eine bessere Wahrscheinlichkeitsverteilung ist, kann für andere Klauseln dann durchaus eine schlechtere Wahrscheinlichkeitsverteilung sein und es ist unklar, ob wir so gewinnen können.

Wir geben unsere Idee hier nicht auf, sondern versuchen sie durch Präzisierung zu retten. Wir nennen zwei Klauseln *unabhängig*, wenn sie keine Variablen gemeinsam haben. Wir nennen eine Menge von Klauseln *unabhängig*, wenn die Klauseln der Menge alle paarweise unabhängig sind. Wir nennen eine Menge von Klauseln *inklusionsmaximal unabhängig*, wenn sie unabhängig ist und man keine Klausel der 3-SAT-Instanz mehr hinzufügen kann, ohne dass die Menge anschließend nicht mehr unabhängig ist. Wir können natürlich effizient eine inklusionsmaximale unabhängige Menge von Klauseln berechnen: Wir starten mit der leeren Menge, testen der Reihe nach alle Klauseln darauf, ob bei Hinzufügung die Menge unabhängig bleibt, und fügen hinzu, wenn das der Fall ist. Für die Variablen aus dieser inklusionsmaximal unabhängigen Menge von Klauseln werden wir jetzt eine andere Initialisierung beschreiben. Dabei können wir wieder davon ausgehen, dass alle Variablen in allen Klauseln nur positiv vorkommen. Wenn das nicht der Fall ist, können wir von den Variablen x_i zu den Variablen $\hat{x}_i$ übergehen und setzen $\hat{x}_i = x_i$, falls x_i positiv vorkommt und $\hat{x}_i = \overline{x_i}$, falls x_i negativ vorkommt. Die Funktion `AssignIndependentClauses` bekommt eine inklusionsmaximale unabhängige Menge von Klauseln C' und drei Parameter p_1, p_2 und p_3 übergeben, dabei gilt $p_1, p_2, p_3 \in [0; 1]$ und $3p_1 + 3p_2 + p_3 = 1$.

`AssignIndependentClauses`(C', p_1, p_2, p_3)
1. Für jede Klausel $c_j = \hat{x}_{j_1} \vee \hat{x}_{j_2} \vee \hat{x}_{j_3} \in C'$
2. Führe ein Zufallsexperiment mit drei Ausgängen durch:
3. Mit Wahrscheinlichkeit $3p_1$
4. Wähle $(b[j_1], b[j_2], b[j_3])$ uniform zufällig aus $\{(0,0,1),(0,1,0),(1,0,0)\}$.
5. Mit Wahrscheinlichkeit $3p_2$
6. Wähle $(b[j_1], b[j_2], b[j_3])$ uniform zufällig aus $\{(0,1,1),(1,0,1),(1,1,0)\}$.
7. Mit Wahrscheinlichkeit p_3
8. Wähle $(b[j_1], b[j_2], b[j_3]) = (1,1,1)$.

Wir setzen diese Funktion in einen Algorithmus ein und überlegen uns dann, was wir gewonnen haben. Klar ist, dass wir alle Klauseln aus C' mit Sicherheit erfüllen. Wir lassen die Wahl von p_1, p_2 und p_3 für den Moment noch offen und behandeln sie als Parameter des Algorithmus.

Algorithmus 10.25.
1. Für alle $i \in \{1, 2, \ldots, T\}$
2. Berechne inklusionsmaximale unabhängige Klauselmenge C'.
3. Berechne eine initiale Belegung $b \in \{0,1\}^n$ durch
`AssignIndependentClauses`(C', p_1, p_2, p_3)
Uniform zufällige Wahl für die restlichen Variablen.
4. `RandomWalk`$(3n+1, b)$
5. If keine erfüllende Belegung gefunden
Then Ausgabe „Es gibt vermutlich keine erfüllende Belegung.“

Wie groß ist die Wahrscheinlichkeit, dass ein Aufruf von `RandomWalk`$(3n+1, b)$ in Algorithmus 10.25 eine erfüllende Belegung findet? Wir nennen dieses Ereignis B und haben natürlich weiterhin

$$\text{Prob}\,(B) \geq \frac{1}{3n+1} \cdot \text{E}\left(\left(\frac{1}{2}\right)^{d(b,b^*)}\right),$$

weil sich an `RandomWalk` selbst ja nichts geändert hat. Die Abschätzung des Erwartungswertes wird jetzt aber schwieriger, weil die Variablen nun nicht mehr vollständig unabhängig gewählt werden. Zur Vereinfachung der Notation nehmen wir an, dass genau die Variablen $\hat{x}_1$, $\hat{x}_2$, …, $\hat{x}_{3|C'|}$ in Klauseln in C' vorkommen. Es ist klar, dass die Variablen $\hat{x}_i$ mit $i > 3\,|C'|$ weiterhin vollständig unabhängig gewählt werden und wir bei den anderen Variablen jeweils unabhängige Dreiergruppen identifizieren können. Wiederum zur Vereinfachung der Notation nehmen wir an, dass jeweils die Variablen $\hat{x}_{3i+1}$, $\hat{x}_{3i+2}$

und $\hat{x}_{3i+3}$ mit $i \in \{0, 1, \ldots, |C'| - 1\}$ gemeinsam gewählt werden. Wir führen jetzt für diese Dreiergruppen neue Zufallsvariablen ein, für die Dreiergruppe $\hat{x}_{3i+1}$, $\hat{x}_{3i+2}$ und $\hat{x}_{3i+3}$ ist das die Zufallsvariable $X_{3i+1,3i+2,3i+3}$ und wir definieren

$$X_{3i+1,3i+2,3i+3} := d\left(b[3i+1]b[3i+2]b[3i+3], b^*[3i+1]b^*[3i+2]b^*[3i+3]\right).$$

Damit haben wir jetzt in Analogie zu unseren früheren Rechnungen

$$d\left(b, b^*\right) = \sum_{i=0}^{|C'|-1} X_{3i+1,3i+2,3i+3} + \sum_{i=3|C'|+1}^{n} X_i$$

und können $\mathrm{E}\left(\left(\frac{1}{2}\right)^{d(b,b^*)}\right)$ ausrechnen.

$$\begin{aligned}
\mathrm{Prob}\left(B\right) &\geq \frac{1}{3n+1} \cdot \mathrm{E}\left(\left(\frac{1}{2}\right)^{d(b,b^*)}\right) \\
&= \frac{1}{3n+1} \cdot \left(\prod_{i=0}^{|C'|-1} \mathrm{E}\left(\left(\frac{1}{2}\right)^{X_{3i+1,3i+2,3i+3}}\right)\right) \cdot \left(\prod_{i=3|C'|+1}^{n} \mathrm{E}\left(\left(\frac{1}{2}\right)^{X_i}\right)\right) \\
&\geq \frac{1}{3n+1} \cdot \left(\prod_{i=0}^{|C'|-1} \mathrm{E}\left(\left(\frac{1}{2}\right)^{X_{3i+1,3i+2,3i+3}}\right)\right) \cdot \left(\frac{3}{4}\right)^{n-3|C'|}
\end{aligned}$$

Zur Bestimmung von $\mathrm{E}\left(\left(\frac{1}{2}\right)^{X_{3i+1,3i+2,3i+3}}\right)$ unterscheiden wir nach der Anzahl der Einsen im entsprechenden Teil von b^* die drei möglichen Fälle.

1. Fall $\boldsymbol{b^*[3i+1] + b^*[3i+2] + b^*[3i+3] = 3}$

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

2. Fall $\boldsymbol{b^*[3i+1] + b^*[3i+2] + b^*[3i+3] = 1}$

$$\begin{aligned}
\mathrm{E}\left(\left(\frac{1}{2}\right)^{X_{3i+1,3i+2,3i+3}}\right) &= p_1 \cdot \left(\frac{1}{2}\right)^0 + 2p_2 \cdot \left(\frac{1}{2}\right)^1 + (2p_1 + p_3) \cdot \left(\frac{1}{2}\right)^2 + p_2 \cdot \left(\frac{1}{2}\right)^3 \\
&= \frac{1}{4}p_3 + \frac{9}{8}p_2 + \frac{3}{2}p_1
\end{aligned}$$

3. Fall $b^*[3i+1] + b^*[3i+2] + b^*[3i+3] = 2$

$$\begin{aligned}\mathrm{E}\left(\left(\frac{1}{2}\right)^{X_{3i+1,3i+2,3i+3}}\right) &= p_2 \cdot \left(\frac{1}{2}\right)^0 + (p_3 + 2p_1) \cdot \left(\frac{1}{2}\right)^1 + 2p_2 \cdot \left(\frac{1}{2}\right)^2 + p_1 \cdot \left(\frac{1}{2}\right)^3 \\ &= \frac{1}{2}p_3 + \frac{3}{2}p_2 + \frac{9}{8}p_1\end{aligned}$$

Wir wollen uns das Leben jetzt etwas erleichtern und legen uns hier auf feste Werte für p_1, p_2, und p_3 fest. Wir wählen $p_1 := 4/21$, $p_2 := 2/21$, $p_3 := 3/21$ und beobachten nicht nur, dass die Wahl zulässig ist, weil $3p_1 + 3p_2 + p_3 = 1$ gilt, sondern dass wir jetzt auch

$$\mathrm{E}\left(\left(\frac{1}{2}\right)^{X_{3i+1,3i+2,3i+3}}\right) = \frac{3}{7}$$

in jedem der drei Fälle haben. Auf den ersten Blick mag 3/7 verwundern, weil der Bruch ja offenbar sogar kleiner als 3/4 ist. Wir haben hier aber immer drei Variablen zusammengefasst, so dass wir $3/7 \approx 0{,}428571$ mit $(3/4)^3 = 29/64 = 0{,}421875$ vergleichen müssen, wir haben also in der Tat etwas gewonnen für die Variablen in unseren unabhängigen Klauseln. Wir setzen das zusammen und haben

$$\begin{aligned}\mathrm{Prob}\,(B) &\geq \frac{1}{3n+1} \cdot \left(\frac{3}{4}\right)^{n-3|C'|} \cdot \left(\frac{3}{7}\right)^{|C'|} \\ &= \frac{1}{3n+1} \cdot \left(\frac{3}{4}\right)^{n} \cdot \left(\frac{64}{63}\right)^{|C'|}\end{aligned}$$

als untere Schranke für die Erfolgswahrscheinlichkeit einer Runde für Algorithmus 10.25 mit $p_1 = 4/21$, $p_2 = 2/21$ und $p_3 = 3/21$. Wenn $|C'|$ groß ist, ist das tatsächlich besser, leider könnte $|C'|$ aber auch sehr klein sein, dann haben wir asymptotisch gar nichts gewonnen.
Wir haben also einen Algorithmus, der gut ist, wenn $|C'|$ groß ist und schlecht, wenn $|C'|$ klein ist. Finden wir vielleicht noch einen Algorithmus, der für kleine $|C'|$ gut ist? Wir könnten dann beide Algorithmen kombinieren und auf einen Algorithmus hoffen, der für alle Größen von C' gut ist. Das ist in der Tat gar nicht so schwierig: Weil C' eine inklusionsmaximale unabhängige Menge ist, enthalten alle Klauseln, die nicht zu C' gehören, mindestens eine Variable, die in C' vorkommt. Wenn wir uns für eine Belegung der Variablen aus C' entschieden haben, können wir diese Variablen als konstant gesetzt betrachten. Wenn wir diese Konstantsetzungen außerhalb von C' durchführen, bleibt nur noch eine 2-SAT-Instanz übrig. Für diese können wir wie

bereits erwähnt deterministisch in Polynomialzeit eine erfüllende Belegung finden – wir erinnern daran, dass wir voraussetzen, dass es eine erfüllende Belegung gibt. Im allgemeinen Fall hätten wir hier schlechte Karten, eine Belegung, die maximal viele Klauseln der 2-SAT-Instanz gleichzeitig erfüllt, können wir dann nämlich leider nicht berechnen, MAX-2-SAT ist, wie erinnern uns, NP-schwierig. Wir definieren jetzt also den Algorithmus, den wir für kleine $|C'|$ verwenden wollen. Wir lassen die Wahl der Parameter für `AssignIndependentClauses` wieder offen.

Algorithmus 10.26.
1. `AssignIndependentClauses`(q_1, q_2, q_3)
2. *Führe die Konstantsetzungen in allen Klauseln durch.*
3. *Löse die entstandende 2-SAT-Instanz deterministisch in Polynomialzeit.*

Die Wahrscheinlichkeit, dass Algorithmus 10.26 eine erfüllende Belegung findet, können wir dadurch nach unten abschätzen, dass die in Zeile 1 gewählte Teilbelegung für b mit b^* übereinstimmt. Mit $q_1 = q_2 = q_3 = 1/7$ ist das mit Wahrscheinlichkeit $(1/7)^{|C'|}$ der Fall. Wir fügen jetzt unsere Ergebnisse abschließend zusammen.

Algorithmus 10.27.
1. *Algorithmus 10.25 mit $T = 1$ und $p_1 = 4/21$, $p_2 = 2/21$, $p_3 = 3/21$.*
2. *Algorithmus 10.26 mit $q_1 = q_2 = q_3 = 1/7$.*

Theorem 10.28. *Algorithmus 10.27 berechnet zur einer erfüllbaren 3-SAT-Instanz über n Variablen eine erfüllende Belegung mit Wahrscheinlichkeit $> 0{,}7517^n$ eine erfüllende Belegung. Er lässt sich durch Wiederholung zu einem Algorithmus mit erwarteter Laufzeit $O(poly(n) \cdot 1{,}33026^n)$ ausbauen.*

Beweis. Wir wissen, dass Algorithmus 10.27 mindestens mit Wahrscheinlichkeit

$$\frac{1}{3n+1} \cdot \max\left\{ \left(\frac{1}{7}\right)^{|C'|}, \left(\frac{3}{4}\right)^n \cdot \left(\frac{64}{63}\right)^{|C'|} \right\}$$

erfolgreich ist. Der erste Term im Maximum fällt streng monoton mit $|C'|$, der zweite Term im Maximum wächst streng monoton mit $|C'|$. Wir erhalten also das Minimum, wenn wir $|C'|$ so bestimmen, dass beide Terme den gleichen Wert annehmen. Wir haben

$$\left(\frac{1}{7}\right)^{|C'|} = \left(\frac{3}{4}\right)^n \cdot \left(\frac{64}{63}\right)^{|C'|}$$

für

$$|C'| = n \cdot \frac{\log(3/4)}{\log(63/(7 \cdot 64))} \approx 0{,}1466525n$$

und erhalten wie behauptet eine untere Schranke von mehr als $0{,}75173411^n$ als untere Schranke für die Wahrscheinlichkeit einer erfüllenden Belegung. Damit brauchen wir im Erwartungswert $O(1{,}33026^n)$ Wiederholungen bis zur ersten erfüllenden Belegung und die behauptete Laufzeitschranke folgt. □

Wir erinnern uns daran, dass wir bei der Wahl von p_1, p_2 und p_3 im Algorithmus 10.25 auf Bequemlichkeit beim Rechnen und nicht besonders gute Wahrscheinlichkeiten geachtet haben. Man kann auch an dieser Stelle noch einmal mehr Mühe investieren und damit die erwartete Laufzeit auf $O(\text{poly}(n) \cdot 1{,}330193^n)$ senken, wir ersparen uns diese marginale Verbesserung hier aber.
Hat sich die Arbeit denn jetzt insgesamt gelohnt? Wir müssen zugeben, dass die letzte Verbesserung von 4/3 auf 1,33026 eher marginal war und besser mit sportlichem Ehrgeiz als wichtiger Anwendung gerechtfertigt werden kann. Die Verbesserung von 2 als Basis auf 1,42 im ersten und 4/3 im zweiten Schritt war aber gerechtfertigt. Wenn wir vereinfachend annehmen, dass wir in einer Implementierung auf einem modernen Rechner in einer Sekunde 1 000 000 Belegungen ausprobieren können und eine 3-SAT-Instanz über 50 Variablen betrachten, dann haben wir im ersten Schritt die erwartete Rechenzeit von über 35 Jahren auf etwas unter 1 Minute und im zweiten Schritt noch weiter auf gut 1 Sekunde gedrückt; dass das für praktische Anwendungen relevant ist, sollte offensichtlich sein.

11 Schnittprobleme

Mit Schnittproblemen haben wir uns schon im Kapitel 4 beschäftigt, als wir die Korrektheit des Algorithmus von Ford und Fulkerson (Algorithmus 4.4) nachgewiesen haben: Im Max-Flow-Min-Cut-Theorem (Theorem 4.6) haben wir gezeigt, dass der Wert eines maximalen Flusses mit dem Wert eines minimalen Q-S-Schnittes übereinstimmt. Schnittprobleme sind aber nicht auf Flussnetzwerke beschränkt, auch in allgemeinen gewichteten Graphen kann man sinnvolle Schnittprobleme definieren. Ungewichtete Graphen stellen einen Spezialfall dar, sie entsprechen gewichteten Graphen, bei denen alle Kanten Gewicht 1 haben.

Ein *Schnitt* ist eine Partitionierung der Knotenmenge V eines Graphen in zwei Mengen V_1 und V_2, es gilt also $V_1 \cup V_2 = V$. Ein solcher Schnitt V_1, V_2 schneidet eine Kante $e \in E$, wenn $|e \cap V_1| = |e \cap V_2| = 1$ gilt, wenn also ein Endpunkt der Kante in V_1 und der andere Endpunkt in V_2 liegt. Der *Wert* $w(V_1, V_2)$ eines Schnitts ist die Summe der Kantengewichte aller Kanten, die geschnitten werden, es ist also $w(V_1, V_2) = \sum_{e \in \{\{u,v\} | u \in V_1, v \in V_2\}} w(e)$.

Beim Flussproblem haben wir uns für Q-S-Schnitte mit möglichst kleinem Wert interessiert. Wir nennen einen Schnitt minimal, wenn kein anderer Schnitt kleineren Wert hat. Analog dazu nennen wir einen Schnitt maximal, wenn kein anderer Schnitt größeren Wert hat. Wir nehmen für den Rest dieses Kapitels an, dass die Knotenmenge V Größe n hat, also $|V| = n$ gilt.

Min Cut ist ein Optimierungsproblem, bei dem die Eingaben gewichtete, ungerichtete Graphen sind, die Kantengewichte sind dabei natürliche Zahlen. Zulässige Lösungen sind Partitionierungen der Knotenmenge in zwei nichtleere disjunkte Mengen, der Wert des Schnitts ist zu minimieren. Natürlich ist Min Cut in $\mathcal{NPO}$. Wir wissen aus dem Kapitel über Flussprobleme (Kapitel 4), dass wir minimale Q-S-Schnitte in polynomieller Zeit berechnen können. Es gibt $\binom{n}{2}$ Möglichkeiten, zwei Knoten aus V als Quelle und Senke auszuzeichnen, wir können natürlich einen minimalen Q-S-Schnitt für jede relevante Wahl von Q und S berechnen, folglich ist Min Cut auch in $\mathcal{PO}$. An Stelle der ja nicht ganz einfachen Flussalgorithmen werden wir uns in diesem Kapitel mit einfachen randomisierten Algorithmen beschäftigen, die zumindest mit gewisser Wahrscheinlichkeit in nicht zu langer Zeit einen minimalen Schnitt berechnen. Wir müssen zugeben, dass es genau wie im Abschnitt über exakte Algorithmen für 3-SAT (Abschnitt 10.3) hier nicht um Approximation geht. Warum ist das Kapitel dann hier platziert? Das liegt an Max Cut, dem ersten Problem, mit dem wir uns in diesem Kapitel auseinandersetzen werden.

11.1 Max Cut

Max Cut ist ein Optimierungsproblem, bei dem die Eingaben ungewichtete, ungerichtete Graphen sind, zulässige Lösungen sind Partitionierungen der Knotenmenge in zwei disjunkte Mengen, der Wert des Schnitts ist zu maximieren. Natürlich ist auch Max Cut in $\mathcal{NPO}$, allerdings ist Max Cut nicht in $\mathcal{PO}$, falls P $\neq$ NP gilt. Es gehört zu der klassischen Liste von 21 Problemen, von denen Richard Karp 1972 nachgewiesen hat, dass sie NP-vollständig sind. Es ist also sinnvoll, sich um approximative Lösungen für Max Cut Gedanken zu machen. Seit 1994 kennt man dank Goemans und Williams eine 1,14-Approximation, seit 1996 weiß man dank Håstad, dass es keine 1,06-Approximation gibt, falls P $\neq$ NP gilt. Beide Ergebnisse sind nur schwierig und aufwendig zu beweisen, wir geben uns hier mit einem sehr viel schwächeren Ergebnis zur Approximation von Max Cut zufrieden, das aber den Charme besitzt, mit einem sehr einfachen Algorithmus und einem sehr einfachen Beweis auszukommen.

Algorithmus 11.1.
1. $V_1 := V_2 := \emptyset$
2. *Für* $v \in V$
3. *Mit Wahrscheinlichkeit* $1/2$ *setze* $V_1 := V_1 \cup \{v\}$
   sonst setze $V_2 := V_2 \cup \{v\}$.
4. *Ausgabe* V_1, V_2

Theorem 11.2. *Algorithmus 11.1 berechnet für einen ungewichteten Graphen in Zeit* $O(n)$ *einen Schnitt* V_1, V_2*, der im Erwartungswert* $|E|/2$ *Kanten schneidet und im Erwartungswert mindestens Güte 2 hat.*

Beweis. Die Aussage über die Laufzeit ist offensichtlich, die Aussage über die Güte folgt direkt, weil auch ein optimaler Schnitt nicht mehr als alle Kanten schneiden kann. Wir müssen also nur zeigen, dass im Erwartungswert die Hälfte der Kanten geschnitten wird.
Für jede Kante $e \in E$ definieren wir eine Indikatorvariable X_e mit

$$X_e := \begin{cases} 1 & \text{falls } |e \cap V_1| = |e \cap V_2| = 1, \\ 0 & \text{sonst.} \end{cases}$$

Offensichtlich gilt $w(V_1, V_2) = \sum_{e \in E} X_e$ und

$$\mathrm{E}\left(w(V_1, V_2)\right) = \mathrm{E}\left(\sum_{e \in E} X_e\right) = \sum_{e \in E} \mathrm{E}\left(X_e\right) = |E| \cdot \mathrm{E}\left(X_e\right)$$

folgt. Es genügt also, $\mathrm{E}(X_e)$ für eine beliebige Kante $e \in E$ zu bestimmen. Weil X_e eine Indikatorvariable ist, gilt $\mathrm{E}(X_e) = \mathrm{Prob}(X_e = 1)$ und gemäß Definition von X_e müssen wir noch noch $\mathrm{Prob}(|e \cap V_1| = |e \cap V_2| = 1)$ bestimmen. Wir schreiben $e = \{u, v\}$ und haben

$$\begin{aligned}
& \mathrm{Prob}(|e \cap V_1| = |e \cap V_2| = 1) \\
= \; & \mathrm{Prob}(((u \in V_1) \wedge (v \in V_2)) \vee ((u \in V_2) \wedge (v \in V_1))) \\
= \; & \mathrm{Prob}((u \in V_1) \wedge (v \in V_2)) + \mathrm{Prob}((u \in V_2) \wedge (v \in V_1)) \\
= \; & \mathrm{Prob}(u \in V_1) \cdot \mathrm{Prob}(v \in V_2) + \mathrm{Prob}(u \in V_2) \cdot \mathrm{Prob}(v \in V_1) \\
= \; & \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4} + \frac{1}{4} = \frac{1}{2},
\end{aligned}$$

weil die Knoten alle jeweils unabhängig mit Wahrscheinlichkeit $1/2$ einer der beiden Mengen V_1, V_2 zugeordnet werden. □

11.2 Min Cut

Wir wollen randomisierte Algorithmen für Min Cut entwerfen, die schrittweise einen Schnitt konstruieren, zu dem wir dann eine Mindestwahrscheinlichkeit bestimmen werden, mit der dieser Schnitt ein minimaler Schnitt ist. Wir setzen voraus, dass der eingegebene Graph G zusammenhängend ist. Wenn das nicht der Fall ist, zerfällt er in mindestens zwei Zusammenhangskomponenten, die wir leicht im Rahmen einer Tiefensuche deterministisch in linearer Zeit bestimmen können. Wenn es mehr als nur eine Zusammenhangskomponente gibt, hat der minimale Schnitt natürlich Wert 0 und wir brauchen uns keine weiteren Gedanken zu machen. Wir beschäftigen uns im Folgenden also nur mit dem interessanten Fall zusammenhängender Graphen. Für unsere Algorithmen wird die folgende Funktion `Kontraktion`(u, v) eine zentrale Rolle spielen, die wir mit beliebigen Knoten $u \neq v \in V$ aufrufen dürfen.

`Kontraktion`(u, v)
1. $V := (V \setminus \{u, v\}) \cup \{z\}$
2. $E := E \setminus \{u, v\}$
3. Für alle $x \in V$ mit $\{\{u, x\}, \{v, x\}\} \cap E \neq \emptyset$
4. If $|\{\{u, x\}, \{v, x\}\} \cap E| = 2$ Then
   $E := (E \setminus \{\{u, x\}, \{v, x\}\}) \cup \{\{x, z\}\}$;
   $w(\{x, z\}) := w(\{u, x\}) + w(\{v, x\})$
5. If $\{u, x\} \in E$ Then
   $E := (E \setminus \{\{u, x\}\}) \cup \{\{x, z\}\}$; $w(\{x, z\}) := w(\{u, x\})$
6. If $\{v, x\} \in E$ Then
   $E := (E \setminus \{\{v, x\}\}) \cup \{\{x, z\}\}$; $w(\{x, z\}) := w(\{v, x\})$

`Kontraktion`(u, v) verschmilzt also die beiden Knoten u und v zu einem neuen Knoten z. Eine eventuell vorhandene Kante $\{u, v\}$ verschwindet dabei. Alle anderen Kanten, an denen u oder v beteiligt sind, bleiben erhalten, dabei ersetzt natürlich der neue Knoten z die alten Knoten u und v. Die Kantengewichte werden übernommen, sollte mehr als eine Kante existiert haben, addieren sich die Kantengewichte.
Die Idee, mit deren Hilfe wir aus `Kontraktion` einen vollständigen Algorithmus machen wollen, ist leicht dargestellt. Wir kontrahieren so lange, bis nur noch zwei Knoten übrig sind. Wenn wir uns jeweils gemerkt haben, welche Knoten wir zu welchem neuen Knoten verschmolzen haben, können wir die beiden übrigbleibenden Knoten als Partition ausgeben.

1. Repeat
2. Wähle $u \neq v \in V$ *geeignet.*
3. `Kontraktion`(u, v)
4. Until $|V| = 2$
5. Gib die durch die beiden Knoten definierte Partition aus.

Natürlich hängt die Performanz dieses „Algorithmus“ (für einen richtigen Algorithmus ist das Verfahren noch zu unbestimmt) entscheidend von der Wahl der Knoten in Zeile 2 ab. Was wollen wir eigentlich erreichen? Wir betrachten gedanklich einen minimalen Schnitt V_1, V_2. Wenn wir jeweils zwei Knoten wählen, die durch eine Kante verbunden sind, die nicht vom Schnitt V_1, V_2 geschnitten wird, so haben wir am Ende den minimalen Schnitt V_1, V_2 gefunden. Wir möchten aber nicht viel Aufwand betreiben. Unser Ziel ist es vor allem, einen einfachen und schnellen Algorithmus zu haben. Dabei wollen wir aber schon vernünftig wählen. Wir wünschen uns einen minimalen Schnitt, darum ist es sicher gut, wenn wir vermeiden, Kanten mit großem Gewicht im

Schnitt zu haben. Wir wählen Knoten u, v mit größerer Wahrscheinlichkeit, wenn die Kante $\{u, v\}$ großes Gewicht hat. Dabei hängt es natürlich von der aktuellen Summe der Kantengewichte ab, was „groß“ konkret bedeutet. Diese einfachen Einsichten führen uns zu folgendem Algorithmus.

Algorithmus 11.3 (**Randomisierte Kontraktion**).
1. *Repeat*
2. *Wähle* $e = \{u, v\}$ *mit Wahrscheinlichkeit* $\frac{w(e)}{\sum_{e' \in E} w(e')}$.
3. `Kontraktion`(u, v)
4. *Until* $|V| = 2$
5. *Gib die durch die beiden Knoten definierte Partition aus.*

Lemma 11.4. *Algorithmus 11.3 hat Laufzeit* $O(n^2)$.

Beweis. Wir bemerken zunächst, dass Algorithmus 11.3 überhaupt funktioniert: Weil wir voraussetzen, dass der Graph zusammenhängend ist, gibt es eine Kante, die wir wählen können. Das Verschmelzen an dieser Kante in `Kontraktion` kann den Zusammenhang offenbar nicht zerstören, so dass wir in jeder Runde eine Kante wählen können.
Für die Laufzeit beobachten wir zunächst, dass wir initial die Summe der Kantengewichte in Zeit $O(|E|) = O(n^2)$ bestimmen können. Weil die Funktion `Kontraktion` die Anzahl der Knoten um 1 verkleinert und wir bei $|V| = 2$ abbrechen, gibt es genau $n - 2$ Aufrufe von `Kontraktion`. In jedem dieser Aufrufe werden im Wesentlichen nur die Nachbarn von u und v betrachtet, mit einem Durchlauf der Adjazenzlisten von u und v können wir also jeden Aufruf in Zeit $O(n)$ erledigen. Es ist nicht schwer, sich zu überlegen, dass die zufällige Wahl der Kante auch nicht länger braucht. Damit haben wir eine Gesamtlaufzeit von $O(n^2)$. □

Wir möchten eine untere Schranke für die Wahrscheinlichkeit bestimmen, mit der Algorithmus 11.3 einen minimalen Schnitt findet. Dazu werden wir zunächst einige Aussagen beweisen, die uns hilfreiche Einsichten in die Struktur des Problems vermitteln. Die Korrektheit aller dieser Aussagen ist intuitiv klar, die Aussagen sind auch tatsächlich nicht schwer zu beweisen. Wir verwenden die Notation $w(E') = \sum_{e \in E'} w(e)$ für alle $E' \subseteq E$, außerdem bezeichne $\text{minCut}(G)$ den Wert eines minimalen Schnittes für den Graphen G. Für einen Graphen G bezeichnen wir mit `Kontraktion`(u, v) den Graphen, der durch Aufruf von `Kontraktion`(u, v) aus G entsteht.

Lemma 11.5. *Für alle zusammenhängenden, gewichteten ungerichteten Graphen* $G = (V, E)$ *gelten die folgenden Aussagen.*

1. *Ein Schnitt V_1, V_2 ist genau dann Ausgabe von Algorithmus 11.3, wenn nie eine Kante zwischen V_1 und V_2 kontrahiert wurde.*
2. $w(E) \geq minCut(G) \cdot |V| / 2$
3. $\forall e = \{u, v\} \in E\colon minCut(G) \leq minCut(\texttt{Kontraktion}(u, v))$

Beweis. 1. Wir betrachten V_1, V_2 mit $V_1 \cup V_2 = V$. Wenn eine zwischen V_1 und V_2 verlaufende Kante kontrahiert wird, wird ein Knoten aus V_1 mit einem Knoten aus V_2 verschmolzen, so dass V_1, V_2 nicht mehr Ausgabe von Algorithmus 11.3 sein kann. Wenn andererseits bis zum Schluss niemals eine Kante zwischen V_1 und V_2 kontrahiert wurde, so verläuft die am Ende des Algorithmus übriggebliebene Kante zwischen V_1 und V_2, so dass wie behauptet der Schnitt V_1, V_2 ausgegeben wird.

2. Für einen Knoten $v \in V$ bezeichne $d_w(v) := \sum_{e=\{v,\cdot\}\in E} w(e)$. Es ist leicht einzusehen, dass $\min\{d_w(v) \mid v \in V\} \geq \text{minCut}(G)$ gilt: Sei sonst v ein Knoten mit $d_w(v) < \text{minCut}(G)$. Wir betrachten den Schnitt V_1, V_2 mit $V_1 := \{v\}$ und $V_2 := V \setminus V_1$. Offensichtlich gilt $w(V_1, V_2) = d_w(v) < \text{minCut}(G)$, ein Widerspruch.

   Wir betrachten $\sum_{v\in V} d_w(v)$ und sehen, dass wir $\sum_{v\in V} d_w(v) = 2w(E)$ haben, da in der Summe jede Kante aus E zweimal gezählt wird. Es gilt also

   $$w(E) = \frac{1}{2}\sum_{v\in V} d_w(v) \geq \frac{1}{2} \cdot |V| \cdot \text{minCut}(G)$$

   wie behauptet.

3. Wir betrachten das Ergebnis von $\texttt{Kontraktion}(u, v)$ für zwei beliebige Knoten $u \neq v$. Wir können jeden Schnitt in dem so entstandenen Graphen mit einem Schnitt gleichen Wertes in G identifizieren, wenn wir den verschmolzenen Knoten in u und v aufspalten. Also gibt es im so entstandenen Graphen keinen Schnitt, den es in G nicht auch schon gegeben hat. Das begründet schon

   $$\forall e = \{u, v\} \in E\colon \text{minCut}(G) \leq \text{minCut}(\texttt{Kontraktion}(u, v)).$$

□

Wir haben jetzt genug Problemverständnis entwickelt, um die Performanz von Algorithmus 11.3 zu analysieren. Wir formulieren und beweisen zunächst das Resultat und überlegen dann, wie zufrieden wir damit sein können.

Theorem 11.6. *Für einen zusammenhängenden, gewichteten ungerichteten Graphen $G = (V, E)$ sei V_1, V_2 ein minimaler Schnitt. Algorithmus 11.3 berechnet in Zeit $O(n^2)$ mit Wahrscheinlichkeit mindestens $2/n^2$ den Schnitt V_1, V_2.*

Beweis. Die Laufzeit haben wir im Beweis von Lemma 11.4 bewiesen. Wir wissen darum auch, dass Algorithmus 11.3 genau $(n-2)$-mal die Funktion `Kontraktion` aufruft. Das Ergebnis des i-ten Aufrufs sei der Graph $G_{i+1} = (V_{i+1}, E_{i+1})$, wir betrachten also die Folge von Graphen $G_1, G_2, \ldots, G_{n-1}$ (mit $G_1 = G$), dabei enthält der Graph G_i genau $n_i := |V| - i + 1$ Knoten. Wir schreiben $n_1 = n$, weil $|V| = n$ gilt.
Wir wissen, dass genau dann der Schnitt V_1, V_2 berechnet wird, wenn niemals eine Kante zwischen V_1 und V_2 kontrahiert wird (Lemma 11.5). Es bezeichne E' die Menge dieser Kanten, die nicht kontrahiert werden dürfen. Uns interessiert also die Wahrscheinlichkeit, dass wir in allen $n-2$ Kontraktionen keine Kante aus E' kontrahieren. Wir betrachten die $n-2$ Kontraktionen einzeln und setzen für die i-te Kontraktion voraus, dass das in den ersten $i-1$ Kontraktionen nicht passiert ist, der Schnitt V_1, V_2 also vor der i-ten Kontraktion noch mögliche Ausgabe ist. Wir möchten natürlich die Gesamtwahrscheinlichkeit, dass in allen Kontraktionen passend gewählt wird, abschätzen. Wir wissen, dass

$$\text{Prob}\left(\bigcup_{i=1}^{n} A_i\right) = \prod_{i=1}^{n} \text{Prob}\left(A_i \mid \bigwedge_{j=1}^{i-1} A_j\right)$$

gilt und defieren uns Ereignisse geschickt so, dass die Notation etwas einfacher wird. Es bezeichne A_i das Ereignis, dass in den ersten $i-1$ Kontraktionen niemals eine Kante aus E' kontrahiert wird. Wir sehen, dass das Ereignis A_i die Ereignisse $A_{i-1}, A_{i-2}, \ldots, A_1$ einschließt, es ist also $\bigwedge_{j=1}^{i-1} A_j = A_{i-1}$. Damit haben wir

$$\begin{aligned}&\text{Prob}\,(\text{Algorithmus 11.3 berechnet } V_1, V_2)\\ &\quad= \prod_{i=1}^{n-2} \text{Prob}\,(i\text{-te Kontraktion kontrahiert keine Kante aus } E' \mid A_i)\end{aligned}$$

und beobachten, dass das Ereignis A_i natürlich $\text{minCut}(G) = \text{minCut}(G_i)$ impliziert: Zumindest der minimale Schnitt V_1, V_2 ist ja erhalten geblieben. Aus $\text{minCut}(G) = \text{minCut}(G_i)$ folgt $w(E_i) \geq \text{minCut}(G) \cdot |V_i|/2$ (Lemma 11.5). Die Wahrscheinlichkeit, in der i-ten Kontraktion eine Kante aus

E' zu kontrahieren, können wir jetzt gut abschätzen. Wir haben

$$\begin{aligned}
& \text{Prob}\,(\text{kontrahiere Kante aus } E' \text{ in } i\text{-ter Kontraktion} \mid A_i) \\
= \; & \sum_{e \in E'} \text{Prob}\,(\text{kontrahiere } e \text{ in } i\text{-ter Kontraktion} \mid A_i) \\
= \; & \sum_{e \in E'} \frac{w(e)}{w(E_i)} = \frac{w(E')}{w(E_i)} = \frac{\text{minCut}(G)}{w(E_i)} \\
\leq \; & \frac{\text{minCut}(G)}{\text{minCut}(G) \cdot |V_i|/2} = \frac{2}{|V| - i + 1} = \frac{2}{n - i + 1}
\end{aligned}$$

und sehen, dass

$$\begin{aligned}
& \text{Prob}\,(\text{Algorithmus 11.3 berechnet } V_1, V_2) \\
= \; & \prod_{i=1}^{n-2} \text{Prob}\,(i\text{-te Kontraktion kontrahiert keine Kante aus } E' \mid A_i) \\
\geq \; & \prod_{i=1}^{n-2} \left(1 - \frac{2}{n-i+1}\right) = \prod_{i=1}^{n-2} \frac{n-i-1}{n-i+1} = \prod_{i=3}^{n} \frac{i-2}{i} \\
= \; & \frac{(n-2)!}{(n!)/2} = \frac{2}{n(n-1)} \geq \frac{2}{n^2}
\end{aligned}$$

folgt wie behauptet. □

Die von uns nachgewiesene untere Schranke für die Erfolgswahrscheinlichkeit konvergiert recht rasant gegen 0; damit lässt sich so natürlich kaum etwas anfangen. Was wollen wir mit einem Algorithmus, der zwar recht einfach und schnell ist, aber fast sicher nicht das gewünschte Ergebnis ausgibt?
Eine naheliegende Idee zur Verbesserung besteht darin, Algorithmus 11.3 nicht nur einmal aufzurufen, sondern unabhängige Wiederholungen durchzuführen. Dieses Verfahren haben wir in GTI bzw. TIFAI als *Probability Amplification* kennengelernt. Wir wissen schon, dass die erwartete Anzahl Wiederholungen, bis erstmalig ein minimaler Schnitt gefunden wird, durch $O(n^2)$ nach oben beschränkt ist. Wir berechnen also im Erwartungswert in Zeit $O(n^4)$ einen minimalen Schnitt. Woran erkennen wir, ob wir erfolgreich waren? Offenbar lässt sich das nicht so einfach erkennen. Wir werden darum wie folgt vorgehen: Wir legen uns auf eine feste Anzahl r unabhängiger Wiederholungen von Algorithmus 11.3 fest. Wir merken uns jeweils den ausgegebenen Schnitt und geben schließlich einen Schnitt mit kleinstem Wert aus. Mit welcher Wahrscheinlichkeit liefert uns dieser Algorithmus einen minimalen Schnitt?

Lemma 11.7. *Gibt man für einen Graphen $G = (V, E)$ einen Schnitt mit kleinstem Wert unter r Ausgaben von unabhängigen Wiederholungen von Algorithmus 11.3 aus, so ist der ausgegebene Schnitt mit Wahrscheinlichkeit mindestens $1 - (1 - 2/n^2)^r \geq 1 - e^{-(2r)/n^2}$ minimal.*

Beweis. Es ist $n = |V|$. Ein Durchlauf von Algorithmus 11.3 liefert mit Wahrscheinlichkeit mindestens $2/n^2$ einen minimalen Schnitt (Theorem 11.6). Wir haben also mit Wahrscheinlichkeit höchstens $1 - 2/n^2$ keinen minimalen Schnitt und sehen, dass in r unabhängigen Wiederholungen mit Wahrscheinlichkeit höchstens $(1 - 2/n^2)^r$ niemals ein minimaler Schnitt gefunden wird. Folglich haben wir mit Wahrscheinlichkeit mindestens $1-(1-2/n^2)^r$ mindestens einmal einen minimalen Schnitt und folglich einen minimalen Schnitt als Ausgabe. Die Abschätzung mit $1 - e^{-(2r)/n^2}$ folgt direkt aus dem Umstand, dass $e^x \geq 1 + x$ für alle x gilt. □

Mit Hilfe von Lemma 11.7 können wir direkt angeben, wie groß die Laufzeit sein sollte, damit wir mit ausreichend großer Wahrscheinlichkeit einen minimalen Schnitt finden. Es ist interessant zu beobachten, dass Laufzeit $O(n^4 \log n)$ für Erfolgswahrscheinlichkeiten, die fast beliebig nahe an 1 sind, ausreichen.

Korollar 11.8. *Durch unabhängige Wiederholungen von Algorithmus 11.3 findet man für einen Graphen $G = (V, E)$ mit $|V| = n$ und jede Konstante $c > 0$ in Zeit $O(n^4 \log n)$ einen minimalen Schnitt mit Wahrscheinlichkeit mindestens $1 - 1/n^{2c}$.*

Beweis. Wir verwenden Lemma 11.7 mit $r := \lceil cn^2 \ln n \rceil$. Die Gesamtlaufzeit ist $O(n^2 \cdot cn^2 \ln n) = O(n^4 \log n)$, die Erfolgswahrscheinlichkeit beträgt mindestens $1 - e^{-(2cn^2 \ln n)/n^2} = 1 - 1/n^{2c}$. □

Mit der Erfolgswahrscheinlichkeit $1 - 1/n^{2c}$ können wir (sogar schon für $c = 1/2$) völlig zufrieden sein, die Laufzeit $O(n^4 \log n)$ ist aber nicht so begeisternd. Wir wollen uns um eine Verbesserung der Laufzeit bemühen und überlegen deshalb, was für die lange Zeit verantwortlich ist. Es gibt offenbar zwei Faktoren: Ein Durchlauf von Algorithmus 11.3 bringt uns Zeit $\Theta(n^2)$ ein; das ist nicht besonders viel, wenn man bedenkt, dass wir den Graphen auf zwei Knoten schrumpfen und es $\Theta(n^2)$ Kanten geben kann. Der zweite Faktor ist die Anzahl der unabhängigen Wiederholungen, die mit $\Theta(n^2 \log n)$ ziemlich groß ist. Wir werden also hier versuchen, zu Verbesserungen zu kommen. Die Anzahl der Wiederholungen, die erforderlich sind für eine akzeptable Erfolgswahrscheinlichkeit, hängt natürlich alleine von der Erfolgswahrscheinlichkeit eines Durchlaufs ab. Wir werden darum noch einmal zur Analyse von Algorithmus 11.3 zurückkehren und nach Wegen Ausschau halten, die

Erfolgswahrscheinlichkeit eines Durchgangs wesentlich zu verbessern. Dabei hilft es sich klarzumachen, wie unsere Abschätzung genau entstanden ist.

Lemma 11.9. *Bricht man Algorithmus 11.3 auf einem Graphen $G = (V, E)$ mit $|V| = n$ ab, wenn die Größe der Knotenmenge bei t angekommen ist (mit $t \in \{2, 3, \ldots, n\}$), so ist mit Wahrscheinlichkeit mindestens $t \cdot (t-1)/(n \cdot (n-1))$ noch keine Kante eines festen minimalen Schnitts V_1, V_2 kontrahiert.*

Beweis. Wir gehen genau wie im Beweis von Algorithmus 11.3 vor, die Anzahl der Iterationen sinkt allerdings von $n-2$ auf $n-t$. Damit haben wir

$$\begin{aligned} & \text{Prob}\,(\text{Schnitt } V_1, V_2 \text{ noch möglich}) \\ \geq\ & \prod_{i=1}^{n-t} \left(1 - \frac{2}{n-i+1}\right) = \prod_{i=1}^{n-t} \frac{n-i-1}{n-i+1} \\ =\ & \frac{(n-2)\cdot(n-3)\cdots(t+1)\cdot t\cdot(t-1)}{n\cdot(n-1)\cdot(n-2)\cdot(n-3)\cdots(t+2)\cdot(t+1)} \\ =\ & \frac{t\cdot(t-1)}{n\cdot(n-1)} \end{aligned}$$

wie behauptet. □

Wir sehen, dass die Erfolgswahrscheinlichkeit anfangs ganz gut ist, sie aber gerade in den letzten Iterationen dann dramatisch sinkt. Es wäre darum sicher besser, den Algorithmus 11.3 nach einer nicht zu großen Anzahl von Runden abzubrechen und dann am Ende irgendwie schlauer vorzugehen.
Lemma 11.9 verrät uns, dass die Wahrscheinlichkeit, sich falsch zu entscheiden, mit abnehmender Restgraphgröße immer größer wird. Wir werden versuchen, dem Rechnung zu tragen, indem wir einen rekursiven Algorithmus vorschlagen, der in jedem Durchlauf die Knotenanzahl um einen konstanten Faktor senkt. Wir führen diese Kontraktion nicht nur einmal durch, wir machen das in zwei parallelen Aufrufen, was die Wahrscheinlichkeit eines Erfolges natürlich (geringfügig) steigert. Dadurch, dass der Algorithmus rekursiv ist, findet diese „Verzweigung“ in zwei unabhängige Versuche auf jeder Rekursionsebene statt, so dass die Gesamtanzahl von Versuchen exponentiell mit der Rekursionstiefe wächst. Das führt dazu, dass wir für die kleinen Graphen, bei denen es schwierig ist, sich gut zu entscheiden, sehr viele Versuche haben, für die großen Graphen, bei denen es weniger wahrscheinlich ist, einen Fehler zu machen, deutlich weniger. Wenn die Graphen in der Rekursion sehr klein geworden sind, brechen wir die Rekursion ab und bestimmen irgendwie deterministisch einen minimalen Schnitt. Wir werden abbrechen, wenn die Anzahl der Restknoten konstant ist, so dass es keine große Rolle

spielt, wie genau wir dann einen minimalen Schnitt bestimmen: Die Laufzeit dafür ist in jedem Fall $O(1)$. Wir stellen jetzt erst den Algorithmus vollständig vor, analysieren dann, wie groß seine Laufzeit ist, und überlegen uns im Anschluss, welche untere Schranke für die Erfolgswahrscheinlichkeit wir nachweisen können.

Algorithmus 11.10 (**Fast Cut**).
Eingabe *zusammenhängender gewichteter Graph* $G = (V, E)$
Ausgabe *Schnitt* V_1, V_2
1. *If* $|V| \leq 6$ *Then*
   Berechne minimalen Schnitt V_1, V_2.
   Ausgabe V_1, V_2
2. *Else*
3. $t := \left\lceil 1 + \frac{|V|}{\sqrt{2}} \right\rceil$
4. $H :=$ `RandomisierteKontraktion`(G), *bis* t *Knoten übrig sind.*
5. $H' :=$ `RandomisierteKontraktion`(G), *bis* t *Knoten übrig sind.*
6. $(V_1, V_2) :=$ `FastCut`(H)
7. $(V_1', V_2') :=$ `FastCut`(H')
8. *Vergleiche* (V_1, V_2) *und* (V_1', V_2'), *gib kleineren Schritt aus.*

Wir beobachten zunächst, dass der Abbruch der Rekursion bei $|V| \leq 6$ durch die Wahl von t motiviert ist: Es gilt $\left\lceil 1 + \frac{n}{\sqrt{2}} \right\rceil \leq n$ erstmals für $n = 6$. Nun wollen wir feststellen, ob die Gesamtlaufzeit von Algorithmus 11.10 akzeptabel ist. Wir erinnern uns daran, dass wir wesentlich schneller als $O(n^4 \log n)$ werden und möglichst dicht an $O(n^2)$ herankommen wollen.

Lemma 11.11. *Algorithmus 11.10 hat auf einem Graphen* $G = (V, E)$ *Laufzeit* $O(n^2 \log n)$.

Beweis. Wir bezeichnen die Laufzeit von Algorithmus 11.10 für einen Graphen, der n Knoten hat, mit $T(n)$. Wir wissen, dass die Laufzeit von Algorithmus 11.3 (also der Aufruf von `RandomisierteKontraktion` in den Zeilen 6 und 7) $O(n^2)$ beträgt. Wir vereinfachen uns die Notation und nehmen an, dass die Laufzeit durch $n^2/2$ nach oben beschränkt ist. Das ändert nichts an der Asymptotik der Laufzeit.
Mit dieser Annahme haben wir

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

für $n > 6$ und $T(n) = O(1)$ für $n \leq 6$. Wir vereinfachen uns die Situation noch einmal, indem wir annehmen, dass $T(n) = 1$ gilt für $n \leq 6$. Auch das verändert die Laufzeit asymptotisch nicht.

