Johannes Blömer, Bernd Gärtner

•

Skript zur Vorlesung

Approximationsalgorithmen

•

ETH Zürich

Sommersemester 1998

Inhaltsverzeichnis

1 Einleitung **5**
- 1.1 Das Job-Scheduling Problem 5
- 1.2 Notation 8
- 1.3 Das Max-Cut Problem 10
- 1.4 Uebungen 12

2 Eine kurze Einführung in NP-Vollständigkeit **13**
- 2.1 Uebungen 20

3 Elementare Approximationsalgorithmen **21**
- 3.1 Das Problem des Handlungsreisenden 21
- 3.2 Job-Scheduling 28
- 3.3 Uebungen 30

4 Approximationsschemata **31**
- 4.1 Das Rucksack-Problem 31
- 4.2 Approximationsschemata 36
- 4.3 Approximationsschema für Job-Scheduling 37
- 4.4 Uebungen 39

5 Max-SAT und Randomisierung **40**
- 5.1 Uebungen 44

6 LP-Relaxierungen **46**
- 6.1 Max-SAT und LP-Relaxierungen 46
- 6.2 Set Cover und LP-Relaxierungen 52
- 6.3 Eine verallgemeinerte Definition von Approximationsalgorithmen . 55
- 6.4 Ein LP-gestützter Approximationsalgorithmus für Job-Scheduling 58
- 6.5 Uebungen 60

7 Semidefinites Programmieren und Approximationsalgorithmen **62**
- 7.1 Semidefinites Programmieren und Max-Cut 62
- 7.2 Semidefinites Programmieren und Max-$\leq$ 2-SAT 68
- 7.3 Semidefinites Programmieren und Max-SAT* 71

7.4 Approximatives Färben von Graphen 74
7.5 Uebungen 79

8 Nichtapproximierbarkeitsresultate 81
8.1 Ein einfaches Nichtapproximierbarkeitsresultat 81
8.2 Lückenerhaltende Reduktionen 84
8.3 Das Oeffnen der Lücke 87
8.4 Uebungen 93

9 Musterlösungen zu den Uebungen 94

Vorwort

Seit längerer Zeit schon werden mathematische Probleme in der Informatik nach ihrem Schwierigkeitsgrad bezüglich algorithmischer Lösbarkeit klassifiziert. Die bekannteste Klassifizierung verläuft anhand der Komplexitätsklasse P, die in gewissem Sinne als Menge aller in polynomieller Zeit lösbarer Probleme definiert ist. Man sagt auch, Probleme in P seien "leicht". Schwere Problem sind dann Probleme, die nicht in P liegen–allerdings ist für viele Probleme gar nicht bekannt, ob sie zu P gehören oder nicht.
Besondere Aufmerksamkeit in dieser Richtung verdient die Klasse NP, die alle Probleme zusammenfasst, für die eine gegebene Lösung in polynomieller Zeit überprüft werden kann. Dies sieht nach einer etwas künstlichen Definition aus, es stellt sich aber heraus, dass viele natürlich formulierbare Probleme sofort in die Klasse NP eingeordnet werden können.
Insbesondere liegen alle Probleme aus P in NP; die wesentliche offene Frage ist aber, ob es Probleme in NP gibt, die schwer sind, also nicht in P liegen. Der Hintergrund dieser Frage ist, dass eine ganze Reihe von Problemen existieren, für die nie ein polynomieller Algorithmus gefunden werden konnte, so dass die Vermutung nahe liegt, dass ein solcher nicht existieren kann. Man ist weit davon entfernt, diese Vermutung für irgendeines dieser Probleme beweisen oder widerlegen zu können.
Die Frage ist insofern vereinfacht worden, dass man zumindest Probleme in NP identifiziert hat (die sogenannten NP-vollständigen Probleme), die schwer sein müssen, falls es überhaupt schwere Probleme in NP gibt. Bei der Suche nach schweren Problemen kann man sich also auf diese einschränken. Andererseits gilt, dass ein polynomieller Algorithmus für irgendein NP-vollständiges Problem polynomielle Algorithmen für *alle* Probleme in NP impliziert. Insofern bilden die NP-vollständigen Probleme die 'Essenz' von NP, und man vermutet von ihnen, dass sie schwer sind.
Möchte man ein NP-vollständiges (oder ein noch schwereres, sogenanntes NP-schweres) Problem in der Praxis lösen, kann man also nicht darauf hoffen, effiziente Algorithmen zu finden. In der Praxis kommen leider viele dieser Probleme vor; ein sehr bekanntes ist das Problem des Handlungsreisenden, der auf einer Rundreise n Städte besuchen möchte, dabei aber zur Minimierung der Reisekosten eine möglichst kleine Gesamtdistanz zurücklegen will.

Hier kommen Approximationsalgorithmen ins Spiel. Das sind effiziente Algorithmen für NP-schwere Probleme, die zwar keine exakten Lösungen berechnen, die Lösung aber so gut approximieren können, dass ihr Ergebnis in der Praxis verwendbar ist. So existiert etwa ein polynomieller Algorithmus, der das Problem des Handlungsreisenden bis auf einen beliebig klein wählbaren Fehler lösen kann. Oft reicht dies in der Anwendung völlig aus. Die vermutete Schwere des Problems liegt also nur im 'letzten ε', das die approximative von der optimalen Lösung unterscheidet.
Während die traditionelle Theorie nur zwischen leichten Problemen (in P) und NP-schweren Problemen unterscheidet, erhält man durch die Theorie der Approximationsalgorithmen feinere Abstufungen, je nachdem wie gut sich die Lösung eines Problems approximieren lässt. Kann man dies beliebig gut (wie im Fall des Handlungsreisenden), ist das Problem fast schon wieder leicht. Hingegen kann man andere Probleme angeben, die sich unter der Voraussetzung, dass sie schwer sind, nicht gut approximieren lassen.
Approximationsalgorithmen dürfen nicht mit Heuristiken verwechselt werden, mit denen ein Problem in der Praxis oft sogar schnell exakt gelöst werden kann. Für solche Heuristiken existieren meistens Eingaben, bei denen sie sehr schlecht abschneiden. Für Approximationsalgorithmen wollen wir *beweisbare* Qualitätsaussagen für *alle* Eingaben herleiten, etwa von der Form: Algorithmus A berechnet für jede Eingabe von n Städten eine Rundreise des Handlungsreisenden, die höchstens doppelt so lang ist wie die optimale Rundreise.
In Kapitel 1 werden wir zwei einfache NP-schwere Probleme angeben, für die beweisbar gute Approximationsalgorithmen existieren. Hier werden wir auch die grundlegenden Definition einführen.
Kapitel 2 wiederholt noch einmal die Konzepte NP, NP-Vollständigkeit und NP-Schwere.
In Kapitel 3 betrachten wir weitere elementare Approximationsalgorithmen, bevor wir in Kapitel 4 zu allgemeineren Konzepten und Techniken übergehen. Wir beginnen dabei mit Approximationsschemata, die uns eine formale Definition dafür liefern, dass ein Problem "beliebig gut" approximierbar ist.
Kapitel 5 betrachtet dann randomisierte (zufallsgesteuerte) Approximationsalgorithmen. Diese sind oft erstaunlich einfach und liefern gute Resultate.
Kapitel 6 und 7 führen in die wichtige Technik der Relaxierungen ein, wobei Kapitel 6 die schon traditionellen LP-Relaxierungen behandelt, während wir in Kapitel 7 die noch ziemlich neue Technik der Relaxierungen basierend auf semidefinitem Programmieren studieren.
In Kapitel 8 schliesslich leiten wir Nichtapproximierbarkeitsresultate her. Das sind Ergebnisse, die zeigen, dass bestimmte Probleme nicht beliebig gut approximiert werden können, sofern sie wirklich schwer sind. Wir verwenden dabei eine neue und sehr überraschende alternative Charakterisierung für NP.
Als "running example", das wir immer wieder benutzen, um Techniken zu erläutern, dient das *Job-Scheduling* Problem, mit dem wir auch in Kapitel 1 beginnen.

Kapitel 1

Einleitung

In diesem Abschnitt werden wir die wichtigsten Definitionen zusammenstellen, und diese an einigen einfachen Beispielen erläutern. Beginnen wollen wir mit dem Job-Scheduling Problem, eines der ersten Probleme überhaupt, die im Zusammenhang mit Approximationsalgorithmen behandelt wurden.

1.1 Das Job-Scheduling Problem

Problem 1.1.1 (Job-Scheduling) *Gegeben sind m Maschinen $M_1, \ldots, M_m$ und n Jobs $J_1, \ldots, J_n$, die auf diesen Maschinen erledigt werden sollen. Für jeden Job J_k ist eine Dauer $d_k > 0$ spezifiziert. Diese besagt, dass jede der Maschinen Zeit d_k benötigt, um den Job J_k auszuführen. Weiter haben wir die Bedingung, dass zu jedem Zeitpunkt jede Maschine nur einen Job ausführen kann, und dass ein einmal angefangener Job nicht abgebrochen werden kann. Gesucht ist dann eine Verteilung der Jobs auf Maschinen (ein Scheduling), die den oben genannten Bedingungen genügt, so dass alle Jobs möglichst schnell ausgeführt sind. Bei einem festen Schedule nennt man die Dauer, bis sämtliche Jobs erledigt sind, den* Makespan *des Schedules. Es gilt also den Makespan zu minimieren.*

Einen Schedule S spezifizieren wir durch n Paare (s_k, M_i). Das Paar (s_k, M_i) bedeutet, dass Job J_k zum Zeitpunkt s_k auf Maschine M_i gestartet wird. Der Makespan des Schedules S ist dann $\max\{s_k + d_k | k = 1, \ldots, n\}$. Das Ziel ist einen Schedule zu finden, der dieses Maximum minimiert.