Wir betrachten die Folge der auftretenden Knotenanzahlen $n_0 := n$, $n_1 := \left\lceil 1 + \frac{n_0}{\sqrt{2}} \right\rceil$, $n_2 := \left\lceil 1 + \frac{n_1}{\sqrt{2}} \right\rceil$, $\ldots$, $n_r := \left\lceil 1 + \frac{n_{r-1}}{\sqrt{2}} \right\rceil$, dabei ist $n_r \leq 6$. Offensichtlich bezeichnet r die maximale Rekursionstiefe. Weil $\left\lceil 1 + \frac{7}{\sqrt{2}} \right\rceil = 6$ gilt, haben wir tatsächlich $n_r = 6$. Wir betrachten die entstehenden Gleichungen und haben

$$\begin{aligned} T(n) &= T(n_0) = 2T(n_1) + n_0^2 = 2\left(2T(n_2) + n_1^2\right) + n_0^2 \\ &= 4T(n_2) + 2n_1^2 + n_0^2 \\ &= 8T(n_3) + 4n_2^2 + 2n_1^2 + n_0^2 \\ &= 2^r T(6) + \sum_{i=0}^{r-1} 2^i n_i^2 = 2^r + \sum_{i=0}^{r-1} 2^i n_i^2. \end{aligned}$$

Zur Abschätzung überlegen wir uns, wie die Summanden $2^i n_i^2$ sich mit wachsendem i entwickeln und betrachten dazu die beiden Faktoren zunächst getrennt. Natürlich gilt $2^{i+1}/2^i = 2$ und wir sind an n_{i+1}^2/n_i interessiert. Es gilt

$$\frac{n_{i+1}^2}{n_i^2} = \frac{\left\lceil 1 + \frac{n_i}{\sqrt{2}} \right\rceil^2}{n_i^2} > \frac{\left(1 + \frac{n_i}{\sqrt{2}}\right)^2}{n_i^2} = \frac{\frac{n_i^2}{2} + \sqrt{2}n_i + 1}{n_i^2} = \frac{1}{2} + \frac{\sqrt{2}}{n_i} + \frac{1}{n_i^2} > \frac{1}{2}$$

und

$$\frac{2^{i+1} n_{i+1}^2}{2^i n_i^2} > 2 \cdot \frac{1}{2} = 1$$

folgt. Also ist $2^i n_i^2$ maximal für $i = r$ und wir haben

$$T(n) = 2^r + \sum_{i=0}^{r-1} 2^i n_i^2 \leq 2^r + r \cdot 2^r n_r^2 = 2^r + r \cdot 2^r \cdot 36 = O(r \cdot 2^r)$$

als obere Schranke für $T(n)$. Wir müssen also nur noch die Rekursionstiefe r nach oben abschätzen.
Wir definieren $q := 1/\sqrt{2}$ und behaupten $n \cdot q^i + 2/(1-q) \geq n_i$. Wir beweisen das mit vollständiger Induktion. Für den Induktionsanfang haben wir

$$n \cdot q^0 + \frac{2}{1-q} = n + \frac{2}{1-q} > n = n_0.$$

Für den Induktionsschluss haben wir

$$\begin{aligned} n \cdot q^{i+1} + \frac{2}{1-q} &= q \cdot \left(n \cdot q^i + \frac{2}{1-q} + \frac{2}{q-q^2} - \frac{2}{1-q} \right) \\ &= q \cdot \left(n \cdot q^i + \frac{2}{1-q} + \frac{2}{q} \right) \\ &\geq q \cdot \left(n_i + \frac{2}{q} \right) = 2 + n_i \cdot q \\ &\geq \lceil 1 + n_i \cdot q \rceil = n_{i+1} \end{aligned}$$

wie gewünscht.
Für die Abschätzung der Induktionstiefe r suchen wir das kleinste i, so dass $n_i < 7$ gilt. Wir wissen jetzt, dass es genügt, ein kleinstes i zu finden, so dass $nq^i + 2/(1-q) < 7$ gilt. Wir rechnen dafür

$$\begin{aligned} & n \cdot q^i + \frac{2}{1-q} < 7 \\ \Leftrightarrow \quad & n \cdot q^i < 7 - \frac{2}{1-q} \\ \Leftrightarrow \quad & \log(n) + i \log(q) < \log \left(7 - \frac{2}{1-q} \right) \\ \Leftrightarrow \quad & -\frac{i}{2} < \log \left(7 - \frac{2}{1-q} \right) - \log n \\ \Leftrightarrow \quad & i > 2 \log(n) - 2 \log \left(7 - \frac{2}{1-q} \right) \end{aligned}$$

und sehen, dass $r \leq 2 \log(n) + 6$ gilt. Die 6 hier hat nichts mit der 6 vom Rekursionsabbruch zu tun, sie entstammt dem Umstand, dass $\left\lceil -2 \log \left(7 - \frac{2}{1-q} \right) \right\rceil = 6$ gilt. Wir erinnern uns, dass wir $T(n) = O(r \cdot 2^r)$ nachgewiesen hatten, so dass $T(n) = O(n^2 \log n)$ folgt. □

Der im Vergleich zum einfach gehaltenen Algorithmus 11.3 (Randomisierte Kontraktion) geschickt vorgehende Algorithmus 11.10 (Fast Cut) hat eine nur um einen Faktor $O(\log n)$ größere Laufzeit. Wir hoffen natürlich, dass wir die Erfolgswahrscheinlichkeit jetzt aber wesentlich besser als nur mit $\Omega(1/n^2)$ abschätzen können und überzeugen uns davon, dass das tatsächlich der Fall ist.

Lemma 11.12. *Algorithmus 11.10 berechnet auf einem Graphen $G = (V, E)$ mit $|V| = n$ einen minimalen Schnitt mindestens mit Wahrscheinlichkeit $\Omega(1/\log n)$.*

Beweis. Wir sehen uns Fast Cut (Algorithmus 11.10) für einen Graphen $G = (V, E)$ mit $|V| > 6$ an, andernfalls berechnet der Algorithmus ja mit Wahrscheinlichkeit 1 einen minimalen Schnitt.
Wir betrachten zum Graphen $G = (V, E)$ einen beliebigen minimalen Schnitt V_1, V_2. Wir wollen für die Wahrscheinlichkeit, dass Algorithmus 11.10 einen Schnitt gleicher Größe berechnet, eine untere Schranke nachweisen.
Mit Blick auf Algorithmus 11.10 identifizieren wir die folgenden für uns relevanten Ereignisse.

$A(s)$**:** Im Graphen H ist keine Kante zwischen V_1, V_2 kontrahiert, dabei hat der Ausgangsgraph G genau s Knoten.

$A'(s)$**:** Im Graphen H' ist keine Kante zwischen V_1, V_2 kontrahiert, dabei hat der Ausgangsgraph G genau s Knoten.

$B(s)$**:** Fast Cut(H) liefert V_1, V_2 unter der Annahme, dass das möglich ist, dabei enthält H genau s Knoten.

$B'(s)$**:** Fast Cut(H') liefert V_1, V_2 unter der Annahme, dass das möglich ist, dabei enthält H' genau s Knoten.

Diese Notation benutzend wollen wir für

$$\text{Prob}\,(B(n)) = \text{Prob}\,((A(n) \wedge B(t)) \vee (A'(n) \wedge B'(t)))$$

mit $t = \left\lceil 1 + \frac{n}{\sqrt{2}} \right\rceil$ eine untere Schranke finden und beobachten zunächst, dass $\text{Prob}\,(A(n)) = \text{Prob}\,(A'(n))$ und $\text{Prob}\,(B(t)) = \text{Prob}\,(B'(t))$ gilt, weil die Ereignisse $A(n)$ und $A'(n)$ auf der einen Seite und $B(t)$ und $B'(t)$ auf der anderen Seite jeweils gleichartig und unabhängig sind. Außerdem sind die Ereignisse $A(n)$ und $B(t)$ auf der einen Seite und $A'(n)$ und $B'(t)$ auf der anderen Seite jeweils unabhängig. Das ausnutzend erhalten wir

$$\begin{aligned}
\text{Prob}\,(B(n)) &= \text{Prob}\,((A(n) \wedge B(t)) \vee (A'(n) \wedge B'(t))) \\
&= 1 - \text{Prob}\left(\overline{A(n) \wedge B(t)} \wedge \overline{A'(n) \wedge B'(t)}\right) \\
&= 1 - \text{Prob}\left(\overline{A(n) \wedge B(t)}\right)^2 = 1 - (1 - \text{Prob}\,(A(n) \wedge B(t)))^2 \\
&= 1 - (1 - \text{Prob}\,(A(n)) \cdot \text{Prob}\,(B(t)))^2
\end{aligned}$$

und machen uns an die Abschätzung von $\text{Prob}\,(A(n))$.
Lemma 11.9 liefert

$$\text{Prob}\,(A(n)) = \text{Prob}\,(A'(n)) \geq \frac{\left\lceil 1 + \frac{n}{\sqrt{2}} \right\rceil \cdot \left(\left\lceil 1 + \frac{n}{\sqrt{2}} \right\rceil - 1 \right)}{n \cdot (n-1)},$$

und wir sehen, dass $\text{Prob}\,(A(n)) = \text{Prob}\,(A'(n)) \geq 1/2$ für alle n folgt.
Wir setzen das ein und erhalten

$$\text{Prob}\,(B(n)) \geq 1 - \left(1 - \frac{1}{2}\text{Prob}\,(B(t))\right)^2.$$

Wir erinnern uns an die Entwicklung der Knotenanzahlen n_i und definieren $p(i) := \text{Prob}\,(B(n_{r-i}))$. Mit dieser Notation haben wir

$$p(i+1) \geq 1 - \left(1 - \frac{p(i)}{2}\right)^2 = p(i+1) - \frac{p(i+1)^2}{4}$$

und es gilt $p(0) = \text{Prob}\,(B(n_r)) = 1$.
Wir betrachten die Funktion $f\colon \mathbb{R} \to \mathbb{R}$ mit $f(x) = x - x^2/4$. Im Intervall $[0;1]$ ist diese Funktion (ihr Funktionsgraph ist eine nach unten geöffnete Parabel mit Scheitelpunkt bei $x_s = 2$) streng monoton wachsend. Wir wollen jetzt zunächst zeigen, dass

$$p(i) \geq \frac{1}{d} \Rightarrow p(i+1) \geq \frac{1}{d+1}$$

gilt. Wir haben $p(i+1) \geq f(p(i))$ und wegen der eben erwähnten Monotonie von f folgt aus $p(i) \geq 1/d$ (unserer Voraussetzung), dass $p(i+1) \geq f(1/d)$ gilt. Es genügt also zu zeigen, dass $f(1/d) \geq 1/(d+1)$ gilt. Wir erhalten

$$\begin{aligned}
& f\left(\frac{1}{d}\right) \geq \frac{1}{d+1} \\
\Leftrightarrow \quad & \frac{1}{d} - \frac{1}{4d^2} \geq \frac{1}{d+1} \\
\Leftrightarrow \quad & d + 1 - \frac{d+1}{4d} \geq d \\
\Leftrightarrow \quad & 1 \geq \frac{d+1}{4d} \\
\Leftrightarrow \quad & d \geq \frac{1}{3}
\end{aligned}$$

und sehen, dass das immer erfüllt ist: $d \geq 1/3$ ist äquivalent zu $1/d \leq 3$, wir hatten $p(i) \geq 1/d$ vorausgesetzt, dabei ist $p(i)$ eine Wahrscheinlichkeit, so dass natürlich $p(i) \leq 1$ gilt. Wir können also sogar $d \geq 1$ voraussetzen.
Wir haben jetzt also $p(0) = 1$ und $(p(i) \geq 1/d) \Rightarrow (p(i+1) \geq 1/(d+1))$, man sieht leicht (und kann natürlich auch leicht induktiv zeigen), dass $p(i) \geq 1/(i+1)$ für alle i gilt. Wir haben also $p(i) = \Omega(1/i)$ und können jetzt wieder

zurückübersetzen, so erhalten wir $\text{Prob}\,(B(n_{r-i})) = p(i) = \Omega(1/i)$, für $i = r$ erhalten wir also insbesondere

$$\text{Prob}\,(B(n_{r-r})) = \text{Prob}\,(B(n_0)) = \text{Prob}\,(B(n)) = p(r) = \Omega\left(\frac{1}{r}\right).$$

Wir erinnern uns daran, dass $r = O(\log n)$ gilt, so dass wir jetzt insgesamt für die Erfolgswahrscheinlichkeit auf einem Graphen mit n Knoten die untere Schranke $\Omega(1/\log n)$ nachgewiesen haben. □

Analog zu Korollar 11.8 können wir durch unabhängige Wiederholungen einen Algorithmus erhalten, der das Problem Min Cut mit Wahrscheinlichkeit praktisch beliebig nahe an 1 optimal löst.

Theorem 11.13. *Durch unabhängige Wiederholungen von Algorithmus 11.10 (Fast Cut) findet man für einen Graphen $G = (V, E)$ mit $|V| = n$ und jede Konstante $c > 0$ in Zeit $O\big(n^2 \log^3 n\big)$ einen minimalen Schnitt mit Wahrscheinlichkeit mindestens $1 - 1/n^c$ und für jede Konstante $\varepsilon > 0$ in Zeit $O\big(n^2 \log^2 n\big)$ einen minimalen Schnitt mit Wahrscheinlichkeit mindestens $1 - \varepsilon$.*

Beweis. Aus Lemma 11.11 und Lemma 11.12 wissen wir, dass es Konstanten $k_1, k_2 > 0$ gibt, so dass wir mit Algorithmus 11.10 in Zeit $\leq k_1 \cdot n^2 \ln n$ mit Wahrscheinlichkeit mindestens $k_2/\ln n$ einen minimalen Schnitt berechnen. Wir wiederholen Algorithmus 11.10 (Fast Cut) unabhängig w-mal und wählen die Anzahl der Wiederholungen $w := \lceil (c/k_2) \ln^2 n \rceil$, damit haben wir Gesamtrechenzeit $O\big(c \cdot (k_1/k_2) n^2 \ln^3 n\big) = O\big(n^2 \log^3 n\big)$ und Fehlschlagswahrscheinlichkeit höchstens

$$\left(1 - \frac{k_2}{\ln n}\right)^w \leq e^{-c \ln n} = \frac{1}{n^c}$$

wie behauptet. Für die zweite Teilaussage wählen wir die Anzahl der Wiederholungen $w := \lceil (1/k_2) \ln(1/\varepsilon) \ln n \rceil$ und erhalten Gesamtlaufzeit $O\big(n^2 \log^2 n\big)$ und Erfolgswahrscheinlichkeit mindestens $1 - \varepsilon$ wie behauptet. □

Für das Schnittproblem ist $\Omega(n + m)$ eine triviale untere Schranke, für dichte Graphen haben wir also $\Omega(n^2)$ als untere Schranke für die Laufzeit. Wir kommen jetzt mit Laufzeit $O\big(n^2 \log^2 n\big)$ aus, haben also in der Tat einen sehr schnellen randomisierten Algorithmus mit Erfolgswahrscheinlichkeit sehr dicht an 1. Man sollte sich jetzt im Rückblick noch einmal davon überzeugen, dass der eigentliche Algorithmus Fast Cut (Algorithmus 11.10) wirklich einfach ist, lediglich die Analyse hat uns ein wenig Mühe gekostet. Wir haben also auch hier Randomisierung als Konzept gesehen, das einfache und sehr effiziente Algorithmen erlaubt.

12 Ein randomisierter Primzahltest

Wir werden uns in diesem Kapitel wieder mit einem besonders effizienten und recht einfachen randomisierten Algorithmus beschäftigen für ein Problem, für das es auch einen deterministischen Algorithmus gibt, der mit polynomieller Rechenzeit auskommt; das erinnert so weit noch an Min Cut aus dem letzten Kapitel (Abschnitt 11.2). Genau wie bei Min Cut werden wir auch hier einen randomisierten Algorithmus besprechen, der zwar mit sehr großer Wahrscheinlichkeit das richtige Ergebnis ausgibt, der aber auch mit kleiner Wahrscheinlichkeit irren kann. Allerdings geht es diesmal um ein Entscheidungsproblem. Zu einer natürlichen Zahl $q \in \mathbb{N}$ wollen wir entscheiden, ob q eine Primzahl ist. Eine Primzahl ist bekanntlich eine natürliche Zahl, die genau zwei Teiler hat. Weil wir wissen, dass 1 und q natürlich Teiler von q sind, müssen wir nur alle Zahlen zwischen 2 und $q-1$ überprüfen, ob sie q teilen. Es geht sogar noch etwas besser: genau dann, wenn eine Zahl z, die kleiner als $\lfloor\sqrt{q}\rfloor$ ist, q teilt, so gibt es auch eine Zahl größer als $\lfloor\sqrt{q}\rfloor$, die q teilt, diese Zahl ergibt sich gerade als q/z. Es genügt also, q/z für jede Zahl $z \in \{2, 3, \ldots, \lfloor\sqrt{q}\rfloor\}$ auszurechnen. Wenn bei einer dieser Divisionen kein Rest entsteht, ist q keine Primzahl, wenn keine Division restfrei ist, ist q prim. Man kann leicht noch sparsamer sein und zum Beispiel alle geraden Zahlen (außer 2) auslassen. Warum also ist so ein $O\bigl(\sqrt{q}\bigr)$-Algorithmus nicht akzeptabel? Weil er nicht polynomiell beschränkte Laufzeit hat. Die Eingabelänge ist die Codierungslänge von q, also $\Theta(\log q)$, ein Algorithmus, der $\Omega\bigl(\sqrt{q}\bigr)$ viele Schritte unternimmt, hat also exponentielle Laufzeit in der Eingabelänge. Aber ist ein Algorithmus mit Laufzeit $O\bigl(\sqrt{q}\bigr)$ nicht trotzdem schnell genug? Um zu verstehen, warum das nicht der Fall ist, muss man sich im Grunde von der Vorstellung trennen, dass q eine natürliche Zahl ist. Wir sollten q lieber als eine Zeichenkette aus $\{1, 2, \ldots, 9\} \cdot \{0, 1, \ldots, 9\}^*$ betrachten und gelegentlich daran denken, dass q auch als natürliche Zahl aufgefasst werden kann. Dann sollte klar sein, dass Algorithmen, die exponentielle Laufzeit in der Länge der „Zeichenkette q“ haben, inakzeptabel sind.

Bevor wir uns wie versprochen dem randomisierten Primzahltest zuwenden, werden wir motivieren, warum sich irgendjemand für derartig große Primzahlen interessieren sollte. Als Motivation dient uns ein Ausflug in die Kryptographie.

12.1 Das RSA-Kryptosystem

Wenn wir uns an unsere einleitende Diskussion über einen naiven Primzahltest erinnern, bei dem wir Primalität durch Division feststellen, haben wir zusätzlich zu der Information, dass q keine Primzahl ist, noch zwei Teiler bekommen. Der Primzahltest, den wir im nächsten Abschnitt kennenlernen werden, wird uns diese Zusatzinformation nicht liefern. Wenn der Algorithmus ausgibt, dass eine Zahl keine Primzahl ist, so wird das zwar mit Sicherheit richtig sein und der Algorithmus könnte das auch belegen, er kennt aber keinen Teiler von q. Wir werden später sehen, wie das sein kann. Im anderen Fall ist die Ausgabe, dass die Zahl vermutlich prim ist, sicher kann man in diesem Fall aber nicht sein. Dass die Information über Teiler fehlt, ist wesentlich: Die Sicherheit des kryptographischen Systems, das wir hier kurz diskutieren wollen, beruht auf der unbewiesenen Annahme, dass es schwierig ist, zu großen natürlichen Zahlen Teiler effizient zu berechnen.

Definition 12.1. *Ein* kryptographisches System *besteht aus den folgenden Komponenten.*

- *endliche* Schlüsselmenge K
- Nachrichtenmenge M
- *Menge der* Kryptogramme Y
- Chiffrierfunktion $E\colon K \times M \to Y$
- Dechiffrierfunktion $D\colon K \times Y \to M$
  mit $\forall k \in K\colon \forall m \in M\colon D(k, E(k, m)) = m$

Für die praktische Anwendung eines kryptographischen Systems wünscht man sich natürlich, dass die Chiffrierfunktion E und die Dechiffrierfunktion D effizient berechenbar sind: die Benutzung eines Kryptosystems soll mit möglichst geringem Aufwand möglich sein. Außerdem soll so ein System aber sicher sein: Jemand, der das System kennt und verschickte Nachrichten abhört, jemand, der also E, D und $E(k, m)$ kennt, soll trotzdem nicht m berechnen können. Dieser Wunsch ist durchaus beweisbar erfüllbar, One-Time-Pads sind ein extremes Beispiel. Die Sicherheit der Kommunikation beruht dann alleine auf der Geheimhaltung des geheimen Schlüssels k. Für die legalen Benutzer entsteht damit allerdings das Problem, den geheimen Schlüssel k sicher zu kommunizieren. Diese offensichtliche Schwachstelle wurde (was die Publikationslage angeht) erst 1976 behoben, als Diffie und Hellmann in „New

Directions in Cryptography“ den Grundstein für Public-Key-Kryptographie legten, für kryptographische Systeme mit öffentlichem Schlüssel. Der Ansatz klingt vielleicht zunächst absurd, hat aber viele Vorzüge. Wir skizzieren den Ablauf der Benutzung eines solchen Public-Key-Kryptosystems.
Wir nehmen an, dass A an B vertrauliche Informationen schicken will und die beiden Kommunikationspartner keine Möglichkeit haben, einen geheimen Schlüssel über einen sicheren Kommunikationskanal auszutauschen. Die gesamte Kommunikation zwischen ihnen ist zumindest potenziell öffentlich. Wir gehen aber davon aus, dass die beiden miteinander kommunizieren können und sich gegenseitig ihrer Identität sicher sind. Wir beschränken die Rolle eines Angreifers also auf das reine Abhören.
Die Kommunikation beginnt damit, dass sich A und B (potenziell) öffentlich über das zu verwendende Kryptosystem einigen. Dann erzeugt B (also der designierte Empfänger der geheimen Nachricht) einen Schlüssel k und eine geheime Zusatzinformation $l(k)$. Den Schlüssel k schickt B an A, die geheime Zusatzinformation $l(k)$ bleibt natürlich geheim. Der Sender A hat die geheime Nachricht m und berechnet nun $E(k, m)$, dabei ist k der von B erhaltene geheime Schlüssel. Das Kryptogramm $E(k, m)$ schickt A nun öffentlich an B. Der Empfänger B entschlüsselt das Kryptogramm $E(k, m)$ mit Hilfe der Dechiffrierfunktion D, des öffentlichen Schlüssels k und der geheimen Zusatzinformation $l(k)$.
Zentral ist, dass der Sender A zum Verschlüsseln einer Nachricht an B den öffentlichen Schlüssel von B benutzt. Natürlich sollen die Funktionen D und E wieder effizient zu berechnen sein; hier fordern wir zusätzlich, dass die Dechiffrierfunktion D mit Kenntnis von $l(k)$ effizient berechenbar sein soll, während die Berechnung ohne Kenntnis von $l(k)$ nicht effizient möglich sein soll.
Ein kryptografisches System, das diese Anforderungen *vermutlich* erfüllt, ist das RSA-Kryptosystem, das 1978 veröffentlicht worden ist (Rivest, Shamir, Adleman (1978): A Method of Obtaining Digital Signatures and Public-Key Cryptosystems). Die Sicherheit beruht auf der Annahme, dass es nicht effizient möglich ist, für eine Zahl n die Primfaktorzerlegung durchzuführen. Wir brauchen konkret, dass für eine große Zahl $n = p \cdot q$, wobei p und q Primzahlen sind, die beiden Primfaktoren p und q nicht effizient bestimmbar sind.
Der Prozess der Schlüsselerzeugung auf der Seite des designierten Empfängers B läuft bei RSA wie folgt ab. Man erzeugt *geheim* zwei Primzahlen p und q, beide größer als 2^l, beide mit genau $l+1$ Bits zu repräsentieren, dabei ist l nicht zu klein. Aus diesen Primzahlen berechnet B nun $n := p \cdot q$. Dazu wird die *Eulerfunktion* $\phi(n)$ berechnet, die durch

$$\phi(n) = |\{a \in \{1, 2, \ldots, n-1\} \mid \mathrm{ggT}(a, n) = 1\}|$$

gegeben ist. Weil $n = p \cdot q$ die Primfaktorzerlegung von n beschreibt, ist $\phi(n)$ effizient berechenbar, es gilt $\phi(n) = (p-1) \cdot (q-1)$. Im nächsten Schritt berechnet B eine Zufallszahl $e \in \{2, 3, \ldots, n-1\}$, für die $\mathrm{ggT}(e, \phi(n)) = 1$ gilt. Schließlich berechnet B nun $d \equiv e^{-1} \bmod \phi(n)$. Keine dieser Berechnungen ist öffentlich. Am Ende dieser Berechnungen wird der designierte Empfänger B die Information $k = (n, e)$ als öffentlichen Schlüssel veröffentlichen, die Information $l(k) = (\phi(n), d)$ bleibt geheim.
Wenn der Sender A die geheime Botschaft $m \in \{0,1\}^*$ versenden will, so zerlegt er sie zunächst in Blöcke der Länge $2l$, also $m = m_1 m_2 \cdots m_j$ mit $m_i \in \{0,1\}^{2l}$ für alle $i \in \{1, 2, \ldots, j\}$. Die Zahl l ist ein Parameter des RSA-Systems und bei der Verhandlung über das zu verwendende Kryptosystem öffentlich vereinbart worden.
Alle diese Blöcke m_i werden auf die gleiche Art behandelt und dann jeweils chiffriert verschickt, auf der anderen Seite wird B die Blöcke jeweils einzeln dechiffrieren und anschließend zur Gesamtnachricht m zusammensetzen. Wir brauchen also nur die Behandlung eines Blocks $m_i \in \{0,1\}^{2l}$ zu betrachten. Natürlich können wir m_i auch als natürliche Zahl mit $0 \leq m_i < 2^{2l}$ interpretieren.
Der Sender A berechnet das Kryptogramm $y_i = E((n,e), m_i) \equiv m_i^e \bmod n$. Der Empfänger B wird dieses Kryptogramm y_i entschlüsseln, indem er $D((n,d), y_i) \equiv y_i^d \bmod n$ berechnet. Weil $m_i < 2^{2l}$ gilt und $n \geq 2^{2l}$ ist, reicht die Kenntnis von $m_i \bmod n$ aus, um m_i genau zu kennen.
Wir machen uns später klar, dass alle Schritte effizient durchführbar sind. Wir wollen hier noch anmerken, dass Kenntnis von n und $\phi(n)$ ausreicht, um die Primfaktoren p und q effizient bestimmen zu können: Wir wissen, dass $\phi(n) = (p-1) \cdot (q-1)$ gilt und können $q = n/p$ einsetzen. Daraus ergibt sich $\phi(n) = n - p - n/p + 1$ und wir können äquivalent zu $p^2 + p \cdot (\phi(n) - n - 1) + n = 0$ umformen. Wir sehen also, dass das Auflösen der Gleichung $x^2 + (\phi(n) - n + 1) \cdot x + n = 0$ nach x als Lösung p ergibt und q dann mit einer weiteren Division bestimmt werden kann. Wenn Faktorisierung effizient möglich ist, ist das RSA-Kryptosystem also in der Tat unsicher. Dass die angegebene Methode überhaupt funktioniert, dass also immer $(m^e \bmod n)^d \bmod n \equiv m \bmod n$ gilt, werden wir uns später überlegen, nachdem wir uns einige zahlentheoretische Grundlagen angeeignet haben.

12.2 Der Solovay-Strassen-Primzahltest

Um das RSA-Kryptosystem effizient implementieren zu können, müssen wir effizient potenzieren, größte gemeinsame Teiler und große Primzahlen be-

stimmen können. Es liegt auf der Hand, dass wir uns dazu zahlentheoretisches Grundlagenwissen aneignen müssen. Wir werden im Folgenden alle Rechnungen modulo m ausführen und annehmen, dass die üblichen Operationen (also Addition, Subtraktion, Multiplikation, Division und modulo-Berechnung) mit Zahlen dieser Länge in konstanter Zeit durchführbar sind. Bei Anwendung im RSA-Kryptosystem wird $m = n$ sein und die Zahlen Länge $\lfloor \log m \rfloor + 1 \approx 2l$ haben. Die üblichen Schlüssellängen l liegen im Bereich von 1024 bis 4096, wir dürfen also nicht damit rechnen, dass hardwareseitig Rechnen mit Zahlen dieser Länge direkt unterstützt wird. Weil auf jeden Fall l aber konstant ist, ist die Annahme der konstanten Zeit für Rechnungen mit Zahlen dieser Länge aber als angemessen zu beurteilen.
Wir wollen in diesem Grundlagenabschnitt möglichst konkret bleiben und beginnen darum mit einfachen Algorithmen, deren Korrektheit und Laufzeit wir nachweisen wollen, vor allem um uns an die Denk- und Argumentationsstrukturen, die sich in diesem Kapitel als nützlich erweisen werden, zu gewöhnen. Wir beginnen mit einem vermutlich bekannten und sehr einfachen Algorithmus zum schnellen Potenzieren.

Algorithmus 12.2.
Eingabe $x, n \in \mathbb{N}$
Ausgabe x^n
1. $r := 1$
2. *While* $n > 1$
3. *If* n *ungerade Then*
4. $r := r \cdot x$
5. $n := n - 1$
6. $x := x \cdot x$
7. $n := n/2$
8. *Ausgabe* $x \cdot r$

Theorem 12.3. *Algorithmus 12.2 berechnet* x^n *mit höchstens* $1 + 2 \log n$ *Multiplikationen.*

Beweis. Wir beginnen mit der Korrektheit und führen zunächst unsere Notation ein. Wir betrachten die durchgeführten Durchläufe der While-Schleife, nach dem t-ten Durchlauf seien die aktuellen Inhalte der auftretenden Variablen n_t, x_t und r_t. Wir haben also $r_0 = 1$, $x_0 = x$ und $n_t = n$. Wir behaupten, dass immer $r_t \cdot x_t^{n_t} = x^n$ gilt. Weil die Ausgabe $x_t^{n_t} \cdot r_t$ mit $n_t = 1$ ist, folgt aus dieser Behauptung die Korrektheit des Algorithmus, da n stets ganzzahlig ist und in jeder Runde echt kleiner wird. Dass die behauptete Gleichheit gilt, zeigen wir durch vollständige Induktion.

Für $t = 0$ haben wir $r_0 \cdot x_0^{n_0} = 1 \cdot x^n = x^n$ wie behauptet. Für den Induktionsschluss betrachten wir $r_{t+1} \cdot x_{t+1}^{n_{t+1}}$ und unterscheiden zwei Fälle nach dem Wert von n. Wenn $n > 1$ gerade ist, so ist $r_{t+1} = r_t$, $x_{t+1} = x_t \cdot x_t$ und $n_{t+1} = n_t/2$. Wir haben also

$$r_{t+1} \cdot x_{t+1}^{n_{t+1}} = r_t \cdot (x_t \cdot x_t)^{n_t/2} = r_t \cdot x_t^{n_t} = x^n$$

wie gewünscht. Ist andernfalls $n > 1$ ungerade, so ist $r_{t+1} = r_t \cdot x_t$, $x_{t+1} = x_t \cdot x_t$ und $n_{t+1} = (n_t - 1)/2$, so dass sich

$$r_{t+1} \cdot x_{t+1}^{n_{t+1}} = r_t \cdot x_t \cdot (x_t \cdot x_t)^{(n_t-1)/2} = r_t \cdot x_t^{n_t} = x^n$$

ergibt.
Für die Abschätzung der Rechenzeit bezeichne $M(n)$ die Anzahl der Multiplikationen in der While-Schleife, wenn n der Exponent ist. Wir wollen $M(n) \leq 2 \log n$ zeigen, da am Ende noch eine Multiplikation durchgeführt wird, folgt daraus das Theorem. Wir führen den Beweis wieder mit vollständiger Induktion, diesmal über n.
Für $n = 1$ haben wir $M(1) = 0 = 2 \log 1 = 2 \log n$, so dass der Induktionsanfang gesichert ist. Für den Induktionsschritt beobachten wir, dass $M(n) = \max\{1 + M(n/2), 2 + M((n-1)/2)\}$ gilt, wobei sich der erste Term aus dem Fall n gerade und der zweite Term aus dem Fall n ungerade ergibt. Wir können auf jeden Fall $M(n) \leq 2 + M(n/2)$ abschätzen und erhalten $M(n) \leq 2 + 2 \log(n/2) = 2 + 2 \log(n) - 2 = 2 \log(n)$ aus der Induktionsvoraussetzung. □

Wir beobachten, dass wir alle Berechnungen auch modulo m ausführen können. Dann werden die beteiligten Zahlen nicht zu groß und wir erhalten aus der Schranke für die Anzahl der Multiplikationen auch eine Schranke für die Rechenzeit.
Der euklidische Algorithmus für die Bestimmung des größten gemeinsamen Teilers zweier Zahlen ist sicher jedem bekannt. Wir geben ihn hier trotzdem der Vollständigkeit halber mit Korrektheitsbeweis und Laufzeitabschätzung an.

Algorithmus 12.4 (Euklidischer Algorithmus)**.**
Eingabe $a \geq b \in \mathbb{N}$
Ausgabe $ggT(a, b)$
1. $c := a \bmod b$
2. *If* $c = 0$
3. *Then Ausgabe* b
4. *Else Ausgabe* $ggT(b, c)$

Theorem 12.5. *Der euklidische Algorithmus (Algorithmus 12.4) berechnet in Zeit $O(\log(a+b))$ mit höchstens $\lfloor \log_{3/2}(a+b) \rfloor$ modulo-Operationen den größten gemeinsamen Teiler von a und b.*

Beweis. Wir beginnen mit der Korrektheit und beobachten, dass aus $c := a \bmod b$ folgt, dass es ein $k \in \mathbb{N}_0$ gibt, so dass $a = k \cdot b + c$ gilt, dabei ist $c \in \{0, 1, \ldots, b-1\}$. Ist $c = 0$, so ist b ein Teiler von a und $\mathrm{ggT}(a,b) = b$ folgt, so dass die Ausgabe in Zeile 3 korrekt ist. Ist $c \geq 1$, so müssen wir $\mathrm{ggT}(a,b) = \mathrm{ggT}(b,c)$ nachweisen.
Sei $r = \mathrm{ggT}(b,c)$. Weil $a = k \cdot b + c$ gilt und r sowohl b als auch c teilt, ist r auch ein Teiler von a, also insbesondere ein gemeinsamer Teiler von a und b. Wir führen einen Beweis durch Widerspruch und nehmen an, dass $r' > r$ der größte gemeinsame Teiler von a und b ist. Weil r' also $a = k \cdot b + c$ und auch b teilt, ist r' auch ein Teiler von c. Wir haben also neben dem $\mathrm{ggT}(b,c) = r$ einen weiteren gemeinsamen Teiler r' von b und c, für den $r' > r$ gilt, ein Widerspruch. Der Algorithmus ist also korrekt.
Für die Laufzeit genügt es zu zeigen, dass die Anzahl der modulo-Operationen durch $\lfloor \log_{3/2}(a+b) \rfloor$ nach oben beschränkt ist. Wir beweisen das mittels vollständiger Induktion über $a+b$. Wir betrachten zunächst $a+b \leq 2$, in diesem Fall gilt $a = b = 1$. Die Anzahl der modulo-Operationen in diesem Fall beträgt $1 = \lfloor \log_{3/2} 2 \rfloor = \lfloor \log_{3/2}(a+b) \rfloor$, so dass der Induktionsanfang gesichert ist. Für größere $a+b$ ist die Anzahl der modulo-Operationen gemäß Induktionsvoraussetzung durch $1 + \lfloor \log_{3/2}(b+c) \rfloor$ gegeben. Wir behaupten, dass $b + c \leq (2/3) \cdot (a+b)$ gilt, daraus folgt dann $1 + \lfloor \log_{3/2}(b+c) \rfloor \leq 1 + \lfloor \log_{3/2}((2/3) \cdot (a+b)) \rfloor \leq \lfloor \log_{3/2}(a+b) \rfloor$ wie gewünscht.
Wir haben $c = a \bmod b$, so dass $c \leq a - b$ folgt, was wir äquivalent zu $a \geq b + c$ umformen können. Andererseits ist auch $b > c$ offensichtlich, woraus wir $b = (b/2) + (b/2) > (b/2) + (c/2) = (b+c)/2$ folgern können. Zusammen ergibt das $a + b > b + c + (b+c)/2 = (3/2) \cdot (b+c)$ und $b + c \leq (2/3) \cdot (a+b)$ folgt wie behauptet. □

Wir beschließen diese Folge sehr einfacher zahlentheoretischer Algorithmen mit einem Algorithmus für die effiziente Berechnung des multiplikativen Inversen $b^{-1} \bmod a$ zu b, dabei setzen wir $\mathrm{ggT}(a,b) = 1$ voraus.

Algorithmus 12.6.
Eingabe $a \geq b \in \mathbb{N}$ *mit* $ggT(a,b) = 1$
Ausgabe $b^{-1} \bmod a$
1. *If* $b = 1$ *Then Ausgabe 1. STOP.*
2. $c := a \bmod b$
3. $c^* := c^{-1} \bmod b$ *(rekursiv)*
4. *Ausgabe* $\left(\frac{1 - c^* \cdot a}{b}\right) \bmod a$

Theorem 12.7. *Algorithmus 12.6 berechnet zu $a > b \in \mathbb{N}$ mit $ggT(a,b) = 1$ das multiplikative Inverse $b^{-1} \bmod a$ zu b in Zeit $O(\log(a+b))$.*

Beweis. Wir bemerken zunächst, dass die Laufzeitschranke ganz analog zum Beweis für den Algorithmus von Euklid (Algorithmus 12.4) folgt, so dass wir nur die Korrektheit beweisen müssen. Wir machen das mit vollständiger Induktion über b und betrachten zunächst den Fall $b = 1$. In diesem Fall ist die Ausgabe 1, wir haben $1 \cdot b = 1 \cdot 1 = 1 \equiv 1 \bmod a$ und der Induktionsanfang ist gesichert. Für größere b ist die Ausgabe $((1 - c^* \cdot a)/b) \bmod a$ und wir haben $c^* \equiv b^{-1} \bmod a$ gemäß Induktionsvoraussetzung. Allerdings müssen wir dafür noch nachweisen, dass der rekursive Aufruf mit b, c überhaupt zulässig ist. Wir haben $c = a \bmod b$ und $b > c$ folgt direkt. Außerdem ist $c > 0$, andernfalls wäre $\mathrm{ggT}(a,b) = 1$ im Widerspruch zur Voraussetzung. Schließlich erinnern wir uns daran (Beweis von Theorem 12.5), dass $\mathrm{ggT}(b,c) = \mathrm{ggT}(a,b)$ gilt und $\mathrm{ggT}(b,c) = 1$ folgt. Der rekursive Aufruf in Zeile 3 ist also zulässig. Wir haben $c^* \equiv c^{-1} \bmod b$, also gibt es ein $k' \in \mathbb{Z}$ mit $c \cdot c^* + k' \cdot b = 1$. Wir erinnern uns außerdem daran, dass $a = k \cdot b + c$ gilt für ein $k \in \mathbb{N}$. Wir fügen das zusammen und erhalten

$$\begin{aligned} 1 - c^* \cdot a = 1 - c^* \cdot (k \cdot b + c) &= 1 - c^* \cdot k \cdot b - c^* \cdot c \\ &= 1 - c^* \cdot k \cdot b - 1 + k' \cdot b = b \cdot (k' - k \cdot c^*), \end{aligned}$$

so dass die Ausgabe $(1 - c^* \cdot a)/b = k' - k \cdot c^*$ jedenfalls ganzzahlig ist. Außerdem ist

$$\frac{1 - c^* \cdot a}{b} \cdot b \bmod a \equiv 1 - c^* \cdot a \bmod a \equiv 1 \bmod a$$

wie gewünscht. □

Das zentrale offene Problem ist die effiziente Berechnung von großen Primzahlen. Wir glauben, dass Faktorisierung ein schwieriges Problem ist (andernfalls ist RSA unsicher und wir brauchen uns nicht damit zu beschäftigen), also müssen wir hoffen, das PRIMES, das Entscheidungsproblem, zu einer Zahl p zu entscheiden, ob p prim ist, einfacher ist als p zu faktorisieren. Tatsächlich ist schon seit 2002 ein deterministischer Polynomialzeitalgorithmus für dieses Problem bekannt (Agrawal, Kayal, Saxena (2002): PRIMES is in P). Allerdings ist dieser Algorithmus recht langsam, so dass in der Praxis randomisierten Verfahren mit einseitigem Fehler, die wesentlich schneller sind und nur sehr kleine Fehlerwahrscheinlichkeit haben, bevorzugt werden. Diese Algorithmen werden niemals eine Primzahl fälschlich als zusammengesetzt bezeichnen, können aber eine zusammengesetzte Zahl fälschlich für eine Primzahl halten. Man kann sich überlegen, dass das im Fall von RSA nicht

die Sicherheit gefährdet, es kann aber dazu führen, dass eine verschlüsselte Nachricht nicht mehr entschlüsselt werden kann. In diesem sehr unwahrscheinlichen Fall wird dann einfach ein neues Paar aus öffentlichem Schlüssel und geheimer Zusatzinformation generiert.
Hat man einen Primalitätstest (gleichgültig, ob der nun randomisiert oder deterministisch ist), so kann man zufällige Primzahlen einfach raten und verifizieren: Wir wählen zufällig eine Zahl im definierten Intervall und testen anschließend, dass es sich um eine Primzahl handelt. Um aussagen zu können, wie oft man im Erwartungswert eine Primalitätstest durchführen muss, wird eine Aussage über die Häufigkeit von Primzahlen benötigt. Zahlentheoretiker antworten auf diese Frage gerne, dass $\lim_{n\to\infty} \pi(n) \cdot n/\ln n = 1$ gilt, dabei ist $\pi(n) = |\{p \leq n \mid p \text{ prim}\}|$ die Anzahl der Primzahlen bis n. Diese elegante Aussage ist leider aus Informatiksicht nicht sehr hilfreich, weil man nichts über die genaue Entwicklung des Grenzwertes für konkrete Werte von n entnehmen kann. Hilfreicher ist da die Aussage

$$\forall n \geq 2\colon \frac{n}{\log n} - 2 \leq \pi(n) \leq \frac{3n}{\log n}.$$

Entscheidend für uns ist die untere Schranke, die obere Schranke dient im Wesentlichen zur Beruhigung: die untere Schranke ist asymptotisch exakt, es lohnt sich für uns nicht, noch genauer zu rechnen. Wir begnügen uns darum hier mit der Herleitung der unteren Schranke. Man könnte das Ergebnis auch einfach zitieren, wir wollen uns aber davon überzeugen, dass der Beweis gar nicht so schwierig ist. Zunächst einmal beobachten wir aber, dass wir bei zufälliger Auswahl einer natürlichen Zahl $\leq n$ im Erwartungswert nur $\Theta(\log n)$ Versuche brauchen, bis wir eine Primzahl erwischt haben. Wenn wir einen effizienten Primzahltest haben, können wir also in der Tat schnell zufällige große Primzahlen bestimmen.
Es wird hilfreich sein, sich die Primfaktorzerlegung von Zahlen anzusehen, darum definieren wir

Definition 12.8. *Für $m \in \mathbb{N}$ und p prim ist* $\nu_p(m) := \max\{k \mid (p^k \mid m)\}$.

und halten damit zwei Hilfsresultate fest, die sich für den Beweis der unteren Schranke als nützlich erweisen werden.

Lemma 12.9. $\forall n \in \mathbb{N}, p$ *prim*: $\nu_p(n!) = \sum_{k\geq 1} \left\lfloor \frac{n}{p^l} \right\rfloor$.

Beweis. Wir definieren $R_{p,n} := \{(i,k) \mid i \in \{1,2,\ldots,n\} \text{ und } p^k \mid i\}$ und schätzen $|R_{p,n}|$ auf zwei verschiedene Weisen ab. Wir stellen uns zu $R_{p,n}$ eine Tabelle mit n Spalten vor, bei der in der i-ten Spalte und k-ten Zeile

eine 1 eingetragen ist, wenn $(i,k) \in R_{p,n}$ gilt und eine 0 sonst. Wir beobachten, dass es ein endliches k gibt, so dass ab der k-ten Zeile alle Zeilen 0-Zeilen sind. Wir können offenbar $|R_{p,n}|$ bestimmen, indem wir die Einsen in dieser Tabelle addieren. Gehen wir spaltenweise vor, so erhalten wir gemäß Definition von $R_{p,n}$

$$|R_{p,n}| = \sum_{i=1}^{n} \left|\left\{k \geq 1 \mid p^k \mid i\right\}\right| = \sum_{i=1}^{n} \max\left\{k \geq 1 \mid \left(p^k \mid i\right)\right\}$$
$$= \sum_{i=1}^{n} \nu_p(i) = \nu_p(n!).$$

Addieren wir andererseits zeilenweise, so erhalten wir

$$|R_{p,n}| \quad = \quad \sum_{k\geq 1} \left|\left\{i \mid i \in \{1,2,\ldots,n\} \text{ und } p^k \mid i\right\}\right| \quad = \quad \sum_{k\geq 1} \left\lfloor \frac{n}{p^k} \right\rfloor$$

und haben zusammen $\nu_p(n!) = \sum\limits_{k\geq 1} \left\lfloor \frac{n}{p^k} \right\rfloor$ gezeigt. □

Lemma 12.10. $\forall p \; prim, l = \nu_p\left(\binom{2n}{n}\right) : p^l \leq 2n.$

Beweis. Wir haben

$$l = \nu_p\left(\binom{2n}{n}\right) = \nu_p\left(\frac{(2n)!}{(n!)\cdot(n!)}\right) = \nu_p((2n)!) - 2\cdot\nu_p(n!)$$

und können Lemma 12.9 verwenden, um

$$l = \sum_{k\geq 1} \left\lfloor \frac{2n}{p^k} \right\rfloor - 2\cdot \sum_{k\geq 1} \left\lfloor \frac{n}{p^k} \right\rfloor = \sum_{k\geq 1} \left\lfloor \frac{2n}{p^k} \right\rfloor - 2\cdot \left\lfloor \frac{n}{p^k} \right\rfloor$$

zu erhalten. Natürlich sind alle Summanden mit $p^k > 2n$ gleich 0. Die einzelnen Summanden sind sich sehr ähnlich, wir beobachten, dass $\lfloor 2y \rfloor - 2\cdot\lfloor y \rfloor \in \{0,1\}$ gilt, so dass tatsächlich nur 0 und 1 als Summanden vorkommen. Folglich gilt

$$l \leq \max\left\{k \geq 1 \mid p^k \leq 2n\right\}$$

und $p^l \leq 2n$ folgt. □

Wir können jetzt eine erste Verbindung zu $\pi(n)$, der Anzahl der Primzahlen bis n, herstellen. Damit kommen wir der unteren Schranke, die wir nachweisen wollen, thematisch erkennbar nahe.

Lemma 12.11. $\forall n \in \mathbb{N}\colon \binom{2n}{n} \leq (2n)^{\pi(2n)}$.

Beweis. Wir schauen uns die eindeutige Primfaktorzerlegung von $\binom{2n}{n}$ an, sei also

$$\binom{2n}{n} = \prod_{i=1}^{r} p_i^{k_i}$$

mit p_i prim und $k_i \in \mathbb{N}$ für alle $i \in \{1, 2, \ldots, r\}$. Natürlich teilt jeder einzelne Primfaktor $(2n)!$, so dass $\max\{p_i \mid 1 \leq i \leq r\} < 2n$ und $r \leq \pi(2n)$ folgen. Lemma 12.10 liefert, dass $p_i^{k_i} \leq 2n$ gilt für alle i. Also haben wir insgesamt

$$\binom{2n}{n} = \prod_{i=1}^{r} p_i^{k_i} \leq (2n)^r \leq (2n)^{\pi(2n)}$$

wie behauptet. □

Damit können wir endlich die schon eingangs erwähnte Schranke von $(n/\log n) - 2$ für die Anzahl der Primzahlen bis n zeigen.

Theorem 12.12. $\forall n \geq 2\colon \pi(n) \geq \frac{n}{\log n} - 2$.

Beweis. Wir führen den Beweis zunächst nur für gerade n. Wir erinnern uns an Lemma 10.20 und haben

$$\frac{2^{2n}}{2n+1} \leq \binom{2n}{n} \leq 2^{2n}$$

als Abschätzung. Wir benutzen Lemma 12.11 und erhalten

$$(2n)^{\pi(2n)} \geq \binom{2n}{n} \geq \frac{2^{2n}}{2n+1},$$

so dass

$$\pi(2n) \geq \log_{2n}\left(\frac{2^{2n}}{2n+1}\right) = \frac{2n}{\log(2n)} - \frac{\log(2n+1)}{\log(2n)} \geq \frac{2n}{\log(2n)} - 2$$

folgt. Für $N := 2n$ (also für gerades N) haben wir damit $\pi(N) \geq N/\log(N) - 2$ wie behauptet. Ist N ungerade, so haben wir $\pi(N) = \pi(N+1)$, weil $N+1 > 2$ gerade ist und darum keine Primzahl. Wir können also die Abschätzung für $\pi(N+1) \geq (N+1)/\log(N+1) - 2$ verwenden und sehen, dass $\pi(N) \geq N/\log(N) - 2$ für alle ungeraden $N > 2$ folgt. □

Nachdem wir uns davon überzeugt, dass wir nicht all zu lange suchen müssen, um zufällig eine Primzahl zu finden, benötigen wir „nur noch“ einen effizienten Primzahltest. Wir hatten schon erläutert, dass wir uns mit einem Test zufriedengeben wollen, der einseitigen Fehler hat, der also eine zusammengesetzte Zahl nicht sicher als zusammengesetzt erkennt. Eine triviale Idee für einen solchen Test für eine Zahl n ist, eine Zahl $t \in \{2, 3, \ldots, n-1\}$ uniform zufällig zu wählen und zu testen, ob $t \mid n$ gilt. Wenn das der Fall ist, ist n sicher zusammengesetzt. Andernfalls können wir keine sichere Aussage machen. Wir sehen sofort, dass dieser Test zumindest für große Zahlen n mit wenig Primfaktoren (also zum Beispiel für $n = p \cdot q$ mit p, q prim) völlig unbrauchbar ist: Die Erfolgswahrscheinlichkeit ist exponentiell klein. Wir brauchen also weit weniger triviale Ideen. Dafür ist es hilfreiche, sich einige zahlentheoretische Grundlagen und ein wenig Gruppentheorie (wieder) anzueignen.

Definition 12.13. *Eine Menge G mit einer binären Operation $\circ$ heißt* Gruppe*, wenn Folgendes gilt.*

- $\forall a, b, c \in G\colon (a \circ b) \circ c = a \circ (b \circ c)$ *(Assoziativität)*
- $\exists e \in G\colon \forall a \in G\colon a \circ e = e \circ a = a$ *(neutrales Element)*
- $\forall a \in G\colon \exists b \in G\colon a \circ b = b \circ a = e$ *(inverse Elemente)*

Gilt zusätzlich $\forall a, b \in G\colon a \circ b = b \circ a$ (Kommutativität), so heißt $(G, \circ, e)$ abelsche Gruppe*.*

Wir sehen sofort, dass das neutrale Element eindeutig ist und das auch das inverse Element zu a eindeutig ist (für alle a), wir nennen es a^{-1}. Außerdem ist leicht zu sehen, dass

$$\forall a, b, c \in G\colon a \circ c = b \circ c \Leftrightarrow a = b \ \Leftrightarrow c \circ a = c \circ b$$

gilt.
Es gibt unendliche Gruppen (z. B. $(\mathbb{Z}, +, 0)$) und endliche Gruppen (z. B. $(\{1\}, \cdot, 1)$). Wir werden uns im Folgenden fast ausschließlich mit endlichen abelschen Gruppen beschäftigen. Ausgehend von einer Gruppe identifizieren wir Untergruppen, die bei endlichen Gruppen automatisch entstehen, wenn wir eine Menge $H \subseteq G$ mit $e \in H$ identifizieren, die unter $\circ$ abgeschlossen ist.

Definition 12.14. *$H \subseteq G$ heißt* Untergruppe *der Gruppe $(G, \circ, e)$, wenn $(H, \circ, e)$ eine Gruppe ist.*
Zu einer Untergruppe H von G definieren wir die Relation $\sim_H$ durch

$$\forall a, b \in G\colon a \sim_H b :\Leftrightarrow b^{-1} \circ a \in H.$$

Man rechnet leicht nach, dass $\sim_H$ eine Äquivalenzrelation ist und es für alle $b \in G$ eine bijektive Abbildung zwischen H und $[b]_H$, der Äquivalenzklasse von b bezüglich $\sim_H$, gibt: $f_b\colon [b]_H \to H$ mit $f(a) = b^{-1} \circ a$ ist eine solche Abbildung.
Mit diesen wenigen Begriffen und Einsichten können wir schon ein erstes Resultat formulieren und beweisen, das später bei der Entwicklung eines effizienten randomisierten Primzahltests nützlich sein wird.

Theorem 12.15 (**Satz von Lagrange**). *Sei H Untergruppe einer endlichen Gruppe G. Dann ist $|H| \mid |G|$.*

Beweis. Wir betrachten die Äquivalenzklassen $C_1, C_2, \ldots, C_r$ von $\sim_H$. Weil G endlich ist, ist natürlich auch H endlich und darum auch die Anzahl der Äquivalenzklassen. Weil es für jedes $b \in H$ eine Bijektion zwischen H und $[b]_H$ gibt, gilt $|C_i| = |H|$ für alle $i \in \{1, 2, \ldots, r\}$. Weil die Äquivalenzklassen alle paarweise disjunkt sind und $\bigcup_{i=1}^{r} C_i = G$ gilt, haben wir $|G| = r \cdot |H|$ und $|H| \mid |G|$ folgt. □

Besonders interessant in unserem Kontext sind zyklische Gruppen. Für die formale Definition definieren wir

$$a^i := \begin{cases} e & \text{falls } i = 0 \\ a \circ a^{i-1} & \text{falls } i > 0 \\ (a^{-1})^{(-i)} & \text{falls } i < 0 \end{cases}$$

für $a \in G$ und $i \in \mathbb{Z}$, dabei sei $(G, \circ, e)$ eine Gruppe. Man sieht direkt, dass für alle $a \in G$ und alle $i, j \in \mathbb{Z}$ stets $(a^i)^{-1} = a^{-i}$, $a^{i+j} = a^i \circ a^j$ gelten, außerdem haben wir für $a, b \in G$ mit $a \circ b = b \circ a$, dass $(a \circ b)^i = a^i \circ b^i$ gilt.

Definition 12.16. *Sei $(G, \circ, e)$ eine Gruppe, $a \in G$.*
Es ist $\langle a \rangle := \{a^i \mid i \in \mathbb{Z}\}$, $ord_G(a) := |\langle a \rangle|$ heißt die Ordnung *von a in G.*
Wir nennen a erzeugendes Element*, wenn $\langle a \rangle = G$ gilt. Die Gruppe G heißt* zyklisch*, wenn sie ein erzeugendes Element enthält.*

Offenbar ist $\langle a \rangle$ die kleinste Untergruppe von G ist, die a enthält. Wir wollen uns jetzt etwas näher mit zyklischen Gruppen auseinandersetzen und definieren dafür $\mathbb{Z}_n := \{0, 1, \ldots, n-1\}$. Wir erinnern uns daran, dass $(\mathbb{Z}, +, 0)$ eine Gruppe ist, außerdem macht man sich leicht klar, dass auch $(\mathbb{Z}_n, + \bmod n, 0)$ eine Gruppe ist. Wir werden nun erklären, warum uns diese Gruppen so wichtig sind.

Lemma 12.17. *Sei $(G, \circ, e)$ eine Gruppe, $a \in G$.*

1. *Wenn alle a^i für $i \in \mathbb{Z}$ verschieden sind, dann ist $ord_G(a) = \infty$ und $(G, \circ, e)$ ist isomorph zu $(\mathbb{Z}, +, 0)$.*

2. *Wenn $a^i = a^j$ für ein $i < j$ ist, dann ist $ord_G(a) \leq j - i$.*

Beweis. 1. Wir betrachten die Abbildung $i \to a^i$ zwischen $\mathbb{Z}$ und $\langle a \rangle$: Es handelt sich offensichtlich um eine Bijektion, außerdem ist die Abbildung ein Isomorphismus, da $0 \to e$, $i + j \to a^i \circ a^j$ und $-i \to (a^i)^{-1}$ gilt.

2. Es ist $a^{j-i} = a^j \circ (a^i)^{-1} = a^j \circ (a^j)^{-1} = a^j \circ a^{-j} = e$. Wir wollen zeigen, dass $|\langle a \rangle| \leq j - i$ gilt. Dazu betrachten wir a^l für ein beliebiges $l \in \mathbb{Z}$. Wir definieren $r = l \bmod (j - i)$ und $q = \lfloor l/(j - i) \rfloor$, es gilt also $l = q \cdot (j - i) + r$ mit $r \in \{0, 1, \ldots, j - i - 1\}$. Wir haben also $a^l = a^{q \cdot (j-i)+r} = a^{(j-i) \cdot r} \circ a^r = a^r$ und $\langle a \rangle$ kann höchstens die Elemente $a^0, a^1, \ldots, a^{j-i-1}$ enthalten.

□

Theorem 12.18. *Sei $(G, \circ, e)$ eine Gruppe, $a \in G$ mit $ord_G(a) = n$ für ein $m \in \mathbb{N}$.*

1. *$\forall i \leq j \colon a^i = a^j \Leftrightarrow n \mid (j - i)$*

2. *$\langle a \rangle$ ist isomorph zu $(\mathbb{Z}_n, + \bmod n, 0)$.*

Beweis. 1. Wir haben $\langle a \rangle = \{a^0, a^1, a^2, \ldots, a^{n-1}\}$, weil $\mathrm{ord}_G(a) = n$ gilt. Folglich gibt es ein $i \in \{0, 1, 2, \ldots, n - 1\}$, so dass $a^n = a^i$ gilt. Wir behaupten, dass $e = a^0 = a^n$ gilt, und führen einen Widerspruchsbeweis. Wir nehmen also an, dass $a^n = a^i$ gilt für ein $i \in \{1, 2, \ldots, n-1\}$. Wir wissen (Lemma 12.17), dass dann $\mathrm{ord}_G(a) \leq n - i < n$ gilt, ein Widerspruch zur Voraussetzung. Es gilt also $a^n = a^0 = e$.

Wir zeigen jetzt die behauptete Äquivalenz getrennt für beide Richtungen. Sei also zunächst n ein Teiler von $j - i$. Es gibt also ein $q \in \mathbb{Z}$, so dass $q \cdot n = j - i$ gilt. Weil $a^n = e$ gilt, haben wir $a^i = a^i \circ (a^n)^q = a^{q \cdot n + i} = a^j$.

Sei nun umgekehrt $a^i = a^j$. Also ist $a^{j-i} = a^j \circ a^{-i} = a^j \circ a^{-j} = e = a^0$. Wir definieren $r = (j - i) \bmod n$ und $q = \lfloor (j - i)/n \rfloor$, damit ist $j - i = q \cdot n + r$ mit $0 \leq r < n$. Wir haben wieder $a^{j-i} = a^{qn} \circ a^r = a^r$, da $a^n = e$ gilt. Es ist aber auch $a^{j-i} = e$, so dass $a^r = e$ folgt, woraus wir $r = 0$ schließen. Es ist also $j - i = q \cdot n$ und wir sehen, dass n ein Teiler von $j - i$ ist.

2. Man rechnet wieder leicht nach, dass die Abbildung $i \to a^i$ für $i \in \mathbb{Z}_n$ ein Isomorphismus ist.

□

Wir erinnern uns daran, dass wir $\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$ definiert hatten und definieren nun $\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \mid \text{ggT}(a, n) = 1\}$. Es gilt also $\mathbb{Z}_n^* = \mathbb{Z}_n \setminus \{0\}$, wenn n eine Primzahl ist. Es ist nicht schwer einzusehen, dass $(\mathbb{Z}_n^*, \cdot \bmod n, 1)$ eine Gruppe ist. Wir hatten die Ordnung von a als $|\langle a \rangle|$ definiert, im Fall der Gruppe $(\mathbb{Z}_n^*, \cdot, 1)$ ist also $\text{ord}_n(a) := \min\{i \geq 1 \mid a^i \equiv 1 \bmod n\}$ der gleiche Ordnungsbegriff.
Wir nennen eine Zahl $a \in \mathbb{Z}_n^*$ einen *quadratischen Rest* modulo n und schreiben $a \in \text{QR}(n)$, wenn die Gleichung $x^2 \equiv a \bmod n$ eine Lösung in $\mathbb{Z}_n^*$ hat.
Der schon eingeführte Begriff des erzeugenden Elements lässt sich natürlich auch hier verwenden, $\mathbb{Z}_n^*$ hat also a als erzeugendes Element, wenn $\langle a \rangle = \{a^i \mid i \geq 0\} = \mathbb{Z}_n^*$ gilt. Man kann sich überlegen, dass für Primzahlen n stets $\mathbb{Z}_n^*$ ein erzeugendes Element besitzt.
Wir betrachten $\mathbb{Z}_n^*$ für ein beliebiges n mit erzeugendem Element g. Für $a \in \mathbb{Z}_n^*$ nennen wir $\text{index}_g(a) = \min\{i \geq 0 \mid g^i \equiv a \bmod n\}$ den *Index* von a.
Eine Zahl $n \in \mathbb{N}$ nennen wir *quadratfrei*, wenn in ihrer Primfaktorzerlegung nur Exponenten ≤ 1 vorkommen, wenn also $\nu_p(n) \in \{0, 1\}$ für alle Primzahlen p gilt.
Wir beenden diese Aufzählung von unnummerierten Definitionen mit der *Carmichael-Funktion* $\lambda\colon \mathbb{N} \to \mathbb{N}$, die durch

$$\lambda(n) = \min\left\{i \geq 1 \mid \forall a \in \mathbb{Z}_n^*\colon a^i \equiv 1 \bmod n\right\}$$

definiert ist. Interessante und für uns wichtige Eigenschaften der Carmichael-Funktion nennen wir hier ohne Beweis.

Fakt 12.19.

- $\forall$ *ungeraden Primzahlen* $p, e \in \mathbb{N}\colon \lambda\left(e^p\right) = p^{e-1} \cdot (p-1)$
- $\lambda(2) = 1,\ \lambda(4) = 2,\ \forall e \geq 3\colon \lambda\left(2^e\right) = e^{e-2}$
- $\forall n$ *mit* $n = \prod\limits_{p\ prim} p^{\nu_p(n)}\colon \lambda\left(\prod\limits_{p\ prim} p^{\nu_p(n)}\right) = kgV\left(p^{\nu_p(n)} \mid p\ prim\right)$

Auch die anderen eingeführten Begriffe wollen wir nicht abstrakt stehen lassen, sondern mit Leben füllen. Über quadratische Reste zum Beispiel lässt sich folgende Aussage machen.

Theorem 12.20. $\forall p \geq 3$ *prim*: $|QR(p)| = \frac{\left|\mathbb{Z}_p^*\right|}{2} = \frac{p-1}{2}$

Beweis. Eine Zahl $a^2 \in \mathbb{Z}_p^*$ ist quadratischer Rest, wenn $x^2 \equiv a^2 \bmod p$ eine Lösung hat. Lösungen sind genau die Zahlen $x \in \mathbb{Z}_p^*$, die $(x-a) \cdot (x+a) \equiv 0 \bmod p$ erfüllen. Natürlich ist das für die Zahlen x der Fall, die $x \equiv a \bmod p$ oder $x \equiv -a \bmod p$ erfüllen. Weil p eine Primzahl ist, gibt es aber auch keine anderen Zahlen; wir sagen, dass $\mathbb{Z}_p^*$ *nullteilerfrei* ist. Also ist $QR(p) = \left\{x^2 \mid x = 1, 2, \ldots, \frac{p-1}{2}\right\}$ und $|\mathrm{QR}(p)| = (p-1)/2$ folgt. □

Theorem 12.21. $\forall a \in \mathbb{Z}_n^*, e \in \mathbb{N}_0 \colon a^e \equiv 1 \bmod n \Leftrightarrow ord_n(a) \mid e$

Beweis. Es lohnt sich, noch einen Blick auf Theorem 12.18 zu werfen. Wir definieren wie gewohnt $j = e \bmod \mathrm{ord}_n(a)$ und $k = \lfloor e/\mathrm{ord}_n(a) \rfloor$, damit haben $e = k \cdot \mathrm{ord}_n(a) + j$ mit $0 \leq j < \mathrm{ord}_n(a)$. Es gilt also

$$a^e = a^{k \cdot \mathrm{ord}_n(a) + j} = a^{k \cdot \mathrm{ord}_n(a)} \cdot a^j \equiv a^j \bmod n,$$

und wir können $a^e \equiv 1 \bmod n \Leftrightarrow a^j \equiv 1 \bmod n$ schließen, dabei ist $0 \leq j < \mathrm{ord}_n(a)$. Deshalb haben wir $a^e \equiv 1 \bmod n \Leftrightarrow j = 0$ und $\mathrm{ord}_n(a) \mid e$ folgt wie behauptet. □

Wir beschließen jetzt zunächst unseren kleinen Ausflug in die Zahlentheorie mit einem Ergebnis, das immerhin so bekannt ist, dass es einen eigenen Namen hat. Es ist zum Glück sogar einfach zu beweisen.

Theorem 12.22 (**Kleiner Satz von Fermat**). *Sei p prim.*
$\forall a \in \mathbb{Z}_p^* \colon a^{p-1} \equiv 1 \bmod p$

Beweis. Es ist $\langle a \rangle = \{a^i \mid i \geq 0\}$ und $\mathrm{ord}_p(a) = \min\{i \geq 1 \mid a^i \equiv 1 \bmod p\}$. Folglich ist $|\langle a \rangle| = \mathrm{ord}_p(a)$ und es gilt $a^{\mathrm{ord}_p(a)} \equiv 1 \bmod p$.
Wir erinnern uns an die Definition einer Untergruppe (Definition 12.14) und sehen direkt, dass $(\langle a \rangle, \cdot, 1)$ eine Untergruppe von $(\mathbb{Z}_p^*, \cdot, 1)$ ist. Also (Theorem 12.15) ist $|\langle a \rangle| = \mathrm{ord}_p(a)$ ein Teiler von $\left|\mathbb{Z}_p^*\right| = p - 1$, wir können also $p - 1 = k \cdot \mathrm{ord}_p(a)$ für ein $k \in \mathbb{N}$ schreiben. Das liefert $a^{p-1} = a^{\mathrm{ord}_p(a) \cdot k} \equiv 1 \bmod p$. □

So interessant diese Ergebnisse auch sein mögen, eigentlich wollten wir ja einen randomisierten Primzahltest entwickeln. Sind wir diesem Ziel denn inzwischen schon irgendwie näher gekommen? Das ist zum Glück tatsächlich der Fall. Der kleine Satz von Fermat (Theorem 12.22) liefert einen Anhaltspunkt. Wenn p eine Primzahl ist, dann gilt für alle $a \in \mathbb{Z}_p \setminus \{0\}$, dass $a^{p-1} \equiv 1 \bmod p$ ist. Wir können also einfach eine Zahl $a \in \{2, 3, \ldots, p-2\}$ zufällig wählen. Wenn wir dabei einen Teiler von p erwischen, ist p natürlich keine Primzahl. Das war der triviale Primzahltest, der aber eine zu große Fehlschlagswahrscheinlichkeit hat. Nun können wir noch zusätzlich testen, ob

$a^{p-1} \equiv 1 \bmod p$ gilt. Wenn das nicht der Fall ist, dann ist a ein *Zeuge* dafür, dass p keine Primzahl ist. Wir halten diese Ideen in einem Algorithmus fest, den wir Fermat-Test nennen.

Algorithmus 12.23 (**Fermat-Test**).
1. *Wähle* $a \in \{2, 3, \ldots, p-2\}$ *uniform zufällig.*
2. *If* $a \mid p$ *Then Ausgabe „p ist keine Primzahl"; STOP.*
3. *Berechne* $t = a^{p-1} \bmod p$.
4. *If* $t \neq 1$ *Then Ausgabe „p ist keine Primzahl"; STOP.*
5. *Ausgabe „p ist* vielleicht *prim"*

Wir wissen, dass Algorithmus 12.23 korrekt ist und polynomielle Laufzeit in $\log p$ hat. Die spannende Frage ist, wie groß die Wahrscheinlichkeit ist, mit der eine zusammengesetzte Zahl als „vielleicht prim" klassifiziert wird. Wenn wir in dieser Frage besonders pessimistisch sein wollen, führt uns das zur folgenden Definition.

Definition 12.24. *Eine natürliche Zahl* $n \geq 2$ *heißt* Carmichael-Zahl*, wenn* n *keine Primzahl ist und*

$$\forall a \in \mathbb{Z}_n \text{ mit } ggT(a, n) = 1 \colon a^{n-1} \equiv 1 \bmod n$$

gilt.

Eine Carmichael-Zahl p ist offensichtlich das Ende für den Fermat-Test (Algorithmus 12.23): Der Test, ob $a^{p-1} \equiv 1 \bmod p$ gilt, liefert keinerlei Zusatzinformation mehr, es bleibt nur der triviale Test, ob a ein Teiler von p ist, übrig, dessen Erfolgswahrscheinlichkeit wir schon als zu klein zurückgewiesen haben. Aber gibt es Carmichael-Zahlen überhaupt? Das ist leider der Fall, man kann sich davon überzeugen, dass zum Beispiel 561 eine Carmichael-Zahl ist. Wir halten folgende Aussagen der Vollständigkeit halber fest.

Fakt 12.25.
- *Die kleinste Carmichael-Zahl ist* $561 = 3 \cdot 11 \cdot 17$.
- *Es gibt nur 8241 Carmichael-Zahlen, die nicht größer als* 10^{12} *sind.*
- *Es gibt unendlich viele Carmichael-Zahlen.*

Dass Carmichael-Zahlen eher selten sind, mag ein Trost sein, es rettet aber den Algorithmus nicht. Weil es unendlich viele solche Zahlen gibt, können wir den Algorithmus nicht um eine Liste dieser Zahlen erweitern und wir kennen keine effiziente Methode, um von einer Zahl festzustellen, ob sie eine Carmichael-Zahl ist. Grundsätzlich können wir Carmichael-Zahlen allerdings schon charakterisieren.

Theorem 12.26. *Eine zusammengesetzte natürliche Zahl $n \geq 2$ ist Carmichael-Zahl genau dann, wenn $n-1$ Vielfaches von $\lambda(n)$ ist.*

Beweis. Wir führen den Äquivalenzbeweis getrennt für die beiden Richtungen und nehmen zunächst an, dass $n-1$ ein Vielfaches von $\lambda(n)$ ist, es gibt also ein $k \in \mathbb{N}$, so dass $n-1 = k \cdot \lambda(n)$ ist. Aus der Definition der Carmichael-Funktion (Definition 12.24) folgt, dass $a^{\lambda(n)} \equiv 1 \bmod n$ für alle $a \in \mathbb{Z}_n$ gilt und $a^{n-1} = a^{\lambda(n)\cdot k} \equiv 1 \bmod n$ folgt.
Sei nun umgekehrt n eine Carmichael-Zahl. Wir erinnern uns (Theorem 12.21), dass $a^e \equiv 1 \bmod n$ genau dann gilt, wenn e ein Vielfaches von $\mathrm{ord}_n(a)$ ist. Wenn wir uns nun alle $a \in \mathbb{Z}_n^*$ anschauen, so gilt $a^e \equiv 1 \bmod n$ genau dann für alle a, wenn e ein Vielfaches von $\mathrm{ord}_n(a)$ für alle a ist. Wir haben also

$$\forall a \in \mathbb{Z}_n \colon a^e \equiv 1 \bmod n \Leftrightarrow \mathrm{kgV}\left\{\mathrm{ord}_n(a) \mid a \in \mathbb{Z}_n^*\right\} \mid e$$

gezeigt. Und weil $\mathrm{kgV}\left\{\mathrm{ord}_n(a) \mid a \in \mathbb{Z}_n^*\right\} = \lambda(n)$ gilt, ist der Beweis damit vollständig. □

Weil der Fermat-Test für Carmichael-Zahlen nicht funktioniert, werden wir uns also noch mehr Mühe geben müssen, um eine zusammengesetzte Zahl zuverlässiger als zusammengesetzt zu entlarven. Bevor wir dazu noch etwas mehr mit Zahlentheorie beschäftigen, wollen wir uns kurz ansehen, wie gut der Fermat-Test für Zahlen funktioniert, die nicht Carmichael-Zahlen sind.

Theorem 12.27. *Sei $n \geq 3$ ungerade und zusammengesetzt, außerdem gebe es ein $a \in \mathbb{Z}_n^*$ mit $a^{n-1} \not\equiv 1 \bmod n$. Der Fermat-Test erkennt n als zusammengesetzt mit Wahrscheinlichkeit mindestens $1/2$.*

Beweis. Wir betrachten $L := \{a \in \mathbb{Z}_n^* \mid a^{n-1} \equiv 1 \bmod n\}$. Natürlich ist $1 \in L$. Außerdem ist leicht einzusehen, dass L unter Multiplikation abgeschlossen ist: $(a \cdot b)^{n-1} = a^{n-1} \cdot b^{n-1} \equiv 1 \cdot 1 \bmod n$ für alle $a, b \in L$. Also ist L eine Untergruppe von $\mathbb{Z}_n^*$ und es folgt $|L| \mid |\mathbb{Z}_n^*|$. Weil $L \neq \mathbb{Z}_n^*$ gilt (andernfalls wäre n ja eine Carmichael-Zahl), können wir jetzt die Versagenswahrscheinlichkeit gut abschätzen. Die Größe von $\mathbb{Z}_n^*$ ist durch $n-2$ nach oben beschränkt, wir haben also $|L| \leq (n-2)/2$. Wir wählen uniform zufällig ein $a \in \{2, 3, \ldots, n-2\}$ und erkennen n nicht als zusammengesetzt, wenn $a \in L$ gilt. Damit ist die Wahrscheinlichkeit, n nicht als zusammengesetzt zu erkennen, durch

$$\frac{|L \setminus \{1, n-1\}|}{|\{2, 3, \ldots, n-2\}|} \leq \frac{(n-2)/2 - 2}{n-3} = \frac{1}{2} \cdot \frac{n-6}{n-3} < \frac{1}{2}$$

nach oben beschränkt. □

Eine Erfolgswahrscheinlichkeit von mindestens $1/2$ ist für praktische Zwecke völlig ausreichend. Probability Amplification liefert mit $O(\log n)$ Wiederholungen eine Erfolgswahrscheinlichkeit von mindestens $1 - O\big(1/n^k\big)$ für jede beliebige Konstante k. Alle folgenden Überlegungen sind also tatsächlich alleine durch die Existenz von Carmichael-Zahlen motiviert. Wir beginnen mit einer Charakterisierung von $a \in \mathbb{Z}_p^*$ (p prim), die man Eulerkriterium nennt. Es wird die darauffolgende Definition des Legendre-Symbols motivieren.

Lemma 12.28. *Sei $p \geq 3$ prim.*

$$\forall a \in \mathbb{Z}_p^*\colon a^{(p-1)/2} = \begin{cases} 1 & \textit{falls } a \in QR(p) \\ -1 & \textit{sonst} \end{cases}$$

Beweis. Eine Zahl $a \in \mathbb{Z}_p^*$ heißt quadratischer Rest, wenn $x^2 \equiv a \bmod p$ eine Lösung hat. Weil p prim ist, wissen wir, dass $x^{p-1} \equiv 1 \bmod p$ ist (Theorem 12.22). Weil $p \geq 3$ ist, können wir auch $y^{(p-1)/2} \bmod p$ für beliebige $y \in \mathbb{Z}_p^*$ betrachten. Wir sehen, dass $a^{(p-1)/2}$ ein quadratischer Rest ist, wenn $x^{p-1} \equiv a^{(p-1)/2} \bmod p$ gilt. Es ist also $a^{(p-1)/2}$ quadratischer Rest genau dann, wenn $a^{(p-1)/2} \equiv 1 \bmod p$ ist. Wenn $a^{(p-1)/2}$ kein quadratischer Rest ist, dann muss $a^{(p-1)/2} \equiv -1 \bmod p$ gelten: Aus $a^{p-1} \equiv 1 \bmod p$ folgt ja, dass $a^{(p-1)/2}$ kongruent zu 1 oder -1 sein muss. □

Definition 12.29. *Für $p \geq 3$ prim und $a \in \mathbb{Z}$ ist*

$$\left(\frac{a}{p}\right) := \begin{cases} 1 & \textit{falls } a \in QR(p) \\ 0 & \textit{falls } p \mid a \\ -1 & \textit{sonst} \end{cases}$$

das Legendre-Symbol *von a und p.*

Wir notieren einige einfache Rechenregeln für das Legendre-Symbol, die jeweils einfach nachzuweisen sind, hier ohne Beweis. Alle Aussagen setzen natürlich voraus, dass $p \geq 3$ eine Primzahl ist.

- $\forall a, b \in \mathbb{Z}\colon \left(\frac{a \cdot b}{p}\right) = \left(\frac{a}{p}\right) \cdot \left(\frac{b}{p}\right)$
- $\forall a, b \in \mathbb{Z}$ mit $p \nmid b\colon \left(\frac{a \cdot b^2}{p}\right) = \left(\frac{a}{p}\right)$
- $\forall a, b \in \mathbb{Z}\colon \left(\frac{a + b \cdot p}{p}\right) = \left(\frac{a}{p}\right)$
- $\forall a \in \mathbb{Z}\colon \left(\frac{a}{p}\right) = \left(\frac{a \bmod p}{p}\right)$

- $\left(\frac{-1}{p}\right) = (-1)^{(p-1)/2}$

Wir können die Definition des Legendre-Symbols so erweitern, dass sie auch für ungerade zusammengesetzte Zahlen definiert ist. Wir machen das so, dass wir die neue Definition auf das Legendre-Symbol der Primfaktoren der zusammengesetzten Zahl zurückführen. Weil das Legendre-Symbol nicht für 2 definiert ist, muss die zusammengesetzte Zahl ungerade sein. Bei der erweiterten Definition sprechen wir vom Jacobi-Symbol.

Definition 12.30. *Sei $n \geq 3$ ungerade mit $n = \prod_{i=1}^{r} p_i$, dabei sei p_i prim für alle $i \in \{1, 2, \ldots, r\}$. Für jedes $a \in \mathbb{Z}$ ist*

$$\left(\frac{a}{n}\right) := \prod_{i=1}^{r} \left(\frac{a}{p_i}\right)$$

das Jacobi-Symbol *von a und n.*

Wir sehen sofort, dass für ungerade Primzahlen n das Jacobi-Symbol gerade mit dem Legendre-Symbol zusammenfällt. Für das Jacobi-Symbol können wir genau wie für das Legendre-Symbol einige einfache Rechenregeln festhalten. Dabei setzen wir voraus, dass $n, m \geq 3$ beide ungerade sind und $a, b \in \mathbb{Z}$.

- $\left(\frac{a \cdot b}{n}\right) = \left(\frac{a}{n}\right) \cdot \left(\frac{b}{n}\right)$, falls $\mathrm{ggT}(b, n) = 1$
- $\left(\frac{a \cdot b^2}{n}\right) = \left(\frac{a}{n}\right)$, falls $\mathrm{ggT}(b, n) = 1$
- $\left(\frac{a}{n \cdot m}\right) = \left(\frac{a}{n}\right) \cdot \left(\frac{a}{m}\right)$, falls $\mathrm{ggT}(a, m) = 1$
- $\left(\frac{a}{n \cdot m^2}\right) = \left(\frac{a}{n}\right)$, falls $\mathrm{ggT}(a, m) = 1$
- $\left(\frac{a + b \cdot n}{n}\right) = \left(\frac{a}{n}\right)$
- $\left(\frac{a}{n}\right) = \left(\frac{a \bmod n}{n}\right)$
- $\forall k \in \mathbb{N}$: $\left(\frac{2^{2k} \cdot a}{n}\right) = \left(\frac{a}{n}\right)$
- $\forall k \in \mathbb{N}$: $\left(\frac{2^{2k+1} \cdot a}{n}\right) = \left(\frac{2}{n}\right) \cdot \left(\frac{a}{n}\right)$
- $\left(\frac{-1}{n}\right) = (-1)^{(n-1)/2}$
- $\left(\frac{0}{n}\right) = 0$

- $\left(\frac{1}{n}\right) = 1$

Es gibt noch zwei weitere Rechengesetze, die für uns nützlich sein werden, die aber nicht so direkt einzusehen sind. Wir geben hier beide ohne Beweis an.

Fakt 12.31 (**Quadratisches Reziprozitätsgesetz**). *Für ungerade $m, n \geq$ gilt:*

$$\left(\frac{m}{n}\right) = \begin{cases} \left(\frac{n}{m}\right) & \text{falls } n \equiv 1 \bmod 4 \text{ oder } m \equiv 1 \bmod 4 \\ -\left(\frac{n}{m}\right) & \text{falls } n \equiv 3 \bmod 4 \text{ und } m \equiv 3 \bmod 4 \end{cases}$$

Fakt 12.32. *Für ungerades $n \geq 3$ gilt:*

$$\left(\frac{2}{n}\right) = \begin{cases} 1 & \text{falls } n \equiv 1 \bmod 8 \text{ oder } n \equiv 7 \bmod 8 \\ -1 & \text{falls } n \equiv 3 \bmod 8 \text{ oder } n \equiv 5 \bmod 8 \end{cases}$$

Wenn man diese Rechenregeln berücksichtigt, kann man das Jacobi-Symbol auf naheliegende Weise rekursiv beschreiben. Der Vorteil dieser Beschreibung ist, dass sie eine Methode zur Berechnung nahelegt. Wir werden zunächst diese rekursive Beschreibung angeben und sie danach zu einem effizienten Algorithmus zur Berechnung des Jacobi-Symbols ausbauen.
Zur Berechnung von $\left(\frac{a}{n}\right)$ für ein beliebiges $a \in \mathbb{Z}$ und ein ungerades $n \geq 3$ kann man wie folgt vorgehen.

1. Falls $a \notin \{1, 2, \ldots, n-1\}$, ist das Ergebnis $\left(\frac{a \bmod n}{a}\right)$.
2. Falls $a = 0$, ist das Ergebnis 0.
3. Falls $a = 1$, ist das Ergebnis 1.
4. Falls $4 \mid a$, ist das Ergebnis $\left(\frac{a/4}{n}\right)$.
5. Falls $2 \mid a$, unterscheiden wir weiter: Falls $n \bmod 8 \in \{1, 7\}$, ist das Ergebnis $\left(\frac{a/2}{n}\right)$, sonst ist das Ergebnis $-\left(\frac{a/2}{n}\right)$.
6. Falls $a \equiv 1 \bmod 4$ oder $n \equiv 1 \bmod 4$, ist das Ergebnis $\left(\frac{n \bmod a}{a}\right)$.
7. Falls $a \equiv 3 \bmod 4$ oder $n \equiv 3 \bmod 4$, ist das Ergebnis $-\left(\frac{n \bmod a}{a}\right)$.

Algorithmus 12.33 (**Berechnung des Jacobi-Symbols**).
Eingabe *ungerades $n \geq 3$, $a \in \mathbb{Z}$*
Ausgabe $\left(\frac{a}{n}\right)$
1. $b := a \bmod c$; $c := n$; $s := 1$

```
2.  While b > 1
3.      While 4 | b
            b := b/4
4.      If 2 | b Then
5.          If c mod 8 ∈ {3, 5} Then s := −s
6.          b := b/2
7.      If b ≠ 1 Then
8.          If (b mod 4) = (c mod 4) = 3 Then s := −s
9.          (b, c) := (c mod b, b)
10. Ausgabe s · b
```

Theorem 12.34. *Algorithmus 12.33 berechnet* $\left(\frac{a}{n}\right)$ *in* $O(\log n)$ *Durchläufen der While-Schleife (Zeilen 2–9).*

Beweis. Wir beginnen mit dem Nachweis der Korrektheit und zeigen, dass jederzeit $\left(\frac{a}{n}\right) = s \cdot \left(\frac{b}{c}\right)$ gilt. Initial haben wir $s = 1$, $b = a \bmod c$ und $c = n$, so dass die Gleichheit gilt. Wir können jetzt induktiv annehmen, dass zu Beginn eines Durchlaufs der While-Schleife die Gleichung erfüllt ist und müssen zeigen, dass sie erfüllt bleibt. Das ist offensichtlich der Fall, da die Zeilen 3–6 genau die entsprechenden Rechenregeln umsetzen, in Zeile 7 sichergestellt wird, dass für den Fall $b = 1$ nichts mehr getan wird, und schließlich in den Zeilen 8 und 9 das quadratische Reziprozitätsgesetzt (Fakt 12.31) angewendet wird. Der Algorithmus ist also korrekt.

Dass die Anzahl der While-Durchläufe durch $O(\log n)$ nach oben beschränkt ist, ist leicht einzusehen. Die zentrale Beobachtung ist, dass am Ende eines Durchlaufs entweder das Paar (b, c) durch das Paar $(c \bmod b, b)$ ersetzt wird oder die While-Schleife beendet wird, weil $b = 1$ gilt. Wir vergleichen das mit dem euklidischen Algorithmus (Algorithmus 12.4): Der Laufzeitbeweis (Theorem 12.5) beruht darauf, dass in jedem Durchlauf das Paar (x, y) durch das Paar $(y \bmod x, x)$ ersetzt wird. Bei der Berechnung des Jacobi-Symbols kann b zusätzlich nur weiter verkleinert werden, was die Laufzeit nur zusätzlich senken kann. □

Der Schlüssel für einen effizienten randomisierten Primzahltest, der auch für Carmichael-Zahlen funktioniert, liegt im Jacobi-Symbol. Um das einzusehen, brauchen wir eine Strukturaussage dazu. Zu deren Beweis brauchen wir den chinesischen Restsatz, der vermutlich bekannt ist. Wir wiederholen ihn der Vollständigkeit halber.

Theorem 12.35 (**Chinesischer Restsatz**). *Seien $m_1, m_2, \ldots, m_k \in \mathbb{N}$ mit $ggT(m_i, m_j) = 1$ für alle $i \neq j$ und $a_1, a_2, \ldots, a_k \in \mathbb{N}$. Das Gleichungssystem*

$$\begin{aligned} x &\equiv a_1 \bmod m_1 \\ x &\equiv a_2 \bmod m_2 \\ &\vdots \\ x &\equiv a_k \bmod m_k \end{aligned}$$

hat eine eindeutige Lösung $x \in \left\{0, 1, \ldots, \left(\prod_{i=1}^{k} m_i\right) - 1\right\}$.

Beweis. Wir definieren $m := \prod_{i=1}^{k} m_i$ und $n_i := m/m_i$. Natürlich ist $n_i \in \mathbb{N}$, außerdem gilt $\mathrm{ggT}(m_i, n_i) = 1$ und wir haben $s_i \cdot m_i + t_i \cdot n_i = 1$ für geeignete $s_i, t_i \in \mathbb{Z}$: Solche s_i und t_i kann man sich zum Beispiel im Rahmen des euklidischen Algorithmus mitberechnen lassen.
Wir betrachten $u_i = t_i \cdot n_i$ und sehen, dass $u_i = 1 - s_i \cdot m_i \equiv 1 \bmod m_i$ gilt. Außerdem ist $u_i \equiv 0 \bmod m_j$ für $j \neq i$. Folglich ist $x = a_1u_1 + a_2u_2 + \cdots + a_ku_k$ eine Lösung des Gleichungssystems.
Sei x' eine andere Lösung. Dann gilt aber $x - x' \equiv 0 \bmod m_i$ für alle i. Weil alle m_i also $x - x'$ teilen und alle m_i paarweise teilerfremd sind, ist auch ihr Produkt ein Teiler von $x - x'$. Folglich kann es in $\left\{0, 1, \ldots, \left(\prod_{i=1}^{k} m_i\right) - 1\right\}$ wie behauptet nur eine Lösung geben. □

Theorem 12.36. *Sei für ungerades $n \geq 3$ die Menge $E(n)$ definiert durch $E(n) := \left\{a \in \mathbb{Z}_n^* \mid a^{(n-1)/2} \equiv \left(\frac{a}{n}\right) \bmod n\right\}$.*

$$\begin{aligned} n \text{ prim} &\Rightarrow E(n) = \mathbb{Z}_n^* \\ n \text{ nicht prim} &\Rightarrow |E(n)| \leq |\mathbb{Z}_n^*|/2 \end{aligned}$$

Wir erinnern uns an den Fermat-Test (Algorithmus 12.23). Die Stärke für Zahlen, die nicht Carmichael-Zahlen sind, beruht darauf, dass die Zahlen a, die nicht als Zeugen dafür dienen, dass die Eingabe nicht prim ist, eine Untergruppe bilden und darum höchstens die Hälfte aller Zahlen diese Eigenschaft haben können (Theorem 12.27). Jetzt haben wir ein ähnliches Resultat, das für alle ungeraden Zahlen $n \geq 3$ funktioniert, also insbesondere auch für Carmichael-Zahlen. Wir können für ein zufällig gewähltes $a \in \mathbb{Z}_n^*$ das Jacobi-Symbol $\left(\frac{a}{n}\right)$ berechnen und $a^{(n-1)/2} \bmod n$. Wenn n nicht prim ist, sind die Ergebnisse mit Wahrscheinlichkeit mindestens $1/2$ verschieden. Bevor wir den entsprechenden Algorithmus ausformulieren, wollen wir Theorem 12.36 aber noch beweisen.

Beweis von Theorem 12.36. Wir beobachten zunächst, dass $a^{(n-1)/2} \bmod n$ wohldefiniert ist, weil $n \geq 3$ und ungerade ist. Aus gleichem Grund ist auch $\left(\frac{a}{n}\right)$ wohldefiniert und als Konsequenz ist die Menge $E(n)$ wohldefiniert.
Wir führen den Beweis zunächst für den Fall, dass n eine Primzahl ist. Für Primzahlen n entspricht das Jacobi-Symbol dem Legendre-Symbol, das Legendre-Symbol ist definiert als $\left(\frac{a}{n}\right) = a^{(n-1)/2} \bmod n$ für alle $a \in \mathbb{Z}_n^*$ (Definition 12.29 und Lemma 12.28). Folglich ist $E(n) = \mathbb{Z}_n^*$ wie behauptet.
Sei nun n zusammengesetzt. Wir behaupten, dass $E(n) \neq \mathbb{Z}_n^*$ gilt und führen einen Beweis durch Widerspruch. Gemäß Definition von $E(n)$ enthält $E(n)$ alle $a \in \mathbb{Z}_n^*$, für die $a^{(n-1)/2} \equiv 1 \bmod n$ ist. Wir nehmen an, dass $E(n) = \mathbb{Z}_n^*$ gilt, also ist n eine Carmichael-Zahl. Wir behaupten zusätzlich, dass n ungerade und quadratfrei ist. Dass n ungerade ist, gehört zu unseren Voraussetzungen, wir müssen also nur nachweisen, dass n quadratfrei ist. Dazu führen wir wiederum einen Widerspruchsbeweis und nehmen an, dass n nicht quadratfrei ist. Es gibt also eine Primzahl p, so dass p^2 ein Teiler von n ist; natürlich ist dann auch p ein Teiler von n. Weil n ungerade ist, ist $p > 2$. Theorem 12.26 liefert, dass p ein Teiler von $n-1$ ist. Aus der Annahme, dass n nicht quadratfrei ist, folgt also, dass $p > 2$ sowohl n als auch $n-1$ teil: ein offensichtlicher Widerspruch.
Wir haben jetzt also, dass $n > 2$ ungerade ist, eine Carmichael-Zahl und quadratfrei. Es gibt also eine Primzahl p und eine natürliche Zahl $r > 1$, so dass $\mathrm{ggT}(r, p) = 1$ gilt und $n = p \cdot r$ ist. Theorem 12.20 liefert, dass $|\mathrm{QR}(p)| = (p-1)/2$ gilt. Es gibt also ein g, so dass g quadratischer Nichtrest modulo p ist. Wir betrachten

$$\begin{aligned} a &\equiv g \bmod p \\ a &\equiv 1 \bmod r \end{aligned}$$

und sehen, dass es ein a gibt, das dieses Gleichungssystem erfüllt (chinesischer Restsatz (Theorem 12.35)). Weil $\mathrm{ggT}(a, n) = 1$ gilt, ist $a \in \mathbb{Z}_n^*$. Wenn unsere Annahme $E(n) = \mathbb{Z}_n^*$ richtig ist, muss auch $a \in E(n)$ gelten. Das ist aber nicht der Fall, wie wir uns jetzt überlegen. Wir benutzen die Rechenregeln für das Jacobi-Symbol und sehen, dass

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p \cdot r}\right) = \left(\frac{a}{p}\right) \cdot \left(\frac{a}{r}\right) = \left(\frac{g}{p}\right) \cdot \left(\frac{1}{r}\right) = (-1) \cdot 1 = -1$$

gilt. Wir haben $a^{(n-1)/2} \equiv 1 \bmod r$ und wissen, dass $a^{(n-1)/2} \equiv 1 \bmod n$ oder $a^{(n-1)/2} \equiv -1 \bmod n$ gilt (Lemma 12.28). Weil $n = p \cdot r$ gilt (mit p prim und $r > 2$), ist $a^{(n-1)/2} \equiv 1 \bmod n$. Damit ist aber $(a^{(n-1)/2} \bmod n) \neq \left(\frac{a}{n}\right)$ und $a \notin E(n)$ folgt, ein Widerspruch.

Bis jetzt haben wir gezeigt, dass $E(n) = \mathbb{Z}_n^*$ genau dann gilt, wenn n prim ist. Wir müssen noch zeigen, dass für zusammengesetzte n zusätzlich $|E(n)| \leq |\mathbb{Z}_n^*|/2$ gilt. Dazu genügt es offenbar zu zeigen, dass $E(n)$ eine Untergruppe ist. Natürlich ist das neutrale Element $1 \in E(n)$, wir müssen also nur noch die Abgeschlossenheit unter Multiplikation nachweisen.
Seien $x, y \in E(n)$, wir wollen zeigen, dass $x \cdot y \in E(n)$ gilt. Gemäß Definition von $E(n)$ haben wir $x^{(n-1)/2} \equiv \left(\frac{x}{n}\right) \bmod n$ und $y^{(n-1)/2} \equiv \left(\frac{y}{n}\right) \bmod n$. Also ist $(x \cdot y)^{(n-1)/2} = x^{(n-1)/2} \cdot y^{(n-1)/2} \equiv \left(\frac{x}{n}\right) \cdot \left(\frac{y}{n}\right) \bmod n$. Weil $\left(\frac{x}{n}\right) \cdot \left(\frac{y}{n}\right) = \left(\frac{x \cdot y}{n}\right)$ gilt, ist wie behauptet $x \cdot y \in E(n)$. □

Wir können nun einen effizienten randomisierten Primzahltest mit einseitigem Fehler präsentieren, in dessen Kern das Jacobi-Symbol und Theorem 12.36 stehen. Der Algorithmus scheint aus einer Folge von teilweise trivialen Primzahltests zu bestehen. Tatsächlich sind die ersten Tests aber notwendig, weil wir in Theorem 12.36 voraussetzen, dass

1. $n \geq 3$ gilt,
2. n ungerade ist und
3. $n \in \mathbb{Z}_n^*$ ist, also $\mathrm{ggT}(a, n) = 1$ gilt.

Algorithmus 12.37 (**Algorithmus von Solovay und Strassen**).
Eingabe $n \in \mathbb{N} \setminus \{1\}$, $k \in \mathbb{N}$
1. If $n = 2$ *Then Ausgabe „n ist prim“; STOP.*
2. If n gerade Then Ausgabe „n ist zusammengesetzt“; STOP.
3. While $k \geq 1$
4. Wähle $a \in \{2, 3, \ldots, n-1\}$ *uniform zufällig.*
5. Berechne $d := ggT(a, n)$.
6. If $d \neq 1$ *Then Ausgabe „n ist zusammengesetzt“; STOP.*
7. Berechne $b \equiv a^{(n-1)/2} \bmod n$.
8. If $b \notin \{-1, 1\} \bmod n$ *Then Ausgabe „n ist zusammengesetzt“; STOP.*
9. Berechne $c := \left(\frac{a}{n}\right)$.
10. If $b \not\equiv c \bmod n$ *Then Ausgabe „n ist zusammengesetzt“; STOP.*
11. $k := k - 1$
12. Ausgabe „n ist vermutlich prim“

Theorem 12.38. *Für den Algorithmus von Solovay und Strassen (Algorithmus 12.37) gelten bei Eingabe von* $n \in \mathbb{N} \setminus \{1, 2\}$ *und* $k \in \mathbb{N}$ *die folgenden Aussagen.*

1. *Die Rechenzeit beträgt* $O(k \cdot \log n)$.
2. *Für Primzahlen n wird „n ist vermutlich prim“ ausgegeben.*

3. *Für zusammengesetzte n wird mit Wahrscheinlichkeit mindestens $1 - 2^{-k}$ „n ist zusammengesetzt“ ausgegeben.*

Beweis. 1. Wir haben schon gezeigt, dass jede Zeile in Zeit $O(\log n)$ ausführbar ist. Offenbar gibt es genau k Durchläufe der While-Schleife, so dass sich Gesamtrechenzeit $O(k \cdot \log n)$ ergibt.

2. Die Ausgabe „n ist zusammengesetzt“ erfolgt nur, wenn n gerade ist (für Primzahlen $n > 2$ nicht möglich), $\mathrm{ggT}(a, n) \neq 1$ gilt (für Primzahlen nicht möglich) oder $a^{(n-1)/2} \not\equiv \left(\frac{a}{n}\right) \bmod n$ gilt, was für $a \in \mathbb{Z}_n^*$ und Primzahlen $n > 2$ nicht der Fall sein kann. Also kann für Primzahlen $n > 2$ nur „n ist vermutlich prim“ ausgegeben werden.

3. Wir betrachten einen Durchlauf der While-Schleife und ein zusammengesetztes n. Es genügt zu zeigen, dass mit Wahrscheinlichkeit mindestens $1/2$ die Ausgabe „n ist zusammengesetzt“ erfolgt. Die Aussage über die Gesamterfolgswahrscheinlichkeit folgt mit den üblichen Argumenten über Probability Amplification, da die Wahl von a jeweils unabhängig uniform zufällig erfolgt.

   Wenn n gerade ist, wird mit Wahrscheinlichkeit 1 die Ausgabe „n ist zusammengesetzt“ erzeugt (Zeile 2), wir dürfen also voraussetzen, dass n ungerade ist. Falls $\mathrm{ggT}(a, n) \neq 1$ gilt, wird wiederum mit Wahrscheinlichkeit 1 „n ist zusammengesetzt“ ausgegeben, wir dürfen also zusätzlich voraussetzen, dass $\mathrm{ggT}(a, n) = 1$ gilt. Damit sind die Voraussetzungen für die Anwendung von Theorem 12.36 erfüllt und wir wissen, dass $a^{(n-1)/2} \equiv \left(\frac{a}{n}\right) \bmod n$ höchstens mit Wahrscheinlichkeit $1/2$ gilt. □

Wir hatten die Entwicklung eines randomisierten Primzahltests mit dem RSA-Kryptosystem motiviert, für das die Erzeugung großer Primzahlen erforderlich ist. Wir weisen noch einmal darauf hin, dass der Primzahltest von Solovay und Strassen (Algorithmus 12.37) keine Hinweise auf Primfaktoren liefert, so dass die Sicherheit des RSA-Systems hier nicht berührt wird. Bevor wir das Kapitel beschließen, wollen wir noch eine Lücke schließen, die wir bei der Vorstellung des RSA-Kryptosystems hinterlassen haben. Wir hatten für $n = p \cdot q$ mit zwei Primzahlen p und q die Eulerfunktion $\Phi(n) = |\mathbb{Z}_n^*| = (p-1) \cdot (q-1)$ berechnet, ein e mit $\mathrm{ggT}(e, \Phi(n)) = 1$ und ein d mit $d \equiv e^{-1} \bmod \Phi(n)$. Für die Codierung eines Blocks m hatten wir $m^e \bmod n$ als Chiffrierungsfunktion und für die Decodierung eines Kryptogramms m^e hatten wir $(m^e)^d \bmod n$ als Dechiffrierungsfunktion angegeben. Wir hatten nicht nachgewiesen, dass wir dadurch tatsächlich wieder die ursprüngliche Nachricht m erhalten. Das können wir jetzt nachreichen, der

chinesische Restsatz, den wir in der Zwischenzeit kennengelernt haben, wird sich dabei als nützlich erweisen.
Wir haben $e \cdot d \equiv 1 \bmod (p-1) \cdot (q-1)$, weil $d \equiv e^{-1} \bmod \Phi(n)$ und $\Phi(n) = (p-1) \cdot (q-1)$ gilt. Es gibt also ein $t \in \mathbb{N}_0$, so dass $e \cdot d = 1 + t \cdot (p-1) \cdot (q-1)$ gilt. Also ist $(m^e)^d = m^{1+t\cdot(p-1)\cdot(q-1)}$. Bevor wir zeigen, dass $m^{1+t\cdot(p-1)\cdot(q-1)} \equiv m \bmod (p-1) \cdot (q-1)$ ist, formulieren wir noch ein hilfreiches Lemma.

Lemma 12.39. *Für alle Primzahlen p und alle $a, k \in \mathbb{N}_0$ gilt $a^{1+k\cdot(p-1)} \equiv a \bmod p$.*

Beweis. Wenn $a \equiv 0 \bmod p$ gilt, ist die Aussage offensichtlich richtig. Sei also $a \not\equiv 0 \bmod p$, also $\mathrm{ggT}(a, p) = 1$. Wir haben offensichtlich $a^{1+k\cdot(p-1)} \equiv a \cdot (a^{p-1})^k \bmod p$. Der kleine Satz von Fermat (Theorem 12.22) liefert, dass $a^{p-1} \equiv 1 \bmod p$ ist, also folgt $a^{1+k\cdot(p-1)} \equiv a \bmod p$. □

Anwendung von Lemma 12.39 liefert also sowohl, dass $(m^e)^d \equiv m \bmod p$ gilt, als auch, dass $(m^e)^d \equiv m \bmod q$ gilt. Der chinesische Restsatz (Theorem 12.35) liefert nun, dass es genau ein $x \in \{0, 1, \ldots, p \cdot q - 1\}$ gilt, das $x \equiv m \bmod p$ und $x \equiv m \bmod q$ erfüllt. Wir sehen, dass m diese Eigenschaft hat und können folgern, dass die eindeutige Lösung wie gewünscht m ist, die Decodierung also tatsächlich funktioniert.