Beispiel 1.1.2 *In Abbildung 1.1 haben wir den optimalen Schedule für ein Problem mit 2 Maschinen und Jobs $J_1, \ldots, J_5$ mit Dauer $d_1 = 1, d_2 = 2, d_3 = 2, d_4 = 4, d_5 = 1$ angegeben. Da beide Maschinen bis zur Ausführung aller Jobs stets ausgelastet sind, muss dieses ein optimaler Schedule sein. Es gibt aber noch andere optimale Schedules. Welche?*

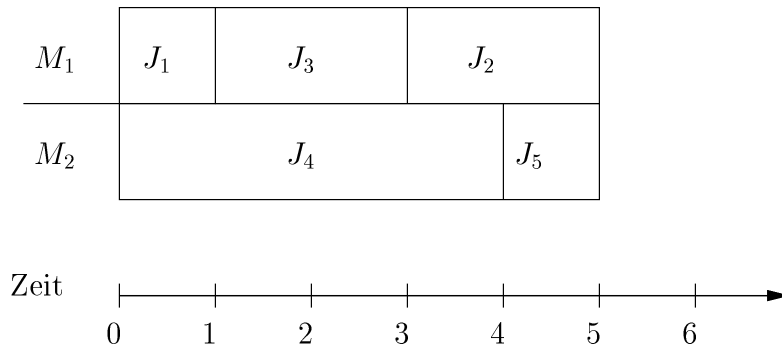

Abbildung 1.1: Ein optimaler Schedule

Im allgemeinen ist das Problem, einen optimalem Schedule zu finden NP-schwer. Das bedeutet, das es aller Voraussicht nach keinen effizienten Algorithmus geben wird, der einen optimalen Schedule findet. Hierzu mehr in Kapitel 2.
Einen Ausweg aus dieser Situation bietet ein Approximationsalgorithmus. Ein Algorithmus also, der zwar nicht unbedingt einen optimalen Schedule findet, aber einen Schedule, dessen Makespan nicht zu sehr vom Optimum abweicht. Dieser Algorithmus sollte einerseits polynomielle Laufzeit haben. Andererseits sollten wir etwas über das Verhältnis des optimalen Makespan zum Makespan des vom Algorithmus gelieferten Schedule *beweisen* können. Wir sind also nicht an Heuristiken interessiert, sondern an Approximationen über deren Güte wir etwas rigoros beweisen können.
Betrachten wir als Beispiel für einen Approximationsalgorithmus den folgenden Algorithmus LS (*list schedule*). In dem von diesem Algorithmus erzeugten Schedule wird einfach der nächste noch nicht gestartete Job auf einer freiwerdenden Maschine gestartet. Um festzustellen, welche Maschine als nächste frei wird, führen wir für jede Maschine M_i eine Invariante a_i ein. Der Algorithmus LS stellt sicher, dass a_i immer angibt, wann Maschine M_i wieder frei wird, und somit für einen neuen Job zur Verfügung stehen wird. Hier zunächst der Algorithmus, im Anschluss dann die Analyse, an der wir schon einige wichtige Techniken kennenlernen werden.

Algorithmus 1.1.3

LS (list schedule):
  $S := \emptyset$
  `FOR` $i = 1$ `TO` m `DO`
    $a_i := 0$
  `END`
  `FOR` $k = 1$ `TO` n `DO`
    Berechne kleinstes i mit $a_i = \min\{a_j | j = 1 \ldots, m\}$
    $s_k := a_i, a_i := a_i + d_k, S := S \cup \{(s_k, M_i)\}$

```
END
Gib S aus
```

In unserem Beispiel von oben wird Algorithmus LS den Schedule in Abbildung 1.2 erzeugen.

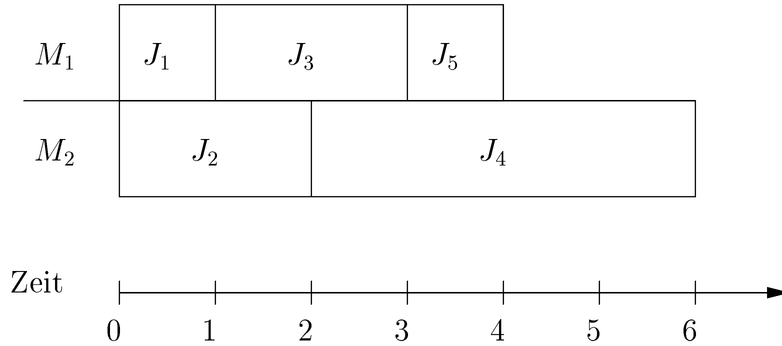

Abbildung 1.2: Der vom Algorithmus LS erzeugte Schedule

Satz 1.1.4 *Der von Algorithmus* LS *erzeugte Schedule hat einen Makespan, der höchstens doppelt so lang ist wie der Makespan eines optimalen Schedules.*

Beweis: Bei einem Job-Scheduling mit m Maschinen $M_1, \ldots, M_m$ und n Jobs $J_1, \ldots, J_n$ mit Dauern $d_1, \ldots, d_n$ sei opt der Makespan eines optimalen Schedules und C der Makespan des von Algorithmus LS erzeugten Schedules. Wie oben, sei s_k der Startzeitpunkt des k-ten Jobs J_k, in dem von Algorithmus LS erzeugten Schedule. Analog sei C_k der Zeitpunkt, zu dem die Ausführung von Job J_k endet. Es gilt also $C_k = s_k + d_k$. Schliesslich sei J_l der Job, der als letzter beendet wird. Dann gilt $C = C_l$.
Die Analyse beruht auf dem folgenden Lemma, das wir bei anderen Gelegenheiten noch benutzen werden.

Lemma 1.1.5

(1) $s_l \leq 1/m \sum_{k \neq l} d_k$

(2) $\text{opt} \geq d_l$

(3) $\text{opt} \geq 1/m \sum_{k=1}^{n} d_k$.

(4) $$C \leq 1/m \sum_{k=1}^{n} d_k + (1 - 1/m) d_l$$

Beweis:

zu (1): Bis zum Zeitpunkt s_l ist jede Maschine durchgehend ausgelastet, sonst hätte J_l früher gestartet werden können. Dann muss aber spätestens zum Zeitpunkt $1/m \sum_{k \neq l} d_k$ eine Maschine frei werden, denn dieser Term entspricht der durchschnittlichen Laufzeit einer Maschine auf allen anderen Jobs, es muss also eine Maschine geben, deren Laufzeit höchstens so gross ist.

zu (2): offensichtlich.

zu (3): Die Summe $\sum_{k=1}^{n} d_k$ ist die gesamte Zeit, die die Jobs in Anspruch nehmen. Auf m Maschinen wird daher mindestens Zeit $1/m \sum_{k=1}^{n} d_k$ für alle Jobs benötigt.

zu (4) Aus (1) erhalten wir

$$C = C_l = s_l + d_l \leq 1/m \sum_{k \neq l} d_k + d_l = 1/m \sum_{k=1}^{n} d_k + (1 - 1/m) d_l.$$

□

Aus (2),(3) und (4) erhalten wir nun

$$C \leq 1/m \sum_{k=1}^{n} d_k + (1 - 1/m) d_l \leq \text{opt} + (1 - 1/\text{m})\text{opt} \leq 2 \cdot \text{opt}.$$

□

Ein immer wiederkehrendes Muster in den Beweisen dieser Vorlesung kann an diesem Beweis schon gut aufgezeigt werden. Wir wollen die Lösung eines Approximationsalgorithmus mit einem optimalen Wert vergleichen, den wir aber gar nicht kennen! Wir brauchen daher gute Schranken für diesen optimalen Wert. Das Finden solcher Schranken ist häufig der entscheidende Punkt in den Analysen von Approximationsalgorithmen. In unserem Beispiel erhalten wir offensichtliche Schranken für den Makespan durch (2) und (3) in Lemma 1.1.5.

1.2 Notation

Nachdem wir ein Beispiel kennengelernt haben, wollen wir nun die wichtigsten Notationen für den Rest der Vorlesung festlegen.
In der Vorlesung betrachten wir stets Optimierungsprobleme. Ein *Optimierungsproblem* ist durch eine Menge $\mathcal{I}$ von *Instanzen* definiert. Im Job-Scheduling z.B. ist eine Instanz spezifiziert durch die Anzahl m der Maschinen, die Anzahl n der

Jobs und die Dauer d_k der einzelnen Jobs. Zu jeder Instanz I haben wir eine Menge $F(I)$ von *zulässigen Lösungen.* Im Job-Scheduling ist eine zulässige Lösung ein Schedule, der alle Jobs genau einer Maschine zuordnet, keine Jobs abbricht und zu jedem Zeitpunkt jeder Maschine nur einen Job zugeordnet hat. Weiter haben wir zu jeder zulässigen Lösung $s \in F(I)$ einen *Wert* $w(s) > 0$. Im Job-Scheduling ist dieses der Makespan des Schedules. Das Ziel ist es dann, bei gegebener Instanz eine zulässige Lösung zu finden, so dass $w(s)$ möglichst gross ist (*Maximierungsprobleme*) oder möglichst klein ist (*Minimierungsprobleme*). Job-Scheduling ist ein Minimierungsproblem.
Ein *Approximationsalgorithmus* für ein Optimierungsproblem Π ist ein Algorithmus, der bei Eingabe einer Instanz I von Π eine in $|I|$ polynomielle Laufzeit hat, und eine zulässige Lösung $s \in F(I)$ ausgibt. Hierbei ist $|I|$ die Beschreibungsgrösse von I. Diese ist bei jedem Optimierungsproblem zu definieren. Beim Job-Scheduling ist die Eingabegrösse $m + n$, die Anzahl der Maschinen plus die Anzahl der Jobs. Die Beschreibungsgrösse der d_k geht nicht ein. Wir schreiben $A(I)$ für die Ausgabe s von A bei Eingabe I.
Bislang scheint diese Definition noch nichts mit Approximationen zu tun zu haben. Darum noch folgende Definitionen. Sei A ein Approximationsalgorithmus für ein Optimierungsproblem Π. Der *Approximationsfaktor* oder die *Approximationsgüte* von A bei Eingabe I ist definiert als

$$\delta_A(I) = \frac{w(A(I))}{\mathrm{opt(I)}},$$

hier ist opt(I) der Wert einer optimalen zulässigen Lösung der Instanz I.
Schliesslich sei $\delta : \mathcal{I} \mapsto \mathbf{R}^+$ eine Funktion. Wir sagen, dass A *Approximationsfaktor* oder *Approximationsgüte* δ hat, wenn für jede Instanz I gilt

$$\delta_A(I) \geq \delta(I) \text{ bei einem Maximierungsproblem}$$
$$\delta_A(I) \leq \delta(I) \text{ bei einem Minimierungsproblem.}$$

Man beachte, dass wir bei einem Maximierungsproblem stets $\delta_A(I) \leq 1$ haben, bei einem Minimierungsproblem dagegen $\delta_A(I) \geq 1$. Analog wird bei Maximierungsproblemen die Funktion δ nur Werte kleiner als 1 annehmen, bei Minimierungsproblemen dagegen nur Werte ≥ 1. Man beachte ausserdem, dass bei einem Maximierungsproblem ein Approximationsalgorithmus, der Approximationsfaktor δ hat, auch Approximationsfaktor δ' hat für jede Funktion δ' mit $\delta'(I) \leq \delta(I)$ für alle $I \in \mathcal{I}$. Bei Minimierungsproblemen gilt dieses für δ', die immer grösser sind als δ. Wir haben diese Ausdrucksweise gewählt, da bei vielen Approximationsalgorithmen der bestmögliche Approximationsfaktor nicht bekannt ist, sondern nur eine Abschätzung. Ein wichtiger Spezialfall ist der Fall, wo δ eine konstante Funktion ist. In diesem Fall werden wir mit δ auch den einzigen Wert bezeichnen, den die Funktion annimmt.
Mit dieser Notation können wir nun bei Job-Scheduling sagen, dass der Algorithmus LS Approximationsfaktor oder Approximationsgüte 2 hat.

1.3 Das Max-Cut Problem

Wir schliessen die Einleitung mit der Behandlung eines weiteren einfachen Algorithmus ab, diesmal für das Maximierungsproblem Max-Cut.

Problem 1.3.1 (MAX-CUT) *Gegeben ist ein Graph $G = (V, E)$ mit Knotenmenge V und Kantenmenge E; gesucht ist eine Partition $(S, V \setminus S)$ der Knotenmenge, so dass die Anzahl der Kanten zwischen S und $V \setminus S$ maximiert wird.*

Die Partition wird üblicherweise als *Schnitt* bezeichnet und mit der Menge S identifiziert, die Kanten zwischen S und $V \setminus S$ sind die *Schnittkanten*, ihre Anzahl die *Grösse* des Schnitts. Das Problem, einen Schnitt maximaler Grösse zu finden, ist NP-schwer.

Beispiel 1.3.2 *Betrachte den Graphen in Abbildung 1.3. Der Schnitt $S = \{1, 3, 5\}$ hat Grösse 5, während $S = \{3, 4\}$ Grösse 6 hat. Dies ist auch gleichzeitig der grösstmögliche Schnitt, wie man sich leicht überlegen kann. Die Schnittkanten sind jeweils fett gezeichnet.*

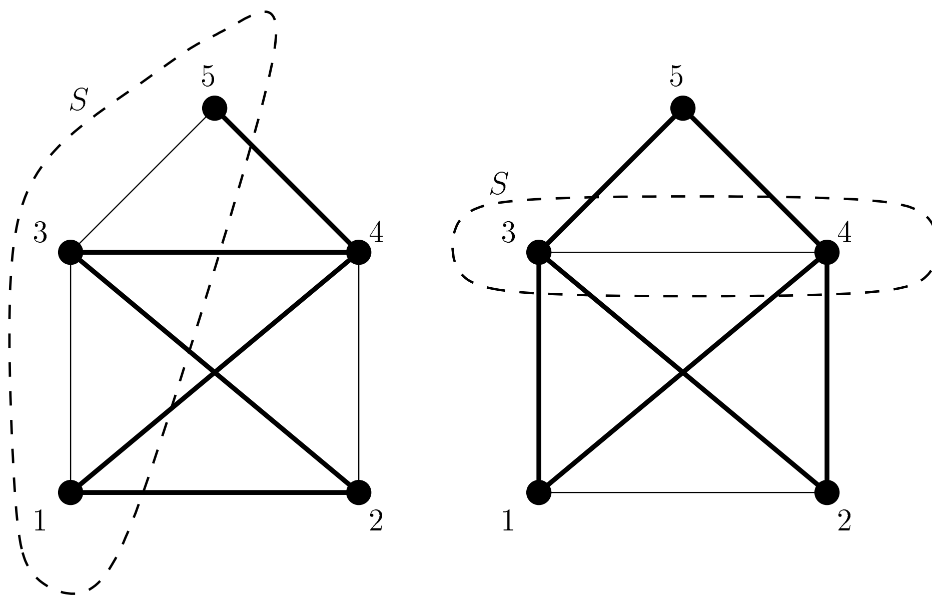

Abbildung 1.3: Schnitte in Graphen

Wir wollen nun einen Approximationsalgorithmus für das Max-Cut Problem angeben. Dazu machen wir uns noch kurz klar, wie wir das Problem als Optimierungsproblem (in diesem Fall als Maximierungsproblem) auffassen können. Eine Instanz I ist ein beliebiger Graph $G = (V, E)$, die zulässigen Lösungen $F(I)$ sind alle Teilmengen S der Knotenmenge V, und der Wert $w(S)$ einer zulässigen Lösung ist die Grösse des durch S definierten Schnittes im Graphen G.

Die Idee des folgenden Algorithmus **LI** (*local improvement*) ist sehr einfach. Man beginnt mit einer beliebigen Menge S (z.B. $S = \emptyset$); solange es noch einen Knoten $v \in V$ gibt, dessen Hinzunahme zu S (bzw. Wegnahme von S) den aktuellen Schnitt vergrössern würde, wird S entsprechend angepasst. Sobald keine solchen lokalen Verbesserungen mehr möglich sind, wird die aktuelle Menge S als Lösung ausgegeben.
Um den Algorithmus formal schön aufschreiben zu können, benötigen wir noch die Notation der *symmetrischen Differenz* $S\triangle\{v\}$, gegeben durch

$$S\triangle\{v\} := \begin{cases} S \cup \{v\}, & \text{falls } v \notin S \\ S \setminus \{v\}, & \text{andernfalls} \end{cases} .$$

Algorithmus 1.3.3

LI (local improvement):

```
  S := ∅
  WHILE es gibt v ∈ V mit w(S△{v}) > w(S) DO
      S := S△{v}
  END
  Gib S aus
```

Der Algorithmus ist polynomiell in $|G|$, denn es gibt höchstens $|E|$ Schleifendurchläufe (jedesmal wird der Schnitt um mindestens eine Kante vergrössert). Der Test, ob $w(S\triangle\{v\}) > w(S)$ gilt, geht natürlich auch in polynomieller Zeit für jeden Knoten v.

Satz 1.3.4 *Algorithmus* LI *hat Approximationsfaktor* $1/2$.