Teil III

Heuristiken

13 Allgemeine randomisierte Suchheuristiken

Wir wissen schon aus dem Grundstudium, dass es in der Praxis wichtige Probleme gibt, die sehr schwierig zu lösen sind. Die Schwierigkeit von Problemen hatte uns dazu motiviert, vom Ziel der exakten Lösung abzurücken und uns mit approximativen Lösungen zufriedenzugeben. Aber auch approximativ lassen sich längst nicht alle Probleme zufriedenstellend lösen: Wir hatten zum Beispiel für das metrische TSP angemerkt (Abschnitt 9.1), dass es nicht beliebig gut approximierbar ist, wenn $P \neq NP$ gilt. Wenn man in der unangenehmen Situation ist, dass ein Problem dringend algorithmisch gelöst werden muss und man möglichst gute Lösungen braucht, man aber außer Stande ist, einen dafür nachweislich geeigneten Algorithmus zu finden oder zu entwerfen, ist man gezwungen, andere Auswege zu finden. Manchmal lässt ein Problem sich umgehen, eine genaue Analyse der Situation offenbart vielleicht, dass tatsächlich nur Teilprobleme gelöst werden müssen, die einfacher zu lösen sind als das allgemeine Problem. Manchmal steht man aber vor einem hartnäckig schwierigen Problem und ist gezwungen, irgendetwas damit zu tun. In solchen Situationen greift man häufig zu heuristischen Verfahren, zu Algorithmen, von denen man hofft, dass sie oft gute Lösungen berechnen können, von denen vielleicht sogar intuitiv plausibel ist, dass sie gute Performanz zeigen, man aber nicht in der Lage ist, einen entsprechenden Beweis zu führen. Wir wollen an dieser Stelle gleich eine Warnung einfügen: Hoffnungen können enttäuscht werden und die Intuition kann völlig fehlgehen. Clevere Heuristiken, die in Versuchen sehr überzeugend abschneiden, können bei schwierigen Instanzen völlig versagen und die schwierigsten Instanzen treten in der Praxis gerne dann auf, wenn man sie am wenigsten brauchen kann. Man sollte also nur mit gebührender Vorsicht zu heuristischen Algorithmen greifen und nicht zu optimistisch bezüglich ihrer Performanz sein. Andererseits darf auch nicht verschwiegen werden, dass in der Praxis gerade heuristische Verfahren besonders performant sein können.

Wir unterscheiden zwei Arten von Heuristiken, wobei die Trennlinie nicht immer ganz scharf gezogen werden kann. Die Unterscheidung ist aber trotzdem grundsätzlich nützlich, weil die beiden Heuristikarten von unterschiedlichen Vorstellungen ausgehen und darum ganz unterschiedliche Ausprägungen haben. Wenn man den Weg zu einer Heuristik geht, wie wir ihn hier vorgezeichnet haben, vom Wunsch, einen nachweislich effizienten Algorithmus zu finden über die Suche nach einem nachweislich gut approximierenden Algorithmus bis schließlich zu einer Heuristik, so denkt man an einen problemspezifischen Algorithmus, den man so entwirft, dass man glaubt und hofft, dass er vie-

le typische Instanzen effizient löst. Man entwickelt den Algorithmus so für das Problem, dass möglichst viele Einsichten in die Problemstruktur algorithmisch ausgenutzt werden, um möglichst schnell möglichst gute Lösungen zu finden. Wir werden ein Beispiel für eine solche Heuristik kennenlernen im Abschnitt 14.1. Zunächst werden wir uns aber mit einer ganz anderen Klasse von Heuristiken beschäftigen, bei denen man einen ganz merkwürdigen Ausgangspunkt nimmt: Wir werden allgemeine randomisierte Suchheuristiken besprechen, bei denen eine Basisidee, wie man möglichst effizient nach Lösungen zu einem Problem sucht, vorhanden ist, man aber auf das konkrete zu lösende Problem gar keinen Bezug nimmt. Man nimmt an, dass alle möglichen Lösungen so in einem Suchraum S codiert sind, dass man Punkte $s \in S$ effizient erzeugen und auch effizient randomisiert variieren kann. Außerdem soll eine Bewertungsfunktion $f\colon S \to \mathbb{R}$ bekannt sein, die zu maximieren ist. Es gibt eine Vielzahl verschiedener solche allgemeiner Suchheuristiken, die sich darin unterscheiden, wie sie in diesem Suchraum S suchen, einige konkrete Beispiele werden wir uns im nächsten Abschnitt ansehen. Hier wollen wir noch vorab klären, wie man eigentlich auf die Idee kommen kann, problemunabhängige Suchverfahren entwickeln und verwenden zu wollen.
Bei der Lösung eines Optimierungsproblems in der Praxis gibt es mehrere Faktoren, die bestimmen, wie lange es dauert, bis man eine ausreichend gute Lösung gefunden hat. Die Perspektive dieser Vorlesung lässt uns in erster Linie an die Rechenzeit denken, die ein möglichst effizienter Algorithmus für die Lösung benötigt. Es wird aber häufig so sein, dass die Zeit, die zur Entwicklung und Implementierung eines solchen effizienten Algorithmus aufgewendet werden muss, die anschließende Ausführungszeit auf dem Computer weit übersteigt. Man muss außerdem bedenken, dass die Entwicklungszeit in dem Sinne „teuer“ ist, dass hierfür (hoffentlich) qualifiziertes Personal benötigt wird, während der anschließenden Ausführungszeit wird nur Rechenzeit eines Computers benötigt, die häufig ohnehin im Überfluss vorhanden ist. Es liegt darum nahe, die Entwicklungszeit reduzieren zu wollen, auch wenn man dabei in Kauf nehmen muss, dass die Ausführungszeit steigt. Damit ist allgemeinen randomisierten Suchheuristiken der Weg geebnet: Diese Verfahren liegen praktisch fertig vor, sie sind in aller Regel in einer Algorithmenbibliothek fertig implementiert vorhanden, die Anpassung an das konkrete Problem beschränkt sich auf die Codierung des Suchraums und der Bewertungsfunktion f, was häufig schnell und einfach zu lösen ist. Dann kann eine gewählte problemunabhängige Heuristik zum Einsatz kommen, die mit etwas Glück in annehmbarer Zeit eine ausreichend gute Lösung findet. Es sollte klar geworden sein, dass man von solchen problemunabhängigen Heuristiken nicht übermäßig viel erwarten darf. Ihr Einsatz ist nur dann gerechtfertigt, wenn zur Entwicklung geeigneter problemspezifischer Algorithmen die nötige Zeit

oder das nötige Wissen fehlt. Es ist bekannt, dass solche Heuristiken oft erstaunlich schnell erstaunlich gute Lösungen finden, man muss sich also nicht übermäßig schlecht fühlen, wenn man sich zum Einsatz einer allgemeinen randomisierten Suchheuristik gezwungen sah.
Es gibt viele verschiedene Suchheuristiken, wir werden hier randomisierte lokale Suche, den Metropolis-Algorithmus, Simulated Annealing und einige sehr einfach evolutionäre Algorithmen kennenlernen. Dabei steht wie immer die Analyse der Effizienz im Vordergrund.
Bei einem problemspezifischen Algorithmus ist klar, was zu analysieren ist: Der Algorithmus ist für ein bestimmtes Problem entworfen, man sucht eine obere Schranke für die Zeit, die der Algorithmus zur Lösung von Instanzen dieses Problems braucht. Bei allgemeinen Suchheuristiken ist weniger klar, wofür man sich eigentlich interessiert. Insbesondere ist es in aller Regel nicht sinnvoll, die Performanz für ein Problem zu analysieren, bei dem man den Einsatz plant. Mit gleichem Aufwand könnte man vermutlich einen problemspezifischen Algorithmus entwerfen, der das Problem vermutlich besser oder schneller lösen könnte. Wir unterscheiden zunächst grob allgemeine von spezifischen Resultaten: Allgemeine Resultate können allgemeine Positivresultate sein, die für eine allgemeine Suchheuristik nachweisen oder widerlegen, dass die Wahrscheinlichkeit, in höchstens t Schritten ein globales Optimum zu finden, mit $t \to \infty$ gegen 1 konvergiert. Solche Konvergenzresultate sind je nach Suchheuristik einfach oder auch schwierig; in jedem Fall sind sie nicht übermäßig interessant: Konkrete hilfreiche Schranken für die Zeit kann man hier nicht erwarten, schließlich gelten solche Resultate für alle Probleme, also insbesondere für besonders schwierige. Andererseits kann man allgemeine Negativresultate besprechen, in erster Linie sind Ergebnisse über die Black-Box-Komplexität von Problemen und die sogenannten No-free-Lunch-Theoreme zu nennen. Wir verweisen diese Themen in eine Vorlesung, deren Schwerpunkt die Komplexitätstheorie ist, hier werden wir uns ausschließlich mit spezifischen Resultaten beschäftigen. Dabei kann es sich um konkrete Ergebnisse für die Performanz einer allgemeinen randomisierten Suchheuristik auf einer ausgewählten, nicht selten passend definierten, paradigmatischen Beispielfunktion handeln. Solche Ergebnisse sollen Effekte, deren Existenz man vermutet, exemplarisch nachweisen. Sie dienen einer Versachlichung der Diskussion und stellen bewiesene Tatsachen als Grundlage einer Bewertung zur Verfügung. In günstigen Fällen lassen sich solche Resultate ausbauen zu Ergebnissen über ganze Klassen von Funktionen. Schließlich kann man auch allgemeine Suchheuristiken im Zusammenhang mit konkreten kombinatorischen Optimierungsproblemen analysieren. Bei all diesen spezifischen Resultaten ist das Ziel, ein besseres Verständnis für die Grenzen und Möglichkeiten des Einsatzes der untersuchten Suchheuristik zu gewinnen.

Suchheuristiken haben manchmal einen festen Suchraum, für den sie definiert sind. Wir werden hier ausschließlich Suchheuristiken besprechen, die mit dem Suchraum $\{0,1\}^n$ arbeiten. Wenn aber Lösungen zur Instanz eines Problems nicht diese Gestalt haben, ist eine Aufbereitung des Problems nötig, bevor eine Suchheuristik angewendet werden kann. Wir sprechen dabei von der *Codierung* des Problems. Wir gehen dabei in der Regel so vor, dass wir Punkte des Suchraums der Suchheuristik auf potenzielle Lösungen des Optimierungsproblems abbilden; die zum Optimierungsproblem gehörende Bewertungsfunktion liefert eine Bewertung, die (gegebenenfalls noch transformiert) als Bewertung für die Suchheuristik gilt. Wir werden das noch konkret an einem Beispiel besprechen und auch dabei auftauchende Schwierigkeiten diskutieren.
Wir interessieren uns wie immer in erster Linie für die Rechenzeit der Suchheuristiken. Weil alle von uns betrachteten Suchheuristiken algorithmisch sehr einfach sind, was für allgemeine Suchheuristiken auch typisch ist, werden wir uns das Leben etwas erleichtern und an Stelle der tatsächlichen Rechenzeit nur die Anzahl der Zielfunktionsauswertungen zählen und annehmen, dass sich die tatsächliche Rechenzeit so gut abschätzen lässt. Es wird unmittelbar klar sein, dass das bei den von uns betrachteten Suchheuristiken ein angemessenes Vorgehen ist. Es sollte aber auch unmittelbar klar sein, dass bei aufwendigeren Heuristiken eine Analyse der tatsächlichen Rechenzeit (also der Anzahl der Elementaroperationen) erforderlich sein kann.
Die erste hier vorgestellte Suchheuristik ist Simulated Annealing. Wir beschreiben Simulated Annealing so, dass es zur Minimierung einer Funktion $f\colon \{0,1\}^n \to \mathbb{R}$ geeignet ist. Wie alle Suchheuristiken dieses Abschnitts beschreiben wir das Verfahren als unendlichen Zufallsprozess, für den Einsatz in der Praxis muss noch ein geeignetes Abbruchkriterium definiert werden, so dass das Verfahren zumindest terminiert. Gut sind Abbruchkriterien, die sinnvoll auf den Prozess eingehen, also etwa Verfahren, die beim Erreichen einer ausreichend guten Lösung den Prozess stoppen oder wenn die Entwicklung vermuten lässt, dass keine besseren Lösungen mehr gefunden werden. Verbreitet sind auch weniger sinnvolle Kriterien, zu nennen ist etwa eine vorher eingestellte Anzahl von Schritten.
Den für den Algorithmus wichtigen Begriff der Nachbarschaft und die konkrete Wahl der Wahrscheinlichkeitsverteilung über dieser Nachbarschaft lassen wir bewusst offen, um uns ausreichende Flexibilität zu erhalten. Natürlich sind diese beiden Faktoren bei jedem Einsatz exakt zu spezifizieren.

Algorithmus 13.1 (**Simulated Annealing**). *Sei* $T\colon \mathbb{N} \to \mathbb{R}_0^+$ *eine Funktion,* T *heißt* Annealingstrategie. *Sei* $N\colon \{0,1\}^n \to \mathcal{P}(\{0,1\}^n)$ *eine* Nachbarschaft. *Die folgende Suchheuristik heißt* Simulated Annealing.

1. $t := 1$
2. *Wähle* $x_t \in \{0,1\}^n$ *uniform zufällig.*
3. *Wähle* $y \in N(x_t)$ *gemäß einer festen Verteilung.*
4. *Mit Wahrscheinlichkeit* $\min\left\{1, e^{-(f(y)-f(x_t))/T(t)}\right\}$ *setze* $x_{t+1} := y$*, sonst setze* $x_{t+1} := x_t$*.*
5. $t := t + 1$
6. *Weiter bei 3.*

Typische Nachbarschaften sind alle Hammingnachbarn (also $N(x) = \{y \in \{0,1\}^n \mid d(x,y) = 1\}$) oder auch Punkte mit Hammingabstand höchstens 2 (also $N(x) = \{y \in \{0,1\}^n \mid 0 < d(x,y) \leq 2\}$). Häufig wird als Verteilung die Gleichverteilung gewählt.
Simulated Annealing ist schon ein recht komplizierter Algorithmus: Es wird in der Nachbarschaft gesucht, wird ein Punkt mit besserem Funktionswert gefunden ($f(y) < f(x_t)$), so ersetzt dieser Punkt y den aktuellen Punkt. Ist y hingegen schlechter ($f(y) > f(x_t)$), so ersetzt y trotzdem mit gewisser Wahrscheinlichkeit den aktuellen Punkt, dabei hängt diese Wahrscheinlichkeit exponentiell von der Differenz der Funktionswerte und der aktuellen *Temperatur* $T(t)$ ab. Hohe Temperaturen vergrößern die Wahrscheinlichkeit, eine Verschlechterung im Funktionswert zu akzeptieren, im Extremfall $T(t) = \infty$ wird jeder Nachbar y als neuer Punkt x_{t+1} akzeptiert. Umgekehrt verkleinern niedrige Temperaturen diese Wahrscheinlichkeit, im Extremfall $T(t) = 0$ wird keinerlei Verschlechterung akzeptiert. Falls der Funktionswert des neuen Punktes gleich dem Funktionswert des alten Punktes ist, ergibt sich für die Wahrscheinlichkeit $\min\left\{1, e^{-0/T}\right\}$, wir erhalten offenbar 1 und definieren, dass das auch für den Extremfall $T = 0$ gilt.
Simulated Annealing ist ein vom Prozess des Abkühlens, bei dem die Moleküle einen energieminimalen Zustand einnehmen, inspiriertes Verfahren. Darum wird man eine monoton sinkende Annealingstrategie T wählen: Anfangs ist man eher bereit, Verschlechterungen zu akzeptieren, später, wenn schon viel erreicht ist, sinkt diese Bereitschaft.
Basierend auf Simulated Annealing lassen sich zwei weitere allgemeine Suchheuristiken leicht formal definieren. Historisch gesehen ist allerdings Simulated Annealing die späteste dieser drei Suchheuristiken, wir werden sehen, dass es als eine natürliche Verallgemeinerung beschrieben werden kann.

Algorithmus 13.2 (**Metropolis-Algorithmus**)**.** *Der* Metropolis-Algorithmus *ist Simulated Annealing mit Annealingstrategie* $T(t) = T$ *für eine feste* Temperatur $T \in \mathbb{R}_0^+ \cup \{\infty\}$*.*

Algorithmus 13.3 (**Randomisierte lokale Suche (RLS)**)**.** Randomisierte lokale Suche *(*RLS*) ist der Metropolis-Algorithmus mit Temperatur* $T = 0$*.*

Wir sehen, dass für den Metropolis-Algorithmus die gleichen Bemerkungen über das Akzeptieren von Nachbarn mit kleinerem Funktionswert wie für Simulated Annealing zutreffen, allerdings ist die Temperatur hier fest und variiert nicht mit der Zeit. Randomisierte lokale Suche akzeptiert im Gegensatz dazu keinerlei Verschlechterungen im Funktionswert. Man sieht direkt, dass das dazu führen kann, dass der Algorithmus in einem lokalen Optimum verharrt und es vorkommen kann, dass ein globales Optimum nicht gefunden wird. Ebenso sieht man direkt, dass der Metropolis-Algorithmus mit Temperatur $T > 0$ sicher ein globales Optimum findet, wenn die Nachbarschaft so definiert ist, dass der gesamte Suchraum durchschritten werden kann: Jeder Schritt hat positive Wahrscheinlichkeit, wir können also immer eine endliche Folge von Schritten angeben, die zu einem globalen Maximum führt und positive Wahrscheinlichkeit hat. Wenn wir alle solche Folgen betrachten, alle Folgen höchstens Länge l haben und p eine untere Schranke für die Wahrscheinlichkeit einer solchen Folge ist, so ist die erwartete Optimierzeit durch l/p nach oben beschränkt, außerdem ist die Wahrscheinlichkeit, nach t Schritten noch kein globales Optimum gefunden zu haben, durch $(1-p)^{\lfloor t/l \rfloor}$ nach oben beschränkt. Für Simulated Annealing ist die Lage eigentlich ähnlich, die sinkende und in der Regel gegen 0 konvergierende Wahrscheinlichkeit macht die Analyse aber wesentlich schwieriger. Man kann aber in der Tat zeigen, dass für ausreichend langsam sinkende Annealingstrategien mit gegen 1 konvergierender Wahrscheinlichkeit ein globales Maximum erreicht wird. Die aufwendigen Beweisdetails wollen wir uns hier ersparen, wir hatten uns ja gegen solche allgemeinen Resultate entschieden.

Als vorerst letzte Suchheuristik führen wir einen sehr einfachen evolutionären Algorithmus ein, den man (1+1)-EA nennt. Evolutionäre Algorithmen sind ebenfalls naturinspirierte Suchverfahren, sie orientieren sich an Ideen aus der natürlichen Evolution. Sie sind populationsbasierte Suchverfahren, sie operieren also auf einer Multimenge von Punkten aus dem Suchraum. Sie generieren neue Suchpunkte durch zufällige Variation aktueller Suchpunkte, dabei kann die Variation sich auf einen einzelnen Suchpunkt beziehen, man spricht dann von *Mutation*, oder mehrere Suchpunkte betreffen (meist zwei), in diesem Fall spricht man von *Crossover*. Wir schauen uns hier einen durchaus untypischen Vertreter an, den man erhält, wenn man die Populationsgröße und die Anzahl in jeder Runde neu erzeugter Suchpunkte jeweils auf 1 reduziert. Der so entstehende evolutionäre Algorithmus hat den Charme, große Ähnlichkeit zu den drei hier vorgestellten Suchheuristiken zu haben, sich aber in einem wichtigen Punkt, der Erzeugung neuer Punkte im Suchraum, die hier durch Mutation geschieht, zu unterscheiden.

Definition 13.4. *Sei $p \in [0;1]$, p heißt* Mutationswahrscheinlichkeit. *Die folgende Suchheuristik heißt* (1+1)-EA.

1. $t := 1$
2. *Wähle $x_t \in \{0,1\}^n$ uniform zufällig.*
3. *Für $i \in \{1,2,\ldots,n\}$*
   Mit Wahrscheinlichkeit p, setze $y[i] := 1 - x_t[i]$, sonst $y[i] := x_t[i]$.
4. *If $f(y) \geq f(x)$ Then setze $x_{t+1} := y$ Else setze $x_{t+1} := x_t$.*
5. $t := t + 1$
6. *Weiter bei 3.*

Wir sind an spezifischen Ergebnissen für diese randomisierten Suchheuristiken interessiert, als Beispielproblem schauen wir uns das Problem an, einen minimalen Spannbaum zu berechnen. Wir wissen schon aus der Vorlesung DAP 2, dass das Problem effizient mit dem Algorithmus von Kruskal gelöst werden kann. Darum ist klar, dass man in der Praxis keine allgemeinen Suchheuristiken für dieses Problem einsetzen soll. Uns interessieren vor allem zwei Aspekte: Zum einen möchten wir sehen, was man bei der Problemcodierung beachten muss; das erkennen wir sicher leichter bei einem Problem, das wir gut verstehen. Außerdem möchten wir uns exemplarisch ansehen, wie die Performanz solcher Suchheuristiken bei eher einfachen Problemen ist.

13.1 RLS und (1+1)-EA zur Suche nach minimalen Spannbäumen

Eingabe für unser Problem ist ein ungerichteter, gewichteter und zusammenhängender Graph $G = (V, E, w)$ mit Kantengewichten $w\colon E \to \mathbb{N}$. Ausgabe ist ein ebenfalls ungerichteter, gewichteter und zusammenhängender Graph $G' = (V, E', w)$ mit $E' \subseteq E$. Wir interessieren uns für $w(E') := \sum_{e \in E'} w(e)$. Unter allen diesen Graphen ist einer mit minimalem Gesamtgewicht $w(E')$ auszugeben. Wir nennen das Problem kurz und prägant *MST*.
Wir haben bei der Problemdefinition nicht gefordert, dass das Ergebnis ein Spannbaum ist, das ergibt sich aber direkt als Folge: Der ausgegebene Graph muss zusammenhängend sein, außerdem soll er minimales Kantengewicht haben. Weil alle Kanten positives Gewicht haben, können aus zusammenhängenden Graphen, die keine Bäume sind, Kanten entfernt werden, ohne dass der Zusammenhang verloren geht, so dass man ebenfalls zusammenhängende Graphen mit echt kleinerem Gewicht erhält. Nur für Bäume ist das nicht mehr möglich, so dass also tatsächlich ein minimaler Spann*baum* gesucht ist.

Wir definieren $m := |E|$ und entscheiden uns zunächst für eine geeignete Problemcodierung. Dabei sei die Kantenmenge $E = \{e_1, e_2, \ldots, e_m\}$. Weil eine Lösung eindeutig durch die aus E gewählten Kanten definiert ist, können wir das Problem als Kantenauswahlproblem codieren und einen Lösungskandidaten als Inzidenzvektor über der Kantenmenge beschreiben: Der Suchraum ist also $\{0,1\}^m$, die Interpretation von $x \in \{0,1\}^m$ ist, dass x die Kantenauswahl E_x codiert, für die $e_i \in E_x \Leftrightarrow x[i] = 1$ gilt. Wir können so offensichtlich alle möglichen Kantenauswahlen darstellen, insbesondere also auch alle Spannbäume. Allerdings sind so auch viele unzusammenhängende Graphen darstellbar.
Wir brauchen eine Zielfunktion $f\colon \{0,1\}^m \to \mathbb{R}$, die wir minimieren wollen. Minima der Zielfunktion sollen dabei minimalen Spannbäumen entsprechen. Wir können

$$f_1(x) := w(E_x)$$

als ersten naiven Versuch betrachten. Den Wunsch, das Gesamtgewicht der Kantenauswahl zu minimieren, haben wir offenbar direkt umgesetzt, leider haben wir aber nicht zum Ausdruck gebracht, dass auch Zusammenhang wichtig ist. Eine allgemeine Suchheuristik dürfte für f_1 recht schnell eine optimale Lösung finden, allerdings ist $E_x = \emptyset$ die optimale Lösung mit Kantengewicht 0. Wir sehen, dass man bei der Formulierung der Zielfunktion sorgfältig sein muss und alle Anforderungen an eine Lösung geeignet berücksichtigen muss.
Wir schlagen

$$f_2(x) := \begin{cases} w(E_x) & \text{falls } (V, E_x) \text{ zusammenhängend} \\ \infty & \text{sonst} \end{cases}$$

als zweiten Versuch vor. Wir sehen direkt, dass jetzt tatsächlich nur minimale Spannbäume Minima von f_2 entsprechen. Allerdings haben wir das Problem jetzt vielleicht unnötig schwierig gemacht: Betrachten wir den Kettengraphen $G = (V, E, w)$ mit $V = \{1, 2, \ldots, n\}$, $E = \bigcup_{i=1}^{n-1} \{e_i, e_{i+1}\}$ und $w(e) = 1$ für alle $e \in E$. Offensichtlich gibt es nur einen Spannbaum, der damit auch minimal ist, er wird durch $x = 1^m$ codiert. Unsere Suchheuristiken starten mit einer Kantenauswahl, die im Durchschnitt nur $(n-1)/2$ Kanten enthält, also mit einem Graphen, der nicht zusammenhängend ist. Solange kein zusammenhängender Graph gefunden ist, wird immer nur ∞ als Funktionswert erreicht. Eine allgemeine Suchheuristik ist also mit einem Problem konfrontiert, bei dem im Suchraum $\{0,1\}^n$ alle Punkte Funktionswert ∞ haben und ein spezieller Punkt zu finden ist, der Funktionswert $n-1$ hat. Es ist offensichtlich, dass man keine Chance hat, dieses Problem effizient zu lösen, wenn

man nicht zufällig weiß, dass 1^n eine zulässige Lösung ist. Wir erkennen, dass man durch ungeschickte Codierung aus einem eigentlich einfachen Problem ein sehr schwieriges machen kann.
Wir geben uns jetzt etwas mehr Mühe und schlagen

$$f_3(x) := (c(x) - 1) \cdot w_b^2 + (|x| - (n-1)) \cdot w_b + w(E_x)$$

als Zielfunktion vor, dabei gibt für $x \in \{0,1\}^m$ die Funktion $c\colon \{0,1\}^m \to \{1,2,\ldots,n\}$ als Funktionswert $c(x)$ die Anzahl der Zusammenhangskomponenten in (V, E_x) an, mit $|x|$ bezeichnen wir die Anzahl der Einsbits in $x \in \{0,1\}^m$, es ist also $|x| = \sum_{i=1}^{m} x[i]$, schließlich ist $w_b := m \cdot \max\{w(e) \mid e \in E\}$.
Der Funktionswert setzt sich aus drei Teilen zusammen: Der erste Teil ist ein Strafterm, der dafür sorgt, dass die Anzahl der Zusammenhangskomponenten auf 1 reduziert wird. Das Strafgewicht w_b ist so groß, dass Ab- oder Zunahme des Funktionswertes um w_b^2 durch Veränderung der Anzahl der Zusammenhangskomponenten alle anderen Änderungen überwiegt. Auch der zweite Term ist ein Strafterm, hier soll die Anzahl der Kanten minimiert werden. Weil das Strafgewicht hier w_b ist, ist $|x| = n - 1$ die beste Wahl: Noch kleinere Kantenanzahlen führen zu einer Vergrößerung der Anzahl der Zusammenhangskomponenten, was größere Funktionswerte impliziert. Wir drängen also in erster Linie darauf, dass ein zusammenhängender Graph gefunden wird, dann ist wichtig, dass der Graph genau $n - 1$ Kanten enthält. Schließlich taucht als dritter Termin $w(E_x)$ auf: Unter allen zusammenhängenden Graphen mit $n - 1$ Kanten ist ein Graph mit minimalem Kantengewicht gesucht. Die Funktion f_3 ist also geeignet gewählt, man darf aber kritisch anmerken, dass recht viel Problemwissen eingeht: Die passende Anzahl der Kanten hatten wir in der Problemstellung selbst ja nicht aufgeführt. Man darf diese Kritik nicht falsch verstehen. Es ist sicher immer gut, möglichst viel Problemwissen in die Codierung der Zielfunktion einfließen zu lassen. Es ist aber nicht klar, dass in praktischen Situationen so viel Problemeinsicht vorhanden ist, so dass man sich fragen sollte, ob f_3 eigentlich noch einer typischen Zielfunktion für eine allgemeine Suchheuristik entspricht.
Wir nehmen diese kritische Überlegung auf und stellen eine weitere Zielfunktion vor, die diesen Hinweis auf die geeignete Kantenanzahl nicht enthält.

$$f_4(x) := (c(x) - 1) \cdot w_b + w(E_x)$$

Die Funktion f_4 erscheint die bis jetzt naheliegendste Wahl für eine Fitnessfunktion, allerdings ist es schon so, dass wir zwei getrennte Kriterien durch Summierung und Gewichtung vereinigen: Eigentlich haben Zusammenhang

und minimales Kantengewicht nicht viel miteinander zu tun, sind sogar entgegengesetzt. Natürlicher wäre es, die beiden Kriterien getrennt zu halten und separat betrachten zu lassen. Das führt uns zu

$$f_5(x) := (c(x), w(E_x))$$

als Zielfunktion, bei der beide Komponenten zu minimieren sind. So natürlich diese Problemformulierung auch ist, sie hat den erheblichen Nachteil, dass unsere vier Suchheuristiken nicht für die Optimierung solcher Probleme ausgerichtet sind und eine Anpassung der Algorithmen erforderlich wird.
Wir kommen jetzt zu unseren ersten konkreten Ergebnissen, dazu betrachten wir zunächst die Zielfunktion f_3 und RLS. Wir legen dafür folgende Nachbarschaften und Wahrscheinlichkeitsverteilung auf dieser Nachbarschaft fest. Zu $x \in \{0,1\}^m$ ist

$$N(x) := \{y \in \{0,1\}^m \mid 0 < d(x,y) \leq 2\}$$

die Menge der 1- und 2-Bit-Hammingnachbarn. Die Wahrscheinlichkeitsverteilung legen wir wie folgt fest: Zunächst entscheiden wir uns mit Wahrscheinlichkeit jeweils $1/2$ dafür, einen 1- oder 2-Bit-Hammingnachbarn zu wählen. Unter diesen Nachbarn wählen wir dann uniform zufällig. Damit wird jedes y mit $d(x,y) = 1$ mit Wahrscheinlichkeit $(1/2) \cdot (1/n) = \Theta(1/n)$ gewählt, jedes y mit $d(x,y) = 2$ wird mit Wahrscheinlichkeit $(1/2) \cdot (1/\binom{n}{2}) = \Theta(1/n^2)$ gewählt. Die Gleichbehandlung von 1- und 2-Bit-Hammingnachbarn impliziert also eine Bevorzugung der direkten Hammingnachbarn.
Wir formulieren unser erstes Ergebnis für RLS auf f_3. Wir erinnern daran, dass wir als Maß für die Zeit der Anzahl der f-Auswertungen (hier also der f_3-Auswertungen) verwenden,

Theorem 13.5. *Die erwartete Zeit, bis RLS auf f_3 einen Spannbaum findet, ist $O(m \log m)$.*

Beweis. Wegen der Selektion fällt $f_3(x_t)$ monoton in t. Wir wollen den Weg von x_t in $\{0,1\}^m$ in etwa nachvollziehen und definieren dazu eine Partitionierung von $\{0,1\}^m$, die in Zusammenhang mit den Funktionswerten steht. Außerdem stellen wir sicher, dass zwischen den Mengen bei Berücksichtigung unserer Nachbarschaft N Erreichbarkeit besteht. Eine Partition $S_1, S_2, \ldots, S_l$ mit solchen Eigenschaft, die wir noch einmal etwas formaler festhalten, nennen wir *f_3-basierte Partition.*

1. $\forall 1 \leq i < j \leq l \colon \forall x \in S_i, x' \in S_j \colon f_3(x) > f_3(x')$
2. $\forall x \in S_l \colon x$ erfüllt Zielanforderung

3. $\forall 1 \leq i < l \colon \forall x \in S_i \colon \exists j > i, y \in S_j \colon d(x,y) \leq 2$

Die Zielanforderung für uns ist, dass alle $x \in S_l$ Spannbäume codieren.
Wir behaupten, dass

$$S_i := \begin{cases} \{x \mid c(x) = n+1-i\} & \text{für } 1 \leq i < n \\ \{x \mid (c(x) = 1) \wedge (|x| = m+n-i)\} & \text{für } n \leq i \leq m+1 \end{cases}$$

eine f_3-basierte Partition definiert. Dazu prüfen wir das Einhalten der drei oben genannten Bedingungen nach.

- Sei zunächst $i < n$. Dann wird der Funktionswert durch die Anzahl der Zusammenhangskomponenten dominiert, und die Verbesserung im Funktionswert beim Überganz zu einer Menge mit größerem Index folgt direkt. Für $i \geq n$ codieren alle $x \in S_i$ zusammenhängende Graphen, die Anzahl der Kanten ist also durch $n-1$ nach unten beschränkt. Der Funktionswert wird hier durch die Anzahl der Kanten dominiert, die Verbesserung im Funktionswert folgt wiederum direkt.

- Für alle $x \in S_{m+1}$ gilt, dass (V, E_x) einen zusammenhängenden Graphen mit $n-1$ Kanten codiert, es handelt sich also wie gewünscht ausschließlich um Spannbäume.

- Die Erreichbarkeit überlegen wir uns wieder getrennt nach dem Wert von i. Ist $i < n$, so können wir die Anzahl der Zusammenhangskomponenten verkleinern, wenn wir eine Kante, die zwischen zwei Zusammenhangskomponenten verläuft, hinzunehmen. Eine solche Kante gibt es für jede Zusammenhangskomponente, weil der Ausgangsgraph G zusammenhängend ist. Es reicht also sogar, nur direkte Hammingnachbarn zu betrachten. Für $i \geq n$ ist immer eine Kante entfernbar, ohne dass der Zusammenhang verloren geht, es genügen also auch hier direkte Hammingnachbarn.

Wir betrachten nun diese f_3-basierte Partition und geben für jede Menge S_i mit $i < m-n+1$ eine untere Schranke für die Wahrscheinlichkeit, S_i zu verlassen, an. Ist $i < n$, so sind wie erwähnt $n-i$ einzelne Kanten wählbar, wir haben also

$$\text{Prob}\,(\text{verlasse } S_i) \geq \frac{1}{2} \cdot \frac{n-i}{m}$$

und erhalten $2m/(n-i)$ als obere Schranke für die erwartete Verweildauer in S_i. Ist $i \geq n$, so sind $m-i-1$ Kanten wählbar, wir haben also

$$\text{Prob}\,(\text{verlasse } S_i) \geq \frac{1}{2} \cdot \frac{m-i-1}{m}$$

und erhalten $2m/(m-i-1)$ als obere Schranke für die erwartete Verweildauer in S_i. Wir erhalten eine obere Schranke für die erwartete Gesamtzeit durch Addition all dieser erwarteten Verweildauern. Wir erhalten also

$$\left(\sum_{i=1}^{n-1} \frac{2m}{n-i}\right) + \left(\sum_{i=n}^{m-n+1} \frac{2m}{m-i-1}\right) = 2m\left(\left(\sum_{i=1}^{n-1} \frac{1}{i}\right) + \left(\sum_{i=n-2}^{m-n-1} \frac{1}{i}\right)\right)$$
$$< 2m\left(\ln(n-1) + \ln(m-n-1) + 1\right) = O(m \log m)$$

als obere Schranke wie behauptet. □

Im Beweis von Theorem 13.5 war die zentrale von uns ausgenutzte Eigenschaft, dass $f_3(x_t)$ monoton in t ist. Das gilt beim Metropolis-Algorithmus und Simulated Annealing im Allgemeinen nicht, wohl aber beim (1+1)-EA. Es liegt darum nahe zu fragen, ob wir ein ähnliches Resultat auch für den (1+1)-EA beweisen können. Das ist tatsächlich ohne weitere Mühe der Fall.

Theorem 13.6. *Die erwartete Zeit, bis der (1+1)-EA auf f_3 einen Spannbaum findet, ist $O(m \log m)$.*

Beweis. Weil $f_3(x_t)$ auch für den (1+1)-EA monoton in t fällt, können wir die gleiche f_3-basierte Partition wie im Beweis von Theorem 13.5 betrachten und auch jeweils untere Schranken für die Verlasswahrscheinlichkeit von S_i bestimmen. Weil wie gesehen jeweils nur einer von k Hammingnachbarn zu erreichen ist (mit passendem k) erhalten wir

$$\binom{k}{1} \cdot \frac{1}{m} \cdot \left(1 - \frac{1}{m}\right)^{m-1} \geq \frac{k}{em}$$

als untere Schranke, so dass sich die Rechnungen aus dem Beweis von Theorem 13.5 fast unverändert wiederholen, wir ersetzen lediglich den Faktor 2 durch den Faktor e, das asymptotische Ergebnis $O(m \log m)$ ändert sich dadurch natürlich nicht. □

Dass wir mit RLS und (1+1)-EA effizient überhaupt Spannbäume finden, ist ja schon einmal schön. Selbstverständlich ist das ja nicht, wie wir uns bei der Diskussion um f_2 schon überlegt hatten. Es schließen sich offensichtlich zwei Fragen an: Funktioniert das auch mit dem Metropolis-Algorithmus und Simulated Annealing? Finden wir auch minimale Spannbäume effizient? Die erste Frage ist je nach Auslegung einfach oder schwer zu beantworten. Es ist nicht schwer einzusehen, dass es Temperaturen und Annealingstrategien gibt, bei denen es lange dauern kann. Die Wahl $T(t) = T = \infty$ zum Beispiel impliziert eine rein zufällige Suche im $\{0,1\}^m$ ohne Berücksichtigung der Funktionswerte, wenn es nicht viele Spannbäume gibt (wie zum

Beispiel beim Kettengraphen), dauert es sicher lange. Andererseits ist natürlich auch $T(t) = T = 0$ eine Annealingstrategie, dann entsprechen Simulated Annealing und der Metropolis-Algorithmus exakt der randomisierten lokalen Suche. Wir haben also gute Performanz bei geeigneter Temperatur bzw. geeigneter Annealingstrategie, das war einfach zu sehen. Schwierig zu beantworten ist die Frage, wie genau man Annealingstrategien und Temperaturen charakterisiert, für die Spannbäume effizient gefunden werden. Wir wollen dies hier nicht vertiefen und geben uns darum auch weiterhin mit der Betrachtung von RLS und dem (1+1)-EA zufrieden. Spannender ist ohnehin die zweite Frage: Finden wir mit RLS und dem (1+1)-EA auch effizient minimale Spannbäume?
Für eine obere Schranke für die erwartete Optimierzeit brauchen wir strukturelle Einsichten in das zu lösende Problem, hier also die Berechnung minimaler Spannbäume. Es sei noch einmal explizit darauf hingewiesen, dass diese Problemeinsichten nur für die Analyse erforderlich sind; Codierung und Einsatz der allgemeinen Suchheuristiken setzen so tiefgehende Einsichten in das Problem nicht voraus. Für das Spannbaumproblem ist das folgende Lemma entscheidend.

Lemma 13.7. *Sei $G = (V, E, w)$ eine MST-Instanz, T^* ein minimaler Spannbaum, T ein Spannbaum mit $w(T) > w(T^*)$.*
Es gibt k verschiedene Spannbäume T' mit $d(T', T) = 2$ und durchschnittlich $w(T) - w(T') \geq ((w(T) - w(T^*))/k$, dabei ist $k \in \{1, 2, \ldots, n-1\}$.

Beweis. Wir betrachten die Spannbäume $T^* = (V, E_{T^*})$ und $T = (V, E_T)$, es sei $k := |E_T \setminus E_{T'}|$. Weil $w(T) > w(T^*)$ gilt und $|E_{T^*}| = |E_T| = n-1$ gilt, ist $k \in \{1, 2, \ldots, n-1\}$. Wir erinnern uns an den Korrektheitsbeweis des Algorithmus von Kruskal: Wir können eine Kante in T, die nicht in T^* vorkommt, gegen eine Kante aus T^* austauschen, ohne dass das Gesamtgewicht wächst. Etwas formaler können wir sagen, dass es eine Bijektion $\alpha\colon E_{T^*} \setminus E_T \to E_T \setminus E_{T^*}$ gibt, so dass für alle $e \in E_{T^*} \setminus E_T$ wir einen Kreis in $T \cup \{e\}$ haben, der die Kante $\alpha(e)$ enthält, für die $w(\alpha(e)) \geq w(e)$ gilt. Wir erhalten einen neuen Spannbaum mit der Kantenmenge $(T \cup \{e\}) \setminus \{\alpha(e)\}$, der als T' geeignet ist: Der Hammingabstand zu T beträgt genau 2, weil die Kante e hinzugefügt und die Kante $\alpha(e) \neq e$ entfernt wird. Weil $w(e) \leq w(\alpha(e))$ gilt, ist $w(T') \leq w(T)$. Wenn wir alle k Kantenaustausche durchführen, sinkt das Gesamtgewicht von $w(T)$ auf $w(T^*)$, so dass wir eine durchschnittliche Gewichtsabnahme von $((w(T) - w(T^*))/k$ haben wie behauptet. □

Für die Anwendung bei der Abschätzung der erwarteten Optimierzeit ist es günstiger, nicht eine unbekannte Zahl k von solchen Kantenaustauschen zu haben. Wir können dieses Problem leicht beheben. Damit lässt sich dann

für RLS tatsächlich ohne große Schwierigkeiten eine obere Schranke für die erwartete Optimierzeit zeigen.

Lemma 13.8. *Sei $G = (V, E, w)$ eine MST-Instanz, T^* ein minimaler Spannbaum, T ein Spannbaum mit $w(T) > w(T^*)$.*
Es gibt n verschiedene Kantenaustauschoperationen, die zu Spannbäumen T' mit $d(T', T) = 2$ und durchschnittlich $w(T) - w(T') \geq ((w(T) - w(T^*))/n$ führen.

Beweis. Wir wissen aus dem Beweis von Lemma 13.7, dass es jedenfalls $k \geq 1$ geeignete Kantenaustauschoperationen gibt. Wir können leicht $n-k$ weitere 2-Bit-Hammingnachbarn von T identifizieren, die schlechteren Funktionswert haben, so dass diese Spannbäume nicht akzeptiert werden. Daraus folgt schon die Behauptung. □

Theorem 13.9. *Die erwartete Optimierzeit von RLS auf MST mit f_3 beträgt $O(m^2 (\log m + \log w_{\max}))$ für einen Graphen $G = (V, E, w)$ mit $|E| = m$ und $w_{\max} := \max\{w(e) \mid e \in E\}$.*

Beweis. Wir wissen, dass wir im Erwartungswert nach $O(m \log m)$ Schritten einen Spannbaum finden, wir brauchen uns die Situation also erst anzusehen, nachdem das geschehen ist. Wir betrachten einen minimalen Spannbaum T^* mit minimalem Gewicht $w_{\text{opt}} := w(T^*)$. Lemma 13.8 liefert, dass es zu x mit $E_x > w_{\text{opt}}$ stets n verschiedene Suchpunkte $x'_1, x'_2, \ldots, x'_n \in N(x)$ gibt, für die im Durchschnitt

$$w(T) - w(E_{x'_i}) \geq (w(T) - w(T^*))/n$$

gilt. Die Wahrscheinlichkeit, einen dieser Nachbarn zu wählen, beträgt $\Theta(n/m^2)$. Anfangs beträgt der Abstand im Gewicht zum Optimum $D := w(T) - w_{\text{opt}}$, und natürlich ist D durch $n \cdot w_{\max}$ nach oben beschränkt. Nach t Schritten dieser Art ist die Gewichtsdifferenz im Erwartungswert auf $(1 - 1/n)^t \cdot D$ gefallen. Wir wählen

$$t := \lceil n \cdot \ln(2) \cdot (\log(D) + 1) \rceil$$

und haben $\mathrm{E}(w(T) - w_{\text{opt}}) \leq 1/2$. Wir erinnern daran, dass die Kantengewichte ganzzahlig sind, also haben wir im Erwartungswert den Funktionswert des Optimums erreicht. Anwendung der Markow-Ungleichung ergibt, dass die Wahrscheinlichkeit, dass der Gewichtsabstand zum Optimum mindestens $2 \cdot (1/2) = 1$ beträgt, durch $1/2$ nach oben beschränkt ist. Wir haben also mit Wahrscheinlichkeit mindestens $1/2$ nach $O(n \log D)$ Schritten das Optimum erreicht. Falls wir es nicht erreicht haben, sind wir sicher nicht in einer

ungünstigeren Lage als als am Anfang: Wir haben irgendeinen Spannbaum als aktuellen Suchpunkt, irgendwelche Voraussetzungen hatten wir ja nicht gemacht in Bezug auf x_t. Wir können also im Fall eines Misserfolgs die folgenden Schritte wie einen unabhängigen Neustart behandeln, die erwartete Anzahl solcher Neustarts, die ausreichen, um einen minimalen Spannbaum zu finden, ist also durch 2 nach oben beschränkt. Damit haben wir als obere Schranke für die erwartete Optimierzeit

$$O\left(t \cdot \frac{m^2}{n}\right) = O\big(m^2 \log D\big) = O\big(m^2 \left(\log(n) + \log(w_{\max})\right)\big)$$

nachgewiesen. □

Der Beweis von Theorem 13.9 basiert zentral darauf, dass $f(x_t)$ nicht sinken kann und einer von k 2-Bit-Hammingnachbarn mit Wahrscheinlichkeit $\Theta(k/m^2)$ erreicht wird. Wir hatten uns schon überlegt, dass beides auch für den (1+1)-EA gilt. Also folgt mit gleichem Beweis die folgende Aussage.

Theorem 13.10. *Die erwartete Optimierzeit des (1+1)-EA auf MST mit f_3 beträgt $O(m^2 (\log m + \log w_{\max}))$ für einen Graphen $G = (V, E, w)$ mit $|E| = m$ und $w_{\max} := \max\{w(e) \mid e \in E\}$.*

Die erwartete Zeit von RLS und dem (1+1)-EA für die Berechnung eines minimalen Spannbaums auf einem Graphen mit polynomiellen Kantengewichten ist $O(m^2 \log n) = O(n^4 \log n)$. Das ist polynomiell und in diesem Sinn „effizient“, im Vergleich zum Algorithmus von Kruskal ist es aber ausgesprochen langsam. Allerdings haben wir nur eine obere Schranke gezeigt und der Beweis war ausgesprochen primitiv. Vielleicht sind die beide Suchheuristiken in Wahrheit ja viel schneller? Wir werden durch Angabe eines konkreten Graphen und die Analyse der erwarteten Optimierzeit für diesen Graphen zeigen, dass das nicht der Fall ist.

Definition 13.11. *Sei $n \in \mathbb{N}$ so gegeben, dass $n/4 \in \mathbb{N}$ gilt. Der Graph $G_n := (V, E, w)$ ist gegeben durch $V := \{1, 2, \ldots, n\}$ und $E := E_\Delta \cup E_k$ mit*

$$E_\Delta := \bigcup_{i=1}^{n/4} \{\{2i-1, 2i\}, \{2i-1, 2i+1\}, \{2i, 2i+1\}\}$$

und

$$E_k := \{\{i, j\} \mid i \neq j \in \{(n/2)+1, (n/2)+2, \ldots, n\}\},$$

außerdem ist

$$w(e) := \begin{cases} 1 & \text{für } e \in E_k, \\ 2n^2 & \text{für } e \in \{\{2i-1, 2i\}, \{2i, 2i+1\}\} \subseteq E_\Delta, \\ 3n^2 & \text{sonst.} \end{cases}$$

Theorem 13.12. *Die erwartete Optimierzeit von RLS auf G_n beträgt $\Theta(n^4 \log n)$.*

Beweis. Die obere Schranke haben wir schon gezeigt (Theorem 13.9), wie brauchen also nur die untere Schranke nachzuweisen. Der Graph G_n besteht aus einer Kette von $n/4$ Dreiecken und einer Clique auf $n/2$ Knoten. Jedes Dreieck enthält eine Kante mit Gewicht $3n^2$ und zwei Kanten mit Gewicht $2n^2$. Ein minimaler Spannbaum enthält offensichtlich ausschließlich diese leichte Kanten in den Dreiecken und eine beliebige Auswahl der Kanten der Clique, die Zusammenhang garantiert. Bezüglich einer Kantenauswahl x bezeichnen wir ein Dreieck als komplett, wenn alle drei Kanten ausgewählt sind, ein Dreieck heißt schlecht, wenn die schwere und eine leichte Kante ausgewählt ist, schließlich heißen Dreiecke gut, wenn die beiden leichten Kanten ausgewählt sind. In allen anderen Fällen ist der Graph nicht mehr zusammenhängend. Es bezeichne $b(x)$ die Anzahl schlechter Dreiecke, $g(x)$ die Anzahl guter Dreiecke und $ct(x)$ die Anzahl kompletter Dreiecke. Außerdem interessieren wir uns für die Anzahl der Zusammenhangskomponenten, die Anzahl aller Zusammenhangskomponenten sei $c_G(x)$, wir unterteilen das in die Anzahl der Zusammenhangskomponenten in den Dreiecken $c_D(x)$ und die Anzahl der Zusammenhangskomponenten in der Clique $c_C(x)$.
Ansatz für den Beweis einer unteren Schranke ist der Satz von der totalen Wahrscheinlichkeit:

$$\mathrm{E}(A) = \sum_B \mathrm{Prob}(B) \cdot \mathrm{E}(A \mid B)$$

Wir können uns also ein Ereignis B aussuchen und $\mathrm{E}(T) \geq \mathrm{Prob}(B) \cdot \mathrm{E}(T \mid B)$ folgern. Die Idee ist, B so zu wählen, dass $\mathrm{E}(T \mid B)$ leichter bestimmbar ist als $\mathrm{E}(T)$ und $\mathrm{Prob}(B)$ nicht zu klein ist. Konkret werden wir Ereignisse B_i identifizieren, für die wir sowohl $\mathrm{Prob}(B_i) = 1 - o(1)$ als auch $\mathrm{Prob}(\bigcap B_i) = 1 - o(1)$ nachweisen werden, außerdem werden wir $\mathrm{E}(T \mid \bigcap B_i) = \Omega(n^4 \log n)$ zeigen. Das reicht offenbar aus. Ein hilfreiches wahrscheinlichkeitstheoretisches Tool sind Chernoff-Schranken, die wir hier kurz als Fakt festhalten wollen. Man kennt das Ergebnis schon aus der Vorlesung GTI bzw. TIfAI.

Fakt 13.13. *Seien $X_1, X_2, \ldots, X_n \in \{0,1\}$ unabhängige Zufallsvariablen mit $0 < Prob(X_i = 1) < 1$ für alle i, außerdem sei $X := \sum_{i=1}^{n} X_i$.*

- $\forall \delta > 0$: $Prob(X \geq (1+\delta)E(X)) \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^{E(X)}$
- $\forall \delta \in]0;1[$: $Prob(X \geq (1+\delta)E(X)) \leq e^{-\delta^2 E(X)/3}$

- $\forall \delta \in]0;1[\colon Prob\left(X \leq (1-\delta)E(X)\right) \leq e^{-\delta^2 E(X)/2}$

Wir konkretisieren unseren Beweis durch Definition des Ereignisses B_1, wir bezeichnen so das Ereignis, dass nach der Initialisierung $b(x) = \Theta(n)$ und $c_C(x) = 1$ gilt. Wir beobachten zunächst, dass bei der Initialisierung $b(x)$ und $c_C(x)$ unabhängig voneinander sind. Zu $b(x)$ reicht es zu beobachten, dass unabhängig für jedes Dreieck gilt, dass es mit Wahrscheinlichkeit $1/4$ schlecht ist. Also ist die erwartete Anzahl initial schlechter Dreiecke $n/16$ und wir haben mit Wahrscheinlichkeit $1 - e^{-\Omega(n)}$ initial mindestens $n/32$ und höchstens $n/8$ schlechte Dreiecke; hier verwenden wir jeweils Chernoff-Schranken, um die Wahrscheinlichkeit der Abweichung vom Erwartungswert nach oben und unten zu beschränken.
Für den initialen Wert von $c_C(x)$ betrachten wir einen Knoten v in der Clique, sein erwarteter Grad beträgt mindestens $((n/2)-1)/2$, wir haben also für jeden Knoten mit Wahrscheinlichkeit $1-e^{-\Omega(n)}$ mindestens Grad $n/8$. Hat dieser Knoten v genau K Nachbarn in der Clique, so gilt für jeden anderen Knoten in der Clique, dass er im Erwartungswert $K/2$ Nachbarn mit v gemeinsam hat. Chernoff-Schranken liefern wieder, dass mit Wahrscheinlichkeit $1 - e^{-\Omega(n)}$ jeder Knoten mindestens $n/32$ Nachbarn mit v gemeinsam hat, so dass mit mindestens dieser Wahrscheinlichkeit die Clique zusammenhängend ist. Insgesamt haben wir also $\text{Prob}\left(B_1\right) = 1 - e^{-\Omega(n)}$.
Nun betrachten wir die Situation während des Laufs und konzentrieren uns auf die Anzahl der Bits, die Dreieckskanten entsprechen und die in x_t und y unterschiedlichen Wert haben. Wenn es k solche Bits gibt, sprechen wir von einem k-Schritt. Weil wir bisher die Beweise für RLS fast unverändert auf den (1+1)-EA übertragen konnten, werden wir das hier auch versuchen. Wir werden darum vorbereitend im Laufe dieses Beweises einige Argumente ausführlicher und allgemeiner ausrollen, als es für RLS eigentlich erforderlich ist. Das erleichtert uns später die Übernahme für den (1+1)-EA.
Wir betrachten jetzt einen solchen k-Schritt für RLS. Für $k \geq 3$ ist die Wahrscheinlichkeit eines k-Schritts natürlich 0 (das ist beim (1+1)-EA anders, an dieser Stelle werden wir also nochmal nachsehen müssen), für $k \in \{1,2\}$ können wir die Wahrscheinlichkeit eines k-Schritts mit

$$\frac{1}{2} \cdot \binom{3(n/4)}{k} \cdot \binom{m}{k}^{-1} = \Theta\left(\frac{n^k}{m^k}\right) = \Theta\left(\frac{1}{n^k}\right)$$

angeben. Wenn wir eine Folge von $n^{5/2}$ Schritten betrachten (wir nennen eine solche Folge eine *Phase*), so haben wir mit Wahrscheinlichkeit $1 - e^{-\Omega\left(\sqrt{n}\right)}$ eine Anzahl von $\Theta\left(n^{3/2}\right)$ 1-Schritten und $\Theta\left(n^{1/2}\right)$ 2-Schritten. Wir dürfen dabei nicht vergessen, dass in solchen k-Schritten grundsätzlich zusätzlich

Kanten in der Clique hinzukommen oder wegfallen können, für $k = 1$ ist das sogar tatsächlich der Fall.
Wir betrachten die erste Phase der Länge $n^{5/2}$ nach der Initialisierung, dabei setzen wir voraus, dass B_1 eingetreten ist, wir also in der Clique nur eine Zusammenhangskomponente haben und die Anzahl schlechter Dreiecke $\Theta(n)$ ist. Wir können ebenfalls voraussetzen, dass wir $\Theta\left(n^{1/2}\right)$ 2-Schritte haben. Natürlich können $\Theta\left(n^{1/2}\right)$ 2-Schritte die Anzahl schlechter Dreiecke um nicht mehr als $O\left(n^{1/2}\right)$ senken, außerdem wird die Anzahl der Zusammenhangskomponenten um nicht mehr als $O\left(n^{1/2}\right)$ vergrößert: Die Zielfunktion stellt sicher, dass c_C nur wachsen kann, wenn c_D gleichzeitig kleiner wird, da c_D in einem 2-Schritt um höchstens 2 kleiner werden kann, folgt die obere Schranke für die Zunahme in c_C.
Wir betrachten nun die $\Theta\left(n^{3/2}\right)$ 1-Schritte. In einem solchen Schritt kann c_C gar nicht wachsen, weil c_D höchstens um 1 kleiner werden kann und die Kante in den Dreiecken schwerer ist als Cliquen-Kanten. Die Anzahl schlechter Dreiecke kann durchaus um 1 sinken in einem solchen Schritt, allerdings wird dazu entweder eine Kante entfernt, so dass c_D wächst, oder es wird eine Kante eingefügt, was das Gewicht um $2n^2$ vergrößert. In beiden Fällen muss also zum Ausgleich c_C sinken, weil c_C anfangs 1 ist und höchstens auf $O\left(n^{1/2}\right)$ wächst, kann auch in den 1-Schritten der Wert von b um höchstens $O\left(n^{1/2}\right)$ wachsen.
Man macht sich analog zum Beweis von Theorem 13.5 klar, dass am Ende der ersten Phase mit großer Wahrscheinlichkeit der Graph zusammenhängend ist, über das Argument der unabhängigen Wiederholung können wir eine Schranke von $1 - e^{-\Omega\left(\sqrt{n}/\log n\right)}$ zeigen. Damit haben wir insgesamt mit dieser Wahrscheinlichkeit einen zusammenhängenden Graphen, für den $b(x) = \Theta(n)$ gilt, am Ende der ersten Phase.
Auf die erste Phase folgt die zweite Phase, eingangs gilt $c_G(x) = 1$, natürlich bleibt der Graph dann auch zusammenhängend. Wir haben am Anfang der zweiten Phase $b(x) = \Theta(n)$, wir wollen zeigen, dass wir am Ende der zweiten Phase mit großer Wahrscheinlichkeit einen Spannbaum haben und dabei $b(x) = \Theta(n)$ bleibt. Natürlich kann b nur in 2-Schritten sinken, andernfalls ginge der Zusammenhang verloren, davon gibt es aber wieder nur $\Theta\left(n^{1/2}\right)$ viele, so dass $b(x) = \Theta(n)$ erhalten bleibt.
Um einen Spannbaum zu bekommen, muss $ct(x)$ auf 0 sinken. Wir haben

$$\frac{1}{2} \cdot \binom{3 \cdot ct(x)}{1} \cdot \frac{1}{m}$$

als Wahrscheinlichkeit, dass das in einem Schritt passiert, weil $ct(x) \leq n/4$ gilt, haben wir, dass im Erwartungswert ct auf 0 sinkt in $O(m \log m)$ Schrit-

ten. In der Phase der Länge $n^{5/2}$ haben wir wieder eine Erfolgswahrscheinlichkeit von $1 - e^{-\Omega\left(\sqrt{n}/\log n\right)}$. Analog können wir zeigen, dass auch in der Clique die Anzahl der Kanten so sinkt, dass wir am Ende der zweiten Phase mit dieser Wahrscheinlichkeit einen Spannbaum haben und $b(x) = \Theta(n)$ gilt. In einem Spannbaum werden 1-Schritte nicht mehr akzeptiert. Fällt eine Kante aus den Dreiecken heraus, so geht der Zusammenhang verloren, wird eine Kante hinzugefügt, steigt das Gesamtgewicht so sehr, dass die Clique das nicht ausgleichen kann. 2-Schritte werden nur dann akzeptiert, wenn aus einem schlechten Dreieck ein gutes Dreieck wird, die Wahrscheinlichkeit dafür beträgt $\Theta\left(\frac{b(x)}{m^2}\right)$. Wir warten also im Durchschnitt $\Theta\left(m^2/b(x)\right)$ Schritte auf einen solchen Schritt und brauchen $\Theta(n)$ Schritte dieser Art. Wir summieren die erwarteten Wartezeiten und erhalten

$$\Theta\left(m^2 \log n\right) = \Theta\left(n^4 \log n\right)$$

als untere Schranke wie behauptet. □

Wie angekündigt werden wir Aussage und Beweis nun auf den (1+1)-EA übertragen. Dabei werden wir nur die Teile des gerade geführten Beweises überarbeiten, die wir nicht unverändert übernehmen können.

Theorem 13.14. *Die erwartete Optimierzeit des (1+1)-EA auf G_n beträgt $\Theta\left(n^4 \log n\right)$.*

Beweis. Die obere Schranke ist schon gezeigt (Theorem 13.10) für die untere Schranke folgen wir dem Beweis von Theorem 13.12. Bei der Initialisierung ändert sich gar nichts, alle Aussagen über Wahrscheinlichkeiten gelten asymptotisch unverändert auch beim (1+1)-EA. Damit ist klar, dass sich in den ersten beiden Phasen gar nichts wesentlich ändert. Wir haben also wieder mit ausreichend großer Wahrscheinlichkeit am Ende der zweiten Phase einen Spannbaum (V, E_x), für den $b(x) = \Theta(n)$ gilt. Die Bemerkungen über 1-Schritte und 2-Schritte bleiben richtig, es kann aber auch k-Schritte mit $k > 2$ geben. Die Wahrscheinlichkeit für einen k-Schritt ist $O\left(n^{-k}\right)$, wir betrachten nur $O(n^4 \log n)$ Schritte, es gibt also mit Wahrscheinlichkeit $1 - O(\log(n)/n)$ keine k-Schritte mit $k \geq 5$. Wir müssen also nur 3-Schritte und 4-Schritte zusätzlich betrachten. Für 3-Schritte gilt genau wie für 1-Schritte, dass sie nicht akzeptiert werden. Wir müssen uns also nur um 4-Schritte kümmern, dabei interessieren uns natürlich nur solche 4-Schritte, die $b(x)$ verändern. Die Wahrscheinlichkeit, dass das in einem 4-Schritt passiert, beträgt

$$\Theta\left(\left(\frac{b(x)}{m^2}\right)^2\right) = \Theta\left(\frac{b(x)^2}{n^8}\right) = O\left(\frac{1}{n^6}\right),$$

und wir sehen, dass mit Wahrscheinlichkeit $1 - O(\log(n)/n^2)$ so etwas in $O(n^4 \log n)$ Schritten gar nicht vorkommt. Damit ist klar, dass auch für den (1+1)-EA die erwartete Optimierzeit durch $\Omega(n^4 \log n)$ nach unten beschränkt ist. □

Was ändert sich, wenn wir nun f_4 betrachten? Der Unterschied zwischen f_4 und f_3 besteht darin, dass nicht mehr explizit belohnt wird, wenn die Anzahl der Kanten auf $n-1$ sinkt. Wir können darum nicht erwarten, dass wir weiterhin so schnell irgendeinen Spannbaum finden, Aussagen wie Theorem 13.5 und Theorem 13.6 lassen sich nicht mehr so leicht zeigen. Das ist aber auch gar nicht so schlimm, wir suchen ja minimale Spannbäume, nicht irgendwelche Spannbäume.
Es ist nicht schwer einzusehen, dass man für f_4 auf G_n ebenfalls sowohl für RLS als auch für den (1+1)-EA eine untere Schranke von $\Omega(n^4 \log n)$ zeigen kann. Die zweite Phase entfällt, sonst ändert sich kaum etwas. Beim Nachweis einer oberen Schranke für beliebige Graphen ändert sich an der erwarteten Zeit, bis man erstmalig einen zusammenhängenden Graphen hat, auch nichts. Auch das zentrale Strukturlemma 13.8 gilt unverändert. Damit kommt man mit ganz analogen Rechnungen auch zu einer oberen Schranke von $O(m^2(\log(n) + \log(w_{\max})))$, womit das folgende Ergebnis gezeigt ist.

Theorem 13.15. *Die erwartete Optimierzeit von RLS und dem (1+1)-EA auf MST mit f_4 ist $O(m^2(\log(n) + \log(w_{\max})))$ für beliebige Graphen und $\Omega(m^2 \log n)$ für G_n.*

13.2 Minimale Spannbäume multikriteriell

Betrachten wir jetzt noch f_5. Für RLS und den (1+1)-EA besteht die Schwierigkeit darin, dass wir $f_5(y) \leq f_5(x)$ sinnvoll definieren müssen. Naheliegend ist

$$(y_1, y_2) \leq (x_1, x_2) :\Leftrightarrow (y_1 \leq x_1) \wedge (y_2 \leq x_2)$$

zu definieren. Es ist nicht schwer einzusehen, dass wir auf diese Art eine *Halbordnung* definieren, also eine reflexive, transitive und antisymmetrische Relation. Wir sagen „y dominiert x“, wenn $f(y) \leq f(x)$ und $f(y) \neq f(x)$ gilt. Es entspricht dem Geist der beiden betrachteten Suchheuristiken, nur solche Suchpunkte zu behalten, die nicht dominiert werden von einem bereits betrachteten Suchpunkt. Was macht man aber mit unvergleichbaren Suchpunkten? Die kann es natürlich geben, $y_1 < x_1$ und gleichzeitig $y_2 > x_2$ ist denkbar, so dass weder $(y_1, y_2) \leq (x_1, x_2)$ noch $(x_1, x_2) \leq (y_1, y_2)$ gilt. Wir werden einen einfachen evolutionären Algorithmus für diesen Fall definieren,

der in seine Population alle nichtdominierten Punkte aufnimmt. Ist das sinnvoll? Wenn man den gesamten Suchraum absucht, wird man am Ende in der Population nur x haben, für die $f(x)$ minimal unter der Relation $\leq$ ist. Wir nennen solche Punkte *Pareto-Optima*. Die Menge aller optimalen Funktionswerte, also $\{f(x) \mid x \text{ ist Pareto-Optimum}\}$, nennen wir *Pareto-Front*. Es ist nicht schwer zu sehen, dass die Pareto-Front unter f_5 aus genau n Elementen besteht, für jede Anzahl von Zusammenhangskomponenten $k \in \{1, 2, \ldots, n\}$ eines. Die leere Menge $\emptyset$ ist offensichtlich ein Pareto-Optimum mit $k = n$ Zusammenhangskomponenten, auf der anderen Seite stehen minimale Spannbäume mit $k = 1$. Die Population des Algorithmus enthält also nie mehr als n Elemente, wird also nicht zu groß. Ist für jeden Punkt der Pareto-Front ein Repräsentant in der Population, befindet sich sicher auch ein minimaler Spannbaum darunter. Wir betrachten nun konkret den folgenden einfachen evolutionären Algorithmus für multikriterielle Probleme.

Algorithmus 13.16 (**Simple Evolutionary Multiobjetive Optimizer (SEMO)**).
1. *Wähle* $x \in \{0,1\}^n$ *uniform zufällig.*
2. $P := \{x\}$
3. *Wähle* $x \in P$ *uniform zufällig.*
4. *Wähle* $y \in N(x)$ *gemäß einer festen Wahrscheinlichkeitsverteilung.*
5. *If* $\nexists x' \in P\colon (f(x') \leq f(y)) \wedge (f(x') \neq f(y))$ *Then*
6. $\quad P := P \setminus \{x' \in P \mid f(y) \leq f(x')\}$
7. $\quad P := P \cup \{y\}$
8. *Weiter bei 3.*

Wegen der lokalen Mutation kann SEMO als multikriterielle Variante von RLS beschrieben werden. Man kann natürlich in Zeile 4 auch die Mutation des (1+1)-EA verwenden, das Ergebnis bezeichnet man als *global SEMO* oder *GSEMO*.
Wir betrachten zunächst SEMO und verwenden als Nachbarschaft nur direkte Hammingnachbarn, also $N(x) = \{y \in \{0,1\}^n \mid d(x,y) = 1\}$, als Wahrscheinlichkeitsverteilung verwenden wir die Gleichverteilung auf $N(x)$. Mit dieser eingeschränkten Nachbarschaft könnte RLS unter Umständen gar keinen minimalen Spannbaum finden, wir werden zeigen, dass das für SEMO kein Problem darstellt.

Theorem 13.17. *Die erwartete Optimierzeit von SEMO und GSEMO auf MST mit* f_5 *beträgt* $O(mn \cdot (n + \log w_{\max}))$.

Beweis. Wir betrachten zwei Phasen flexibler Länge, die erste Phase umfasst alle Schritte von der Initialisierung bis erstmals die leere Menge in P repräsentiert ist, die zweite Phase umfasst die restlichen Schritte bis zum Finden eines minimalen Spannbaums. Wir betrachten einen Schritt der ersten

Phase, sei $x^* \in P$ so gewählt, dass $w(x^*) = \min\{w(x) \mid x \in P\}$ gilt. Wir beobachten, dass x^* eine maximale Anzahl von Zusammenhangskomponenten hat. Hätte ein $x' \in P$ eine größere Anzahl von Zusammenhangskomponenten, so würde x' von x^* dominiert und könnte gar nicht in P vorhanden sein. Wir können eine beliebige Kante aus x^* entfernen, dann sinkt das Gewicht, die Zahl der Zusammenhangskomponenten kann gleichzeitig wachsen. Wir überlegen uns wieder, dass es m Wahlen eines Nachbarn gibt, bei denen im Durchschnitt das Gewicht um $w(x^*)/m$ sinkt. Wir betrachten jetzt entsprechend $t = \lceil m \ln(2) (\log(mw_{\max}) + 1) \rceil$ solche Schritte und sehen, dass dann das Gewicht $< 1/2$ ist. Weil $|P| \leq n$ gilt, beträgt die Wahrscheinlichkeit, für einen solchen Schritt, x^* auszuwählen, $\Omega(1/n)$. Wir erhalten damit insgesamt $O(mn (\log(n) + \log(w_{\max})))$ als obere Schranke für die erwartete Länge der ersten Phase.
In der zweiten Phase nehmen wir an, dass P die Elemente R_n, R_{n-1}, ..., R_i enthält, dabei seien alle R_i Pareto-Optima und R_j enthalte genau j Zusammenhangskomponenten. Anfangs ist $i \leq n$, weil $\emptyset$ sicher in P enthalten ist.
Analog zum Algorithmus von Kruskal gibt es immer einen Hammingnachbarn, der aus R_i ein geeignetes R_{i-1} erzeugt. Die Wahrscheinlichkeit für einen solchen Schritt ist immer durch $\Omega((1/n) \cdot (1/m))$ beschränkt, so dass wir $O(mn)$ als obere Schranke für die Wartezeit haben, bis i um 1 sinkt. Damit ergibt sich $O(mn^2)$ als obere Schranke für die erwartete Länge der zweiten Phasen und die obere Schranke $O(mn \cdot (n + \log w_{\max}))$ folgt. □

Vergleicht man die Performanz mit f_4 und f_5 auf Graphen mit polynomiellen Kantengewichten, so erhält man bei dichten Graphen (also Graphen mit $\Theta(n^2)$ Kanten) eine Laufzeit von $\Theta(n^4 \log n)$ für f_4 und eine obere Schranke $O(n^4)$ für f_5, also ein leichter Vorteil für die multikriterielle Variante. Für dünnbesetzte Graphen (also Graphen mit $\Theta(n)$ Kanten) ergibt sich eine Laufzeit von $\Theta(n^2 \log n)$ für f_4 und eine obere Schranke $O(n^3)$ für f_5, also ein erkennbarer Nachteil für die multikriterielle Variante. Allerdings haben wir dort keine untere Schranke, so dass wir eher keine Aussage treffen können.

13.3 Minimale Spannbäume mit dem Metropolis-Algorithmus und Simulated Annealing

Bis jetzt haben wir noch kein Ergebnis für Simulated Annealing und den Metropolis-Algorithmus mit relevanter Annealingstrategie bzw. relevanter Temperatur. Wir werden das jetzt nachholen und dabei zwei interessante

Fragen beantworten: Simulated Annealing und der Metropolis-Algorithmus können beide auch Verschlechterungen im Funktionswert akzeptieren. Führt das dazu, dass direkte Hammingnachbarn als Nachbarschaft ausreichen, um f_3 oder f_4 effizient zu optimieren? Es ist klar, dass Simulated Annealing komplexer ist als der Metropolis-Algorithmus und die Wahl einer Annealingstrategie schwieriger ist als die Wahl einer festen Temperatur. Ist zum Ausgleich Simulated Annealing auch mächtiger als der Metropolis-Algorithmus, kann Simulated Annealing erheblich schneller oder erfolgreicher sein?
Wir ersparen uns für diese beiden Algorithmen das Finden irgendeines zusammenhängenden Graphen und führen stattdessen noch etwas Problemwissen ein: Wir wählen für beide Algorithmen (abweichend von der Definition als Algorithmus 13.1 bzw. Algorithmus 13.2) 1^n als festen Startpunkt x_1. Natürlich ist $E_{x_1} = E$) und (V, E_{x_1}) darum zusammenhängend. Wir benutzen jetzt die Zielfunktion f_2, so dass sichergestellt ist, dass wir immer nur zusammenhängende Graphen als (V, E_{x_t}) haben. Wir wählen wie angekündigt $N(x) := \{y \in \{0,1\}^n \mid d(x,y) = 1\}$ und die Gleichverteilung als Wahrscheinlichkeitsverteilung für beide Algorithmen.
Wir definieren jetzt zunächst einen weiteren Beispielgraphen, den wir D_n nennen. Die Analyse von Simulated Annealing und des Metropolis-Algorithmus für D_n wird uns zentrale Einsichten bescheren.

Definition 13.18. *Für $n \in \mathbb{N}$ ist der Graph $D_n = (V, E, w)$ definiert durch $V := \{1, 2, \ldots, 4n+1\}$,*

$$E := \bigcup_{i=1}^{2n} \big\{\{2i-1, 2i\}, \{2i-1, 2i+1\}, \{2i, 2i+1\}\big\}$$

und

$$w(e) := \begin{cases} 1 & \text{falls } e \in \{\{2i-1,2i\},\{2i,2i+1\}\} \text{ für ein } i \leq n, \\ m & \text{falls } e = \{2i-1, 2i+1\} \text{ für ein } i \leq n, \\ m^2 & \text{falls } e \in \{\{2i-1,2i\},\{2i,2i+1\}\} \text{ für ein } i > n, \\ m^3 & \text{falls } e = \{2i-1, 2i+1\} \text{ für ein } i > n. \end{cases}$$

Offenbar besteht D_n aus n Dreiecken mit Kantengewichten m und 1 sowie aus n Dreiecken mit Kantengewichten m^3 und m^2, dabei enthält jedes Dreieck zwei leichte und eine schwere Kante. Die letztgenannten Dreiecke nennen wir schwer, die anderen Dreiecke heißen leicht. Wie vorhin heißt ein Dreieck gut, wenn die beiden leichten Kanten gewählt sind, sind alle Kanten gewählt, heißt es komplett, ist eine leichte und eine schwere Kante gewählt, heißt es schlecht. Wir vorhin bezeichnen wir mit $b(x)$ die Anzahl der schlechten

Dreiecke in (V, E_x), mit $g(x)$ die Anzahl der guten Dreiecke in (V, E_x) und mit $ct(x)$ die Anzahl der kompletten Dreiecke in (V, E_x). Weil der Graph zusammenhängend ist, gilt $b(x) + g(x) + ct(x) = 2n$.
Am Anfang sind alle Dreiecke komplett, aus kompletten Dreiecken kann beliebig eine Kante entfernt werden. Wenn das geschieht, haben wir nach einiger Zeit $\Theta(m)$ schlechte Dreiecke: Es ist für ein Dreieck doppelt so wahrscheinlich, dass es schlecht wird, weil es zwei leichte und nur eine schwere Kante gibt. Weil wir nur direkte Hammingnachbarn betrachten, muss ein schlechtes Dreieck erst komplett werden, bevor es gut werden kann. Dabei wächst in einem leichten Dreiecke der Funktionswert um 1, in einem schweren Dreiecke wächst er um m^2. Wir wünschen uns also eine Temperatur, bei der eine Zunahme des Funktionswertes um m^2 noch recht wahrscheinlich ist. Andererseits können wir natürlich nur dann jemals zum minimalen Spannbaum kommen, wenn wir gute Dreiecke nicht direkt wieder verlieren. Wir verlieren ein gutes Dreieck, wenn wir aus ihm ein komplettes Dreieck machen. Das erhöht in einem leichten Dreieck den Funktionswert um m, in einem schweren Dreieck erhöht es den Funktionswert um m^3. Wir wünschen uns also eine Temperatur, bei der eine Zunahme des Funktionswertes um m schon recht unwahrscheinlich ist. Weil wir uns gleichzeitig wünschen, dass Funktionswertvergrößerungen um m^2 wahrscheinlich sind, liegt es nahe, dass wir keine Temperatur finden werden, die uns in allen Belangen glücklich macht. Der Graph scheint also für den Metropolis-Algorithmus schwierig zu sein. Für Simulated Annealing besteht aber noch Hoffnung: Solange die Temperatur noch recht hoch ist, können die schweren Dreiecke richtig werden, wenn die Temperatur sinkt, wird es wahrscheinlich, dass sie auch alle gut bleiben und wir können bei niedriger Temperatur darauf hoffen, dass nun auch alle leichten Dreiecke gut werden. Bevor wir diese Ideen konkretisieren, werden wir noch ein Resultat aus der Wahrscheinlichkeitstheorie herleiten, das in vielen Bereichen nützlich ist und sich mit Hilfe des Optional-Stopping-Theorems (Theorem 10.13) einfach beweisen lässt.

Theorem 13.19 (Gambler's-Ruin-Theorem). *Seien $s \in \mathbb{N}$, $a \in \{0, 1, \ldots, s\}$, $p_A \in]0;1[\setminus\{1/2\}$, $p_B := 1 - p_A$. Betrachte den Zufallsprozess $(X_i)_{i\geq 0}$ mit $X_0 := a$ und*

$$X_{i+1} = \begin{cases} X_i + 1 & \text{mit Wahrscheinlichkeit } p_A, \\ X_i - 1 & \text{mit Wahrscheinlichkeit } p_B. \end{cases}$$

Es bezeichne $T := \min\{X_t \mid X_t \in \{0, s\}\}$, $q := p_B/p_A$ und $q_A := Prob\,(X_T = 0)$. Es gilt $q_A = \frac{q^a - q^s}{1 - q^s}$ und $E(T) = \frac{(1-q_A)\cdot s - a}{p_A - p_B}$.

Beweis. Wir benutzen das Optional-Stopping-Theorem (Theorem 10.13) und beobachten zunächst, dass T in der Tat eine Stoppzeit ist. Wir behaupten, dass $M_t := q^{X_t}$ ein Martingal in Bezug auf $(X_i)_{i\geq 0}$ ist. Dazu rechnen wir nach

$$\begin{aligned}
\mathrm{E}\left(M_{t+1} \mid X_1, X_2, \ldots, X_t\right) &= \mathrm{E}\left(M_{t+1} \mid X_t\right) = \mathrm{E}\left(q^{X_{t+1}} \mid X_t\right) \\
&= p_A \cdot q^{X_t+1} + p_B \cdot q^{X_t-1} = q^{X_t} \cdot \left(p_A \cdot q + \frac{p_B}{q}\right) \\
&= q^{X_t} \cdot \left(p_A \cdot \frac{p_B}{p_A} + p_B \cdot \frac{p_A}{p_B}\right) \\
&= q^{X_t} \cdot (p_B + p_A) = q^{X_t}
\end{aligned}$$

und sehen, dass die Martingal-Eigenschaft erfüllt ist. Mit dem Optional-Stopping-Theorem folgt $\mathrm{E}\left(M_T\right) = \mathrm{E}\left(M_0\right) = \mathrm{E}\left(q^{X_0}\right) = q^a$ einerseits, andererseits gilt aber auch

$$\mathrm{E}\left(M_T\right) = q_A \cdot q^0 + q_B \cdot q^s = q_A + (1 - q_A) \cdot q^s = (1 - q^s) \cdot q_A + q^s$$

und

$$q^a = (1 - q^s) \cdot q_A + q^s$$

folgt. Wir haben $q_A = \frac{q^a - q^s}{1 - q^s}$ als direkte Folge wie behauptet.
Für $\mathrm{E}\left(T\right)$ betrachten wir zunächst $N_t := X_t - t \cdot (p_A - p_B)$ und behaupten, dass auch N_t ein Martingal in Bezug auf $(X_i)_{i\geq 0}$ ist. Dazu rechnen wir wieder nach

$$\begin{aligned}
&\mathrm{E}\left(N_{t+1} \mid X_1, X_2, \ldots, X_t\right) = \mathrm{E}\left(N_{t+1} \mid X_t\right) \\
= \;& p_A \cdot (X_t + 1 - (t+1) \cdot (p_A - p_B)) + p_B \cdot (X_t - 1 - (t+1) \cdot (p_A - p_B)) \\
= \;& (p_A + p_B) \cdot X_t + p_A - p_B - (p_A + p_B) \cdot (t+1) \cdot (p_A - p_B) \\
= \;& X_t + p_A - p_B - (t+1) \cdot (p_A - p_B) = X_t - t \cdot (p_A - p_B) = N_t
\end{aligned}$$

und sehen, dass die Martingal-Eigenschaft erfüllt ist. Wir wenden wieder das Optional-Stopping-Theorem (Theorem 10.13) an und haben dadurch $\mathrm{E}\left(N_T\right) = \mathrm{E}\left(N_0\right) = \mathrm{E}\left(X_0\right) = a$. Andererseits gilt aber auch

$$\mathrm{E}\left(N_T\right) = q_A \cdot (0 - \mathrm{E}\left(T\right) \cdot (p_A - p_B)) + q_B \cdot (s - \mathrm{E}\left(T\right) \cdot (p_A - p_B))$$

und

$$a = q_A \cdot (0 - \mathrm{E}\left(T\right) \cdot (p_A - p_B)) + q_B \cdot (s - \mathrm{E}\left(T\right) \cdot (p_A - p_B))$$

folgt. Daraus können wir wie behauptet

$$\mathrm{E}\left(T\right) = \frac{(1 - q_A) \cdot s - a}{p_A - p_B}$$

schließen. □

Nach diesem kleinen Ausflug kommen wir nun zum Metropolis-Algorithmus auf D_n zurück. Wir zeigen, dass es tatsächlich keine gute Temperatur gibt.

Theorem 13.20. *Es gibt eine Konstante $c > 0$, so dass der Metropolis-Algorithmus auf D_n mit Wahrscheinlichkeit mindestens $1-e^{-\Omega(n)}$ mindestens e^{cn} Schritte zum Finden eines minimalen Spannbaums braucht.*

Beweis. Weil anfangs alle Dreiecke komplett sind und ein komplettes Dreieck mit Wahrscheinlichkeit $2/3$ schlecht wird, wenn es das erste Mal verändert wird, haben wir mit Wahrscheinlichkeit $1 - e^{-\Omega(n)}$ mindestens ein schlechtes schweres Dreieck, bevor ein minimaler Spannbaum gefunden wird. Wir unterscheiden zwei Fälle nach der festgelegten Temperatur T.
Ist die Temperatur T niedrig, gilt konkret $T < m$, so ist die Wahrscheinlichkeit, dass wir dieses schlechte Dreieck wieder komplett machen, was notwendig ist, wenn es gut werden soll, was es im minimalen Spannbaum ist, durch

$$\frac{1}{m} \cdot e^{-m^2/T} \leq \frac{e^{-m}}{m} = e^{-\Omega(n)}$$

nach oben beschränkt. Daraus folgt die Behauptung alleine durch die Wartezeit auf einen solchen Schritt.
Ist andererseits die Temperatur hoch, gilt also konkret $T \geq m$, so betrachten wir nur die n leichten Dreiecke. Es sei X_t die Anzahl guter Dreiecke nach dem t-ten Schritt. Wir haben anfangs $X_t = 0$ und suchen $T = \min\{t \mid X_t = n\}$. Es ist leicht einzusehen, dass X_{t+1} ausschließlich von $b(x)$, $g(x)$ und $ct(x)$ abhängt. Wir betrachten ein Dreieck in einem Schritt und unterscheiden drei Fälle nach seinem Typ. Ist das Dreieck komplett, so bleibt es mit Wahrscheinlichkeit $1-3/m$ komplett, mit Wahrscheinlichkeit $1/m$ wird ein gutes Dreieck daraus und mit Wahrscheinlichkeit $2/m$ wird ein schlechtes Dreieck daraus. Ist das Dreiecke anfangs gut, so wird es mit Wahrscheinlichkeit $(1/m) \cdot e^{-m/T}$ komplett und bleibt mit der Gegenwahrscheinlichkeit $1 - (1/m) \cdot e^{-m/T}$ gut. Schlecht kann es in einem Schritt nicht werden. Analog wird ein schlechtes Dreieck mit Wahrscheinlichkeit $(1/m) \cdot e^{-1/T}$ komplett und bleibt mit der Gegenwahrscheinlichkeit $1 - (1/m) \cdot e^{-1/T}$ schlecht, es kann in einem Schritt nicht gut werden. Wir gewinnen aus diesen Transitionswahrscheinlichkeiten pessimistische Abschätzungen, die die Wahrscheinlichkeiten, ein gutes Dreieck zu erhalten, überschätzen. Wir bekommen

$$\text{Prob}\,(X_{t+1} = a + 1 \mid X_t = a) \leq \frac{n-a}{m}$$

und

$$\text{Prob}\,(X_{t+1} = a - 1 \mid X_t = a) \geq \frac{a}{m} \cdot e^{-m/T} \geq \frac{a}{3m}.$$

Wir benutzen diese Schranken als Transitionswahrscheinlichkeiten für einen Zufallsprozess $\tilde{X}_t$ und sehen, dass $\tilde{X}_{t+1}$ nur noch von $\tilde{X}_t$ abhängig ist. Wir interessieren uns natürlich für Schritte, in denen die Anzahl der guten Dreiecke sich ändert, darum betrachten wir $\text{Prob}\,(X_{t+1} = X_t - 1 \mid X_{t+1} \neq X_t)$. Wir erhalten

$$\text{Prob}\,(X_{t+1} = X_t - 1 \mid X_{t+1} \neq X_t) \geq \frac{a/(3m)}{a/(3m) + (n-a)/m} = \frac{a}{3n - 2a}$$

und entscheiden uns dafür, die Situation erst zu betrachten, wenn es erstmalig $(10/11)n$ gute Dreiecke gibt. Wir beenden die Betrachtung, wenn erstmalig $(9/11)n$ oder n gute Dreiecke erreicht werden. Damit haben wir

$$\text{Prob}\,(X_{t+1} = X_t - 1 \mid X_{t+1} \neq X_t) \geq \frac{(9/11)n}{3n - 2 \cdot (9/11)n} = \frac{3}{5}.$$

Wir betrachten also einen Zufallsprozess, der bei $(10/11)n$ startet, sich in jedem Schritt um genau 1 verändert, immer zwischen $(9/11)n$ und n ist und für den die Wahrscheinlichkeit einer Zunahme $2/5$ und die Wahrscheinlichkeit einer Abnahme $(3/5)$ beträgt. Wir befinden uns in der Situation von Theorem 13.19 mit $s = (2/11)n$, $a = (1/11)n$ und $p_A = 3/5$, dabei interessieren wir uns für q_A. Wir haben

$$q_A = \frac{(2/3)^{(2/11)n} - (2/3)^{(1/11)n}}{(2/3)^{(2/11)n} - 1} = \left(\frac{2}{3}\right)^{n/11} \cdot \frac{1 - (2/3)^{n/11}}{1 - (2/3)^{(2/11)n}} < \left(\frac{2}{3}\right)^{n/11} = e^{-\Omega(n)}$$

und sehen, dass das behauptete Ergebnis folgt. □

Wir sehen, dass es für den Metropolis-Algorithmus für D_n keine geeignete Temperatur gibt. Das scheint für Simulated Annealing kein gutes Vorzeichen zu sein. Aber wir sollten nicht zu pessimistisch sein. Wir dürfen die Hoffnung haben, dass wir mit einer hohen Temperatur die schweren Dreiecke alle gleichzeitig gut bekommen, Abkühlen dafür sorgt, dass diese Eigenschaft erhalten bleibt und dann die leichten Dreiecke auch alle gut werden. Wir werden zeigen, dass das mit großer Wahrscheinlichkeit auch passiert. Allerdings werden wir kein Resultat für die erwartete Optimierzeit zeigen: Wenn nicht alle schweren Dreiecke gut sind, wenn die Temperatur deutlich absinkt, was mit kleiner Wahrscheinlichkeit passiert, dann haben wir keine Hoffnung mehr, eine optimale Lösung in akzeptabler Zeit zu finden. Wir erinnern uns an Probability Amplification als Methode, um in solchen Situationen mit unabhängigen Restarts auch eine gute erwartete Optimierzeit zu erreichen.

Theorem 13.21. *Sei p ein Polynom in m. Für jede ausreichend große Konstante c findet Simulated Annealing mit Annealingstrategie $T(t) = m^3 \cdot$*

$(1-1/(cm))^{t-1}$ *einen minimalen Spannbaum auf* D_n *in höchstens* $3cm\ln m$ *Schritten mit Wahrscheinlichkeit mindestens* $1-1/p(n)$.

Beweis. Wir betrachten einen Lauf von Simulated Annealing auf D_n und unterteilen ihn gedanklich in vier Phasen. Die erste Phase beginnt mit der Initialisierung und endet, wenn erstmalig $T(t) \leq m^{5/2}$ gilt. Die zweite Phase beginnt im Anschluss und endet, wenn erstmalig $T(t) < m^2$ gilt. Die dritte Phase beginnt im Anschluss und endet, wenn erstmalig $T(t) \leq \sqrt{m}$ gilt. Die vierte und letzte Phase umfasst die Schritte, bis erstmalig $T(t) < 1$ gilt. Wir beenden also unseren Betrachtungen nach t Schritten, wenn $m^3 \cdot (1-1/cm)^{t-1} < 1$ gilt und das für kein kleineres $t' \in \mathbb{N}$ der Fall ist. Wir sehen, dass wir nicht mehr als $3cm\ln m$ Schritte betrachten wie behauptet. Analog überlegt man sich, dass jede der vier Phasen mindestens Länge $(c/4)m\ln m$ hat.
Wir ignorieren die erste Phase völlig, uns ist gleichgültig, was in dieser Zeit passiert. Wir warten nur darauf, dass die Temperatur sinkt. In der zweiten Phase gilt $T(t) \in [m^2; m^{5/2}]$ und wir betrachten ausschließlich schwere Dreiecke. Die Wahrscheinlichkeit, dass ein gutes schweres Dreieck komplett wird, ist durch $e^{-m^3/T(t)} \leq e^{-m^{1/2}}$ nach oben beschränkt, so etwas kommt also mit ausreichend großer Wahrscheinlichkeit nicht vor. Die Wahrscheinlichkeit, dass in einem Schritt eine bestimmte Kante betroffen ist, beträgt $1/m$. Jede Kante ist also im Erwartungswert in dieser zweiten Phase mindestens $((c/4)m\ln m)/m = (c/4)\ln m$-mal betroffen. Wir haben also mit ausreichend großer Wahrscheinlichkeit mindestens $c''\ln m$ Versuche, eine Kante hinzuzunehmen oder herauszunehmen für jede Kante. Die Wahrscheinlichkeit, aus einem schlechten schweren Dreieck ein gutes schweres Dreieck zu machen in den nächsten beiden Schritten, die dieses Dreieck betreffen, ist durch

$$\frac{1}{3} \cdot e^{-m^2/T(t)} \cdot \frac{1}{3} \geq \frac{1}{9e}$$

nach unten beschränkt. Wenn c groß genug ist, was wir voraussetzen, passiert das mit Wahrscheinlichkeit $1-n^{-k}$ mit allen schweren Dreiecken im Laufe der zweiten Phase, dabei ist k eine beliebige Konstante, die nur von c abhängt. Wir haben also mit ausreichend großer Wahrscheinlichkeit am Ende der zweiten Phase alle schweren Dreiecke gut.
In der dritten Phase warten wieder wiederum nur darauf, dass die Temperatur sinkt, weil sie immer kleiner ist als in der zweiten Phase, bleiben in der dritten Phase alle guten schweren Dreiecke mit großer Wahrscheinlichkeit gut. Wir können also zu Beginn der vierten Phase voraussetzen, dass alle schweren Dreiecke gut sind und wir uns nur noch um die leichten Dreiecke sorgen müssen: Dass die guten schweren Dreiecke gut bleiben, folgt wie in der zweiten und dritten Phase. In der vierten Phase gilt $T(t) \in [1; \sqrt{m}]$

und wir betrachten nur leichte Dreiecke. Ein gutes leichtes Dreieck wird nur mit Wahrscheinlichkeit $\leq e^{-m/T(t)} \leq e^{-m^{1/2}}$ komplett, so dass uns mit ausreichend großer Wahrscheinlichkeit alle guten Dreiecke erhalten bleiben. Wir können die Anzahl der versuchten Änderungen jeder Kante genau wie die Wahrscheinlichkeit, aus einem schlechten leichten Dreieck ein gutes leichtes Dreieck zu machen in den nächsten beiden Schritten, die dieses Dreieck betreffen, abschätzen, wie wir das in der zweiten Phase für die schweren Dreiecke gemacht haben. Daraus folgt die Behauptung. □

Wenn man das Ergebnis verallgemeinern möchte, kann man sich zunächst D_n noch einmal etwas genauer ansehen. Wesentlich waren die unterschiedlichen Kantengewichte innerhalb eines Dreiecks, außerdem natürlich die Unterschiede zwischen leichten und schweren Dreiecken. Dass im Beweis von Theorem 13.21 die Dreiecke eines Typs alle optimiert werden konnten, lag daran, dass die Kantengewichte innerhalb eines Dreiecks so unterschiedlich sind, dass es wahrscheinlich ist, eine leichte Kante hinzuzunehmen, es aber unwahrscheinlich ist, eine schwere Kante wieder hinzuzunehmen. Wir betrachten nun eine modifizierte Version von D_n, die nur noch Dreiecke gleichen Typs enthält, dabei haben die beiden leichten Kanten Gewicht w und die schwere Kante hat Gewicht $(1 + \varepsilon(m))w$ in jedem Dreieck. Wir nennen diese modifizierte Instanz D'_n und beweisen zunächst, dass sie für nicht zu kleine Werte von ε durch den Metropolis-Algorithmus optimal gelöst werden können.

Theorem 13.22. *Sei $\varepsilon > 0$ konstant. Für $\varepsilon(m) \geq \varepsilon$ findet der Metropolis-Algorithmus mit Temperatur $T := \varepsilon w/(3 \ln m)$ einen minimalen Spannbaum in erwarteter Polynomialzeit.*

Beweis. Leichte Kanten werden mit Wahrscheinlichkeit $e^{-w/T} = m^{-3/\varepsilon}$ hinzugefügt, schwere Kanten hingegen werden nur mit Wahrscheinlichkeit $e^{-(1+\varepsilon(m))w/T} \leq e^{-(1+\varepsilon)w/T} = m^{-3-3/\varepsilon}$ hinzugefügt.
Wir betrachten nun $m^{2+3/\varepsilon}$ viele Schritte, die Wahrscheinlichkeit, in dieser Zeit eine schwere Kante hinzuzufügen, ist durch $1/m$ nach oben beschränkt. Wir haben mit großer Wahrscheinlichkeit in $m^{2+3/\varepsilon}$ Schritten für jedes Dreieck $\Omega\left(m^{1+(3/\varepsilon)}\right)$ viele Versuche, Kanten des Dreiecks zu verändern. Die Wahrscheinlichkeit, in zwei aufeinanderfolgenden Versuchen aus einem schlechten Dreieck ein gutes zu machen, ist durch $\Omega\left(m^{-3/\varepsilon}\right)$ nach unten beschränkt. Also werden mit großer Wahrscheinlichkeit alle Dreiecke gut in dieser Zeit. Wenn das misslingt, können wir alle Argumente wiederholen, sind also in der Situation unabhängiger Restarts. Weil die Misserfolgswahrscheinlichkeit $o(1)$ ist, haben wir damit insgesamt die behauptete erwartete polynomielle Optimierzeit. □

Für $\varepsilon(m) \geq \varepsilon > 0$ ist also selbst der Metropolis-Algorithmus auf D'_n erfolgreich, dabei ist $\varepsilon > 0$ eine beliebig kleine positive Konstante. Was ist, wenn $\lim_{m\to\infty} \varepsilon(m) = 0$ gilt? Wir zeigen, dass selbst Simulated Annealing dann versagt. Das ist ein ernüchterndes Resultat: Simulated Annealing kann bei Benutzung der direkten Hammingnachbarschaft nicht minimale Spannbäume in polynomieller Zeit berechnen, jedenfalls nicht, wenn die Unterschiede zwischen den Kantengewichten eher gering sind.

Theorem 13.23. *Für $\varepsilon(m) = o(1)$ findet Simulated Annealing mit beliebiger Annealingstratgie für jedes Polynom p und jede Konstante k einen minimalen Spannbaum von D'_n in $O(p(m))$ Schritten nur mit Wahrscheinlichkeit $O\big(m^{-k}\big)$.*

Beweis. Wir betrachten zunächst den Metropolis-Algorithmus mit beliebiger Temperatur T, beschränken uns also zunächst auf den Fall, dass die Temperatur fest ist. Wir machen uns leicht klar, das wir mit ausreichend großer Wahrscheinlichkeit in eine Situation mit $\Theta(m)$ schlechten Dreiecken kommen werden. Sei $p(T)$ die Wahrscheinlichkeit, eine w-Kante zusätzlich aufzunehmen und $p_\varepsilon(T)$ die Wahrscheinlichkeit, eine $(1+\varepsilon(m))w$-Kante zusätzlich aufzunehmen. Wir haben

$$\frac{p(T)}{p_\varepsilon(T)} = \frac{e^{-w/T}}{e^{-(1+\varepsilon(m))w/T}} = e^{\varepsilon(m)w/T}$$

und wollen zeigen, dass dieser Unterschied zu klein ist. Wir müssen natürlich in einem Lauf polynomieller Länge in der Lage sein, leichte Kanten hinzuzufügen, dazu muss $T \geq w/(\gamma \cdot \ln m)$ gelten für eine Konstante $\gamma > 0$. Wir setzen das ein und erhalten, dass

$$\frac{p(T)}{p_\varepsilon(T)} = e^{\varepsilon(m)w/T} = m^{\varepsilon(m)\gamma} < m^\delta$$

gilt für jede beliebig kleine Konstante $\delta > 0$.
Zur Beschreibung des ablaufenden Zufallsprozesses genügt es, die Anzahl schlechter Dreiecke $b(x)$ und die Anzahl kompletter Dreiecke $ct(x)$ zu kennen. Wir definieren eine Potenzialfunktion $\Phi(x) = 2b(x) + ct(x)$ und analysieren deren Verlauf, anfangs gilt dabei natürlich $\Phi(x) = n$ und wir sind fertig, wenn $\Phi(x) = 0$ gilt.
Wir beobachten, dass die Potenzialfunktion Φ ihren Wert in einem Schritt nur um höchstens 1 ändern kann. Sie wächst, wenn in einem kompletten Dreieck eine leichte Kante entfernt wird oder in einem guten Dreieck eine schwere Kante hinzukommt. Folglich gilt

$$\text{Prob}\left(\Phi(x_{t+1}) = a+1 \mid \Phi(x_t) = a\right) = \frac{2ct(x_t)}{m} + \frac{n - b(x_t) - ct(x_t)}{m} \cdot p_\varepsilon(T).$$

Die Potenzialfunktion fällt, wenn eine schwere Kante aus einem kompletten Dreieck entfernt wird oder eine leichte Kante zu einem schlechten Dreieck hinzugefügt wird. Folglich gilt

$$\text{Prob}\left(\Phi(x_{t+1}) = a - 1 \mid \Phi(x_t) = a\right) = \frac{ct(x_t)}{m} + \frac{b(x_t)}{m} \cdot p(T).$$

Mit der übrigbleibenden Wahrscheinlichkeit ändert die Potenzialfunktion ihren Wert nicht.
Wenn wir zeigen könnten, dass

$$\frac{\text{Prob}\left(\Phi(x_{t+1}) = a + 1 \mid \Phi(x_t) = a\right)}{\text{Prob}\left(\Phi(x_{t+1}) = a + 1 \mid \Phi(x_t) = a\right) + \text{Prob}\left(\Phi(x_{t+1}) = a - 1 \mid \Phi(x_t) = a\right)} > \frac{1}{2} + \delta$$

für eine positive Konstante $\delta > 0$ gilt, so könnten wir durch Anwendung von Theorem 13.19 zeigen, dass der Metropolis-Algorithmus mit ausreichend großer Wahrscheinlichkeit in einer polynomiellen Anteil von Schritten nicht fertig wird. Tatsächlich kann man sogar durch einfaches Nachrechnen 3/5 als untere Schranke für diese bedingte Wahrscheinlichkeit zeigen. Damit ist das Resultat für jede beliebige feste Temperatur gezeigt. Dadurch gilt dies hier auch für Simulated Annealing: Es gibt für keines der Dreiecke eine gute Temperatur, also ist auch Simulated Annealing erfolglos. □

Wir wollen unsere Betrachtungen von allgemeinen Suchheuristiken hier beenden. Wir erwähnen noch ohne Beweis, dass Simulated Annealing tatsächlich genau auf den Graphklassen erfolgreich ist, bei denen alle verschiedenen Gewichte sich mindestens um einen Faktor $1 + \varepsilon$ mit beliebig kleiner positiver Konstanten ε unterscheiden, dabei muss natürlich eine geeignete Annealingstrategie gewählt werden. Dazu muss man zeigen, dass bei ausreichend langsamen Abkühlen gute Kanten nach und nach hinzugefügt werden, dabei betrachtet man die Kanten sortiert nach Gewicht in absteigender Reihenfolge. Außerdem macht man sich klar, dass zu schwere Kanten später mit großer Wahrscheinlichkeit auch nicht mehr hinzugefügt werden.