Beweis: Sei S der von LI berechnete Schnitt. Dann gilt für jeden Knoten $v \in V$, dass mindestens die Hälfte der zu v inzidenten Kanten Schnittkanten bzgl. S sind (andernfalls wäre $S\triangle\{v\}$ ein grösserer Schnitt als S, was im Widerspruch dazu steht, dass LI die Menge S ausgegeben hat). Dann gilt aber, dass mindestens die Hälfte *aller* Kanten Schnittkanten bzgl. S sind. Das heisst, es gilt

$$\mathrm{LI}(G) = w(S) \geq |E|/2 \geq \mathrm{opt}(G)/2,$$

denn die Gesamtanzahl aller Kanten ist eine triviale obere Schranke für die maximale Schnittgrösse. Daraus folgt dann

$$\frac{\mathrm{LI}(G)}{\mathrm{opt}(G)} \geq \frac{1}{2},$$

was den Satz beweist, da diese Ungleichung für jeden Graphen (also jede Instanz des Max-Cut Problems) gilt. □

Es sei noch einmal explizit bemerkt, dass wir eine *obere Schranke* für den Wert einer Optimallösung benötigen, um bei einem Maximierungsproblem einen Approximationsfaktor herleiten zu können. Bei Minimierungsproblemen hingegen hilft uns nur eine *untere Schranke* weiter. Die obere Schranke, die wir im Fall von Max-Cut verwendet haben, ist völlig offensichtlich und ergibt sich direkt aus dem Problem. Wir werden aber später sehen (Kapitel 6 und 7), dass auch der Approximationsalgorithmus selbst geeignete Schranken für seine eigene Analyse liefern kann.

1.4 Uebungen

Uebung 1.1 "Billig zügeln": n Umzugsgüter sollen in möglichst wenige Umzugskartons verpackt werden. Dabei nehmen wir an, dass die Kapazität eines Kartons genau 1 ist, und dass das i-te Umzugsgut G_i Grösse $a_i \leq 1$ hat. Ferner soll gelten, dass ein Karton eine Menge von Umzugsgütern genau dann aufnehmen kann, wenn diese in der Summe höchstens Grösse 1 haben.
Betrachte nun den folgenden Algorithmus zum Packen der Umzugsgüter $G_1, \ldots, G_n$ in die Kartons $K_1, K_2, \ldots$

Algorithmus **Next-Fit**:
```
j := 1
FOR i := 1 TO n DO
    IF G_i passt nicht mehr in K_j THEN
        j := j + 1
    END
    packe G_i in K_j
END
```

Zeige, dass Next-Fit einen Approximationsfaktor von 2 erreicht, das heisst, er benötigt höchstens doppelt soviele Umzugskartons wie bei einer optimalen Packung.

Kapitel 2

Eine kurze Einführung in NP-Vollständigkeit

Im letzten Kapitel haben wir schon kurz erwähnt, dass es für das Problem Job-Scheduling vermutlich keinen effizienten Algorithmus gibt, weil das Problem NP-schwer ist. In diesem Kapitel möchten wir diese Aussage mit Hilfe des Konzepts der NP-Vollständigkeit formalisieren.
Betrachten wir das Maximierungsproblem Unabhängige Menge.

Problem 2.0.1 (Unabhängige Menge) *Gegeben ist ein Graph $G = (V, E)$. Gesucht ist eine möglichst grosse Teilmenge $W \subseteq V$, so dass keine Elemente aus W in G durch eine Kante verbunden sind.*

Für alle $v, w \in W$ soll also die Menge $\{v, w\}$ nicht in der Kantenmenge E enthalten sein. Eine Teilmenge von V mit dieser Eigenschaft wird *unabhängige Menge* genannt.
Statt des Maximierungsproblems können wir aber auch folgendes *Entscheidungsproblem* betrachten. Gegeben ist ein Graph $G = (V, E)$ und zusätzlich eine Schranke $K \in \mathbf{N}$. Zu entscheiden ist, ob G eine unabhängige Menge der Grösse mindestens K besitzt, d.h. ob es eine Teilmenge W von V gibt, so dass $|W| \geq K$ und W eine unabhängige Menge ist. Dieses Problem wird Entscheidungsproblem genannt, da die möglichen Antworten nur "Ja" und "Nein" sind. Wir werden auch das Entscheidungsproblem Unabhängige Menge nennen und dann bei Bedarf anmerken, ob die Optimierungs- oder die Entscheidungsvariante gemeint ist. Es sollte klar sein, dass die Optimierungsvariante von Unabhängige Menge mindestens so schwer ist wie die Entscheidungsvariante. Haben wir nämlich einen polynomiellen Algorithmus für die Optimierungsvariante, so liefert uns dieser Algorithmus unmittelbar einen polynomiellen Algorithmus für das Entscheidungsproblem. Um also zu zeigen, dass das Optimierungsproblem Unabhängige Menge schwer ist, genügt es zu zeigen, dass schon das Entscheidungsproblem Unabhängige Menge schwer ist.

Nun kann nicht nur zum Optimierungsproblem Unabhängige Menge ein entsprechendes Entscheidungsproblem definiert werden. Dieses kann ganz allgemein für Optimierungsprobleme gemacht werden. Ist Π ein Optimierungsproblem, so können wir in der Problemstellung zusätzlich zur Instanz I einen Parameter $K \in \mathbf{N}$ aufnehmen. Gesucht ist dann nicht mehr eine zulässige Lösung s mit optimalem Wert $w(s)$, gefragt ist dann, ob es eine zulässige Lösung s mit $w(s) \leq K$ gibt (Minimierungsprobleme) oder eine zulässige Lösung mit $w(s) \geq K$ (Maximierungsprobleme). Auch in diesem allgemeinen Fall überlegt man sich, dass die Optimierungsvariante eines Problems immer mindestens so schwer ist wie die Entscheidungsvariante.
Um Probleme gemäss ihrem Schwierigkeitsgrad zu charakterisieren, ist es einfacher, sich auf Entscheidungsprobleme zu beschränken. Alle Entscheidungsprobleme haben ja dieselben möglichen Ausgaben, nämlich "Ja" und "Nein". Auf der anderen Seite sind die jeweiligen Optimierungsprobleme immer mindestens genauso schwer. Falls wir also zeigen wollen, dass ein Problem schwer ist, so können wir uns auf Entscheidungsprobleme beschränken.
Um Entscheidungsprobleme gut handhaben zu können, abstrahieren wir noch etwas weiter und gehen zu Sprachen über. Mit $\{0,1\}^*$ bezeichnen wir die Menge aller *endlichen* Folgen von 0 und 1. Eine *Sprache* L ist eine Teilmenge von $\{0,1\}^*$. Gegeben eine Sprache L, können wir ein Entscheidungsproblem Π_L wie folgt definieren. Jede Instanz von Π_L ist definiert durch ein $x \in \{0,1\}^*$. Die Antwort bei Instanz x ist "Ja" genau dann, wenn $x \in L$.

Beispiel 2.0.2 *Sei L die Sprache bestehend aus allen endlichen* $0,1$*-Folgen, die Paare von Graphen und natürlichen Zahlen codieren. Hierzu nehmen wir irgendeine standardisierte Form an, um Graphen als* $0,1$*-Folgen zu beschreiben. Eine Möglichkeit sind Adjazenzmatrizen. Natürliche Zahlen sind durch ihre Binärdarstellung gegeben.* $x \in \{0,1\}^*$ *ist in der Sprache L genau dann, wenn x die korrekte Codierung eines Paares bestehend aus einem Graphen G und einer natürlichen Zahl K ist, und ausserdem der Graph eine unabhängige Menge der Grösse mindestens K besitzt.*

Wir sehen an diesem Beispiel, dass die Menge der Entscheidungsprobleme, die durch Sprachen definiert sind , unser Entscheidungsproblem Unabhängige Menge enthält. Nun kann man auch alle anderen interessanten Probleme durch Sprachen beschreiben. Damit haben wir eine mathematische Formalisierung, um über Entscheidungsprobleme Aussagen zu treffen.
Wir kommen nun zu den entscheidenden Definitionen dieses Abschnitts.

Definition 2.0.3 *Ein Algorithmus A mit Eingaben der Form* $(x,w) \in \{0,1\}^* \times \{0,1\}^*$*, der als Ausgabe* 0 *oder* 1 *liefert, heisst* Verifizierer *für die Sprache L, falls*

(1) *Die Laufzeit von A bei Eingabe* (x,w) *ist polynomiell in der Länge* $|x|$ *von* x*.*

(2) *Falls* $x \in L$*, so existiert ein* $w \in \{0,1\}^*$ *mit* $A(x,w) = 1$*. Dabei ist* $A(x,w)$ *die Ausgabe von* A *bei Eingabe* (x,w)*.*

(3) *Falls* $x \notin L$*, so gilt* $A(x,w) = 0$ *für alle* $w \in \{0,1\}^*$*.*

NP *ist dann die Menge aller Sprachen, die einen Verifizierer haben.*

An dieser Definition muss auf mehrere Dinge geachtet werden. In (1) fordern wir, dass die Laufzeit von A nur von der Länge des ersten Teils der Eingabe abhängt. Insbesondere kann sich der Algorithmus nur polynomiell (in $|x|$) viele Bits von der zweiten Eingabe anschauen. Wir können deshalb annehmen, dass die Länge der zweiten Eingabe polynomiell in der Länge der ersten Eingabe ist.
In (2) fordern wir, dass es im Fall $x \in L$ nur *ein* $w \in \{0,1\}^*$ geben muss, so dass die Ausgabe von A bei Eingabe (x,w) 1 ist. (3) wiederum bedeutet, dass bei $x \notin L$ für alle $w \in \{0,1\}^*$ $A(x,w) = 0$ gelten muss. Ein $w \in \{0,1\}^*$ mit $A(x,w) = 1$ ist also ein *Beweis* oder ein *Zeuge* dafür, dass $x \in L$ gilt.
Ist $x \in L$ und ist $w \in \{0,1\}^*$ ein Zeuge hierfür, so sagen wir in der Definition *nichts* darüber aus, wie w gefunden werden kann. Insbesondere fordern wir nicht, dass es einen polynomiellen Algorithmus hierfür gibt. Man sagt deshalb auch, dass für die Sprachen in NP Beweise oder Lösungen leicht zu überprüfen sind, aber nicht unbedingt leicht zu finden sind.
Wir wollen uns nun einige Beispiele für Sprachen in NP anschauen. Zunächst betrachten wir wieder das Problem Unabhängige Menge. Diese Sprache hat einen Verifizierer, den wir nun beschreiben.