14 Lineare Programmierung

Als wir im Kapitel 10 randomisierte Approximationen für MAX-SAT-Probleme gesucht haben (Abschnitt 10.1), haben wir bei der Methode des randomisierten Rundens (Algorithmus 10.5) ein lineares Programm formuliert und angenommen, dass wir es effizient lösen können. Wir werden uns diesem Problem (in Algorithmus 10.5 ein Teilproblem) in diesem Kapitel zuwenden, dabei werden wir uns aber mit einer heuristischen Lösung zufriedengeben. Wir werden den Simplex-Algorithmus besprechen, einen Algorithmus zur Lösung linearer Programme, der im Worst Case keine polynomiell beschränkte Laufzeit hat, der aber einfach ist und sich in der Praxis als so schnell erwiesen hat, dass er in aller Regel noch immer das Mittel der Wahl ist.
Wir erinnern noch einmal an die Problemstellung. Bei der linearen Optimierung ist eine lineare Zielfunktion Z über den Variablen $x_1, x_2, \ldots, x_n$ zu minimieren, dabei gibt es m lineare Nebenbedingungen. Etwas formaler schreiben wir

Minimiere	$Z(x) := c_1x_1 + c_2x_2 + \cdots c_nx_n$
unter $\boldsymbol{m}$ **Nebenbedingungen**	$a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,n}x_n \leq b_i$ mit $i \in \{1, 2, \ldots, m\}$
und	$x_j \geq 0$ für $j \in \{1, 2, \ldots, n\}$

Eingabe ist also

$$c = \begin{pmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{pmatrix} \quad A = \begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & & \cdots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{pmatrix} \quad b = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{pmatrix}$$

und Ausgabe ist ein

$$x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \geq 0$$

mit $Ax \leq b$ und $Z(x) = c^Tx = \min\{c^Tx' \mid x'\}$. Grundsätzlich sind alle vorkommenden Zahlen reelle Zahlen, bei Implementierungen haben wir es im Rechner natürlich immer nur mit rationalen Zahlen zu tun, wir werden diese Schwierigkeit genau wie bei der Besprechung des euklidischen TSP (Abschnitt 9.2) hier nicht problematisieren.

Man könnte versucht sein einzuwenden, dass das Problem doch sehr eingeschränkt ist, insbesondere passt es gar nicht zu der Art und Weise, wie wir es in Algorithmus 10.5 benutzt haben, dort müssen wir nämlich maximieren. Tatsächlich ist das Problem aber viel allgemeiner, als es auf den ersten Blick wirkt. Wir können Maximierungsprobleme lösen, indem wir aus einer zu maximierenden Zielfunktion Z' eine Zielfunktion $Z := -Z'$ machen, die zu minimieren ist. Wir können eine Gleichung $a_i \cdot x = b_i$ durch die zwei Ungleichungen $a_i \cdot x \leq b_i$ und $a_i \cdot x \geq b_i$ ersetzen. Wir können eine Ungleichung $a'_i \cdot x \geq b'_i$ durch Multiplikation mit -1 in die Ungleichung $a_i \cdot x \leq b_i$ mit $a_i := -a'_i$ und $b_i := -b'_i$ überführen, die passende Form hat. Schließlich können wir eine Variable x_i, für die wir uns $x_i \in \mathbb{R}$ wünschen (die also im Gegensatz zu anderen Variablen auch negative Werte annehmen darf), durch zwei „normale“ Variable x'_i und x''_i mit $x'_i, x''_i \in \mathbb{R}_0^+$ ersetzen, wenn wir zusätzlich die Gleichung $x_i = x'_i - x''_i$ einfügen. Übrigens können wir auch aus Ungleichungen Gleichungen machen, in dem wir für jede Ungleichung

$$a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,n}x_n \leq b_i \tag{9}$$

eine neue Variable u_i, die wir *Schlupfvariable* nennen, einführen und die Ungleichung (9) durch die Gleichung (10)

$$a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,n}x_n + 1 \cdot u_i = b_i \tag{10}$$

ersetzen. Weil für die neue Variable auch $u_i \geq 0$ gilt, haben wir offensichtlich Äquivalenz.
Wir sehen, dass es keine Rolle spielt, ob wir lieber mit Gleichungen oder Ungleichungen arbeiten möchten und dass viele der genannten Einschränkungen nicht wesentlich sind. Wir haben allerdings nichts dazu gesagt, wie wir mit echten Ungleichungen umgehen können. Das liegt darin, dass echte Ungleichungen in gewisser Weise „unnatürlich“ und „hässlich“ sind. Wir begründen unsere Abneigung mit einem kleinen Beispiel, das nur eine Variable $x_1 \in \mathbb{R}_0^+$ und eine echte Ungleichung enthält. Es ist $Z(x_1) = -1 \cdot x_1$ zu minimieren, dabei ist wie immer $x_1 \geq 0$, außerdem ist die Nebenbedingung $x_1 < 1$ zu beachten. Wir sehen sofort, dass es hier keine Lösung gibt, der Grenzwert $x_1 = 1$ ist durch die echte Ungleichung gerade ausgeschlossen. Wir werden also ohne echte Ungleichungen auskommen müssen.
Wenn wir uns eine Gleichung

$$a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,n}x_n = b_i$$

ansehen und geometrisch interpretieren, so erkennen wir eine $(n-1)$-dimensionale Hyperebene im $\mathbb{R}^n$, im $\mathbb{R}^2$ ist das eine Gerade, im $\mathbb{R}^3$ eine Ebene.

Wenn wir uns eine Nebenbedingung

$$a_{i,1}x_1 + a_{i,2}x_2 + \cdots + a_{i,n}x_n \leq b_i$$

ansehen und geometrisch interpretieren, so erkennen wir einen Halbraum „unter“ der Hyperebene. Wenn wir die Menge der *zulässigen* Lösungen mit M bezeichnen, so erhalten wir M als Schnittmenge all dieser Halbräume. Wir können bezüglich dieser Menge der zulässigen Lösungen vier Fälle unterscheiden: Wenn $M = \emptyset$ gilt, so gibt es keine zulässige Lösung, wir sprechen von einem unzulässigen Problem. Ist $M \neq \emptyset$, so kann M beschränkt sein, dann gibt es eine Lösung und der Wert dieser Lösung ist eindeutig. Ist $M \neq \emptyset$ aber unbeschränkt, so kann Z in M trotzdem beschränkt sein, in dem Fall gibt es eine Lösung der Wert dieser Lösung ist eindeutig. Schließlich kann $M \neq \emptyset$ unbeschränkt sein und Z ist nach unten unbeschränkt in M (also $Z(x) \rightsquigarrow -\infty$ in M), dann gibt es keine Lösung.
Wenn wir uns die Situation im zwei- oder dreidimensionalen Raum vorstellen ist klar, dass die Menge zulässiger Lösungen durch Geraden oder Ebenen abgetrennt ist und darum Ecken und Kanten hat. Weil die Zielfunktion Z linear ist, muss eine optimale Lösung auf einer Ecke oder Kante liegen, wir können uns darauf beschränken, Ecken zu betrachten. Wir werden uns später überlegen, dass diese Intuition über den dreidimensionalen Raum hinaus auch in den $\mathbb{R}^n$ trägt. Ein naheliegender Lösungsansatz besteht darin, sich einfach alle Ecken anzusehen, den Funktionswert zu bestimmen und eine beste Ecke als Lösung auszugeben. Wir werden später sehen, dass man so tatsächlich optimale Lösungen findet, wenn es denn eine gibt. Wie viele Ecken kann es denn geben? Ein Punkt im $\mathbb{R}^n$ ist eindeutig bestimmt durch den Schnitt von n Hyperebenen. Wir haben insgesamt $n + m$ Bedingungen (die m Nebenbedingungen und die Bedingung $x_i \geq 0$ für jede Variable), aus diesen können wir also n auswählen. Es kann also bis zu $\binom{n+m}{n}$ verschiedene Ecken geben. Wenn wir diesen Ansatz verfolgen wollen, sollten wir uns bemühen, die Ecken geschickt auszusuchen um nicht immer alle Ecken betrachten zu müssen. Eine Möglichkeit, das zu machen, implementiert der Simplex-Algorithmus.

14.1 Der Simplex-Algorithmus

Man kann den Simplex-Algorithmus als einfache lokale Suche beschreiben: Wir starten in einer Ecke der Lösungsmenge M, zu der wir den Funktionswert berechnen. Wir betrachten die Nachbarecken und wählen eine mit besserem Funktionswert. Das wird so lange iteriert, bis ein lokales Minimum gefunden wird, das wird dann als Lösung ausgegeben. Wir werden in diesem

Abschnitt das notwendige Vokabular einführen und die notwendigen Hintergründe besprechen, um genau zu verstehen, wie man das macht, und auch um einzusehen, dass die gefundene lokal optimale Lösung auch tatsächlich eine global optimale Lösung darstellt. Wir beginnen mit einigen grundlegenden Definitionen.

Definition 14.1.

- *Für* $x = (x_1, x_2, \ldots, x_n) \in \mathbb{R}^n$ *heißt* $|x| := \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$ Länge *von* x*.*
- *Für* $x_1, x_2 \in \mathbb{R}^n$ *heißt* $|x_1 - x_2|$ Abstand *von* x_1 *und* x_2*.*
- *Für* $m \in \mathbb{R}^n$ *und* $r \in \mathbb{R}_0^+$ *heißt die Menge* $K := \{x \in \mathbb{R}^n \mid |x - m| \leq r\}$ Kugel *um* Mittelpunkt m *mit* Radius r*.*
- *Für* $x \in \mathbb{R}$ *und* $y \in \mathbb{R}^n \setminus \{0\}$ *heißt die Menge* $g := \{x + \lambda y \mid \lambda \in \mathbb{R}\}$ Gerade *mit* Richtung y*.*
- *Eine Menge* $M \subseteq \mathbb{R}^n$ *heißt* beschränkt*, wenn* $\exists c \in \mathbb{R} \colon \forall x \in M \colon\ |x| \leq c$ *gilt.*
- *Eine Menge* $M \subseteq \mathbb{R}^n$ *heißt* abgeschlossen*, wenn* $\lim\limits_{i \to \infty} x_i \in M$ *für alle konvergenten Folgen* $(x_i \in M)_{i \geq 1}$ *gilt.*
- *Eine Menge* $M \subseteq \mathbb{R}^n$ *heißt* kompakt*, wenn* M *beschränkt und abgeschlossen ist.*

Die in Definition 14.1 eingeführten Begriffe sind vermutlich für niemanden neu und stimmen auch mit unseren intuitiven Vorstellungen überein. Wir fahren fort, indem wir mit diesen Begriffen umgehen und sie so mit Leben füllen. Dabei werden wir neue Begriffe einführen und die Begriffe in Beziehung setzen.

Lemma 14.2. *Sei* $g = \{x + \lambda y \mid \lambda \in \mathbb{R}\}$ *eine Gerade. Für alle Geradenpunkte* $g(\lambda) = x + \lambda y$ *gilt eine der beiden folgenden Aussagen.*

- $\forall \lambda' > \lambda \colon\ |g(\lambda')| > |g(\lambda)|$
- $\forall \lambda' < \lambda \colon\ |g(\lambda')| > |g(\lambda)|$

Beweis. Es gilt

$$(|g(\lambda')| > |g(\lambda)|) \Leftrightarrow \left(|g(\lambda')|^2 > |g(\lambda)|^2\right),$$

und gemäß Definition der Länge ist

$$|g(\lambda')|^2 = \sum_{i=1}^{n} (x_i + \lambda' y_i)^2$$
$$= \underbrace{\left(\sum_{i=1}^{n} y_i^2\right)}_{=:c_1} \cdot \lambda'^2 + \underbrace{\left(\sum_{i=1}^{n} 2x_i y_i\right)}_{=:c_2} \cdot \lambda' + \underbrace{\left(\sum_{i=1}^{n} x_i^2\right)}_{=:c_3}.$$

Gemäß Definition einer Geraden ist $y \neq 0$, also ist $c_1 > 0$. Wenn wir $|g(\lambda')|^2$ als Funktion von λ' auffassen, sehen wir, dass es sich um eine nach oben geöffnete Parabel, die rechts und links ihres Scheitelpunkts monoton ist, so dass eine der beiden Ungleichungen gelten muss. □

Definition 14.3. *Für $x, y \in \mathbb{R}^n$ ist $\{\lambda x + (1-\lambda)y \mid 0 \leq \lambda \leq 1\}$ die* Verbindungsstrecke *zwischen x und y.*
Eine Menge $M \subseteq \mathbb{R}^n$ heißt konvex*, wenn*

$$\forall x, y \in M\colon \{\lambda x + (1-\lambda)y \mid 0 \leq \lambda \leq 1\} \subseteq M$$

gilt.

Man macht sich leicht klar, dass man aus konvexen Mengen neue konvexe Mengen gewinnt, indem man sie schneidet. Konvexität und diese Einsicht scheinen hier im Moment noch etwas abstrakt zu sein, wir ändern das im Anschluss.

Lemma 14.4. *Wenn $M_1, M_2, \ldots, M_l$ konvexe Mengen sind, so ist auch $M := \bigcup_{i=1}^{l} M_i$ konvex.*

Beweis. Für alle $x, y \in M$ gilt $x, y \in M_i$ für alle $i \in \{1, 2, \ldots, l\}$. Weil alle M_i konvex sind, ist die Verbindungsstrecke zwischen x und y in allen M_i. Also ist die Verbindungsstrecke auch in M. □

Wir erinnern uns daran, dass alle Nebenbedingungen die Form $ax \leq b$ hatten mit $a, x \in \mathbb{R}^n$ und $b \in \mathbb{R}$. Eine solche Nebenbedingung definiert einen Halbraum (die Menge der Punkte unter der Hyperebene, die der Gleichung entspricht), diese Halbräume definieren unsere Menge zulässiger Lösungen. Es ist darum wichtig zu beobachten, dass solche Halbräume konvex sind.

Lemma 14.5. *Sei $a \in \mathbb{R}^n$, $b \in \mathbb{R}$. Der Halbraum $H := \{x \mid ax \leq b\}$ ist konvex.*

Beweis. Wir betrachten $x, y \in H$ und haben $ax \leq b$ und $ay \leq b$ gemäß Definition von H. Punkte z auf der Verbindungsstrecke zwischen x und y haben die Form $z = \lambda x + (1-\lambda)y$ mit $\lambda \in [0;1]$. Wir rechnen

$$a \cdot z \;=\; a \cdot (\lambda x + (1-\lambda)x) \;=\; \lambda ax + (1-\lambda)ay \;\leq\; \lambda b + (1-\lambda)b \;=\; b$$

und sehen, dass $z \in H$ gilt. □

Aus Lemma 14.5 und Lemma 14.4 folgt nun unmittelbar, dass die Menge der zulässigen Lösungen konvex ist. Weil die Aussage wichtig ist, nennen wir sie Theorem, Korollar wäre vielleicht angemessener.

Theorem 14.6. *Die Menge der zulässigen Lösungen eines Problems der linearen Optimierung ist der Durchschnitt von $n+m$ abgeschlossenen Halbräumen, konvex und abgeschlossen.*

Natürlich sind nicht alle Mengen konvex. Für nichtkonvexe Mengen wünschen wir uns eine Einbettung in eine möglichst kleine konvexe Menge. Man spricht von der konvexen Hülle.

Definition 14.7. *Die* konvexe Hülle *einer Menge $M \subseteq \mathbb{R}^n$ ist $KoHü(M) := \bigcap_{S \supseteq M, S \text{ konvex}} S$.*

Der Begriff der Linearkombination ist wahrscheinlich bekannt. Wir können konvexe Linearkombinationen benutzen, um konvexe Mengen zu beschreiben.

Definition 14.8. *Ein Punkt $x \in \mathbb{R}^n$ ist* konvexe Linearkombination *(kL) von $x_1, x_2, \ldots, x_p \in \mathbb{R}^n$, wenn es $\lambda_1, \lambda_2, \ldots, \lambda_p \geq 0$ gibt, so dass $\sum_{i=1}^{p} \lambda_i = 1$ und $x = \sum_{i=1}^{p} \lambda_i x_i$ gilt.*

Theorem 14.9. *$\forall M \subseteq \mathbb{R}^n$: $KoHü(M) = \{x \mid x$ ist kL von Punkten aus $M\}$.*

Beweis. Wir definieren $S := \{x \mid x \text{ ist kL von Punkten aus } M\}$ und zeigen zunächst, dass $S \subseteq \text{KoHü}(M)$ gilt. Für jeden Punkt $x \in S$ gilt gemäß Definition von x, dass es $x_1, x_2, \ldots, x_p \in M$ gibt und dazu passende $\lambda_1, \lambda_2, \ldots, \lambda_p \in [0;1]$, so dass $x = \sum_{i=1}^{p} \lambda_i x_i$ gilt. Für wollen zeigen, dass $x \in \text{KoHü}(M)$ gilt und führen den Beweis mit vollständiger Induktion über p. Für $p = 1$ ist $x = 1 \cdot x_1 = x_1 \in M \subseteq \text{KoHü}(M)$, also ist der Induktionsanfang gesichert. Für den Induktionsschritt betrachten wir $x = \sum_{i=1}^{p} \lambda_i x_i$.

Wenn $\lambda_p = 1$ ist, sind alle anderen $\lambda_i = 0$ und $x \in \text{KoHü}(M)$ folgt wie beim Induktionsanfang. Sei also $\lambda_p < 1$. Dann können wir auch

$$x = (1-\lambda_p) \cdot \underbrace{\left(\frac{\lambda_1}{1-\lambda_p} x_1 + \frac{\lambda_2}{1-\lambda_p} x_2 + \cdots + \frac{\lambda_{p-1}}{1-\lambda_p} x_{p-1} \right)}_{=:x'} + \lambda_p x_p$$

schreiben und haben dank Induktionsvoraussetzung, dass $x' \in \text{KoHü}(M)$ gilt. Natürlich ist $\text{KoHü}(M)$ konvex, also liegt x, das auf der Verbindungsstrecke von $x' \in \text{KoHü}(M)$ und $x_p \in \text{KoHü}(M)$ liegt, ebenfalls $\in \text{KoHü}(M)$.
Nun müssen wir noch zeigen, das $\text{KoHü}(M) \subseteq S$ gilt. Für alle $x \in M$ ist natürlich $x = 1 \cdot x$, also eine konvexe Linearkombination von sich selbst, somit ist zumindest $M \subseteq S$. Wenn wir zeigen können, dass S konvex ist, so folgt, dass auch $\text{KoHü}(M) \subseteq S$ ist. Wir betrachten $y, z \in S$. Gemäß Definition von S können wir $y = \sum_{i=1}^{p} \lambda_i x_i$ und $z = \sum_{i=1}^{q} \lambda'_i x'_i$ schreiben. Wir wollen zeigen, dass alle Punkte auf der Verbindungsstrecke zwischen y und z zu S gehören, also dass für alle $\lambda \in [0;1]$ stets $\lambda y + (1-\lambda)z \in S$ gilt. Wir können

$$\begin{aligned} &\lambda y + (1-\lambda) z \\ &= \lambda\lambda_1 x_1 + \lambda\lambda_2 x_2 + \cdots + \lambda\lambda_p x_p + (1-\lambda)\lambda'_1 x'_1 + (1-\lambda)\lambda'_2 x'_2 + \cdots + (1-\lambda)\lambda'_q x'_q \end{aligned}$$

schreiben und erinnern daran, dass $\lambda, \lambda_i, \lambda'_j \in [0;1]$ gilt. Folglich sind auch $\lambda\lambda_i \in [0;1]$ ebenso wie $(1-\lambda)\lambda'_j \in [0;1]$. Außerdem gilt

$$\left(\sum_{i=1}^{p} \lambda\lambda_i \right) + \left(\sum_{j=1}^{q} (1-\lambda)\lambda'_j \right) = \lambda \left(\sum_{i=1}^{p} \lambda_i \right) + (1-\lambda) \left(\sum_{i=1}^{q} \lambda'_j \right) = \lambda + (1-\lambda) = 1$$

und die Behauptung ist gezeigt. □

Als wir über Beispiele für Instanzen im zwei- und dreidimensionalen Raum diskutiert haben, hatten wir uns überlegt, dass wir Optima ausschließlich in Ecken der Menge der zulässigen Lösungen erwarten. Intuition, die auf niedrigdimensionalen Räumen beruht, kann trügerisch sein und sich einer Verallgemeinerung auf n Dimensionen widersetzen. Wir werden darum einen formalen Beweis führen müssen, der unsere Intuition verifiziert. Dazu werden wir formalisieren, was wir eigentlich mit „Ecke“ meinen und was wir über die Lage aussagen können.

Definition 14.10. *Sei $M \subseteq \mathbb{R}^n$ konvex, sei $x \in M$. Der Punkt x heißt* Randpunkt, *wenn für alle $\varepsilon > 0$ die ε-Kugel um x nicht ganz in M liegt. Der Punkt x heißt* Extrempunkt, *wenn für alle $y \neq 0$ stets $x + y \notin M$ oder $x - y \notin M$ gilt.*

Lemma 14.11. *Sei $M \subseteq \mathbb{R}^n$ konvex, sei $x \in M$ ein Extrempunkt von M. Dann ist x auch Randpunkt von M.*

Beweis. Wir führen einen Widerspruchsbeweis und nehmen an, dass $x \in M$ Extrempunkt, aber nicht Randpunkt ist. Weil x nicht Randpunkt ist, gibt es ein $\varepsilon > 0$, so dass die ε-Kugel um x ganz in M ist. Wir betrachten dieses ε und ein $y \neq 0$ mit $|y| < \varepsilon$. Offenbar liegen $x+y$ und $x-y$ in der ε-Kugel um x, also ist $\{x+y, x-y\} \subseteq M$ und x ist *kein* Extrempunkt im Widerspruch zur Voraussetzung. □

Lemma 14.12. *Seien $M_1, M_2, \ldots, M_t$ konvex, sei $M := \bigcup_{i=1}^{t} M_i$. Ist x Randpunkt von M, so gibt es ein i, so dass x Randpunkt von M_i ist.*

Beweis. Wir führen einen Widerspruchsbeweis und nehmen an, dass x Randpunkt von M, aber für alle M_i nicht Randpunkt ist. Es gibt dann also $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_t > 0$, so dass die ε_i-Kugel um x ganz in M_i enthalten ist für alle $i \in \{1, 2, \ldots, t\}$. Wir definieren $\varepsilon := \min\{\varepsilon_i \mid i \in \{1, 2, \ldots, t\}\}$ und haben eine ε-Kugel um x, die ganz in M enthalten ist. Damit ist x kein Randpunkt von M im Widerspruch zur Voraussetzung. □

Lemma 14.13. *Sei $M \subseteq \mathbb{R}^n$ konvex, sei $K \subseteq \mathbb{R}^n$ eine Kugel. Jeder Extrempunkt von $M \cap K$ ist Extrempunkt von M oder Extrempunkt von K.*

Beweis. Wir beobachten, dass K konvex ist. Sei $x \in \mathbb{R}^n$ Extrempunkt von $M \cap K$. Aus Lemma 14.11 folgt, dass x auch Randpunkt von $M \cap K$ ist. Aus Lemma 14.12 folgt, dass x Randpunkt von M oder Randpunkt von K ist. Für die Kugel K gilt offenbar, dass jeder Randpunkt auch Extrempunkt (und wegen Lemma 14.11 sowieso auch umgekehrt) ist. Falls x also Randpunkt von K ist, ist nichts mehr zu zeigen. Sei also x ein Randpunkt von M, der nicht Randpunkt von K ist. Wir führen einen Widerspruchsbeweis und nehmen an, dass x kein Extrempunkt von M ist. Dann gibt es gemäß Definition 14.10 ein $y \neq 0$, so dass $\{x+y, x-y\} \subseteq M$ gilt. Wir erinnern uns daran, dass x kein Randpunkt von K ist, es gibt also ein $\varepsilon > 0$, so dass eine ε-Kugel $B(\varepsilon)$ um x ganz in K liegt. Wir betrachten den Schnitt von $B(\varepsilon)$ mit der Verbindungsstrecke von $x+y$ und $x-y$, das ist offenbar eine Verbindungsstrecke von zwei Punkten $x+y'$ und $x-y'$ für ein $y' \neq 0$, die ganz in $M \cap K$ liegt. Folglich ist x kein Extrempunkt von $M \cap K$ im Widerspruch zur Voraussetzung. □

Lemma 14.14. *Sei $M \subseteq \mathbb{R}^n$ nicht leer, konvex und kompakt. Dann existiert $\max\{|x| \mid x \in M\}$ und für jedes $m \in M$ mit $|m| = \max\{|x| \mid x \in M\}$ gilt, dass m Extrempunkt von M ist.*

Beweis. Die Existenz von $\max\{|x| \mid x \in M\}$ ist gesichert, weil $|\cdot|$ stetig ist und M kompakt. Wir betrachten ein $m \in M$ mit $|m| = \max\{|x| \mid x \in M\}$ und nehmen für einen Widerspruchsbeweis an, dass m kein Extrempunkt von M ist. Dann ist $\{m - y, m + y\} \subseteq M$ für ein $y \neq 0$. Lemma 14.2 impliziert, dass $|m + y| > \max\{|x| \mid x \in M\}$ oder $|m - y| > \max\{|x| \mid x \in M\}$ gilt, ein Widerspruch. □

Wir erinnern uns daran, dass eine Nebenbedingung eine $(n-1)$-dimensionale Hyperebene definiert und die Lösungsmenge „unter" dieser Hyperebene liegt. Wir definieren formal, was wir damit meinen und setzen Hyperebenen mit konvexen Mengen in Beziehung.

Definition 14.15. *Eine Menge $M \subseteq \mathbb{R}^n$ liegt ganz auf einer Seite der durch $u \in \mathbb{R}^n$, $c \in \mathbb{R}$ definierten Hyperebene $H = \{y \in \mathbb{R}^n \mid u \cdot y = c\}$, wenn $\forall m \in M : u \cdot m \leq c$ oder $\forall m \in M : u \cdot m \geq c$ gilt.*

Lemma 14.16. *Sei $M \subseteq \mathbb{R}^n$ eine nichtleere, konvexe und abgeschlossene Menge, sei $z \in \mathbb{R}^n \backslash M$. Es gibt ein $z^* \in M$ mit $|z - z^*| = \min\{|z - x| \mid x \in M\}$. Für die Hyperebene $H := \left\{y \in \mathbb{R}^n \mid \frac{z^*-z}{|z^*-z|} \cdot y = \frac{z^*-z}{|z^*-z|} \cdot z^*\right\}$ gilt*

$$\frac{z^* - z}{|z^* - z|} \cdot m \geq \frac{z^* - z}{|z^* - z|} \cdot z^*$$

für alle $m \in M$.

Beweis. Die Existenz von z^* ist gesichert, weil $|z - x|$ auf M stetig ist und durch 0 nach unten beschränkt. Wir definieren $u := \frac{z^*-z}{|z^*-z|}$, natürlich gilt $|u| = 1$. Es ist $u \cdot (z^* - z) = u^2 > 0$, also ist $u \cdot z < u \cdot z^*$. Wir müssen zeigen, dass für alle $m \in M$ stets $u \cdot m \geq u \cdot z^*$ gilt. Dazu betrachten wir ein $m \in M$ und dazu einen Punkt auf der Verbindungsstrecke zwischen m und z^*, also $w_\lambda := \lambda m + (1 - \lambda)z^*$ mit $\lambda \in [0; 1]$. Weil M konvex ist, ist $w_\lambda \in M$. Wir betrachten $f(\lambda) := |w_\lambda - z|^2$ und rechnen nach, dass

$$\begin{aligned} f(\lambda) = (w_\lambda - z)^2 = (\lambda m + (1 - \lambda)z^* - z)^2 = (\lambda(m - z^*) + (z^* - z))^2 \\ = \lambda^2 (m - z^*)^2 + 2\lambda (m - z^*) + (z^* - z)^2 \end{aligned}$$

gilt. Wir betrachten die erste Ableitung von f

$$f'(\lambda) = 2\lambda (m - z^*)^2 + 2 (m - z^*) (z^* - z)$$

und sehen, dass $f'(0) = 2 (m - z^*) (z^* - z) = d \cdot u \cdot (m - z^*)$ für ein $d > 0$ gilt. Wir sehen, dass $f'(0) \geq 0$ gilt. Wäre nämlich $f'(0) < 0$, so gäbe es ein $\lambda_0 > 0$ mit $f(\lambda_0) < f(0)$, also wäre $|z - w_{\lambda_0}| < |z - z^*|$, ein Widerspruch zur Wahl von z^*. Aus $f'(0) \geq 0$ folgt unmittelbar $u \cdot (m - z^*) \geq 0$, also ist $u \cdot m \geq u \cdot z^*$ wie behauptet. □

Wir halten noch eine Aussage fest, die Lemma 14.16 sehr ähnlich ist, jetzt betrachten wir aber nicht einen Punkt außerhalb von M, sondern einen Randpunkt von M.

Lemma 14.17. *Sei $M \subseteq \mathbb{R}^n$ konvex und abgeschlossen, sei m_{Rand} ein Randpunkt von M. Es gibt eine Hyperebene $H = \{y \in \{0,1\}^n \mid u \cdot y = u \cdot m_{Rand}\}$ durch m_{Rand}, so dass $u \cdot m \geq u \cdot m_{Rand}$ für alle $m \in M$ gilt.*

Beweis. Wir betrachten eine konvergente Folge $(z_i)_{i\geq 1}$, mit $z_i \notin M$ für alle $i \geq 1$ und $\lim_{i\to\infty} z_i = m_{\text{Rand}}$. Eine solche Folge gibt es, weil m_{Rand} ein Randpunkt ist; es gibt darum eine ε-Kugel um m_{Rand}, die nicht ganz in M liegt, in dem Teil außerhalb von M kann $(z_i)_{i\geq 1}$ definiert werden. Weil jedes z_i nicht zu M gehört, können wir für jedes z_i Lemma 14.16 anwenden und erhalten jeweils eine durch u_i beschriebene Hyperebene H_i. Wir haben $|u_i| = 1$ für alle i, die u_i sind also eine beschränkte Folge, so dass es eine konvergente Teilfolge $(u_{i(j)})_{j\geq 1}$ gibt, sie konvergiere gegen u. Weil $(z_i)_{i\geq 1}$ gegen m_{Rand} konvergiert, gilt das auch für die Teilfolge $(z_{i(j)})_{j\geq 1}$. Wir haben $u_{i(j)} \cdot m \geq u_{i(j)} \cdot z^*_{i(j)}$ für alle $m \in M$, also gilt $u \cdot m \geq u \cdot m_{\text{Rand}}$ für alle $m \in M$. Wir sehen, dass $H := \{y \in \mathbb{R}^n \mid u \cdot y = u \cdot m_{\text{Rand}}\}$ eine Hyperebene mit passenden Eigenschaften ist. □

Theorem 14.18 (**Theorem von Krein und Milman**). *Sei $M \subseteq \mathbb{R}^n$ konvex und kompakt. Dann gilt für alle $x \in M$, dass x als konvexe Linearkombination von höchstens $n+1$ Extrempunkten von M darstellbar ist.*

Beweis. Für $M = \emptyset$ ist nichts zu zeigen, wir können darum zusätzlich voraussetzen, dass M nicht leer ist. Wir führen einen Induktionsbeweis über die Dimension n. Für $n = 1$ ist die Aussage trivial, da dann $M = [a; b]$ gilt mit $a \leq b \in \mathbb{R}$. Der Induktionsanfang ist also gesichert. Für $n > 1$ wissen wir, dass M jedenfalls einen Extrempunkt x_0 hat (Lemma 14.14). Ist $x = x_0$, so ist die Aussage wieder trivial, wir setzen darum im Folgenden zusätzlich $x \neq x_0$ voraus. Wir betrachten die Gerade g durch x und x_0, natürlich ist eine Gerade selbst auch abgeschlossen und konvex. Wir betrachten $g \cap M$, eine konvexe, kompakte und eindimensionale Menge. Wir erkennen, dass $g \cap M$ die Verbindungsstrecke ist zwischen x_0 und einem Punkt x', der Randpunkt von M ist.
Wir betrachten die Hyperebene $H := \{y \in \{0,1\}^n \mid u \cdot y = u \cdot x'\}$, offenbar ist $x' \in H$ und M liegt ganz auf einer Seite von H, es gilt also $u \cdot m \geq u \cdot x'$ für alle $m \in M$. Die Hyperebene H ist $n-1$-dimensional, konvex und abgeschlossen, also ist $H \cap M$ kompakt, konvex und ebenfalls $n-1$-dimensional. Wir können die Induktionsvoraussetzung anwenden und folgern, dass x' darstellbar ist als konvexe Linearkombination von höchstens n Extrempunkten von

$H \cap M$. Damit uns das etwas nützt, weisen wir jetzt nach, dass Extrempunkte von $H \cap M$ auch Extrempunkte von M sind. Dazu betrachten wir einen Extrempunkt von $H \cap M$, sei das $z \in H \cap M$. Für einen Widerspruchsbeweis nehmen wir an, dass z kein Extrempunkt von M ist. Dann gibt es ein $y \neq 0$, so dass $\{z + y, z - y\} \subseteq M$ gilt. Folglich gilt dann $u \cdot (z + y) \geq u \cdot x'$ und $u \cdot (z - y) \geq u \cdot x'$. Nehmen wir verschärfend an, dass sogar $u \cdot (z + y) > u \cdot x'$ gilt. Dann gilt $u \cdot z = (1/2) \cdot u \cdot (z + y + z - y) > (1/2) \cdot (u \cdot x' + u \cdot x') = u \cdot x'$. Wir schließen, dass $z \notin H$ gilt, ein Widerspruch zur Voraussetzung. Also muss die Annahme falsch gewesen sein, wir können also folgern, dass $u \cdot (z + y) = u \cdot x'$ gilt. Auf die gleiche Art folgt, dass $u \cdot (z - y) = u \cdot x'$ gilt. Damit ist also $\{z + y, z - y\} \subseteq H \cap M$, folglich ist z kein Extrempunkt von $H \cap M$ und wir haben einen Widerspruch zur Voraussetzung. Insgesamt haben wir damit gezeigt, dass Extrempunkte von $H \cap M$ auch Extrempunkte von M sind.

Wir haben jetzt also $x = \lambda x' + (1 - \lambda)x_0$, dabei ist $x' = \sum_{i=1}^{p} \lambda_i x_i$ eine konvexe Linearkombination von $p \leq n$ Extrempunkten von M. Damit ist x eine konvexe Linearkombination von $\leq n + 1$ Extrempunkten von M wie behauptet. □

Nach so viel Vorarbeit können wir endlich die Früchte der Arbeit ernten: Wir können Extrempunkte der Menge zulässiger Lösungen lokalisieren und nachweisen, dass wir Minima der Zielfunktion tatsächlich dort finden.

Theorem 14.19. *Sei M die Menge der zulässigen Lösungen eines linearen Optimierungsproblems. Wenn $M \neq \emptyset$ gilt, so hat M mindestens einen Extrempunkt, außerdem ist dann jedes $x \in M$ darstellbar als konvexe Linearkombination $x = \sum_{i=1}^{p} \lambda_i x_i$ mit $x_i \in M$ für alle $i \in \{1, 2, \ldots, p\}$, dabei ist x_1 ein Extrempunkt von M und $\lambda_1 > 0$.*

Beweis. Natürlich folgt der erste Teil der Aussage aus dem zweiten Teil, wir müssen also nur den zweiten Teil der Aussage beweisen. Für den zweiten Teil führen wir eine Fallunterscheidung durch. Wir wissen, dass M konvex und abgeschlossen ist (Theorem 14.6). Falls M beschränkt ist, so ist M kompakt und die Behauptung folgt direkt aus dem Theorem von Krein und Milman (Theorem 14.18). Man kann dann x sogar als konvexe Linearkombination von Punkten darstellen, die alle Extrempunkte sind. Es bleibt also nur der Fall, dass M unbeschränkt ist. Wenn $x = 0$ ist, so ist x selbst ein Extrempunkt, die Nebenbedingungen $x_i \geq 0$ für alle $i \in \{1, 2, \ldots, n\}$ stellen das sicher. Wir dürfen also voraussetzen, dass $x \in M \setminus \{0\}$ gilt. Wir betrachten eine Kugel K um 0 mit Radius $(n + 2) \cdot |x|$ und den Schnitt $M \cap K$. Natürlich ist der Schnitt $M \cap K$ konvex und kompakt, wir können also das Theorem von

Krein und Milman (Theorem 14.18) anwenden und sehen, dass $x = \sum_{i=1}^{p} \lambda_i x_i$ gilt für $p \leq n+1$ Extrempunkte von $M \cap K$. Wir dürfen annehmen, dass alle λ_i echt positiv sind, durch Verkleinerung von p kann das erreicht werden. Wir haben $p > 0$, weil wir $x \neq 0$ voraussetzen. Wir wissen (Lemma 14.13), dass alle x_i Extrempunkte von M oder K sind. Ist mindestens ein x_i Extrempunkt von M, so ist der Beweis schon geführt. Nehmen wir also an, dass alle x_i nur Extrempunkte von K sind. Dann gilt für alle x_i natürlich $|x_i| = (n+2) \cdot |x|$, weil alle x_i Punkte der Kugeloberfläche sind. Wir haben $x = \sum_{i=1}^{p} \lambda_i x_i$, also gilt auch $|x| = \left| \sum_{i=1}^{p} \lambda_i x_i \right|$. Wir haben $\lambda_i > 0$ für alle $i \in \{1, 2, \ldots, n\}$ und auch $x_i \geq 0$ für alle i, weil alle x_i nicht nur zu K, sondern auch zu M gehören. Daraus folgt $|x| = \left| \sum_{i=1}^{p} \lambda_i x_i \right| \geq |\lambda_j x_j| = \lambda_j \, |x_j| = \lambda_j \cdot (n+2) \cdot |x|$ für alle j. Wir wählen das $j \in \{1, 2, \ldots, n\}$ mit maximalem λ_j. Weil $\sum_{i=1}^{n} \lambda_i = 1$ gilt, ist $\lambda_j \geq 1/p$ (Schubfachprinzip). Wir erinnern uns, dass $p \leq n+1$ gilt und folgern, dass $\lambda_j \geq 1/(n+1)$ gilt. Wir setzen das ein und erhalten $|x| \geq \lambda_j \cdot (n+2) \cdot |x| \geq |x| \cdot (n+2)/(n+1) > |x|$, einen Widerspruch. Folglich muss mindestens ein x_i Extrempunkt von M sein und die Behauptung folgt durch Umnummerierung. □

Theorem 14.20. *Sei M die Menge der zulässigen Lösungen eines linearen Optimierungsproblems mit Zielfunktion Z. Wenn Z auf M ein Minimum annimmt, dann nimmt Z das Minimum in einem Extrempunkt von M an.*

Beweis. Wir setzen für ein $x \in M$ voraus, dass $Z(x) = \min \{Z(x') \mid x' \in M\}$ gilt. Aus Theorem 14.19 folgt, dass $x = \sum_{i=1}^{p} \lambda_i x_i$ gilt mit $\lambda_i > 0$ für alle i und einem Extrempunkt x_1 von M, außerdem ist $x_i \in M$ für alle $i \in \{1, 2, \ldots, p\}$. Weil die Zielfunktion Z linear ist, können wir $Z(x) = Z\left(\sum_{i=1}^{p} \lambda_i x_i\right) = \sum_{i=1}^{p} \lambda_i Z(x_i)$ schreiben. Weil $Z(x)$ minimal ist, haben wir $Z(x_i) \geq Z(x)$ für alle x_i und folgern, dass $Z(x) \geq \sum_{i=1}^{p} \lambda_i Z(x) = Z(x) \cdot \sum_{i=1}^{p} \lambda_i = Z(x)$ gilt. Weil ganz links und ganz rechts in der Ungleichungskette $Z(x)$ steht, haben wir überall Gleichheit, es gilt also $\sum_{i=1}^{p} \lambda_i Z(x_i) = \sum_{i=1}^{p} \lambda_i Z(x)$. Wir dürfen annehmen, dass alle λ_i echt positiv sind, man kann p verkleinern, um das zu erreichen. Damit haben wir also $Z(x_i) = Z(x)$ für alle x_i gezeigt, insbesondere also auch für x_1, einen Extrempunkt von M. □

Wir wissen jetzt, dass es genügt, sich alle Extrempunkte der Menge zulässiger Lösungen anzusehen, um ein Minimum der Zielfunktion zu finden, wenn die Zielfunktion auf dieser Menge überhaupt ein Minimum annimmt. Wir haben noch nicht geklärt, wie wir das entscheiden. Aber selbst wenn wir das können, sind wir nicht sehr weit gekommen. Es kann $\binom{n+m}{n}$ Extrempunkte geben, die wollen wir uns sicher nicht alle ansehen. Schön wäre es, einem Extrempunkt ansehen zu können, dass er minimal ist. Das ist in der Tat möglich und der Schlüssel zum Simplex-Verfahren.

Definition 14.21. *Sei eine lineare Zielfunktion Z definiert auf einer Menge M. Wir nennen $m \in M$* lokales Minimum*, wenn es eine ε-Kugel K um m gibt (mit Radius $\varepsilon > 0$), so dass $Z(x) \geq Z(m)$ gilt für alle $x \in M \cap K$.*

Theorem 14.22. *Sei Z Zielfunktion eines linearen Optimierungsproblems mit Menge M der zulässigen Lösungen. Jedes lokale Minimum von Z auf M ist auch globales Minimum.*

Beweis. Wir betrachten ein lokales Minimum $x \in M$ und nehmen an, dass x nicht global minimal ist. Es gibt also ein $x^* \in M$ mit $Z(x^*) < Z(x)$. Weil M konvex ist, ist die Verbindungsstrecke zwischen x und x^* ganz in M. Für ein λ mit $0 < \lambda < 1$ betrachten wir den Funktionswert des entsprechenden Punktes auf dieser Verbindungsstrecke und sehen, dass $Z(\lambda x + (1-\lambda)x^*) = \lambda Z(x) + (1-\lambda)Z(x^*) < Z(x)$ gilt. Das gilt für jedes $\lambda < 1$, es kann also kein $\varepsilon > 0$ geben, so dass in der ε-Kugel um x jeder Punkt durch $Z(x)$ nach unten beschränkten Funktionswert hat. Also ist x im Widerspruch zur Voraussetzung kein lokales Minimum. □

Wir können jetzt das Simplex-Verfahren formulieren, wobei wir allgemein formulieren und viele Details noch offen lassen. Insbesondere klären wir nicht vorab, wie man feststellt, ob Z in M unbeschränkt fällt oder M leer ist. Wir werden uns im Anschluss um diese Fragen kümmern und dann das Simplex-Verfahren zum Simplex-Algorithmus (Algorithmus 14.29) ausbauen.

Simplex-Verfahren

1. Finde zulässigen Extrempunkt x als Schnittpunkt von n Hyperebenen, oder entscheide, dass $M = \emptyset$ gilt.
   If $M = \emptyset$, Then gib „keine Lösung“ aus. STOP.
2. If x lokal optimal, Then gib $Z(x)$ als Lösung aus. STOP.
3. Tausche eine der n Hyperebenen so aus, dass ein neuer zulässiger Extrempunkt x mit kleinerem Z-Wert gefunden wird.
   Teste dabei, ob $Z \to -\infty$ in M gilt.
   If $Z \to -\infty$ in M, Then Ausgabe „$Z \to -\infty$“. STOP.
4. Weiter bei 2.

Wir schauen uns jetzt zwei kleine Beispiele an, die wir mit dem Simplex-Verfahren lösen wollen. Wir werden dabei erkennen, wie wir die noch offenen Punkte im Simplex-Verfahren konkret umsetzen können.
Wir betrachten

$$\begin{aligned} & Z(x) = -3x_1 - 5x_2 \to \min \\ & \begin{array}{rrrrr} -x_1 & + & x_2 & \leq & 2 \\ 2x_1 & - & 3x_2 & \leq & 3 \\ 2x_1 & + & 3x_2 & \leq & 12 \end{array} \\ & x_1 \geq 0,\ x_2 \geq 0 \end{aligned} \quad \text{unter}$$

und sehen, dass das Problem die günstige Eigenschaft hat, dass alle rechten Seiten nichtnegativ sind. Darum ist 0 ein zulässiger Extrempunkt und wir brauchen uns um den ersten Schritt des Simplex-Verfahrens keine Gedanken zu machen. Wir haben $Z(0) = -3 \cdot 0 - 5 \cdot 0 = 0$. Wir wollen das Rechnen mit Ungleichungen vermeiden und formen darum lieber in ein Gleichungssystem um. Dazu können wir wie in der Einleitung besprochen für jede der drei Ungleichungen eine Schlupfvariable einführen. Wir nennen die neuen Variablen u_1, u_2, u_3 und erhalten

$$\begin{aligned} & Z \to \min \\ & Z + 3x_1 + 5x_2 = 0 \\ & \begin{array}{rrrrrrrrrrr} -x_1 & + & x_2 & + & u_1 & & & & = & 2 \\ 2x_1 & - & 3x_2 & & & + & u_2 & & = & 3 \\ 2x_1 & + & 3x_2 & & & & & + \ u_3 & = & 12 \end{array} \\ & x_1 \geq 0,\ x_2 \geq 0,\ u_1 \geq 0,\ u_2 \geq 0,\ u_3 \geq 0 \end{aligned} \quad \text{unter}$$

als Resultat dieser Transformation. Unser zulässiger Extrempunkt ist jetzt $(0, 0, 2, 3, 12)$, der Z-Wert ist natürlich weiterhin 0. Die Variablen u_1, u_2, u_3, die alle mit Faktor 1 in genau einer der Gleichungen auftauchen, heißen *Basisvariablen*, die anderen Variablen heißen *Nichtbasisvariablen*. Wir haben also m Basisvariablen und n Nichtbasisvariablen. Ist in unserem Extrempunkt eine Variable 0 (so wie hier x_1 und x_2), so sind die entsprechenden Ungleichungen exakt erfüllt, der Extrempunkt entspricht also genau dem Schnitt dieser Hyperebenen. Wir wollen jetzt entscheiden, ob unser Extrempunkt lokal optimal ist und falls das nicht der Fall einen Extrempunkt mit kleinerem Z-Wert wählen. Wir machen das, indem wir eine Nichtbasisvariable zur Basisvariablen machen (sie *betritt die Basis*) und eine Basisvariable zur Nichtbasisvariablen machen (sie *verlässt die Basis*). Wir können uns aussuchen, ob wir die Nichtbasisvariable x_1 oder die Nichtbasisvariable x_2 die Basis betreten lassen. Dabei wird der Wert der gewählten Variablen positiv werden, wir erkennen an der Gleichung mit Z, dass das Z verkleinern wird. Wir können also unser Ziel mit beiden Variablen erreichen und entscheiden uns für x_2, weil x_2 in der Z-Gleichung mit Faktor 5 auftaucht, x_1 hingegen nur mit Faktor 3, wir erhoffen uns aus diesem Grund beim Eintritt von x_2 in die

Basis eine stärkere Änderung von Z. Welche Basisvariable soll die Basis verlassen? Auch hier haben wir grundsätzlich die freie Wahl. Wir betrachten die drei Gleichungen und stellen Folgendes fest, wobei wir berücksichtigen, dass $x_1 = 0$ gilt: Aus der ersten Gleichung folgt $x_2 = 2 - u_1 \leq 2$, da alle Variablen nichtnegativ sind. Aus der zweiten Gleichung folgt $x_2 = 1 + u_2/3$ und somit keine Beschränkung für x_2. Aus der dritten Gleichung folgt $x_3 = 4 - u_3/3 \leq 4$. Alle drei Gleichungen müssen gleichzeitig erfüllt sein, es folgt also $x_2 \leq 2$ aus der ersten Gleichung. Wir erreichen $x_2 = 2$ genau dann, wenn $u_1 = 0$ gilt und entscheiden darum, dass u_1 die Basis verlassen soll. Wir erhalten aus der ersten Gleichung $x_2 = 2 + x_1 - u_1$ und setzen das in die zweite und dritte Gleichung sowie die Z-Gleichung ein. Damit haben wir jetzt

$$
\begin{array}{lllllllllll}
 & Z \to \min & & & & & & & & & \\
\text{unter} & Z + 8x_1 - 5u_1 = -10 & & & & & & & & & \\
 & -x_1 & + & u_1 & + & x_2 & & & & = & 2 \\
 & -x_1 & + & 3u_1 & & & + & u_2 & & = & 9 \\
 & 5x_1 & - & 3u_1 & & & & & + \; u_3 & = & 6 \\
 & x_1 \geq 0,\ x_2 \geq 0,\ u_1 \geq 0,\ u_2 \geq 0,\ u_3 \geq 0 & & & & & & & & &
\end{array}
$$

und unser neuer Basispunkt ist $(0, 2, 0, 9, 6)$ mit Z-Wert -10. Wir führen einen weiteren Schritt des Simplex-Verfahrens durch. Weil u_1 in der Z-Gleichung mit negativem Faktor vorkommt, macht eine Vergrößerung von u_1 den Z-Wert größer. Wir sehen ein, dass wir erkennen können, wenn wir eine optimale Lösung gefunden haben: Kommen die Nichtbasisvariablen alle mit nichtpositiven Koeffizienten in der Z-Gleichung vor, kann Z bei einem Basiswechsel nicht weiter sinken, der aktuelle Basispunkt ist also lokal optimal und mit Theorem 14.22 auch global optimal. Hier ist das nicht der Fall, x_1 soll also unsere Basis betreten. Um zu entscheiden, welche Variable die Basis verlassen soll, betrachten wir wieder die drei Gleichungen und erkennen, dass nur die dritte Gleichung mit $x_1 = (6/5) - u_3/5 \leq 6/5$ eine Beschränkung für x_1 liefert. Wir erkennen, dass es grundsätzlich möglich ist, dass gar keine Gleichung die neue Basisvariable beschränkt. In dem Fall können wir diese Variable unbegrenzt vergrößern und dadurch Z unbegrenzt verkleinern. Wir haben also ein Kriterium gefunden, das uns entscheiden lässt, ob Z in M unbeschränkt fällt. Hier ist das nicht der Fall und wir lassen x_1 die Basis betreten und u_3 die Basis verlassen. Wir erhalten aus der dritten Gleichung $x_1 = (6/5) + (3/5)u_1 - u_3/5$ und setzen in die übrigen Gleichungen ein. Damit erhalten wir

$$
\begin{array}{lrcrcrcrcl}
 & Z & & & & & & & & \to \min \\
\text{unter} & Z - (1/5)u_1 & & - (6/5)u_3 & & & & & & = -98/5 \\
 & (2/5)u_1 & + & (1/5)u_3 & + & x_2 & & & & = 16/5 \\
 & (22/5)u_1 & + & (1/5)u_3 & & & + & u_2 & & = 51/5 \\
 & -(3/5)u_1 & + & (1/5)u_3 & & & & & + x_1 & = 6/5 \\
 & x_1 \geq 0, & & x_2 \geq 0, & & u_1 \geq 0, & & u_2 \geq 0, & & u_3 \geq 0
\end{array}
$$

und $(6/5, 16/5, 0, 51/5, 0)$ als neuen Basispunkt mit Z-Wert $-98/5 = -19{,}6$. Weil in der Z-Gleichung die beiden Nichtbasisvariablen u_1 und u_3 beide mit negativem Koeffizienten vorkommen, ist unsere Lösung optimal. Für unser ursprüngliches Problem haben wir also $x_1 = 6/5$ und $x_2 = 16/5$ als optimale Lösung mit Wert $-19{,}6$.