Algorithmus 2.0.4

Verifizierer für Unabhängige Menge:

```
IF x ist keine zulässige Codierung eines Graphen G = (V, E)
und einer natürlichen Zahl K ∈ N THEN
    Ausgabe 0
ELSE
    IF w ist keine zulässige Codierung einer Teilmenge W
    der Grösse mindestens K von V THEN
        Ausgabe 0
    ELSE
        IF es gibt zwei Elemente v, w ∈ W mit {v, w} ∈ E THEN
            Ausgabe 0
        ELSE
            Ausgabe 1
        END
    END
END
```

Man überzeugt sich leicht, dass dieser Algorithmus sowohl Bedingung (1) als auch Bedingungen (2) und (3) aus Definition 2.0.3 erfüllt. Unabhängige Menge liegt also in NP.
Als nächstes betrachten wir das Problem 3-SAT. Eine Boolesche Formel φ in *3-konjunktiver Normalform* über den Booleschen Variablen $x_1, \ldots, x_n$ ist eine Formel der Gestalt $\varphi = \bigwedge_{i=1}^{m} C_i$, wobei $C_i = l_{i1} \vee l_{i2} \vee l_{i3}$ und $l_{ij} = x$ oder $l_{ij} = \neg x$ für ein $x \in \{x_1, \ldots, x_n\}$. Die C_i heissen *Klauseln* und die l_{ij} heissen *Literale*. Eine Formel φ heisst erfüllbar, falls es eine Belegung der Variablen in φ mit "wahr" und "falsch" gibt, so dass die Formel den Wert "wahr" annimmt.

Problem 2.0.5 (3-SAT) *Die Sprache* 3-SAT *besteht aus allen erfüllbaren Booleschen Formeln in 3-Konjunktiver Normalform.*

Jede Boolesche Formel φ in 3-Konjunktiver-Normalform kann als ein Element in $\{0,1\}^*$ codiert werden, so dass die Länge der Codierung polynomiell in der Anzahl der Variablen n und der Anzahl der Klauseln m ist. Wir identifizieren eine Formel φ mit ihrer Codierung und bezeichnen auch diese mit φ. Wir werden "wahr" mit 1 und "falsch" mit 0 identifizieren. Eine Belegung einer Booleschen Formel über n Variablen ist also ein Element von $\{0,1\}^n$. Der folgende Algorithmus ist dann ein Verifizierer für 3-SAT.

Algorithmus 2.0.6

Verifizierer für 3-SAT:
```
IF x ist keine zulässige Codierung einer Booleschen Formel
φ in 3-Konjunktiver Normalform THEN
    Ausgabe 0
ELSE
    IF Die Länge von w stimmt nicht mit der Anzahl der Variablen in φ
    überein THEN
        Ausgabe 0
    ELSE
        IF w erfüllt φ nicht THEN
            Ausgabe 0
        ELSE
            Ausgabe 1
        END
    END
```

Wiederum sieht man leicht, dass dieser Algorithmus ein Verifizierer für 3-SAT ist. Auch 3-SAT liegt also in NP.
In NP gibt es Probleme, die einfach zu lösen sind; das bedeutet für uns, dass es einen polynomiellen Lösungsalgorithmus für sie gibt. (Man kann sich fragen, ob ein $O(n^{25})$-Algorithmus noch als effizient zu bezeichnen ist; diese Definition von

Effizienz hat sich aber durchgesetzt.) Die Menge aller Sprachen, die in diesem Sinne als einfach zu lösen gelten, bilden die Klasse P. Es ist klar, dass P $\subseteq$ NP gilt, denn ein polynomieller Lösungsalgorithmus kann insbesondere als Verifizierer eingesetzt werden.
Es gibt aber auch Probleme in NP, von denen man nicht weiss, ob sie effizient zu lösen sind, und von denen man vermutet, dass dieses nicht der Fall ist. Hierzu gehören die sogenannten *NP-vollständigen* Probleme. Um diese zu definieren, benötigen wir den Begriff der Reduktion.

Definition 2.0.7 *Seien L_1, L_2 Sprachen. Eine Algorithmus mit Ein- und Ausgaben aus $\{0,1\}^*$ heisst eine Reduktion von L_1 auf L_2, falls*

(1) $x \in L_1 \Leftrightarrow A(x) \in L_2$.

(2) Bei Eingabe x ist die Laufzeit von A polynomiell in der Länge $|x|$ von x.

L_1 heisst reduzierbar auf L_2, wenn es eine Reduktion von L_1 auf L_2 gibt.

Da die Laufzeit von A polynomiell ist, ist insbesondere die Länge von $A(x)$ polynomiell in der Länge von x. Ist L_1 auf L_2 reduzierbar, so bedeutet das anschaulich, dass L_2 mindestens so schwer ist (in Bezug auf polynomielle Lösbarkeit) wie L_1. Wir wollen als Beispiel zeigen, dass sich 3-SAT auf Unabhängige Menge reduzieren lässt. Der folgende Algorithmus erwartet als Eingabe also eine Boolesche Formel in 3-konjunktiver Normalform.

Algorithmus 2.0.8 *Sei $\varphi = \bigwedge_{i=1}^{m} C_i$.*

Reduktion von 3-SAT auf Unabhängige Menge:

```
V := ∅, E := ∅
K:=m
FOR i := 1 TO m DO
    Füge jedes der 3 Literal aus C_i zu V hinzu.
    Füge zu E eine Kante zwischen je zwei dieser Literale hinzu.
END
WHILE es gibt in V zwei Literale l = x, l' = ¬x DO
    E = E ∪ {l, l'}
END
Gib ((V, E), K) als Instanz von Unabhängiger Menge aus.
```

In Abbildung 2.1 ist der Graph gezeichnet, der bei Anwendung dieser Reduktion auf die Formel $\varphi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3)$ erzeugt wird. Die einzelnen Klauseln sind mit C_1, C_2, C_3 bezeichnet. Die Knoten einer unabhängigen Menge der Grösse 3 sind schwarz gezeichnet.
Wir wollen nun zeigen, dass Algorithmus 2.0.8 eine Reduktion von 3-SAT auf Unabhängige Menge ist. Die Laufzeit ist sicherlich polynomiell in der Eingabegrösse.

Es bleibt zu zeigen, dass eine Boolesche Formel φ mit m Klauseln genau dann erfüllbar ist, wenn der konstruierte Graph eine unabhängige Menge der Grösse m hat. Ist w eine erfüllende Belegung für φ, so ist in jeder Klausel von φ mindestens ein Literal erfüllt. Die Knoten, die diesen erfüllten Literalen entsprechen, bilden eine unabhängige Menge der Grösse m. Hat andererseits der konstruierte Graph eine unabhängige Menge der Grösse m, so muss in dieser unabängigen Menge für jedes der Dreiecke, die den Klauseln in φ entsprechen, ein Knoten enthalten sein. Nun kann man eine Belegung der Variablen wählen, die die Literale erfüllt, die den Knoten der unabhängigen Menge entsprechen. Diese Belegung erfüllt die Formel φ.

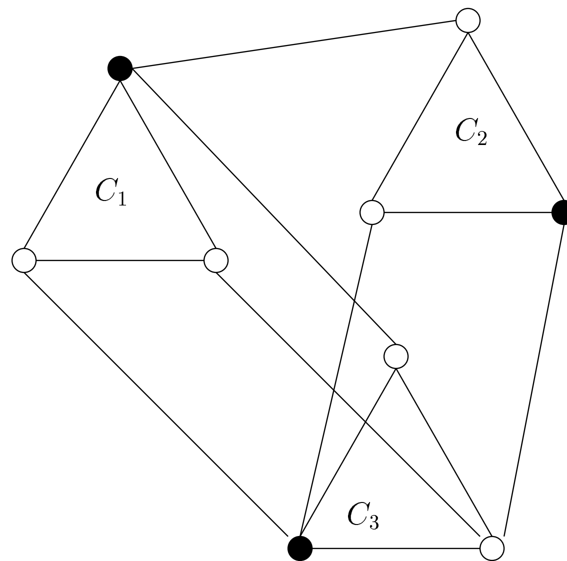

Abbildung 2.1: Beispiel einer Reduktion von 3-SAT auf Unabhängige Menge

Wir kommen nun zur Definition von NP-Vollständigkeit.

Definition 2.0.9 *Eine Sprache $L \in$* NP *heisst* NP-vollständig*, wenn sich jede andere Sprache in* NP *auf L reduzieren lässt.*

Das bedeutet, die NP-vollständigen Sprachen sind mindestens so schwer wie alle Sprachen in NP.
Es gilt nun der folgende Satz, der von Cook und Levin Anfang der siebziger Jahre bewiesen wurde.