Falls wir initial keine triviale zulässige Basislösung haben, können wir trotzdem so vorgehen und hoffen, durch Basiswechsel eine zulässig Basislösung zu finden. Wir werden später diskutieren, warum die Hoffnung gerechtfertigt sein kann. Zunächst werden wir das am kleinen Beispiel durchgeführte Verfahren etwas stärker formal fassen und beschreiben, um es einer Implementierung zugänglicher zu machen. Dazu führen wir den Begriff des Simplextableaus ein.

Definition 14.23. *Zum linearen Optimierungsproblem* $Z \to \min$ *mit* $c^T \cdot x + \delta = Z$, $Ax - b = -u$ *und* $x \geq 0, u \geq 0$ *heißt*

x_1	x_2	$\cdots$	x_n		
$a_{1,1}$	$a_{1,2}$	$\cdots$	$a_{1,n}$	$-b_1$	$-u_1$
$a_{2,1}$	$a_{2,2}$	$\cdots$	$a_{2,n}$	$-b_2$	$-u_2$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\vdots$
$a_{m,1}$	$a_{m,2}$	$\cdots$	$a_{m,n}$	$-b_m$	$-u_m$
c_1	c_2	$\cdots$	c_n	δ	Z

das zugehörige Simplextableau.

Definition 14.24. *Zwei Simplextableaus heißen* äquivalent*, wenn die zugehörigen Gleichungssysteme gleiche Lösungsmengen haben.*

Wir werden also einen Basiswechsel als Übergang von einem Simplextableau zu einem äquivalenten Simplextableau beschreiben. Zunächst werden wir unsere bisherigen Erkenntnisse in Simplextableaus „wiederfinden“.

Theorem 14.25. *Sind in einem Simplextableau alle b-Werte nichtnegativ, so ist der Basispunkt zulässig. Sind zusätzlich alle c-Werte nichtnegativ, so ist der Basispunkt optimal.*

Beweis. Sind alle b-Werte nichtnegativ, so ist der Basispunkt $(0, b)$ (also $x = 0$ und $u = b$) offensichtlich zulässig. Ist zusätzlich $c \geq 0$, so ist die Wahl $x = 0$ offensichtlich optimal. □

Theorem 14.26. *Ist ein Simplextableau zur Basis $\{x_i, u_j \mid i \in B_x, j \in B_u\}$ äquivalent zum Simplextableau eines linearen Optimierungsproblems, so sind die Hyperebenen $x_i = 0$ für alle $i \in \{1, 2, \dots n\} \setminus B_x$ und $u_i = 0$ für alle $i \in \{1, 2, \dots, m\} \setminus B_u$ linear unabhängig. Der Schnittpunkt dieser Hyperebenen ist ein Extrempunkt der zulässigen Menge, falls er zulässig ist.*

Beweis. Wir schreiben zunächst die betrachteten Hyperebenen als Gleichungen in den ursprünglichen x-Variablen. Dabei wird aus $u_i = 0$ für $i \in \{1, 2, \dots, m\} \setminus B_u$ die Gleichung $\sum_{j=1}^{n} a_{i,j} x_j = b_i$. Die Gleichungen $x_i = 0$ für $i \in \{1, 2, \dots, n\} \setminus B_x$ haben bereits die gewünschte Form. Wir betrachten die Koeffizientenmatrix A. Zur Vereinfachung der Notation nehmen wir an, dass $B_x = \{1, 2, \dots, k\}$ und $B_u = \{k+1, k+2, \dots, m\}$ gilt. Wir können das einfach durch Umbenennung erreichen. Damit haben wir

$$A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,k} & a_{1,k+1} & \cdots & a_{1,n} \\ \vdots & \cdots & \vdots & \vdots & \cdots & \vdots \\ a_{k,1} & \cdots & a_{k,k} & a_{k,k+1} & \cdots & a_{k,n} \\ & & & 1 & \cdots & 0 \\ & 0 & & \vdots & \ddots & \vdots \\ & & & 0 & \cdots & 1 \end{pmatrix}$$

als Koeffizientenmatrix. Natürlich sind die betrachteten Hyperebenen genau dann linear unabhängig, wenn A invertierbar ist. Die spezielle Form von A lässt uns direkt erkennen, dass A genau dann invertierbar ist, wenn die Submatrix

$$A^{(k)} = \begin{pmatrix} a_{1,1} & \cdots & a_{1,k} \\ \vdots & \cdots & \vdots \\ a_{k,1} & \cdots & a_{k,k} \end{pmatrix}$$

invertierbar ist. Wir kehren jetzt zum Simplextableau zur durch B_x und B_u gegebenen Basis zurück, dabei nehmen wir weiter an, dass $B_x = \{1, 2, \dots, k\}$ und $B_u = \{k+1, k+2, \dots, m\}$ gilt. Seine Koeffizientenmatrix sei A', außerdem sei $A'^{(k)}$ analog zu $A^{(k)}$ definiert. Der Bequemlichkeit halber benutzen wir die Definition $y^{(k)}$, um den aus den ersten k Komponenten von y bestehenden Teilvektor zu bezeichnen. Mit dieser Notation folgt aus $x_{k+1} = x_{k+2} = \cdots x_n = 0$ nun $-x^{(k)} = A'^{(k)} u^{(k)} - b'^{(k)}$, außerdem haben wir aus dem Simplextableau mit Koeffizientenmatrix A noch $-u^{(k)} = A^{(k)} x^{(k)} - b^{(k)}$. Wir können ineinander einsetzen und erhalten

$$-x^{(k)} = A'^{(k)} \left(-A^{(k)} x^{(k)} + b^{(k)}\right) - b'^{(k)},$$

so dass $x^{(k)} = A'^{(k)}A^{(k)}x^{(k)} - A'^{(k)}b^{(k)} + b'^{(k)}$ folgt. Diese Gleichung gilt natürlich insbesondere für $x^{(k)} = 0^{(k)}$, also erhalten wir $A'^{(k)}b^{(k)} = b'^{(k)}$. Wir setzen das ein, so dass wir $x^{(k)} = A'^{(k)}A^{(k)}x^{(k)}$ erhalten für alle $x^{(k)}$. Daraus folgt $A'^{(k)} = \left(A^{(k)}\right)^{-1}$ und $A^{(k)}$ ist offensichtlich invertierbar. □

Wir sehen, dass wir unbesorgt mit Simplextableaus arbeiten können. Wir müssen uns jetzt also um einen Basiswechsel kümmern, bei dem eine Nichtbasisvariable r_k die Basis betritt und eine Basisvariable s_i die Basis verlässt.

Definition 14.27. *Betrachte ein zum Simplextableau zum linearen Optimierungsproblem $Z \to \min$ mit $c^T \cdot x + \delta = Z$, $Ax - b = -u$ und $x \geq 0, u \geq 0$ äquivalentes Simplextableau:*

r_1	r_2	$\cdots$	r_k	$\cdots$	r_n		
$a_{1,1}$	$a_{1,2}$	$\cdots$	$a_{1,k}$	$\cdots$	$a_{1,n}$	$-b_1$	$-s_1$
$a_{2,1}$	$a_{2,2}$	$\cdots$	$a_{2,k}$	$\cdots$	$a_{2,n}$	$-b_2$	$-s_2$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\vdots$
$a_{i,1}$	$a_{i,2}$	$\cdots$	$a_{i,k}$	$\cdots$	$a_{i,n}$	$-b_i$	$-s_i$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\vdots$
$a_{m,1}$	$a_{m,2}$	$\cdots$	$a_{m,k}$	$\cdots$	$a_{m,n}$	$-b_m$	$-s_m$
c_1	c_2	$\cdots$	c_k	$\cdots$	c_n	δ	Z

Wird ein Basiswechsel durchgeführt, bei dem die Nichtbasisvariable r_k die Basis betritt und die Basisvariable s_i die Basis verlässt, so heißt $a_{i,k}$ Pivotelement.

Bei einem Basiswechsel setzen wir voraus, dass $a_{i,k} \neq 0$ für das Pivotelement $a_{i,k}$ gilt. Wir vollziehen jetzt den Basiswechsel, wie wir ihn vorher an einem konkreten Beispiel besprochen haben, nun für das Simplextableau aus Definition 14.27 nach, dabei sei $a_{i,k} \neq 0$ das Pivotelement, die Nichtbasisvariable r_k betritt also die Basis und die Basisvariable s_i verlässt die Basis. Wir betrachten zunächst die i-te Zeile und sehen, dass wir

$$-r_k = \frac{a_{i,1}}{a_{i,k}}r_1 + \cdots + \frac{a_{i,k-1}}{a_{i,k}}r_{k-1} + \frac{1}{a_{i,k}}s_i + \frac{a_{i,k+1}}{a_{i,k}}r_{k+1} + \cdots + \frac{a_{i,n}}{a_{i,k}}r_n - \frac{b_i}{a_{i,k}}$$

erhalten. Für die j-te Zeile (mit $j \neq i$) haben wir

$$a_{j,1}r_1 + \cdots + a_{j,k-1}r_{k-1} + a_{j,k}r_k + a_{j,k+1}r_{k+1} + \cdots + a_{j,n}r_n = -s_j$$

und erhalten durch Einsetzen von r_k in die j-te Zeile (mit $j \neq i$)

$$\left(a_{j,1} - a_{j,k}\frac{a_{i,1}}{a_{i,k}}\right)r_1 + \cdots + \left(a_{j,k-1} - a_{j,k}\frac{a_{i,k-1}}{a_{i,k}}\right)r_{k-1} - \frac{a_{j,k}}{a_{i,k}}s_i$$
$$+\left(a_{j,k+1} - a_{j,k}\frac{a_{i,k+1}}{a_{i,k}}\right)r_{k+1} + \cdots + \left(a_{j,n} - a_{j,k}\frac{a_{i,n}}{a_{i,k}}\right)r_n - \left(b_j - a_{j,k}\frac{b_i}{a_{i,k}}\right) = -s_j.$$

Wir fassen das zusammen: Aus dem Pivotelement $a_{i,k}$ wird $1/a_{i,k}$. In der Pivotzeile i erhalten wir in der Spalte j (mit Ausnahme der Pivotspalte k) aus dem Koeffizienten $a_{i,j}$ den Koeffizienten $a_{i,j}/a_{i,k}$, außerdem wird aus $-b_i$ nun $-b_i/a_{i,k}$. In der Pivotspalte k erhalten wir in der Zeile j (mit Ausnahme der Pivotzeile i) aus dem Koeffizienten $a_{j,k}$ den Koeffizienten $-a_{j,k}/a_{i,k}$, außerdem wird aus c_k nun $-c_k/a_{i,k}$. Schließlich erhalten wir für allen anderen Elemente, die weder in der Pivotzeile i noch der Pivotspalte k stehen, aus $a_{j,l}$ nun $a_{j,l} - a_{j,k}a_{i,l}/a_{i,k}$, aus $-b_j$ nun $-b_j - a_{j,k}b_i/a_{i,k}$, aus c_l nun $c_l - c_k a_{i,l}/a_{i,k}$ und aus δ nun $\delta - c_k b_i/a_{i,k}$. Wir können uns das kurz und prägnant zu

$$\begin{bmatrix} p & q \\ r & s \end{bmatrix} \to \begin{bmatrix} \frac{1}{p} & \frac{q}{p} \\ -\frac{r}{p} & s - \frac{rq}{p} \end{bmatrix}$$

zusammenfassen, dabei steht p für das Pivotelement.

Nach diesen Vorbereitungen kommen wir jetzt zum Simplex-Algorithmus. Um den Algorithmus kompakt notieren zu können, definieren wir noch, wann wir eine Zeile des Simplextableaus *gut* nennen. Die Definition ist naheliegend, sie drückt aus, welche Zeilen der Zulässigkeit unseres Basispunktes nicht im Wege stehen.

Definition 14.28. *In einem Simplextableau heißt die i-te Zeile* gut *genau dann, wenn $b_i \geq 0$ gilt, andernfalls heißt die Zeile* schlecht. *Es sei die Menge der guten Zeilenindizes $G := \{i \mid i \in \{1, 2, \ldots, m\}$ und i gut$\}$.*

Algorithmus 14.29 (Simplex-Algorithmus)**.**

1. Phase: Finden eines zulässigen Basispunkts

1. *If $G = \{1, 2, \ldots, m\}$ $\{*$ zulässiger Basispunkt gefunden $*\}$*
   Then Weiter mit 2. Phase.
2. *$s := \min\{i \mid b_i < 0\}$ $\{*$ oberste schlechte Zeile $*\}$*
3. *If $\min\{a_{s,1}, a_{s,2}, \ldots, a_{s,n}\} \geq 0$*
   Then Ausgabe „keine zulässige Lösung“. STOP.
4. *Wähle k_0 mit $a_{s,k_0} < 0$.*
5. *Wähle Pivotzeile p so, dass $\frac{b_p}{a_{p,k_0}} = \min\left\{\frac{b_j}{a_{j,k_0}} \mid j \in G \text{ und } a_{j,k_0} > 0\right\}$.*
6. *If $\min\left\{\frac{b_j}{a_{j,k_0}} \mid j \in G \text{ und } a_{j,k_0} > 0\right\} = \emptyset$, Then $p := s$.*
7. *Führe Basiswechsel um Pivotelement a_{p,k_0} durch.*
8. *Fahre mit dem neuen Simplextableau in Zeile 1 fort.*

2. Phase: Minimieren von Z auf zulässigen Basispunkten

1. *If* $\min\{c_1, c_2, \ldots, c_n\} \geq 0$
   Then Ausgabe „Basispunkt optimal". STOP.
2. *Wähle* k_0 *mit* $c_{k_0} < 0$.
3. *Wähle Pivotzeile* p *so, dass* $\frac{b_p}{a_{p,k_0}} = \min\left\{\frac{b_j}{a_{j,k_0}} \mid j \in G \text{ und } a_{j,k_0} > 0\right\}$.
6. *If* $\min\left\{\frac{b_j}{a_{j,k_0}} \mid j \in G \text{ und } a_{j,k_0} > 0\right\} = \emptyset$
   Then Ausgabe „$Z \rightsquigarrow -\infty$*. STOP.*
7. *Führe Basiswechsel um Pivotelement* a_{p,k_0} *durch.*
8. *Fahre mit dem neuen Simplextableau in Zeile 1 fort.*

Wir beobachten, dass der Basiswechsel korrekt durchgeführt wird. Die Wahl der Pivotzeile p in der dritten Zeile der zweiten Phase stellt sicher, dass die neue Basisvariable nur so weit wie möglich vergrößert wird. Insgesamt stellen wir fest, dass alle Ausgaben, die in der ersten und zweiten Phase getätigt werden, korrekt sind. Damit ist klar, dass die Ausgabe des Simplex-Algorithmus korrekt ist, falls er eine Ausgabe erzeugt. Wir halten das als Ergebnis fest.

Theorem 14.30. *Der Simplex-Algorithmus (Algorithmus 14.29) ist partiell korrekt. Ein Basiswechsel ist in Zeit* $O(nm)$ *durchführbar.*

Natürlich genügt uns partielle Korrektheit nicht, wir müssen also noch zeigen, dass der Simplex-Algorithmus auch endlich ist, also in endlicher Zeit eine Ausgabe erzeugt und stoppt. Das wird nicht so einfach sein, um die Schwierigkeiten zu verstehen, definieren wir, was wir unter einem degenerierten linearen Optimierungsproblem verstehen.

Definition 14.31. *Ein lineares Optimierungsproblem heißt* degeneriert*, wenn es unter den* $n + m$ *Hyperebenen* $n + 1$ *Hyperebenen mit nichtleerem Schnitt gibt.*

Wir beobachten, dass ein lineares Optimierungsproblem jedenfalls dann degeneriert ist, wenn wir in der b-Spalte eine 0 haben. Zusätzlich zu den n Nichtbasisvariablen kann die Basisvariable der entsprechenden Zeile 0 gesetzt werden. Die Beziehung gilt aber auch umgekehrt: Wenn ein lineares Optimierungsproblem degeneriert ist, gibt es ein äquivalentes Simplextableau mit einer 0 in der b-Spalte.
Wir wollen zeigen, dass der Simplex-Algorithmus zumindest für lineare Optimierungsprobleme, die nicht degeneriert sind, korrekt ist. Dafür zeigen wir zunächst, dass $b_p/a_{p,k_0} \geq 0$ gilt, wenn das Pivotelement a_{p,k_0} im Simplex-Algorithmus gewählt wird.

Lemma 14.32. *Wenn der Simplex-Algorithmus (Algorithmus 14.29) das Pivotelement a_{p,k_0} wählt, so gilt $b_p/a_{p,k_0} \geq 0$.*

Beweis. Wir unterscheiden zwei Fälle. Ist $\left\{ \frac{b_j}{a_{j,k_0}} \mid j \in G \text{ und } a_{j,k_0} > 0 \right\} \neq \emptyset$, so wird eine Zeile p mit $b_p \geq 0$ und $a_{p,k_0} > 0$ gewählt, so dass $b_p/a_{p,k_0} \geq 0$ folgt. Es ist also nur der Fall $\min \left\{ \frac{b_j}{a_{j,k_0}} \mid j \in G \text{ und } a_{j,k_0} > 0 \right\} = \emptyset$ zu untersuchen, der nur in der ersten Phase eine Rolle spielt, weil in der zweiten Phase in diesem Fall der Algorithmus stoppt. In der ersten Falle wird in diesem Fall $p = s$ gewählt, dabei ist s die kleinste schlechte Zeile, es gilt also $b_p < 0$. Gemäß Wahl von k_0 gilt aber auch $a_{s,k_0} < 0$, so dass $b_p/a_{p,k_0} > 0$ folgt. □

Theorem 14.33. *Der Simplex-Algorithmus ist für nichtdegenerierte Probleme korrekt.*

Beweis. Wir müssen nur zeigen, dass der Algorithmus endlich ist, partielle Korrektheit hatten wir schon eingesehen (Theorem 14.30). Wir machen das getrennt für beide Phasen und zeigen zunächst, dass die erste Phase endliche Länge hat. Wir betrachten dazu einen Basiswechsel am Simplextableau, also

$$\begin{array}{c} r \\ \begin{array}{|c|c|} \hline a & -b \\ \hline c & \delta \\ \hline \end{array} \end{array} \begin{array}{l} -s \\ Z \end{array} \quad \rightsquigarrow \quad \begin{array}{c} r' \\ \begin{array}{|c|c|} \hline a' & -b' \\ \hline c' & \delta' \\ \hline \end{array} \end{array} \begin{array}{l} -s' \\ Z \end{array}$$

und behaupten, dass für jede gute Zeile g stets $b_g \geq b'_g$ gilt, gute Zeilen also gut bleiben. Wir betrachten zunächst die Pivotzeile p und beobachten, dass $b'_p = b_p/a_{p,k_0}$ gilt, außerdem wissen wir, dass $b_p/a_{p,k_0} \geq 0$ gilt (Lemma 14.32). Weil das Problem nicht degeneriert ist, gilt sogar $b'_p > 0$. Wir sehen, dass die Pivotzeile auf jeden Fall gut ist nach dem Basiswechsel. Nun betrachten wir eine andere Zeile $i \neq p$. Wir haben $b'_i = b_i - a_{i,k_0} b_p/a_{p,k_0}$ und definieren $\Delta_i := b_i - b'_i = a_{i,k_0} b_p/a_{p,k_0}$. Wir unterscheiden zwei Fälle nach dem Wert von a_{i,k_0}: Ist $a_{i,k_0} \leq 0$, so ist $\Delta_i \leq 0$ und wir haben $b'_i = b_i - \Delta_i \geq b_i$, die Zeile i bleibt also wie behauptet gut, wenn sie vorher gut war. Ist andererseits $a_{i,k_0} > 0$, so wird bei der Wahl von p auch die Zeile i betrachtet, wir dürfen also sicher sein, dass $b_i/a_{i,k_0} \geq b_p/a_{p,k_0}$ gilt. Also ist

$$b'_i = b_i - a_{i,k_0} b_p/a_{p,k_0} = a_{i,k_0} \cdot \left(\frac{b_i}{a_{i,k_0}} - \frac{b_p}{a_{p,k_0}} \right) \geq 0$$

und wir sehen, dass auch in diesem Fall eine gute Zeile gut bleibt.
Wir behaupten für die erste Phase außerdem, dass $b'_s > b_s$ gilt, die schlechte Zeile s wird also auf jeden Fall „weniger schlecht“. Gilt $s = p$, so haben wir

sogar $b'_s > 0$ und die schlechte Zeile ist nachher sogar gut. Wir können also annehmen, dass $s \neq p$ gilt. Wir haben

$$b'_s = b_s - a_{i,k_0} \cdot \frac{b_p}{a_{p,k_0}}$$

und wissen, dass $a_{i,k_0} < 0$ gilt und $b_p/a_{p,k_0} > 0$. Also ist $b'_s > b_s$ wie behauptet. Weil die Anzahl der Basispunkte endlich ist, wird jede schlechte Zeile in endlicher Zeit gut, weil zusätzlich wie gesehen gute Zeilen immer gut bleiben, endet die erste Phase in endlicher Zeit mit einem zulässigen Basispunkt oder der Ausgabe, dass es keine Lösung gibt.
Für die zweite Phase beobachten wir zunächst, dass wir mit einem zulässigen Basispunkt starten und wie in der ersten Phase alle guten Zeilen gut bleiben, wir haben also immer einen zulässigen Basispunkt. Wir behaupten, dass nach einem Basiswechsel $\delta' < \delta$ gilt. Wir haben

$$\delta' = \delta + c_{k_0} \cdot \frac{b_p}{a_{p,k_0}}$$

und haben $c_{k_0} < 0$ und $a_{p,k_0} > 0$, weil wir k_0 so wählen. Weil das Problem nichtdegeneriert ist, dürfen wir $b_p > 0$ voraussetzen. Also ist $\delta' < \delta$ wie behauptet. Weil die Anzahl der Basispunkte endlich ist, haben wir nach endlichen vielen Schritten ein minimales δ' erreicht oder geben „$Z \to -\infty$“ aus. □

Bei degenerierten Problemen können wir natürlich Glück haben und der Algorithmus hält. Dann ist auf jeden Fall die Ausgabe korrekt, das sichert ja Theorem 14.30. Wir können aber auch Pech haben und das Algorithmus stoppt nie. Das ist natürlich unbefriedigend. Kann man das nicht ändern? Wenn der Simplex-Algorithmus nicht stoppt, kann das nur daran liegen, dass er Basispunkte mehrfach erreicht. Es wird ja in jeder Runde ein Basiswechsel ausgeführt und es gibt nur endlich viele verschiedene Basispunkte. Wir können also Endlichkeit erzwingen, indem wir uns merken, welche Basispunkte wir schon gesehen haben und einen Wechsel zu einem Basispunkt, den wir schon gesehen haben, vermeiden. In der Praxis wird man das nicht tun, weil der Speicherplatzbedarf dabei zu groß ist. Stattdessen wird man in der Praxis die im Simplex-Algorithmus vorhandene Wahlfreiheit zu randomisierten Entscheidungen nutzen. Dann ist zumindest die erwartete Rechenzeit endlich. Nur Endlichkeit nachzuweisen, ist natürlich auch nicht befriedigend. Tatsächlich kann man zeigen, dass die Worst-Case-Rechenzeit exponentiell ist. Allerdings sind solche Worst-Case-Instanzen schwierig zu finden und wir sparen uns die Mühe hier. Wir halten fest, dass der Simplex-Algorithmus in der

Praxis noch immer der schnellste Algorithmus für die lineare Programmierung ist, obwohl Polynomialzeitalgorithmen bekannt sind. Darum findet man ihn auch im Teil über Heuristiken.
Der Simplex-Algorithmus (Algorithmus 14.29) ist in der hier vorgestellten Grundform zur Lösung linearer Programme in der Praxis durchaus brauchbar. Je nach Anwendung kann es sich aber anbieten, den Algorithmus zu modifizieren und auf spezielle Bedürfnisse abzustimmen. Wir behandeln solche Modifikationen hier als heuristisch, wir hoffen also nur, dass sie den Grundalgorithmus verbessern, ohne einen Beweis zu versuchen. Das ist konsequent, denn der Simplex-Algorithmus selbst ist ebenfalls nur als Heuristik vertretbar. Wir diskutieren die Modifikationen nur kurz an, ohne die Details herauszuarbeiten. Die erste Modifikation betrifft Nebenbedingungen in Gleichungsform. Wir hatten in der Einleitung diskutiert, dass wir eine Nebenbedinung $a_i x = b_i$ durch zwei Ungleichungen $a_i x \leq b_i$ und $-a_i x \leq -b_i$ ersetzen können. Wir erkennen jetzt, dass das direkt zur Folge hat, dass wir ein degeneriertes lineares Optimierungsproblem erzeugt haben, was angesichts unserer Ergebnisse unschön ist. Man überlegt sich leicht, dass es eine bessere Lösung gibt, die nicht automatisch Degeneration impliziert, wir wollen die Idee und ihre Ausformulierung hier auslassen und in die Übungen verweisen.
Ebenfalls schon in der Einleitung hatten wir diskutiert, dass wir eine unbeschränkte Variable $x_i \in \mathbb{R}$ durch zwei Variablen $x_i', x_i'' \in \mathbb{R}_0^+$ ersetzen können, wenn wir $x_i = x_i' - x_i''$ setzen. Alternativ können wir den Algorithmus so modifizieren, dass er direkt mit unbeschränkten Variablen umgehen kann. In der ersten Phase ändert sich nicht viel, wir können die unbeschränkten Variablen bei der Wahl des Pivotelements ignorieren. In der zweiten Phase sehen wir, dass $c \geq 0$ als Optimalitätskriterium bezüglich der unbeschränkten Variablen nicht mehr greift. Eine optimale Lösung liegt nun vor, wenn keine Variable mehr wählbar ist. Außerdem müssen wir bei unbeschränkten Variablen alle Variablen mit $c_k \neq 0$ berücksichtigen, nicht nur die mit $c_k < 0$. Wenn wir eine unbeschränkte Variable als Pivotzeile wählen, müssen wir auf die Zulässigkeit des Basispunktes achten, andere Änderungen ergeben sich nicht. Der mäßige algorithmische Mehraufwand wird dadurch entschädigt, dass wir nicht die Anzahl der Variablen erhöhen müssen.
Wenn wir nach oben beschränkte Variablen haben, also $x_i \leq \alpha_i$, können wir diese Beschränkung einfach als Nebenbedingung aufnehmen. Man kann aber solche Variablen zulassen, die Beschränkung im Algorithmus berücksichtigen und dafür eine Nebenbedingung einsparen. Im Zentrum der Anpassung steht eine Erweiterung des Begriffs des Basispunkts: Wir lassen nun zusätzlich zum Wert 0 auch den Randwert α_i zu. Eine Basispunkt ist optimal, wenn keine erlaubte Änderung die Zielfunktion vermindert, dass lässt sich nach wie vor lokal prüfen. Bei der Überprüfung, ob Z in M unbeschränkt fällt, spielen die

beschränkten Variablen offenbar keine Rolle. In der ersten Phase müssen wir berücksichtigen, dass auch $x_i > \alpha_i$ einen Basispunkt unzulässig macht. In der zweiten Phase müssen wir bei einem Basiswechsel die Richtung der erlaubten Änderung beachten und bei der Wahl der neuen Nichtbasisvariablen wie immer die stärkste Beschränkung wählen, dabei aber natürlich die oberen Schranken α_i berücksichtigen.

Als Letztes betrachten wir eine spezielle Erweiterung, die in der Praxis relevant sein kann. Nehmen wir an, dass wir ein lineares Optimierungsproblem $Z = c^T x \to \min$ mit $Ax = b$ und $x \geq 0$ mit dem Simplex-Algorithmus (Algorithmus 14.29) gelöst haben. Jetzt entsteht in einer Anwendung ein neues Problem einfach dadurch, dass eine neue Variable x_{n+1} und eine weitere Spalte zu A hinzugefügt wird. Müssen wir das so entstandene lineare Optimierungsproblem wieder „ganz von vorne“ mit dem Simplex-Algorithmus lösen, oder können wir davon profitieren, dass wir das etwas kleinere Problem schon optimal gelöst haben? Wir nehmen an, dass wir die optimale Lösung x^* und die Basis B^* dazu kennen. Die heuristische Idee ist, dass $(x^*, 0)$ vermutlich ein guter Startpunkt ist. Dazu müssen wir aber den Basiswechsel zu B durchführen, und zwar bezogen auf die gesamte Matrix A. Wir erinnern uns dafür an Theorem 14.26 und übernehmen auch die Notation von dort. Wir können die Inverse zu $A^{(k)}$ links oben ablesen und müssen uns noch überlegen, was mit der $(n+1)$-ten Spalte passiert. Wir betrachten den Anfang der Länge k diese Spalte, in unserer Notation ist das $(a^{n+1})^{(k)}$. Wir schauen uns den relevanten Teil von A an, den wir B nennen, es ist also

$$B := \begin{pmatrix} a_{1,k+1} & \cdots & a_{1,n} \\ a_{2,k+1} & \cdots & a_{2,n} \\ \vdots & \cdots & \vdots \\ a_{k,k+1} & \cdots & a_{k,n} \end{pmatrix},$$

außerdem sei $x' := (x_{k+1}, \ldots, x_n)$. Mit dieser Notation haben wir im Starttableau

$$A^{(k)} x^{(k)} + Bx' + (a^{n+1})^{(k)} x_{n+1} - b^{(k)} = -u^{(k)}$$

und können folgern, dass

$$\left(A^{(k)}\right)^{-1} u^{(k)} + \left(A^{(k)}\right)^{-1} Bx' + \left(A^{(k)}\right)^{-1} (a^{n+1})^{(k)} x_{n+1} - \left(A^{(k)}\right)^{-1} b^{(k)} = -x^{(k)}$$

gilt. Wir erwähnt, ist $\left(A^{(k)}\right)^{-1}$ direkt ablesbar, so dass nur $\left(A^{(k)}\right)^{-1} (a^{n+1})^{(k)}$ neu zu berechnen ist. Dieses Produkt ist offenbar leicht in Zeit $O(k^2)$ berechenbar. Uns fehlen jetzt noch die übrigen Elemente der $(n+1)$-ten Spalte, das betrifft sowohl die Elemente $a_{i,n+1}$ als auch c_{n+1}. Natürlich waren diese Elemente niemals in der Pivotzeile, da sie ja bei den Basiswechseln gar

nicht vorhanden waren. Also sind wir beim Basiswechsel immer bei dem Fall, den wir mit „$s \to s - rq/p$“ abgekürzt hatten. Wir erhalten damit $c_{n+1}^{(\text{neu})} = c_{n+1} - (a_{\text{neu}}^{n+1})_1 \, c_1$ für $k = 1$, analog $c_{n+1}^{(\text{neu})} = c_{n+1} - (a_{\text{neu}}^{n+1})_1 \, c_1 - (a_{\text{neu}}^{n+1})_2 \, c_2$ für $k = 2$ und $c_{n+1}^{(\text{neu})} = c_{n+1} - \sum_{i=1}^{k} (a_{\text{neu}}^{n+1})_i \, c_i$ für allgemeines k, was sich mit vollständiger Induktion auch beweisen lässt. Für die $a_{i,n+1}$ ergeben sich die neuen Werte durch analoge Rechnungen. Wir können also effizient das vorherige Ergebnis x^* als neuen Startpunkt nutzen.

14.2 Primal-duale Approximation

Wir bleiben bei der linearen Programmierung und nehmen eine Art Perspektivenwechsel vor. Dazu schauen wir uns ein konkretes Beispiel an, an dem wir uns die Ideen herausarbeiten wollen. Wir betrachten dazu ein lineares Optimierungsproblem und werden zur Abwechslung einmal maximieren. Konkret sei

$$\begin{aligned} & Z(x_1, x_2) = 3x_1 - 2x_2 \to \max \\ & \quad x_1 - x_2 \leq 0 \quad (1) \\ \text{unter} \quad & 2x_1 + x_2 \leq 7 \quad (2) \\ & -x_1 + 3x_2 \leq 8 \quad (3) \\ & x_1, x_2 \geq 0 \end{aligned}$$

unser lineares Optimierungsproblem. Wir erkennen, dass wir aus den Nebenbedingungen direkt Schranken für den Wert einer optimalen Lösung ablesen können. Addieren wir zum Beispiel die Nebenbedingungen (1) und (2), so erhalten wir $3x_1 \leq 7$. Weil $x_2 \geq 0$ gilt, ist damit auch $Z(x_1, x_2) = 3x_1 - 2x_2 \leq 7$. Wenn wir das Zweifache von Nebenbedingung (3) und das Fünffache von Nebenbedingung (1) addieren, so erhalten wir $3x_1 + x_2 \leq 16$ und es folgt hier, dass $Z(x_1, x_2) \leq 16$ gilt. Wir können die Idee verallgemeinern: Für jede Nebenbedingung führen wir eine neue Variable ein, hier die Variablen y_1, y_2, y_3. Wir können nun das y_1-Fache der ersten Nebenbedingung, das y_2-Fache der zweiten Nebenbedingung und das y_3-Fache der dritten Nebenbedingung addieren, dann erhalten wir

$$y_1 \cdot (x_1 - x_2) + y_2 \cdot (2x_1 + x_2) + y_3 \cdot (-x_1 + 3x_2) \leq y_1 \cdot 0 + y_2 \cdot 7 + y_3 \cdot 8$$

oder auch

$$(y_1 + 2y_2 - y_3) \cdot x_1 + (-y_1 + y_2 + 3y_3) \cdot x_2 \leq 7y_2 + 8y_3.$$

Wir werfen einen vergleichenden Blick auf die Zielfunktion $Z(x_1, x_2) = 3x_1 - 2x_2$ und sehen, dass wenn $y_1 + 2y_2 - y_3 \geq 3$ und $-y_1 + y_2 + 3y_3 \geq -2$ gilt, $Z(x_1, x_2) \leq 7y_2 + 8y_3$ folgt. Wir können unser ursprüngliches Problem jetzt also auch in unseren neuen y-Variablen formulieren:

$$\begin{array}{rll} & Z'(y_1, y_2, y_3) = 7y_2 + 8y_3 \to \min & \\ \text{unter} & y_1 + 2y_2 - y_3 \geq 3 & \\ & -y_1 + y_2 + 3y_3 \geq -2 & \\ & y_1, y_2, y_3 \geq 0 & \end{array}$$

Für jede Variable unseres ursprünglichen Problems haben wir nun eine Nebenbedingung, für jede Nebenbedingung des ursprünglichen Problems haben wir nun eine Variable. Eine Lösung $y \in \left(\mathbb{R}_0^+\right)^3$ für unser neues lineares Programm liefert uns direkt $Z(x_1, x_2) \leq Z'(y)$ für alle $x_1, x_2 \in \mathbb{R}_0^+$. Wir nennen dieses neue lineare Optimierungsproblem das zu unserem Ausgangsproblem *duale Problem* und werden diesen Begriff jetzt auch allgemein formal fassen.

Definition 14.34. *Das zu einem* primalen Problem $Z(x) = c^T X \to \min$ *unter* $Ax \geq b$ *und* $x \geq 0$ *ist das Problem* $Z'(y) = b^T y \to \max$ *unter* $A^T y \leq c$ *und* $y \geq 0$.

Wir haben es am Beispiel schon angedeutet, es gibt zwischen einem primalen Problem und seinem dualen Problem enge Beziehungen. Wir werden diese Beziehungen hier festhalten und beweisen.

Theorem 14.35. *Sei P ein primales Problem der linearen Optimierung, sei P' das zu P duale Problem, sei P'' das zu P' duale Problem. Dann sind P und P'' äquivalent.*

Beweis. Gemäß Definition 14.34 ist P' gegeben durch $Z'(y) = b^T y \to \max$ unter $A^T y \leq c$ und $y \geq 0$. Das ist äquivalent zu dem Problem $-Z'(y) = (-b)^T y \to \min$ unter $(-A)^T y \geq -c$ und $y \geq 0$. Gemäß Definition 14.34 ist das dazu duale Problem gegeben durch $Z''(z) = (-c)^T z \to \max$ unter $\left((-A)^T\right)^T z \leq -b$ und $z \geq 0$. Das ist offenbar äquivalent zu $-Z''(z) = c^T z \to \min$ unter $Az \geq b$ und $z \geq 0$. Also ist P'' äquivalent zu P. □

Theorem 14.36. *Sei x zulässig für das lineare Minimierungsproblem P mit Zielfunktion Z, sei y zulässig für das zu P duale Maximierungsproblem P' mit Zielfunktion Z'. Dann gilt $Z(x) \geq Z'(y)$.*

Beweis. Wir haben $Z(x) = c^T x$ gemäß Definition der Zielfunktion, die Nebenbedingungen von P' stellen sicher, dass $c^T x \geq \left(A^T y\right)^T x$ gilt. Natürlich gilt $\left(A^T y\right)^T x = y^T A x$. Die Nebenbedingungen von P stellen sicher, dass $y^T A x \geq y^T b$ gilt und wir beobachten, dass $Z'(y) = y^T b$ gilt. Insgesamt haben wir also $Z(x) \geq Z'(y)$ gezeigt wie behauptet. □

Theorem 14.36 hat eine interessante direkte Konsequenz: Jeder zulässige Punkt für P' definiert eine untere Schranke für den Wert zulässiger Lösungen für P. Natürlich hat jeder zulässige Punkt endlichen Funktionswert, folglich kann Z nicht unbeschränkt sein, wenn es einen zulässigen Punkt für P' gibt. Umgekehrt können wir sagen, dass die Menge der zulässigen Punkte für P' leer ist, wenn Z nach unten unbeschränkt ist.

Theorem 14.37 (**Dualitätstheorem**). *Sei P ein primales Problem der linearen Optimierung mit Zielfunktion Z, sei P' das zu P duale Problem mit Zielfunktion Z'. Es gilt:*

1. *P hat eine Lösung $\Leftrightarrow$ P' hat eine Lösung*
2. *x Lösung von P und y Lösung von $P' \Rightarrow Z(x) = Z(y)$*
3. *Aus dem Lösungssimplextableau von P kann die Lösung von P' abgelesen werden und umgekehrt.*

Beweis. Für den Beweis schauen wir uns die zu den linearen Optimierungsproblemen P und P' gehörigen Simplextableaus an. Wir führen den Simplex-Algorithmus (Algorithmus 14.29) für P aus und vollziehen jeden Basiswechsel, den der Simplex-Algorithmus für P durchführt, parallel auch im Simplextableau zu P' durch. Dabei betrachten wir die Beziehungen zwischen den beiden Simplextableaus.
Zum primalen Problem P mit Zielfunktion $Z(x) = c^T x \to \min$ und Nebenbedingungen $(-A)x \leq b$ sowie $x \geq 0$ gehört das folgende Anfangstableau:

x_1	x_2	$\cdots$	x_k	$\cdots$	x_n		
$-a_{1,1}$	$-a_{1,2}$	$\cdots$	$-a_{1,k}$	$\cdots$	$-a_{1,n}$	b_1	$-u_1$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\vdots$
$-a_{i,1}$	$-a_{i,2}$	$\cdots$	$-a_{i,k}$	$\cdots$	$-a_{i,n}$	b_i	$-u_i$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\vdots$
$-a_{m,1}$	$-a_{m,2}$	$\cdots$	$-a_{m,k}$	$\cdots$	$-a_{m,n}$	b_m	$-u_m$
c_1	c_2	$\cdots$	c_k	$\cdots$	c_n	$\delta = 0$	Z

Zum dualen Problem P' mit Zielfunktion $Z'(y) = (-b)^T y \to \min$ und Nebenbedingungen $A^T y \leq c$ sowie $y \geq 0$ gehört das folgende Anfangstableau:

y_1	y_2	$\cdots$	y_i	$\cdots$	y_m		
$a_{1,1}$	$a_{2,1}$	$\cdots$	$a_{i,1}$	$\cdots$	$a_{m,1}$	$-c_1$	$-v_1$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\vdots$
$a_{1,k}$	$a_{2,k}$	$\cdots$	$a_{i,k}$	$\cdots$	$a_{m,k}$	$-c_k$	$-v_k$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\vdots$
$a_{1,n}$	$a_{2,n}$	$\cdots$	$a_{i,n}$	$\cdots$	$a_{m,n}$	$-c_n$	$-v_n$
$-b_1$	$-b_2$	$\cdots$	$-b_i$	$\cdots$	$-b_m$	$\Delta = 0$	$-Z'$

Wir betrachten jetzt einen Basiswechsel in P, dabei sei $-a_{i,k}$ das Pivotelement. Wir erhalten dann das folgende Simplextableau als Ergebnis:

x_1	x_2	$\cdots$	u_i	$\cdots$	x_n		
$-a_{1,1} + a_{1,k}\frac{a_{i,1}}{a_{i,k}}$	$-a_{1,2} + a_{1,k}\frac{a_{i,2}}{a_{i,k}}$	$\cdots$	$-\frac{a_{1,k}}{a_{i,k}}$	$\cdots$	$-a_{1,n} + a_{1,k}\frac{a_{i,n}}{a_{i,k}}$	$b_1 - a_{1,k}\frac{b_1}{a_{i,k}}$	$-u_1$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\vdots$
$\frac{a_{i,1}}{a_{i,k}}$	$\frac{a_{i,2}}{a_{i,k}}$	$\cdots$	$-\frac{1}{a_{i,k}}$	$\cdots$	$\frac{a_{i,n}}{a_{i,k}}$	$-\frac{b_i}{a_{i,k}}$	$-x_k$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\vdots$
$-a_{m,1} + a_{m,k}\frac{a_{i,1}}{a_{i,k}}$	$-a_{m,2} + a_{m,k}\frac{a_{i,2}}{a_{i,k}}$	$\cdots$	$-\frac{a_{m,k}}{a_{i,k}}$	$\cdots$	$-a_{m,n} + a_{m,k}\frac{a_{i,n}}{a_{i,k}}$	$b_m - a_{m,k}\frac{b_m}{a_{i,k}}$	$-u_m$
$c_1 - c_k\frac{a_{i,1}}{a_{i,k}}$	$c_2 - c_k\frac{a_{i,2}}{a_{i,k}}$	$\cdots$	$\frac{c_k}{a_{i,k}}$	$\cdots$	$c_n - c_k\frac{a_{i,n}}{a_{i,k}}$	$\delta + c_k\frac{b_i}{a_{i,k}}$	Z

Wir führen jetzt im Simplextableau zu P' den entsprechenden Basiswechsel durch, benutzen dafür das entsprechende Pivotelement $-a_{i,k}$, lassen also y_i die Basis betreten und v_k die Basis verlassen. Wir notieren das Ergebnis wieder in Form eines Simplextableaus.

y_1	y_2	$\cdots$	v_k	$\cdots$	y_m		
$a_{1,1} - a_{i,1}\frac{a_{1,k}}{a_{i,k}}$	$a_{2,1} - a_{i,1}\frac{a_{2,k}}{a_{i,k}}$	$\cdots$	$-\frac{a_{i,1}}{a_{i,k}}$	$\cdots$	$a_{m,1} - a_{i,1}\frac{a_{m,k}}{a_{i,k}}$	$-c_1 + a_{i,1}\frac{c_1}{a_{i,k}}$	$-v_1$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\vdots$
$\frac{a_{1,k}}{a_{i,k}}$	$\frac{a_{2,k}}{a_{i,k}}$	$\cdots$	$\frac{1}{a_{i,k}}$	$\cdots$	$\frac{a_{m,k}}{a_{i,k}}$	$-\frac{c_k}{a_{i,k}}$	$-y_i$
$\vdots$	$\vdots$	$\cdots$	$\vdots$	$\cdots$	$\vdots$	$\vdots$	$\vdots$
$a_{1,n} - a_{i,n}\frac{a_{1,k}}{a_{i,k}}$	$a_{2,n} - a_{i,n}\frac{a_{2,k}}{a_{i,k}}$	$\cdots$	$-\frac{a_{i,n}}{a_{i,k}}$	$\cdots$	$a_{m,n} - a_{i,n}\frac{a_{m,k}}{a_{i,k}}$	$-c_n + a_{i,n}\frac{c_n}{a_{i,k}}$	$-v_n$
$-b_1 + a_{i,1}\frac{b_1}{a_{i,k}}$	$-b_2 + a_{i,2}\frac{b_2}{a_{i,k}}$	$\cdots$	$\frac{b_i}{a_{i,k}}$	$\cdots$	$-b_m + a_{i,m}\frac{b_m}{a_{i,k}}$	$\Delta - b_i\frac{c_k}{a_{i,k}}$	$-Z'$

Wir beobachten, dass bei dem Basiswechsel viele Beziehungen zwischen den Simplextableaus zu P und P' erhalten bleiben. Die Beziehungen zwischen den Basis- und Nichtbasisvariablen entsprechen einander weiterhin, die Matrix des Simplextableaus zu P' ist weiterhin die negative Transponierte der Matrix des Simplextableaus zu P, die letzte Zeile des Simplextableaus zu P' stimmt weiterhin mit der negierten letzten Spalte des Simplextableaus zu P überein, insbesondere ist auch weiterhin $\Delta = -\delta$. Weil wir keine besonderen Voraussetzungen an den Basiswechsel gestellt haben, gilt das für jeden Basiswechsel und natürlich auch nach jeder Folge von Basiswechseln.

Nehmen wir nun an, dass P eine Lösung hat und der Simplex-Algorithmus für P eine Lösung (also einen Basispunkt mit minimalem Funktionswert) berechnet. Die Variablen und Koeffizienten im zugehörigen Simplextableau seien durch einen Stern ($\square^*$) gekennzeichnet. Der optimale Basispunkt $x^* = 0$ und $u^* = -b^*$ ist natürlich zulässig, also ist $-b^* \geq 0$. Da Z nicht weiter verringert werden kann, gilt außerdem $c^* \geq 0$. Wir betrachten jetzt den Basispunkt $y^* = 0$, $v^* = c^*$ für das duale Problem. Weil $c^* \geq 0$ gilt, ist der Basispunkt zulässig für P'. Da außerdem $-b^* \geq 0$ gilt, kann $-Z'$ nicht weiter verringert werden, so dass der Basispunkt optimal ist. Der Wert der Lösung für P ist δ^*, der Wert der Lösung für P' ist $-\Delta^* = -(-\delta^*) = \delta^*$, die Lösungen haben also den gleichen Wert. Weil wir auch gesehen haben, wie die Lösung für P' aus dem Lösungstableau für P abgelesen werden kann (und umgekehrt), ist der Beweis damit geführt. □

Wir fassen zusammen, was wir jetzt über ein lineares Optimierungsproblem P und sein duales Problem P' aussagen können.

Korollar 14.38. *Sei P ein lineares Optimierungsproblem, sei P' sein duales Problem. Es gilt stets eine der folgenden vier Aussagen.*

1. *P und P' haben beide Lösungen, diese Lösungen haben den gleichen Wert.*
2. *Die Zielfunktion von P ist auf der zulässigen Menge nach unten unbeschränkt und die zulässige Menge von P' ist leer.*
3. *Die Zielfunktion von P' ist auf der zulässigen Menge nach unten unbeschränkt und die zulässige Menge von P ist leer.*
4. *Die zulässigen Mengen von P und P' sind beide leer.*

Wir können diese engen Beziehungen zwischen einem linearen Optimierungsproblem P und seinem dualen Problem nutzen, um schneller zulässige Basispunkte zu finden. Wenn wir P lösen wollen, der erste Basispunkt aber unzulässig ist und dabei $c \geq 0$ gilt, dann hat offensichtlich das duale Problem sofort einen zulässigen Basispunkt. Wir können uns also die erste Phase des Simplex-Algorithmus (Algorithmus 14.29) ersparen, wenn wir das duale Problem P' lösen und daraus die Lösung für P ablesen.
Wir finden für die Beziehungen zwischen primalen und dualen Problemen aber auch noch eine andere Anwendung, die weit weniger offensichtlich und dadurch vielleicht überraschend ist. Wir können Approximationsalgorithmen entwerfen. Wir wollen das hier besprechen, auch wenn wir das Thema „Approximation“ eigentlich in Teil II behandelt haben.

Wir diskutieren unser Vorgehen an einem konkreten Beispiel und schauen uns dazu das Problem an, ein gewichtetes Vertex Cover zu berechnen. Eingabe ist dabei ein ungerichteter gewichteter Graph $G = (V, E, c)$ mit ganzzahligen Knotengewichten $c\colon V \to \mathbb{N}$. Ausgabe ist eine Knotenmenge $V' \subseteq V$, die alle Kanten abdeckt (also gilt $\forall e \in E\colon e \cap V' \neq \emptyset$) und dabei minimales Gewicht hat, es ist also $\sum_{v \in V'} c(v)$ zu minimieren. Wir wissen, dass dieses Optimierungsproblem NP-schwierig ist.
Wir können Vertex Cover leicht als ganzzahliges lineares Programm formulieren, es ist die Zielfunktion $\sum_{v \in V} c(v) \cdot x_v$ zu minimieren unter den Nebenbedingungen $\sum_{v \in e} x_v \geq 1$ für alle $e \in E$ und $x_v \in \{0, 1\}$ für alle $v \in V$. Wenn wir die letzten Nebenbedingungen in $x_v \geq 0$ ändern, erhalten wir als relaxiertes Problem ein lineares Programm. Wir können dazu das duale Problem formulieren, es ergibt sich als Maximierung von $\sum_{e \in E} y_e$ unter den Nebenbedingungen $\sum_{e=\{u,v\} \in E} y_e \leq c(v)$ für alle $v \in V$ und $y_e \geq 0$ für alle $e \in E$.
Wir betrachten eine zulässige Lösung y des dualen Problems. Der Wert dieser Lösung y beträgt $Z'(y) := \sum_{e \in E} y_e$. Wir wissen, dass der Wert einer optimalen Lösung für das primale Problem durch diesen Wert nach unten beschränkt ist (Theorem 14.36). Außerdem ist natürlich der Wert einer optimalen Lösung für das primale Problem eine untere Schranke für den Wert einer optimalen Lösung für Vertex Cover, weil ja das duale Problem Relaxierung von Vertex Cover ist. Folglich ist der Wert einer beliebigen zulässigen Lösung y des dualen Problems eine untere Schranke für den Wert einer optimalen Lösung von Vertex Cover.
Nehmen wir an, wir haben ein Vertex Cover V' mit $\sum_{v \in V} c(v) \leq \rho \cdot Z'(y)$ für irgendeinen Faktor ρ. Analog zur Ungleichungskette oben ergibt sich, dass $\sum_{v \in V} c(v) \leq \rho \cdot \mathrm{OPT}$ gilt, wobei OPT den Wert einer optimalen Lösung für Vertex Cover bezeichnet. Ein solches Vertex Cover V' ist also eine ρ-Approximation. Wir benutzen das, um eine Art „Kochrezept“ für die Erstellung von Approximationsalgorithmen zu beschreiben.