Satz 2.0.10 3-SAT *ist* NP*-vollständig.*

Der Beweis dieses Satzes ist aufwendig, deshalb verzichten wir auf ihn. Mittels dieses Satzes können aber häufig andere Probleme recht einfach als NP-vollständig nachgewiesen werden. Es gilt nämlich

Lemma 2.0.11 *L, L' seien Sprachen in* NP*. Ist L* NP*-vollständig, und lässt sich L auf L' reduzieren, so ist auch L'* NP*-vollständig.*

Beweis: Sei $\hat{L}$ eine beliebige Sprache in NP. Wir müssen zeigen, dass sich $\hat{L}$ auf L' reduzieren lässt. Nun lässt sich $\hat{L}$ mittels einer Reduktion R' auf L reduzieren, und L lässt sich mittels einer Reduktion R auf L' reduzieren. Führt man R' und R hintereinander aus, erhält man eine Reduktion von $\hat{L}$ auf L'. □

Hier verwenden wir die Tatsache, dass die Komposition polynomieller Algorithmen einen polynomiellen Algorithmus ergibt, weil das Produkt zweier Polynome wieder ein Polynom ist.
Mittels dieses Lemmas können wir nun beweisen, dass Unabhängige Menge NP-vollständig ist. Wir haben ja schon gesehen, dass Unabhängige Menge in NP liegt, und wir haben auch schon eine Reduktion von 3-SAT auf Unabhängige Menge konstruiert. Damit ist nach dem vorangegangenen Lemma Unabhängige Menge NP-vollständig.
Die NP-vollständigen Probleme sind die schwersten Probleme in NP, denn wir haben das folgende Lemma.

Lemma 2.0.12 *Ist L* NP*-vollständig und existiert ein polynomieller Algorithmus für L, d.h. ein Algorithmus A, der bei Eingabe $x \in \{0,1\}^*$ in polynomieller Zeit in $|x|$ entscheidet, ob $x \in L$, so existiert ein polynomieller Algorithmus für jede Sprache L' in* NP*.*

Beweis: Sei L' eine beliebige Sprache in NP. Um zu entscheiden, ob $x \in \{0,1\}^*$ in L' liegt, wenden wir die Reduktion R von L' auf L mit Eingabe x an. Wir erhalten dann in polynomieller Zeit ein $y \in \{0,1\}^*$. Auf dieses y wenden wir den Algorithmus A an. Nach Definition der Reduktion R und nach Voraussetzung über A, ist $x \in L'$ genau dann, wenn $A(y) = 1$. Da $|y|$ polynomiell ist in $|x|$ und A polynomielle Laufzeit besitzt, liefert uns dies den gewünschten Algorithmus für L'. □

Um auch von Optimierungsproblemen sagen zu können, dass sie mindestens so schwer sind wie NP-vollständige Probleme, nehmen wir dieses Lemma zum Anlass für die folgende Definition.

Definition 2.0.13 *Ein Problem Π, sei es ein Entscheidungsproblem oder ein Optimierungsproblem, heisst* NP-schwer*, falls die Existenz eines polynomieller Algorithmus für Π die Existenz eines polynomiellen Algorithmus für alle Sprachen in* NP *impliziert.*

Mit dieser Definition sind alle NP-vollständigen Probleme NP-schwer. Aber auch die Optimierungsvariante von Unabhängige Menge ist NP-schwer. Denn mit ei-

nem polynomiellen Algorithmus für das Optimierungsproblem Unabhängige Menge haben wir einen polynomiellen Algorithmus für das Entscheidungsproblem Unabhängige Menge, und dann einen polynomiellen Algorithmus für jede beliebige Sprache in NP.
Nun stellt sich natürlich die Frage, wie schwer NP-vollständige oder NP-schwere Probleme denn wirklich zu lösen sind. Die Antwort ist unbekannt. Es gibt allerdings inzwischen über 1000 NP-vollständige Probleme. Für keines dieser Probleme ist je ein polynomieller Algorithmus entworfen worden. Es wird deshalb vermutet, das NP-vollständige oder NP-schwere Probleme nicht effizient gelöst werden können. Dass ein Problem NP-vollständig oder NP-schwer ist, wird darum auch als starkes Indiz dafür angesehen, dass dieses Problem nicht effizient zu lösen ist.

2.1 Uebungen

Uebung 2.1 Gegeben sei ein Graph $G = (V, E)$. Ein *Vertex-Cover* in G ist eine Teilmenge C der Knotenmenge V mit der Eigenschaft, dass jede Kante $e \in E$ zu mindestens einem Knoten $v \in C$ inzident ist. Eine *Clique* ist eine Teilmenge K der Knotenmenge mit der Eigenschaft, dass je zwei Knoten aus K durch eine Kante aus E verbunden sind.

Zeige dass es NP-vollständig ist, bei Eingabe von G und einer natürlichen Zahl k zu entscheiden, ob G

(a) ein Vertex-Cover der Grösse k

(b) eine Clique der Grösse k

besitzt.

Uebung 2.2 Nimm an, dass es einen polynomiellen Algorithmus für das Entscheidungsproblem Unabhängige Menge gibt. Zeige, dass es dann auch einen polynomiellen Algorithmus für das Optimierungsproblem Unabhängige Menge gibt.

Uebung 2.3 Sei $G = (V, E)$ ein Graph mit $|V| = n$. Konstruiere eine Boolesche Formel ϕ, die genau dann erfüllbar ist, wenn G einen Hamiltonschen Kreis besitzt, das ist ein Kreis im Graphen, der jeden Knoten genau einmal berührt. Hierzu führe Boolesche Variablen $x_{ij}, i, j = 1, \ldots, n$, ein. Die Variable x_{ij} steht für die Aussage "Der j-te Knoten des Hamiltonschen Kreises ist der Knoten i". Die Formel muss nicht in 3-konjunktiver Normalform sein.

Kapitel 3

Elementare Approximationsalgorithmen

Bevor wir zu allgemeineren Techniken übergehen, werden wir in den folgenden beiden Kapiteln noch elementare Approximationsalgorithmen kennenlernen.

3.1 Das Problem des Handlungsreisenden

Beginnen wollen wir mit dem Problem des Handlungsreisenden.

Problem 3.1.1 (Euklidischer Handlungsreisender (ETSP))
Ein Handlungsreisender möchte beginnend in der Stadt s_1 die Städte $s_1, \ldots, s_n$ jeweils genau einmal besuchen, um am Ende zu s_1 zurückzukehren. Dabei möchte er eine möglichst kleine Distanz zurücklegen. Wir nehmen an, dass die Städte s_i gegeben sind als Punkte in der Ebene $\mathbf{R}^2$. Die Distanz zwischen zwei Städten s_i, s_j ist durch die euklidische Distanz der Punkte s_i, s_j im $\mathbf{R}^2$ gegeben.

Die Abkürzung ETSP steht für "euclidean traveling salesman problem". ETSP ist wie üblich NP-schwer. Wir werden im Laufe dieses Kapitels zwei Approximationsalgorithmen für das Problem kennenlernen. Der erste hat Approximationsgüte 2, der zweite Approximationsgüte 3/2. Doch bevor wir zu diesen Algorithmen kommen, wollen wir noch Verallgemeinerungen von ETSP definieren.
Hierzu benötigen wir den Begriff der *Rundreise* oder des *Hamiltonschen Kreises*. In einem Graphen $G = (V, E), V = \{v_1, \ldots, v_n\}$ ist eine Rundreise ein Kreis in G, der an einem beliebigen Knoten startet, jeden Knoten genau einmal besucht und dann zum Ausgangspunkt zurückkehrt. Genauer ist eine Rundreise eine Permutation π der Menge $\{1, \ldots, n\}$, so dass

$$\{v_{\pi(i)}, v_{\pi(i+1)}\} \in E, i = 1, \ldots, n-1, \text{ und } \{v_{\pi(n)}, v_{\pi(1)}\} \in E.$$

Problem 3.1.2 (Handlungsreisender (TSP)) *Eine Instanz I ist gegeben durch einen gewichteten Graphen. Also durch einen Graphen $G = (V, E), V = \{v_1, \ldots, v_n\}$, und zusätzlich eine Funktion $w : E \to \mathbf{R}^+$, die jeder Kante $e \in E$ ein Länge $w(e)$ zuordnet. Gesucht ist dann eine Rundreise mit minimaler Gesamtlänge. D.h., gesucht ist eine Permutation π, die eine Rundreise in G definiert und für die*

$$w(\pi) = \sum_{i=1}^{n-1} w\left(\{v_{\pi(i)}, v_{\pi(i+1)}\}\right) + w\left(\{v_{\pi(n)}, v_{\pi(1)}\}\right)$$

minimal ist.

Um ETSP als Spezialfall von TSP zu erhalten, wählen wir als Graphen G den vollständigen Graphen auf n Knoten. Wir nehmen ausserdem an, dass jeder Knoten in den $\mathbf{R}^2$ eingebettet ist, und dass die Länge einer Kante gegeben ist durch die euklidische Distanz zwischen seinen beiden Endpunkten. Da ETSP NP-schwer, gilt dieses auch für die Verallgemeinerung TSP.
Während es beim ETSP immer eine Rundreise gibt, gibt es Instanzen von TSP, in denen es keine Rundreise gibt. In diesem Fall setzen wir ∞ als Länge der optimalen Rundreise. Jeder Algorithmus, der TSP löst, muss also folgendes, als Hamiltonscher Kreis bekanntes Entscheidungsproblem lösen.

Problem 3.1.3 (Hamiltonscher Kreis (HK)) *Gegeben sei ein Graph $G = (V, E)$. Zu entscheiden ist, ob G einen Hamiltonschen Kreis besitzt.*

HK ist NP-vollständig. Wir werden später auf TSP und HK noch zurückkommen. Doch nun wollen wir Approximationsalgorithmen für ETSP beschreiben.
Man erinnere sich, dass eine *spannender Baum* in einem Graphen $G = (V, E)$ ein Baum T auf den Knoten in V ist, so dass jede Kante in T in der Kantenmenge E von G enthalten ist. In einem gewichteten Graphen ist das Gesamtgewicht eines spannenden Baumes die Summe der Kantengewichte der Kanten des Baumes. Ein *minimal-spannender Baum* ist dann ein spannender Baum mit minimalem Gesamtgewicht. Ein minimal-spannender Baum einer Punktemenge im $\mathbf{R}^2$ ist ein minimal-spannender Baum für den vollständigen Graphen zwischen diesen Punkten, wobei das Gewicht einer Kante durch die euklidische Distanz zwischen den Endpunkten der Kante gegeben ist. Ein minimal-spannender Baum in einem Graphen mit m Kanten kann in Zeit $\mathcal{O}(m \log m)$ Zeit gefunden werden. Der Algorithmus von Kruskal z.B. erreicht diese Laufzeit.
Der folgende Algorithmus erwartet als Eingabe eine Instanz I von ETSP spezifiziert durch n Punkte $s_1, \ldots, s_n \in \mathbf{R}^2$.

Algorithmus 3.1.4

MSB *(minimal-spannender Baum)*:

Berechne einen minimal-spannenden Baum T auf den Punkten $s_1, \ldots, s_n$.
Konstruiere aus T den Graphen H, in dem jede Kante in T verdoppelt wird.
Finde in H einen Eulerkreis K.
Berechne die Reihenfolge $s_{\pi(1)}, \ldots, s_{\pi(n)}$ der ersten Auftritte der Knoten $s_1, \ldots, s_n$ in K.
Gib $s_{\pi(1)}, \ldots, s_{\pi(n)}$ aus.

Zu diesem Algorithmus einige Bemerkungen. Da wir H aus T durch verdoppeln jeder Kante erhalten, hat in H jeder Knoten geraden Grad. Dann besitzt H nach Uebung 3.2 einen Eulerkreis, das ist ein Kreis im Graphen, der jede Kante genau einmal benutzt. Dieser kann in polynomieller Zeit gefunden werden. Da auch ein minimal-spannender Baum in polynomieller Zeit gefunden werden kann, ist MSB also ein Approximationsalgorithmus. In Abbildung 3.1 ist das Verhalten von MSB an einem Beipiel mit 8 Städten illustriert. Die Reihenfolge der Städte auf dem Eulerkreis ist

$$s_1, s_5, s_2, s_3, s_2, s_4, s_2, s_6, s_7, s_6, s_8, s_6, s_2, s_5, s_1.$$

Die Reihenfolge der ersten Auftritte ist dann

$$s_1, s_5, s_2, s_3 . s_4, s_6, s_7, s_8.$$

Satz 3.1.5 MSB *hat Approximationsgüte* 2.

Beweis: Sei $\mathrm{opt}(I)$ die Länge einer optimalen Rundreise R. Sei $|T|$ das Gewicht des minimal-spannenden Baumes T, der in MSB berechnet wird. Durch Entfernen einer beliebigen Kante wird R zu einem spannenden Baum. Daher gilt $|T| \leq \mathrm{opt}(I)$. Die Gesamtlänge der Kanten in H ist daher höchstens $2\mathrm{opt}(I)$. Daher hat der Eulerkreis K höchstens Gesamtgewicht $2\mathrm{opt}(I)$. Die Rundreise $MSB(I)$ entsteht aus K, indem "Abkürzungen" genommen werden. Da für die euklidische Distanz in der Ebene die Dreiecksungleichung gilt, erhalten wir $|MSB(I)| \leq 2\mathrm{opt}(I)$. □

Wenn wir uns den Algorithmus MSB und seine Analyse genauer anschauen, so bemerken wir, dass das Verdoppeln der Kanten in T für die Approximationsgüte 2 verantwortlich ist. Dieses Kantenverdoppeln soll nun aber nur sicherstellen, dass im Graphen H jeder Knoten geraden Grad hat, und H somit einen Eulerkreis besitzt. Um einen Graphen H zu konstruieren, in dem jeder Knoten geraden Grad hat, können wir aber auch anders vorgehen. In T betrachten wir nur die Teilmenge der Knoten, die in T ungeraden Grad haben. Um den Graphen H zu erhalten, sollte jeder dieser Knoten noch eine zusätzliche inzidente Kante erhalten. Da es

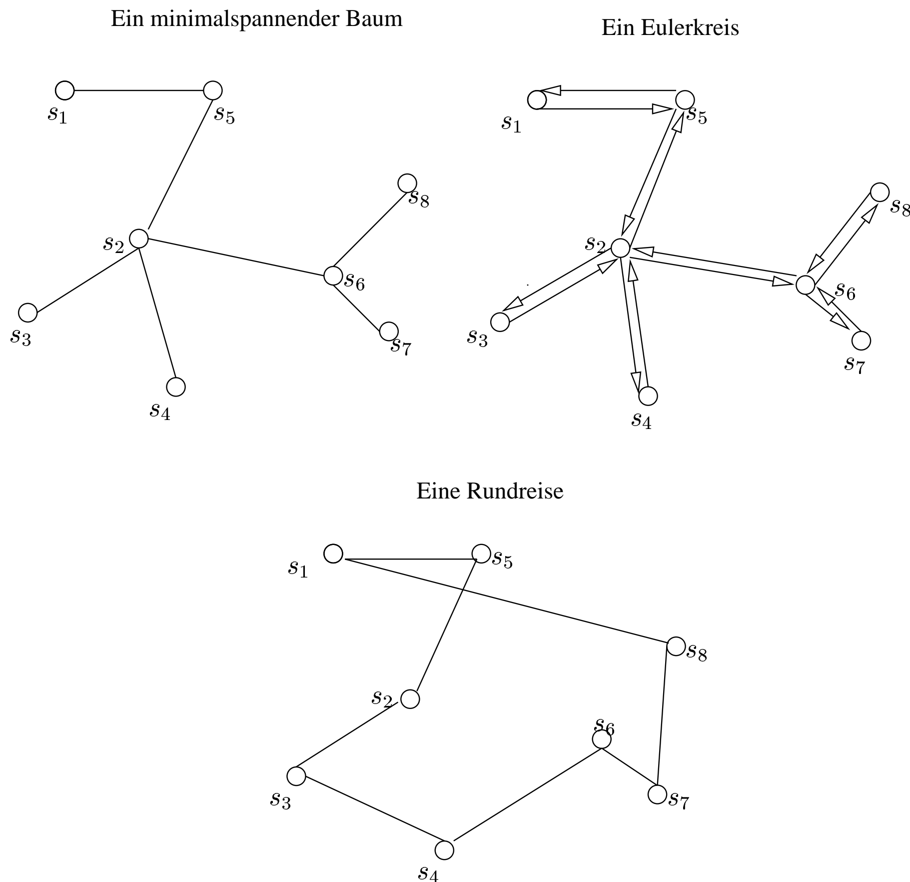

Abbildung 3.1: Beispiel für Berechnung einer Rundreise mit Algorithmus MSB

geradzahlig viele Knoten ungeraden Grades in T gibt (Uebung 3.1), kann dieses durch Hinzufügen eines *perfekten Matchings* auf den Knoten ungeraden Grades erreicht werden.
Es sei daran erinnert, dass ein perfektes Matching in einem Graphen $G = (V, E)$ eine Teilmenge $F \subseteq E$ der Kanten ist, so dass jeder Knoten aus V zu genau einer Kante aus F inzident ist. Offensichtlich muss nicht immer ein perfektes Matching existieren. Eine notwendige Bedingung ist, dass die Anzahl der Knoten des Graphen gerade ist. In einem gewichteten Graphen ist das Gewicht eines Matchings, die Summe der Kantengewichte der Kanten des Matchings. Ein *minimales perfektes Matching* ist dann ein Matching mit minimalem Gesamtgewicht. Ein minimales perfektes Matching für eine Punktemenge in der Ebene ist ein perfektes Matching in dem vollständigen Graphen auf diesen Punkten, wobei Kantengewichte wie üblich durch die euklidischen Distanzen definiert sind. Ein perfektes Matching existiert genau für Punktemengen mit geradzahlig vielen Punkten. Es gibt polynomielle Algorithmen zur Berechnung von minimalen perfekten Matchings in allgemeinen Graphen.

Nun können wir die obige Idee zur Verbesserung von MSB mit Hilfe des folgenden Algorithmus formulieren. Der Algorithmus stammt von Christofides.