1. Formuliere das Problem als ganzzahliges lineares Programm, formuliere seine Relaxierung P und das duale Problem P'.

2. Starte mit den Lösungen $x = 0$ und $y = 0$.
   Beobachte, dass $y = 0$ zulässig ist, $x = 0$ hingegen ist meist unzulässig.

3. Solange die primale Lösung x unzulässig ist,

(a) Erhöhe eine Variable y_i, bis eine duale Bedingung exakt erfüllt ist, (also $\sum_j y_i a_{i,j} = c_i$ gilt).
Beobachte, dass y dabei zulässig bleibt.

(b) Wähle solche exakt erfüllten dualen Bedingungen aus und erhöhe die entsprechende primale Variable ganzzahlig.

4. Beweise in der Analyse des Verfahrens, dass $c^T x \leq \rho \cdot y^T b$ gilt für ein möglichst kleines ρ.

In der Formulierung des Verfahrens gibt es natürlich viele Freiheiten, insbesondere zum vierten Schritt gibt es keinerlei Konkretisierung; das bedeutet keinesfalls, dass dieser Schritt einfach ist.
Wir wenden diesen Ansatz zur Konstruktion von Approximationsalgorithmen auf Vertex Cover an. Die Formulierung des ganzzahligen linearen Programms, seiner Relaxierung und des dualen Problems haben wir schon oben erledigt.

Algorithmus 14.39 (**Primal-dualer VC-Approximationsalgorithmus**).
Eingabe *ungerichteter gewichteter Graph* $G = (V, E, c)$
Ausgabe *Vertex Cover* M

1. $x := 0$; $y := 0$
2. *While* $E \neq \emptyset$
3. *Wähle* $E' \subseteq E$ *beliebig.*
4. *Erhöhe* y_e *für alle* $e \in E'$ *gleichförmig, bis* $\sum_{e=\{u,v\} \in E} y_e = c(v)$ *gilt für ein* $v \in V$.
5. $S := \left\{ v \in V \mid \sum_{e=\{u,v\} \in E} y_e = c(v) \right\}$
6. *Für alle* $v \in S$
7. $x_v := 1$
8. $E := E \setminus \{\{u, v\} \in E\}$
9. $M := \{v \in V \mid x_v = 1\}$
10. *Ausgabe* M

Wir machen uns zunächst klar, dass Algorithmus 14.39 überhaupt korrekt ist. Dazu zeigen wir, dass x und y stets zulässig sind. Danach machen wir uns an den vierten Schritt des „Kochrezepts“ und analysieren die Approximationsgüte.

Lemma 14.40. *Der primal-duale Approximationsalgorithmus für Vertex Cover (Algorithmus 14.39) berechnet x und y so, dass x zulässig für das lineare Programm P ist, das Relaxierung des ganzzahligen Programms zu Vertex Cover ist, und y zulässig für dessen duales Problem P'.*

Beweis. Wir sehen, dass eine Kante $\{u, v\}$ genau dann entfernt wird, wenn $x_u = 1$ oder $x_v = 1$ gilt. Der Algorithmus terminiert, wenn $E = \emptyset$ gilt, also ist x am Ende zulässig für P. Die Variable y_e wird explizit nur bis zur Grenze der Zulässigkeit erhöht, bei Erreichen dieser Grenze wird die zugehörige Kante e entfernt. Also bleibt y stets zulässig für P'. □

Theorem 14.41. *Der primal-duale Approximationsalgorithmus für Vertex Cover (Algorithmus 14.39) ist eine 2-Approximation.*

Beweis. Aus Lemma 14.40 folgt unmittelbar, dass M ein Vertex Cover ist. Wir müssen also nur noch die Approximationsgüte nachweisen. Dazu betrachten wir M und das zugehörige x, es gilt dafür $c^T x = \sum_{v \in M} c(v)$, dabei wird $c(v)$ im Algorithmus 14.39 so gewählt, dass

$$\sum_{v \in M} c(v) = \sum_{v \in M} \left(\sum_{e=\{u,v\}} y_e \right)$$

gilt. Wir können auch anders summieren, ohne die Summe zu verändern, damit erhalten wir

$$\sum_{v \in M} c(v) = \sum_{v \in M} \left(\sum_{e=\{u,v\}} y_e \right) = \sum_{e \in E} \left(\left(\sum_{v \in M \cap e} 1 \right) \cdot y_e \right).$$

Natürlich gilt $|e| = 2$, also wird für jede Kante e die Variable y_e höchstens zweimal addiert, wir erhalten also

$$\sum_{e \in E} \left(\left(\sum_{v \in M \cap e} 1 \right) \cdot y_e \right) \leq 2 \cdot \sum_{e \in E} y_e.$$

Wir haben jetzt insgesamt gezeigt, dass der Wert von x höchstens doppelt so groß ist wie der Wert von y, wir haben uns schon oben überlegt, dass daraus folgt, dass wir eine 2-Approximation garantieren können. □

Teil IV

Ausgewählte Themen

15 Hashing

Im Grundstudium in der Vorlesung „Datenstrukturen, Algorithmen und Programmierung 2“ ist das Wörterbuchproblem besprochen worden: Man möchte Daten, die aus einem Schlüssel und dem eigentlichen Datum so speichern, dass man Daten effizient speichern, effizient nach ihnen suchen und Daten auch wieder effizient löschen kann. Weil es offenbar nicht auf die Daten selbst ankommt, betrachtet man bei der Implementierung nur dem Umgang mit den Schlüsseln. Es gibt ganz verschiedene Lösungen zu diesem Problem, sie reichen von sehr primitiven Varianten (zum Beispiel lineare Listen), die sehr einfach zu implementieren, aber sehr langsam sind, zu aufwendigen Varianten (zum Beispiel AVL-Bäume), die gute Laufzeit auch im Worst Case garantieren, dafür aber erheblichen Aufwand bei der Implementierung verursachen. In gewisser Weise „dazwischen“ stehen randomisierte Varianten, die mit mäßigem Aufwand zumindest im Erwartungswert sehr effizient sind, Beispiele sind Skiplisten und Hashing. Das Thema Hashing wollen wir hier noch einmal aufnehmen und vertiefen.

Wir beginnen mit einer Wiederholung bzw. Festlegung der Notation. Wir betrachten Schlüssel aus einem festen *Universum* $\mathcal{U}$, wir haben also eine *Schlüsselmenge* $S \subseteq \mathcal{U}$, dabei ist $|\mathcal{U}|$ sehr groß, $|S|$ ist hingegen von handhabbarer Größe $n = |S|$, es gilt also $|S| \ll |\mathcal{U}|$. Beim Hashing speichern wir die Schlüssel in einem Array der Größe M, das wir *Hashtabelle* nennen. Eine *Hashfunktion* h ist eine Abbildung aus dem Universum auf Arrayplätze, also $h\colon \mathcal{U} \to \{0, 1, \ldots, M-1\}$. Natürlich muss das Array handhabbare Größe haben, es gilt also auch $M \ll |\mathcal{U}|$. Wir brauchen offenbar $M \geq |S|$, andernfalls ist ja gar nicht für alle Schlüssel Platz im Array. Andererseits soll das Array auch nicht zu groß sein, es soll ja kein Speicherplatz unnötig benutzt werden. Wir benutzen als Kenngröße für das Verhältnis zwischen Anzahl der Schlüssel und Arraygröße den *Lastfaktor* $\alpha = n/M$ und möchten nicht zu große Lastfaktoren haben, typisch ist $1/\alpha = O(1)$.

Wenn jeder Schlüssel $x \in S$ seinen eigenen Platz im Array bekommt, ist Hashing ganz einfach. Schwierig wird es durch sogenannte *Kollisionen*, verschiedenen Schlüssel, die den gleichen Arrayplatz beanspruchen, also $x, y \in \mathcal{U}$ mit $h(x) = h(y)$. Weil $|\mathcal{U}| \gg n$ gilt, sind Kollisionen unvermeidlich. Wenn wir keine Annahmen über die Schlüsselmenge S machen, gilt das auch bei Einschränkung auf $x, y \in S$. Aber sogar bei zufälliger Schlüsselmenge sind Kollisionen wahrscheinlich, wir kennen das vom Geburtstagsparadoxon, bei dem die Größe der Klasse $n = |S|$ entspricht und die Größe des Arrays $M = 365$ ist, das erstaunliche Resultat lautet dort, dass es schon bei 23 Kin-

dern in einer Klasse eher wahrscheinlich ist, dass es eine „Kollision“ gibt, die genaue Wahrscheinlichkeit beträgt $1 - (M!)/(M^n \cdot (M-n)!)$, wie man sich leicht überlegt.
Wir müssen also mit Kollisionen umgehen, dafür gibt es zwei grundsätzlich verschiedene Möglichkeiten: Beim *offenen Hashing* verwalten wir in jeder Arrayposition der Hashtabelle eine lineare Liste und speichern in ihr alle Schlüssel, die auf diese Position abgebildet werden. Beim *geschlossenen Hashing* hat jede Arrayposition nur Platz für ein Element, im Fall einer Kollision muss für später kommende Schlüssel ein anderer Platz gefunden werden. Es gibt dazu eine ganze Reihe von Möglichkeiten, als Beispiele seien lineares Sondieren, quadratisches Sondieren, multiplikatives Sondieren und doppeltes Hashing genannt. Weil all diese Techniken für geschlossenes Hashing schon im Grundstudium diskutiert wurden, wollen wir das hier nicht weiter vertiefen. Wir erinnern uns daran, dass die durchschnittliche Performanz von Hashing sehr gut ist, der Worst Case aber extrem schlecht. Dabei nehmen wir Durchschnitt und Worst Case über die Schlüsselmenge, also über unsere Eingabe. Wir machen uns zur Wiederholung kurz klar, warum das so ist und betrachten dazu offenes Hashing. Mit B_i bezeichnen wir die Elemente, die in Position i der Hashtabelle gespeichert werden, also $B_i = \{x \in S \mid h(x) = i\}$. Die Anzahl dieser Elemente sei b_i, es ist also $b_i = |B_i|$. Im ungünstigsten Fall gilt natürlich $h(x) = h(y)$ für alle $x, y \in S$, dann ist brauchen wir Zeit $O(n)$ für eine Operation. Für den Average Case machen wir eine weitreichende *Uniformitätsannahme*, wir nehmen an, dass für $x \in S$ die Hashwerte $h(x)$ uniform zufällig verteilt sind. Diese Annahme ist bei zufällig gewählten Schlüssel gerechtfertigt, andernfalls (also insbesondere in realen Anwendungen) ist sie fragwürdig.
Unter dieser Uniformitätsannahme wollen wir die erwartete Zeit für eine erfolglose und eine erfolgreiche Suche abschätzen. Bei der erfolglosen Suche berechnen wir $h(x)$ (für ein $x \notin S$) und durchsuchen dann $B_{h(x)}$. Wir schreiben nun kurz $i := h(x)$ und sehen, dass die erwartete Zeit $\Theta(1 + \mathrm{E}\,(b_i))$ beträgt. Dank Uniformitätsannahme gilt $\mathrm{E}\,(b_i) = n/M = \alpha$, wir haben also erwartete Zeit $\Theta(1 + \alpha)$ für eine erfolglose Suche. Eine erfolgreiche Suche könnte schneller sein, weil nicht unbedingt ganz B_i durchsucht werden muss. Für die Abschätzung der erwarteten Dauer einer erfolgreichen Suche nehmen wir zusätzlich an, dass jeder Schlüssel $x \in S$ mit gleicher Wahrscheinlichkeit gesucht wird. Wir müssen alle Elemente in B_i durchsuchen, die vor dem gesuchten Element $x \in S$ in B_i stehen, das sind bei der üblichen Implementierung linearer Listen alle die Elemente, die später als x_i in B_i eingefügt worden sind. Wir definieren eine Indikatorvariable $X_{i,j} = 1$, wenn $h(x_i) = h(x_j)$ gilt und $X_{i,j} = 0$ sonst. Unsere Uniformitätsannahme liefert, dass für alle $x_i \neq x_j \in S$ stets $\mathrm{Prob}\,(X_{i,j} = 1) = 1/M$ gilt. Wir erinnern daran, dass für Indikatorva-

riable $X_{i,j}$ natürlich $\mathrm{E}(X_{i,j}) = \mathrm{Prob}(X_{i,j} = 1)$ gilt. Mit dieser Notation und Einsicht ergibt sich für die erwartete Länge einer erfolgreichen Suche

$$\begin{aligned} & 1 + \mathrm{E}\left(\sum_{i=1}^{n} \frac{1}{n} \cdot \sum_{j=i+1}^{n} X_{i,j}\right) = 1 + \frac{1}{n} \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathrm{E}(X_{i,j}) \\ = \;& 1 + \frac{1}{n} \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{1}{M} = 1 + \frac{1}{M \cdot n} \sum_{i=1}^{n} (n - i + 1) \\ = \;& 1 + \frac{1}{M \cdot n} \sum_{i=1}^{n-1} i = 1 + \frac{n(n-1)}{2Mn} = 1 + \frac{n-1}{2M} = \Theta(1 + \alpha). \end{aligned}$$

Wir brauchen also für die Suche Zeit $\Theta(1 + \alpha)$, bei vernünftiger Wahl des Lastfaktors also konstante Zeit. Die Überlegungen setzen voraus, dass unsere Schlüssel gut verteilt sind, was wir in Anwendungen nicht annehmen dürfen. Wir werden darum wie häufig unser Vorgehen randomisieren und uns so gegen den Worst Case wappnen, verlagern in gewisser Weise den Zufall von unseren Eingangsdaten in unseren Algorithmus. Wir wählen die Hashfunktion zufällig.

Wir setzen voraus, dass $|\mathcal{U}|$ endlich ist, natürlich ist dann auch $|S|$ endlich. Eine einfache Idee ist, eine Hashfunktion $h \in \{f : \mathcal{U} \to \{0, 1, \ldots, M-1\}\}$ uniform zufällig zu wählen. Leider ist das Vorgehen nicht praktikabel, es gibt $M^{|\mathcal{U}|}$ solcher Hashfunktionen, zur Repräsentation sind dann $\Omega(|\mathcal{U}| \log M)$ Bits erforderlich, das ist viel zu viel. Wir müssen uns mit der zufälligen Auswahl aus einer kleineren Menge begnügen.

Was erhoffen wir uns eigentlich bei zufälliger Wahl von $h \in \mathcal{H}$ für eine Klasse von Hashfunktionen $\mathcal{H}$? Unsere Uniformitätsannahme hatte für $x \neq y$ stets $\mathrm{Prob}(h(x) = h(y)) = 1/M$ geliefert. Von dieser günstigen Eigenschaft möchten wir uns nicht zu weit entfernen. Wir erlauben eine Abweichung um einen konstanten Faktor c und kommen damit zur Definition c-universeller Hashklassen.

Definition 15.1. *Eine Hashklasse $\mathcal{H} \subseteq \{h : \mathcal{U} \to \{0, 1, \ldots, M-1\}\}$ heißt* c-universell*, wenn bei uniform zufälliger Wahl von $h \in \mathcal{H}$ für beliebige Schlüssel $x \neq y \in \mathcal{U}$*

$$\mathrm{Prob}(h(x) = h(y)) \leq \frac{c}{M}$$

gilt.

Wir wünschen uns also c-universelle Hashklassen mit kleinem c, am schönsten wäre eine 1-universelle Klasse. Es fragt sich, ob man praktisch handhabbare Klassen findet, die diese Eigenschaft haben. Wir widmen uns dieser Frage

und werden, um derart konkret werden zu können, Annahmen über die Struktur unserer Schlüssel machen. Weil wir letztlich annehmen dürfen, dass die Schlüssel $x \in S \subseteq \mathcal{U}$ binär codiert sind und wir jede Binärcodierung auch als natürliche Zahl interpretieren können (wie man das schon im ersten Semester in „Rechnerstrukturen“ gesehen hat), können wir ohne Beschränkung der Allgemeinheit annehmen, dass $\mathcal{U} = \{0, 1, 2, \ldots, |\mathcal{U}| - 1\}$ gilt. Wir wählen nun eine Primzahl $p > |\mathcal{U}|$ (aber nicht viel größer) und erlauben uns, mod p zu rechnen. Damit können wir jetzt eine Klasse von Hashfunktionen definieren.

Definition 15.2. *Sei* $\mathcal{U} = \{0, 1, 2, \ldots, |\mathcal{U}| - 1\}$, $p \geq |\mathcal{U}|$. *Die Klasse*

$$\mathcal{H}_l := \{h_{a,b} \mid 0 < a < p, 0 \leq b < p\}$$

ist eine Klasse von Hashfunktionen mit $h_{a,b}(x) := ((a \cdot x + b) \bmod p) \bmod M$.

Die Klasse von $\mathcal{H}_l$ ist offenbar effizient handhabbar und die Hashfunktionen sind effizient berechenbar. Außerdem ist sie in Bezug auf c-Universalität erstaunlich leistungsstark.

Theorem 15.3. $\mathcal{H}_l$ *ist 1-universell.*

Beweis. Wir betrachten $x \neq y \in \mathcal{U}$ beliebig. Es seien $a \in \{1, 2, \ldots, p-1\}$ und $b \in \{0, 1, \ldots, p-1\}$ unabhängig uniform zufällig gewählt. Weil p prim ist, rechnen wir in $\mathbb{Z}_p^*$. Wir wenden den chinesischen Restsatz an (Theorem 12.35) und sehen, dass $x' \neq y'$ für

$$\begin{aligned} x' &:= (ax + b) \bmod p \\ y' &:= (ay + b) \bmod p \end{aligned}$$

gilt. Also ist $h_{a,b}(x) = h_{a,b}(y)$ genau dann, wenn $x' \bmod M \equiv y' \bmod M$ für $x' \neq y'$ gilt. Natürlich gilt

$$\forall i \in \mathcal{U}\colon\ |\{j \in \mathcal{U} \mid i \equiv j \bmod M\}| \leq \left\lceil \frac{|\mathcal{U}|}{M} \right\rceil$$

und wir sehen, dass

$$\text{Prob}\left(h_{a,b}(x) = h_{a,b}(y)\right) \leq \frac{\lceil |\mathcal{U}| / M \rceil - 1}{|\mathcal{U}| - 1} \leq \frac{\lceil (|\mathcal{U}| - 1)/M \rceil}{|\mathcal{U}| - 1} = \frac{1}{M}$$

gilt. □

Wenn man bei offenem Hashing die erwartete Zeit für erfolgreiche und erfolglose Suche bei Verwendung uniform aus einer c-universellen Hashklasse gewählten Hashfunktion abschätzt, wiederholen sich die eben durchgeführten Rechnungen für die Abschätzung, allerdings ändert sich die Interpretation der Wahrscheinlichkeiten und Erwartungswerte: Waren oben alle zufälligen Ereignisse auf die zufällige Wahl der Schlüsselmenge bezogen, ist jetzt die Wahl der Schlüsselmenge beliebig und wir beziehen uns auf die zufällige Wahl der Hashfunktion. Abgesehen davon zieht nur der Faktor c konstant durch die Rechnungen, so dass wir das Ergebnis festhalten können, ohne den vollständig analog verlaufenen Beweis zu wiederholen.

Theorem 15.4. *Die erwartete Suchzeit bei uniform zufälliger Wahl einer Hashfunktion $h \in \mathcal{H}$ aus einer c-universellen Klasse von Hashfunktionen $\mathcal{H}$ beträgt bei erfolgreicher und erfolglose Suche $\Theta(1+\alpha)$ bei offenem Hashing mit Lastfaktor α.*

Wir haben bis jetzt vorausgesetzt, dass $n = |S|$ vorab bekannt ist: Nur so kann M passend gewählt werden, was für die Festlegung der Hashfunktion vorab erforderlich ist. In Anwendungen wird n nicht vorab bekannt sein, man muss sich also anders verhalten. Dazu wird man vorab einen gewünschten Lastfaktor α' fixieren, eine beliebige nicht zu große initiale Hashtabellengröße M wählen und dann über den tatsächlichen Lastfaktor α Buch führen. Gilt $\alpha > \alpha'$, so wird eine neue Hashtabelle doppelter Größe angelegt und alle Werte werden in die neue Hashtabelle (natürlich unter neuer Hashfunktion) übertragen, wir nennen das *Rehashing*. Ist n die maximale Anzahl von Schlüsseln, so kommen wir offenbar mit Platz $O(n)$ aus. Eine einzelne Insert-Operation kann durch das Rehashing jetzt zwar sehr langwierig werden, man macht sicher aber leicht (und mit Hilfe amortisierter Analyse auch formal) klar, dass man k Operationen in Zeit $\Theta(k)$ ausführen kann. Schwankt die Anzahl der Schlüssel stark, kann es sinnvoll sein, die Hashtabellengröße auch wieder zu verkleinern. Das kann man zum Beispiel machen, wenn $2\alpha < \alpha'$ gilt. Man überlege sich auch hier, warum die amortisierte Rechenzeit je Operation weiterhin konstant ist, das aber nicht mehr gälte, wenn man $\alpha < \alpha'$ zum Anlass nähme, die Hashtabelle wieder zu verkleinern.

15.1 Statisches perfektes Hashing

Das erwartete gute Laufzeitverhalten bei Benutzung einer uniform zufällig gewählten Hashfunktion aus einer c-universellen Klasse von Hashfunktionen

ist mit dem Makel behaftet, nur erwartet zu sein. In zeitkritischen Anwendungen kann das zu wenig sein, Situationen, in denen Laufzeitgarantien gefragt sind, sind vorstellbar. Man kann sich vorstellen, dass auch im Worst Case Ausführungszeit $O(1)$ gefordert wird. Bevor wir uns überlegen, wie man das realisieren kann, wollen wir diskutieren, was das eigentlich genau bedeutet.

Bei Ausführungszeit $O(1)$ im Average Case muss die Ausführungszeit im Durchschnitt $O(1)$ sein, für einzelne Operationen gibt es keine Zeitschranken, allerdings müssen zeitaufwendige Operationen so selten sein, dass der Durchschnitt noch konstant ist. Das gilt aber im Durchschnitt über die zufällige Wahl der Hashfunktion; ist die Hashfunktion unglücklich gewählt, kann es auch wesentlich länger dauern. Davon zu unterscheiden ist eine amortisierte Worst-Case-Rechenzeit von $O(1)$: Auch hier haben wir für eine einzelne Operation keine gute obere Schranke, allerdings müssen wiederum zeitaufwendige Operationen so selten sein, dass der Durchschnitt noch konstant ist. Jetzt nehmen wir aber den Durchschnitt über alle Operationen, die Aussage gilt also für jede Wahl der Hashfunktion. Noch stärker ist die Forderung nach Ausführungszeit $O(1)$ im Worst Case: Hier muss für jede Wahl der Hashfunktion und jede Operation die Ausführungszeit für eine Konstante nach oben beschränkt sein. Es ist nicht a priori klar, dass das überhaupt möglich ist.

Wir werden uns hier mit *statischem perfekten Hashing* beschäftigen: Die Schlüsselmenge S liegt fest, die Schlüssel müssen nur einmal in die Hashtabelle eingefügt werden, danach erfolgen ausschließlich Search-Operationen. Nachträgliche Einfügungen oder nachträgliches Löschen ist ausdrücklich ausgeschlossen. Diese Voraussetzungen scheinen sehr eingeschränkt zu sein, es gibt aber durchaus Anwendungsszenarien: Wenn man zum Beispiel eine große Datensammlung auf einer DVD speichern will, kann man den Wunsch haben, dass Benutzern der DVD eine besonders schnelle Suchfunktion zur Verfügung steht. Der zeitliche Aufwand bei der Erstellung der DVD entsteht nur einmal und darf ruhig etwas größer sein. Wir verlangen also bei statischem perfekten Hashing, dass der Aufbau der Hashtabelle in erwarteter Polynomialzeit geschieht, dass dann in der Anwendung jedes Suchen auch im Worst Case in Zeit $O(1)$ funktioniert.

Die Idee zur Realisierung ist so einfach wie naheliegend: Wir müssen das Hashing so organisieren, dass es keine Kollisionen gibt. Man kann das erreichen, indem man die Größe der Hashtabelle M groß genug wählt. Wir zeigen exemplarisch, dass quadratische Größe ausreicht.

Lemma 15.5. *Beim Hashen von n Schlüsseln in eine Hashtabelle der Größe $M := n^2$ mit einer uniform zufällig gewählten Hashfunktion $h \in \mathcal{H}_l$ gibt es mit Wahrscheinlichkeit mindestens $1/2$ keine Kollisionen.*

Beweis. Es gibt eine Kollision, wenn es ein Paar $(x, y) \in S^2$ mit $x \neq y$ gibt, so dass $h(x) = h(y)$ gilt. Es gibt $\binom{n}{2}$ solcher Paare, für jedes Paar beträgt die Kollisionswahrscheinlichkeit höchsten $1/M$, weil $\mathcal{H}_l$ 1-universell ist (Theorem 15.3). Also ist die erwartete Anzahl der Kollisionen durch $\binom{n}{2} \cdot (1/M) = n(n-1)/(2n^2) < 1/2$ nach oben beschränkt. Die Anzahl der Kollisionen ist offenbar eine nichtnegative Zufallsvariable, wir können also die Markowungleichung anwenden und sehen, dass die Wahrscheinlichkeit, mindestens eine Kollision zu haben, durch $1/2$ nach oben beschränkt ist. Darum gibt es mit Wahrscheinlichkeit mindestens $1/2$ gar keine Kollision. □

Eine Hashtabelle quadratischer Größe ist für Anwendungen viel zu groß: Wir lassen nur linearen Platzbedarf zu, wollen also mit Platz $O(n)$ für die Hashtabelle auskommen. Natürlich sagt Lemma 15.5 nicht aus, dass kleine Hashtabellen nicht vielleicht auch schon reichen können. Die Abschätzungen sind auch einigermaßen grob, die Markowungleichung ist nicht besonders scharf. Man macht sich aber leicht klar, dass bei wesentlich kleineren Hashtabellen Kollisionen mit großer Wahrscheinlichkeit auftreten werden. Bei einer Hashtabelle der Größe M haben wir unter $\binom{n}{2}$ Paaren nur mit Wahrscheinlichkeit $(1 - 1/M)^{\binom{n}{2}}$ gar keine Kollision. Wir rechnen

$$\left(1 - \frac{1}{M}\right)^{\binom{n}{2}} \leq e^{-\binom{n}{2}/(M-1)} = e^{-\Theta\left(n^2/M\right)}$$

und sehen, dass bei Hashtabellengröße $M = o\left(n^2/\log n\right)$ Kollisionen sehr wahrscheinlich sind. Wir müssen also etwas geschickter vorgehen, als nur die Hashtabelle zu vergrößern.
Wenn wir die Hashtabellengröße von M' auf $k \cdot M'$ vergrößern, so kann man das so interpretieren, dass wir für jeden Hashwert einen Platz der Größe k anlegen, in dem in konstanter Zeit gesucht werden kann. Weil wir uns beim Aufbau der Datenstruktur durchaus etwas Zeit lassen dürfen und die Schlüsselmenge feststeht, könnten wir aber besser adaptiv vorgehen: Wir durchlaufen einmal die Schlüsselmenge und zählen für jeden Platz in der Hashtabelle der Größe $M' := n$, wie viele Schlüssel auf diesen Platz abgebildet werden, wir bestimmen also b_i für alle $i \in \{0, 1, \ldots, n-1\}$. Dann legen wir eine zweistufige Hashtabelle an, in der ersten Stufe haben wir eine Tabelle der Größe $M' = n$, für jedes Feld dieser Hashtabelle haben wir ein eigene Hashtabelle, dabei hat die Hashtabelle des i-ten Feldes Größe b_i^2. Wir wissen, dass wir im Erwartungswert dann in jeder dieser kleinen Hashtabellen keine Kollision haben. Wir fragen uns zunächst, wie groß die erwartete Größe der gesamten Datenstruktur bei diesem Vorgehen ist.

Lemma 15.6. *Sei $M = n$, $h \in \mathcal{H}_l$ uniform zufällig gewählt, $b_i := |\{x \in S \mid h(x) = i\}|$ für alle $i \in \{0, 1, \ldots, M-1\}$. Es gilt $E\left(\sum_{i=0}^{M-1} b_i^2\right) < 2n$.*

Beweis. Wir hashen insgesamt n Schlüssel in die Hashtabelle, jeder Schlüssel trägt genau 1 zu genau einem b_i bei, es gilt darum $\sum_{i=0}^{M-1} b_i = n$. Also ist $\mathrm{E}\left(\sum_{i=0}^{M-1} b_i\right) \leq \mathrm{E}(n) = n$.
Es gilt $a^2 = 2\binom{a}{2} + a$, darum (und wegen der Linearität des Erwartungswertes) gilt

$$\mathrm{E}\left(\sum_{i=0}^{M-1} b_i^2\right) = \mathrm{E}\left(\sum_{i=0}^{M-1} b_i\right) + \mathrm{E}\left(\sum_{i=0}^{M-1} 2\binom{b_i}{2}\right) = n + 2\mathrm{E}\left(\sum_{i=0}^{M-1} \binom{b_i}{2}\right)$$

und es genügt, $\mathrm{E}\left(\sum_{i=0}^{M-1} \binom{b_i}{2}\right)$ nach oben abzuschätzen. Offenbar bezeichnet $\sum_{i=0}^{M-1} \binom{b_i}{2}$ die Anzahl der Paare $x, y \in S$ mit $x \neq y$, für die $h(x) = h(y)$ gilt. Wir definieren wieder die Indikatorvariablen $X_{x,y}$ für $x \neq y$ mit

$$X_{x,y} := \begin{cases} 1 & \text{falls } h(x) = h(y), \\ 0 & \text{sonst} \end{cases}$$

und sehen, dass $\sum_{i=0}^{M-1} \binom{b_i}{2} = \sum_{x \neq y \in S} X_{x,y}$ gilt. Weil $\mathcal{H}_l$ 1-universell ist, gilt $\mathrm{E}(X_{x,y}) \leq 1/M$ (Theorem 15.3). Wir haben also

$$\mathrm{E}\left(\sum_{i=0}^{M-1} \binom{b_i}{2}\right) = \sum_{x \neq y \in S} \mathrm{E}(X_{x,y}) \leq \binom{n}{2} \cdot \frac{1}{M} = \frac{n(n-1)}{2n} < \frac{n}{2}$$

und insgesamt folgt

$$\mathrm{E}\left(\sum_{i=0}^{M-1} b_i^2\right) < n + 2 \cdot \frac{n}{2} = n.$$

□

Wir können jetzt einen Algorithmus zum Aufbau der Hashtabelle angeben und dann leicht nachweisen, dass wir so perfektes statisches Hashing realisieren. Es bezeichnet S die Menge der Schlüssel mit $|S| = n$, außerdem ist $M := n$ die Größe der Hashtabelle der ersten Stufe.

Algorithmus 15.7.

1. *Repeat*
2. *Wähle $h \in \mathcal{H}_l$ uniform zufällig.*
3. *Für alle $i \in \{0, 1, \ldots, M-1\}$*
   $b_i := 0$; $B_i := \emptyset$
4. *Für alle $x \in S$*
   $b_{h(x)} := b_{h(x)} + 1$; $B_i := B_i \cup \{x\}$
5. *Until* $\sum_{i=0}^{M-1} b_i^2 < 4n$.
6. *Für alle $i \in \{0, 1, \ldots, M-1\}$*
7. *Repeat*
8. *gut := true*
9. *Initialisiere leere Hashtabelle H_i der Größe b_i^2.*
10. *Wähle $h_i \in \mathcal{H}_l$ uniform zufällig.*
11. *Für alle $x \in B_i$*
    If $H_i[h_i(x)]$ belegt
    Then gut := false
    Else Speichere x in $H_i[h_i(x)]$.
12. *Until gut*

Theorem 15.8. *Algorithmus 15.7 konstruiert in erwarteter Zeit $O(n)$ eine Datenstruktur der Größe $O(n)$ mit zusätzlichem Speicherbedarf $O(n)$ und wählt dabei Hashfunktionen so, dass jede anschließende Search-Operation nur konstante Zeit braucht.*

Beweis. Wenn der Algorithmus stoppt, ist eine Datenstruktur konstruiert worden, für die jede Search-Operationen nur die Auswertung zweier Hashfunktionen erfordert, für Search(x) ist lediglich $h(x)$ und $h_{h(x)}(x)$ auszuwerten. Die Aussage für den Speicherplatz folgt direkt, da die Hashtabellen zusammen weniger als Platz $5n$ belegen und zusätzlich zu den $n+1$ Hashfunktionen nur n Elemente in den B_i gespeichert werden müssen. Es ist also nur die Aussage über die erwartete Laufzeit zu beweisen.
Wir betrachten zunächst die erste Phase (Zeilen 1–5). Wir wissen aus Lemma 15.6, dass $\mathrm{E}\left(\sum_{i=0}^{M-1} b_i^2\right) < 2n$ gilt. Weil $\sum_{i=0}^{M-1} b_i^2$ eine nichtnegative Zufallsvariable ist, können wir die Markowungleichung anwenden und sehen, dass die Wahrscheinlichkeit, dass die Repeat-Schleife wiederholt wird, durch $1/2$ nach oben beschränkt ist. Weil jeder Durchlauf in Zeit $O(n)$ durchführbar ist, folgt $O(n)$ als Schranke für die erwartete Länge der ersten Phase.
Für die zweite Phase (Zeilen 6–12) betrachten wir einen Durchlauf der Repeat-Until-Schleife. Wir wissen, dass mit Wahrscheinlichkeit mindestens $1/2$ am Ende gut = true gilt (Lemma 15.5). Also ist die erwartete Gesamtdauer dieser

Schleife $O(b_i^2)$. Wir haben also als erwartete Gesamtdauer der zweiten Phase $O\left(\sum_{i=0}^{M-1} b_i^2\right)$ und starten in die zweite Phase nur dann, wenn $\sum_{i=0}^{M-1} b_i^2 < 4n$ gilt. Folglich hat auch die zweite Phase erwartete Gesamtdauer $O(n)$. □

Man kann diese Strategie auch zu dynamisch perfektem Hashing ausbauen. Bei dynamisch perfektem Hashing fordern wir eine Worst-Case-Rechenzeit für Search von $O(1)$ und eine konstante amortisierte erwartete Rechenzeit für alle übrigen Operationen. Man passt dafür die Gesamttabellengröße dynamisch an, wie wir das am Ende der Einleitung besprochen haben. Wenn die Hashfunktionen ungünstig waren (oder die Schlüssel ungünstig sind – das ist nur eine Frage der Perspektive), führen wir ein vollständiges Rehash durch, was natürlich zeitaufwendig ist. Ebenso kommt es zu einem vollständigen Rehash nach n verändernden Zugriffen, damit uns die Änderungen an den Hashtabellen nicht schleichend die Performanz verschlechtern. Man müsste jetzt die zu wählenden Werte konkretisieren und sorgfältig nachrechnen, dass alles passt. Wir verzichten hier darauf und betrachten lieber einen wesentlich einfacheren modernen Ansatz, der ähnlich gute Performanz verspricht.

15.2 Cuckoo-Hashing

Wir werden in diesem Abschnitt wieder davon ausgehen, dass die Anzahl der Schlüssel $n = |S|$ vorab bekannt ist. Sollte das nicht der Fall sein, kann wieder die bereits mehrfach skizzierte Verdoppelungsstrategie für die Größe der Hashtabellen benutzt werden. Wir werden als Hashtabellengröße $M := (1 + \varepsilon) \cdot n$ verwenden, dabei ist $\varepsilon > 0$ konstant. Neu ist, dass wir nicht mehr eine, sondern jetzt zwei Hashtabellen verwenden, die beide Größe M haben werden. Unser Lastfaktor α wird also kleiner als $1/2$ sein. Wenn wir an die Größe der Datenstruktur für statisches perfektes Hashing denken (dort war der Auslastungsfaktor $\geq 1/4$), ist das absolut akzeptabel. Für jede der beiden Hashtabellen T_1 und T_2 gibt es eine eigene Hashfunktion, wir verwenden also zwei Hashfunktionen h_1, h_2, die wir geeignet zufällig auswählen. Wir diskutieren die Details dazu später.
Mit diesen zwei Hashtabellen lässt sich Search(x) realisieren, indem in $T_1[h_1(x)]$ und $T_2[h_2(x)]$ nachgesehen wird. Ist x dort nicht zu finden, war die Suche erfolglos, andernfalls natürlich erfolgreich. Die Operation „Delete“ lässt sich wie üblich durch eine erfolgreiche Suche und anschließende Markierung des Feldes als „gelöscht“ realisieren. Eigentlich interessant ist vor allem das Einfügen von Daten. Wir beschreiben dazu einen Algorithmus, der zusätzlich zu x noch

einen Parameter $k_{\max}$ hat, den wir später global für alle Insert-Operationen gleich festlegen.

Insert(x)
1. Search(x); If vorhanden, STOP.
2. Für $kin\{1, 2, \ldots, k_{\max}\}$
3. Vertausche x und den Inhalt von $T_1[h_1(x)]$.
4. If x leer Then STOP.
5. Vertausche x und den Inhalt von $T_2[h_2(x)]$.
6. If x leer Then STOP.
7. RehashAll
8. Insert(x)

Die Funktion RehashAll (Zeile 2) macht genau das, was ihr Name befürchten lässt: Es werden neue Hashfunktionen h_1, h_2 zufällig gewählt und damit die Hashtabellen neu gefüllt. Das funktioniert offenbar in Zeit $O(n)$. Wir werden außer bei der Ausführung von Insert noch in einer zweiten Situation Gebrauch von RehashAll machen: Wir zählen die Anzahl insgesamt durchgeführter Insert-Operationen und werden jeweils nach M^2 Inserts RehashAll durchführen.
Natürlich hängt die Performanz entscheidend von den Eigenschaften der Hashfunktionen ab, die erwartete Performanz hängt also entscheidend von den Eigenschaften der Hashklasse ab, aus der die Hashfunktionen uniform zufällig gewählt werden. Wir führen zu diesem Zweck den Begriff der (c, k)-universellen Hashklasse ein.

Definition 15.9. *Eine Hashklasse $\mathcal{H} \subseteq \{h\colon \mathcal{U} \to \{0, 1, \ldots, M-1\}\}$ heißt* (c, k)-universell*, wenn bei uniform zufälliger Wahl von $h \in \mathcal{H}$ für alle paarweise verschiedenen Schlüssel $x_1, x_2, \ldots, x_k \in \mathcal{U}$ und für alle Hashwerte $y_1, y_2, \ldots, y_k \in \{0, 1, 2, \ldots, M-1\}$*

$$Prob\left(\bigwedge_{i=1}^{k} h(x_i) = y_i\right) \leq \frac{c}{M^k}$$

gilt.

Offensichtlich ist (c, k)-universell eine Verallgemeinerung von c-universell, insbesondere bezeichnet $(c, 2)$-universell genau c-universell. Man macht sich analog zur 1-Universalität von $\mathcal{H}_l$ leicht klar, dass die Klasse der h mit

$$h(x) := \left(\sum_{i=0}^{k-1} \left(a_i \cdot x^i\right) \bmod p\right) \bmod M$$

mit $a_i \in \{0, 1, \ldots, p-1\}$ für alle $i \in \{0, 1, \ldots, k-1\}$ $(2, k)$-universell ist. Wir halten außerdem als Fakt fest, dass es eine Klasse von Hashfunktionen gibt, deren Funktionen mit $o(n)$ Wörtern beschreibbar sind, deren Funktionen in konstanter Zeit auswertbar sind und die eingeschränkt auf M^2 Schlüssel mit Wahrscheinlichkeit $1 - O(1/n^2)$ für eine Konstante $\delta > 0$ $(1, n^\delta)$-universell ist. Wir werden $k_{\max} \leq n^\delta$ wählen. Weil wir nach M^2 Insert-Operationen RehashAll durchführen, haben wir also mit Wahrscheinlichkeit $1 - O(1/n^2)$ eine für unsere Zwecke gute Hashfunktion, die sich wie eine ideale (uniform zufällig aus allen Hashfunktionen gewählte) Hashfunktion verhält.
Für die Analyse der Ausführungszeiten ist entscheidend, was bei einem Insert passiert. Die Kosten der anderen Operationen sind $O(1)$, die amortisierten Kosten für das RehashAll nach M^2 Insert-Operationen betragen $O(n/M^2) = O(1/n)$, fallen also nicht ins Gewicht.
Beim Cuckoo-Hashing wird ein Schlüssel x_2 vom neuen Schlüssel x_1 aus seinem „Nest“ verdrängt, dieser Schlüssel verdrängt dann seinerseits einen Schlüssel x_3 aus seinem „Nest“, dieser verdrängt einen Schlüssel x_4 und so fort, bis entweder ein leeres Feld erreicht wird oder x_i mit $i = k_{\max}$ verdrängt wurde. Problematisch ist, dass die verdrängten Schlüssel x_i nicht paarweise verschieden sein müssen. Es ist durchaus möglich, dass bei diesem Prozess ein geschlossener Kreis durchlaufen wird und es mit Sicherheit erst nach $k_{\max}$ Schritten zum Abbruch kommt. Von solchen geschlossenen Kreisen verschieden sind Kreise, in denen zwar ein Element mehrfach in der Folge der verdrängten Schlüssel auftaucht, es danach aber anders weitergeht. Bei der Abschätzung der erwarteten Dauer hilft uns das folgende Lemma, das uns etwas über die Struktur dieser Folgen von Schlüsseln verrät.

Lemma 15.10. *Sei $x_1, x_2, \ldots$ eine Sequenz von Schlüsseln bei Insert, die nicht Teil eines geschlossenen Kreises sind. In jedem Präfix $x_1, x_2, \ldots, x_p$ dieser Folge gibt es eine ununterbrochene Teilfolge von mindestens $p/3$ paarweise verschiedenen Schlüsseln, die mit x_1 beginnt.*

Beweis. Falls in der Sequenz $x_1, x_2, \ldots$ alle x_i paarweise verschieden sind, dann ist die Sequenz $x_1, x_2, \ldots, x_p$ selbst eine Sequenz mit den geforderten Eigenschaften. Wir können also im Folgenden davon ausgehen, dass es Indizes $i < j$ gibt, so dass $x_i = x_j$ gilt. Es seien i und j so gewählt, dass die Elemente $x_1, x_2, \ldots, x_{j-1}$ alle paarweise verschieden sind. Wir unterscheiden zwei Fälle, sei zunächst $p < i+j$. Weil $j \geq i+1$ gilt, ist $j-1 \geq (i+j-1)/2$. Aus $p < i+j$ folgt damit $j - 1 \geq p/2$. Also ist die Sequenz $x_1, x_2, \ldots, x_{j-1}$ eine Sequenz mit den geforderten Eigenschaften. Es bleibt noch der Fall, dass $p \geq i+j$ gilt. Wir betrachten die Sequenz $x_1, x_2, \ldots, x_j$. Wir wissen, dass für alle Indizes k in dieser Sequenz $h_{1+(k \bmod 2)}(x_k) = h_{1+(k \bmod 2)}(x_{k+1})$ gilt, andernfalls hätte diese Sequenz durch Verdrängung so nicht entstehen können. Nun führt uns

x_{j-1} wieder zu $x_i = x_j$. Wir probieren nun die andere Hashfunktion, die uns aber wieder zu x_{i-1} führt. Wir erreichen auf diese Weise wieder x_1 und wir probieren den anderen Hashwert von x_1. Dazu gibt es zwei Möglichkeiten. Entweder führt uns dieser Hashwert ebenfalls in eine Schleife, dann haben wir aber einen geschlossenen Kreis und die Bedingungen des Lemmas sind nicht erfüllt. Wir dürfen also voraussetzen, dass wir in der anderen Situation sind und die neue Folge endet mit einem Schlüssel x_l. Das bedeutet, das entweder die Sequenz $x_1, x_2, \ldots, x_{j-1}$ oder die Sequenz $x_{i+j-1}, x_{i+j}, \ldots, x_l$ die geforderte Eigenschaft hat. □

Für die erwartete Rechenzeit einer Insert-Anweisung interessiert uns, mit welcher Wahrscheinlichkeit bei einem Insert eine Sequenz verdrängter Schlüssel der Länge t vorkommt. Wir unterscheiden bezüglich der Werte von t vier Fälle.

1. **1. Fall: $t > k_{\max}$** Da Insert explizit bei Erreichen der Länge $k_{\max}$ abgebrochen wird, kann dieser Fall nicht eintreten, er hat also Wahrscheinlichkeit 0.

2. **2. Fall: $t \leq k_{\max}$ und eine Hashfunktion ist nicht $(1, k_{\max})$-universell** Wir können die Wahrscheinlichkeit abschätzen durch die Wahrscheinlichkeit, dass eine der beiden Hashfunktionen h_1, h_2 nicht $(1, k_{\max})$-universell ist. Wir hatten für jede Hashfunktion Wahrscheinlichkeit $O(1/n^2)$ als obere Schranke festgehalten, das gilt damit auch für beide Hashfunktionen.

3. **3. Fall: $t = k_{\max}$ und geschlossener Kreis aufgetreten** Diesen Fall werden wir diskutieren müssen.

4. **4. Fall: $t \leq k_{\max}$ und mindestens $(2t-1)/3$ verschiedene Schlüssel beginnend mit x_1 angefasst** Auch für diesen Fall werden wir die Wahrscheinlichkeit näher analysieren.

Lemma 15.10 garantiert, dass diese Fallunterscheidung erschöpfend ist. Wir müssen also jetzt noch die Wahrscheinlichkeiten für den dritten und vierten Fall abschätzen.
Wir beginnen mit dem dritten Fall und nennen das Element, das den Kreis schließt, x_l. Vorher werden $v \leq l$ verschiedene Schlüssel berührt. Wir zählen die Anzahl der Möglichkeiten, den Kreis so zu schließen, und verwenden dabei die Notation aus dem Beweis von Lemma 15.10: Es gibt höchstens v^2 Möglichkeiten, i und j zu wählen. Dann gibt es höchstens v_l Möglichkeiten, x_l zu positionieren. Für jede der $v-1$ übrigen Zellen gibt es M mögliche Positionen

in den Hashtabellen, so dass es hier insgesamt höchstens M^{v-1} Möglichkeiten gibt, schließlich können wir für jeden der Schlüssel einen von n Werten wählen, es gibt also dort höchstens n^{v-1} Möglichkeiten, so dass wir insgesamt $v^3 \cdot M^{v-1} \cdot n^{v-1}$ als obere Schranke für die Anzahl der Möglichkeiten, solche Kreise zu bilden, haben. Weil wir unter der Annahme operieren, dass h_1 und h_2 sich wie ideale Hashfunktionen verhalten, tritt jede dieser Möglichkeiten mit Wahrscheinlichkeit $1/(M^{2v})$ auf. Folglich erhalten wir

$$\sum_{v=3}^{l} \frac{v^3 \cdot M^{v-1} \cdot n^{v-1}}{M^{2v}} < \frac{1}{M \cdot n} \cdot \sum_{v=3}^{\infty} v^3 \cdot \left(\frac{n}{M}\right)^v = O\big(1/n^2\big)$$

als obere Schranke für die Wahrscheinlichkeit im dritten Fall.
Wir diskutieren nun noch den vierten Fall, in dem kein Kreis auftritt, insgesamt $t \leq k_{\max}$ Schlüssel berührt werden, davon sind mindestens $(2t-1)/3$ Schlüssel alle paarweise verschieden, der erste dieser Schlüssel ist der ursprünglich einzufügende Schlüssel x_1. Die mindestens $\lceil (2t-1)/3 \rceil$ verschiedenen Schlüssel seien $b_1, b_2, \ldots,$, dabei sei $b_1 = x_1$. Weil diese Schlüsselsequenz so aufgetreten ist, gilt $h_{\beta_1}(b_1) = h_{\beta_2}(b_2)$, $h_{\beta_2}(b_2) = h_{\beta_2(b_3)}$, $\ldots$, dabei ist $(\beta_1, \beta_2) \in \{(1,2),(2,1)\}$. Wie gesagt steht $b_1 = x_1$ fest, für die übrigen b_i gibt es n^{v-1} mögliche Schlüsselbelegungen. Wir haben wie vorhin für jede Möglichkeit Wahrscheinlichkeit $1/M^{v-1}$, so dass wir insgesamt

$$2 \cdot \left(\frac{n}{M}\right)^{v-1} \leq 2 \cdot (1+\varepsilon)^{-(2t-1)/3+1}$$

als obere Schranke für die Wahrscheinlichkeit erhalten.
Wir nennen die Länge einer Insert-Sequenz T und bestimmen den Erwartungswert dieser Zufallsvariablen. Mit den Abschätzungen für die vier Fälle haben wir

$$\begin{aligned}
\mathrm{E}\,(T) &= \sum_{t=1}^{\infty} t \cdot \mathrm{Prob}\,(T=t) = \sum_{t=1}^{\infty} \mathrm{Prob}\,(T \geq t) \\
&\leq 1 + \sum_{t=2}^{k_{\max}} \mathrm{Prob}\,(T \geq t) = 1 + \sum_{t=2}^{k_{\max}} \Big(2 \cdot (1+\varepsilon)^{-(2t-1)/3+1} + O\big(1/n^2\big)\Big) \\
&= 1 + O\left(\frac{k_{\max}}{n^2}\right) + 2\sum_{t=2}^{\infty} \Big((1+\varepsilon)^{-2/3}\Big)^t \\
&= O\left(1 + \frac{1}{1-(1+\varepsilon)^{-2/3}}\right) = O(1+1/\varepsilon)
\end{aligned}$$

gezeigt. Wir hatten schon erwähnt, dass die Zeit für ReshashAll $O(n)$ beträgt. Bei schlechter Wahl der Hashfunktionen kann das auftreten, das hat einen

Beitrag von $O(n) \cdot O(1/n^2) = O(1/n)$, den Beitrag für geschlossene Kreise können wir ebenfalls durch $O(n) \cdot O(1/n^2) = O(1/n)$ nach oben abschätzen, schließlich haben wir noch den Beitrag für „normale“ Insert-Operationen, die Insert-Sequenzen der Länge $k_{\max}$ erzeugen. Wir haben gesehen, dass das mit Wahrscheinlichkeit höchstens $2 \cdot (1+\varepsilon)^{-(2k_{\max}-1)/3+1}$ auftritt. Wir entscheiden uns, $k_{\max} := \lceil 3 \log_{1+\varepsilon} n \rceil$ zu wählen, damit erhalten wir $O(1/n^2)$ als obere Schranke für die Wahrscheinlichkeit und wiederum $O(n) \cdot O(1/n^2) = O(1/n)$ als erwarteten Beitrag. Insgesamt haben wir also eine erwartete amortisierte Rechenzeit je Insert-Operation, die durch $O(1)$ nach oben beschränkt ist.
Wir haben damit insgesamt eine Worst-Case-Rechenzeit für Search und Delete, die konstant ist und eine erwartete amortisierte Rechenzeit für Insert, die ebenfalls konstant ist. Cuckoo-Hashing realisiert also in der Tat ebenfalls dynamisch perfektes Hashing.