Algorithmus 3.1.6

CH (Christofides):
Berechne einen minimlspannenden Baum T auf den Punkten $s_1, \ldots, s_n$.
Bestimme die Menge der Punkte ungeraden Grades in T.
Auf dieser Punktemenge berechne ein minimales perfektes Matching M.
Füge T und M zu einem Graphen H zusammen.
Finde in H einen Eulerkreis K.
Berechne die Reihenfolge $s_{\pi(1)}, \ldots, s_{\pi(n)}$ der ersten Auftritte der Knoten $s_1, \ldots, s_n$ in K.
Gib $s_{\pi(1)}, \ldots, s_{\pi(n)}$ aus.

In Abbildung 3.2 ist das Verhalten dieses Algorithmus am selben Beispiel wie in Abbildung 3.1 illustriert.

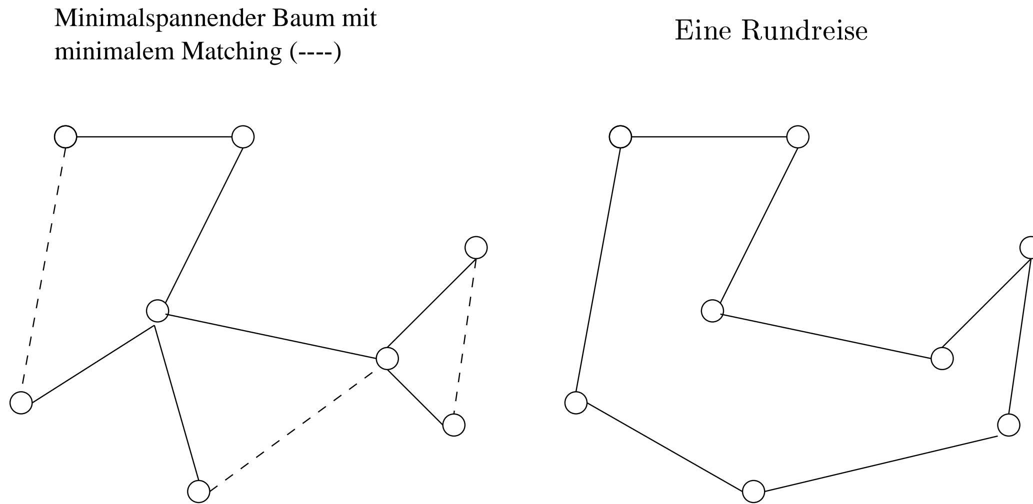

Abbildung 3.2: Beispiel für Berechnung einer Rundreise mit Algorithmus CH

Aus den bereits gemachten Beobachtungen folgt, dass Algorithmus CH in polynomieller Zeit eine Rundreise findet.

Satz 3.1.7 *Algorithmus* CH *hat Approximationsgüte* 3/2.

Beweis: Bei Instanz I sei $\text{opt}(I)$ die Länge einer optimalen Rundreise R. Wie bei der Analyse von Algorithmus MSB gilt $|T| \leq \text{opt}(I)$. Weiter gilt $|M| \leq \text{opt}(I)/2$, denn eine optimale Rundreise R' auf den Punkten, die in T ungeraden Grad haben, ist sicherlich kürzer als $\text{opt}(I)$. Die Rundreise R' kann nun aber in zwei perfekte Matchings aufgespalten werden. Eines davon hat sicher Länge

höchstens $|R'|/2$. Somit $|M| \leq |R'|/2 \leq \text{opt}(I)/2$. Damit erhalten wir, dass die Gesamtlänge der Kanten in H höchstens $\frac{3}{2}\text{opt}(I)$ ist. Dann ist aber auch die Länge des Eulerkreises durch $\frac{3}{2}\text{opt}(I)$ beschränkt. Und mit der Dreiecksungleichung gilt $|\text{CH}(I)| \leq \frac{3}{2}\text{opt}$. ◻

Wann immer wir bislang bewiesen haben, dass ein Algorithmus eine gewisse Approximationsgüte besitzt, so waren dieses immer nur obere Schranken. Wir wussten dann immer, dass die gelieferte Lösung höchstens um einen bestimmten Faktor schlechter war als das Optimum. Wir haben aber bislang nie gezeigt, dass es auch wirklich Instanzen gibt, bei denen die gefundene Lösung so viel schlechter ist als das Optimum. Häufig ist es auch sehr schwer oder unmöglich solche Beispiele anzugeben. Im Fall von Algorithmus 3.1.6 ist dieses jedoch möglich. In Abbildung 3.3 ist ein Beispiel gezeichnet, wo die Länge der von Algorithmus 3.1.6 gefundenen Rundreise etwa um den Faktor 3/2 vom Optimum abweicht.

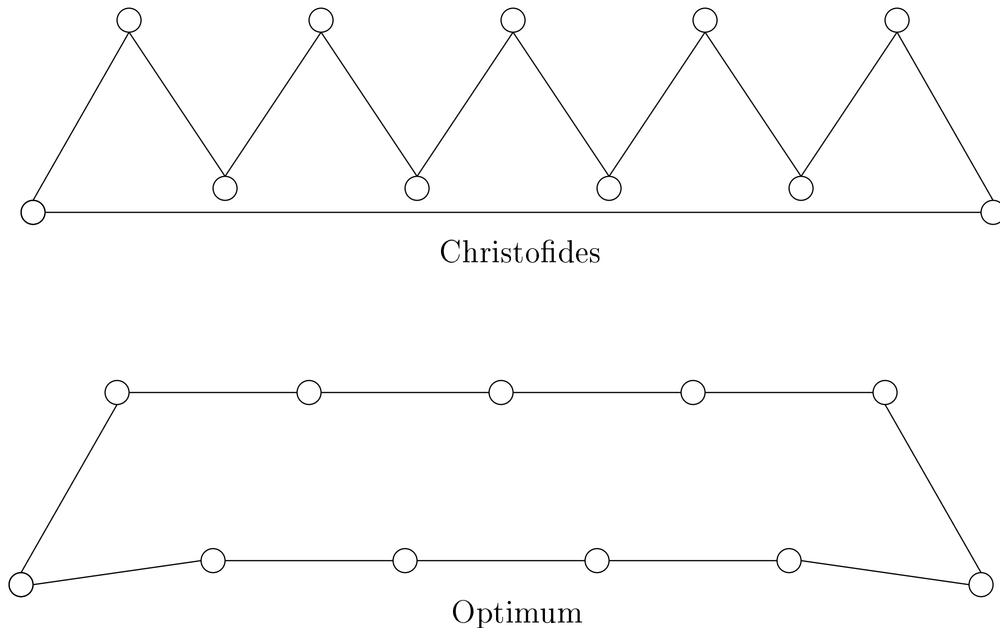

Abbildung 3.3: Beispiel einer Instanz mit $|\text{CH}(I)| \approx \frac{3}{2}\text{opt}(I)$

Zu diesem Beispiel einige Erläuterungen. Die Punkte im unteren Teil des Beispiels liegen nicht auf einer horizontalen Linie. Stattdessen liegen die beiden äusseren Punkte auf einer horizontalen Geraden, die unter der horizontalen Geraden liegt, die die restlichen Punkte des unteren Teils verbindet. Diese beiden horizontalen Geraden dürfen beliebig nahe aneinander sein, jedoch nicht zusammenfallen. Weiter sollen jeder der Punkte in der oberen Hälfte mit den rechts und links unter ihm liegenden Punkten “fast” ein gleichseitiges Dreieck. Man kann dann relativ leicht zeigen, dass die Ausgabe von CH und das Optimum wie in Ab-

bildung 3.3 aussehen. Ausserdem ist die Länge von $\mathrm{CH}(I)$ etwa $3/2\mathrm{opt}(I)$. Das "etwa" bezieht sich darauf, wie weit jetzt die beiden horizontalen Geraden, auf denen die Punkte der unteren Hälfte liegen, voneinander entfernt sind. Je näher sie aneinander liegen, umso näher liegt $|\mathrm{CH}(I)|/\mathrm{opt}(I)$ an $3/2$.
Wir merken noch an, dass seit etwa zwei Jahren bekannt ist, dass sich ETSP beliebig gut approximieren lässt. Was das genau bedeutet, werden wir im nächsten Kapitle sehen.
Zum Abschluss dieses Kapitels zeigen wir noch, dass das allgemeine TSP nicht gut approximiert werden kann.

Satz 3.1.8 *Sei $c > 1$. Falls* $\mathrm{P} \neq \mathrm{NP}$*, so gibt es für* TSP *keinen Approximations-algoritmus mit Güte c.*

Beweis: Wir zeigen, dass es mit einem Approximationsalgorithmus mit Güte c auch einen polynomiellen Algorithmus für das NP-vollständige Problem Hamiltonscher Kreis gibt. Daraus folgt die Aussage des Satzes.
Sei also A ein Approximationsalgorithmus für TSP mit Güte c. Dann ist der folgende Algorithmus, der als Eingabe einen Graphen $G = (V, E)$ erwartet, ein Algorithmus für Hamiltonschen Kreis.

Algorithmus 3.1.9

```
Konstruiere den vollständigen Graphen G' auf den Knoten in V.
FOR i = 1 TO |V| - 1 DO
    FOR j = 2 TO |V| DO
        IF {i, j} ∈ E THEN
            w({i, j}) = 1
        ELSE
            w({i, j}) = c|V|
        END
    END
END
Verwende A mit Eingabe G' und Gewichtsfunktion w,
um eine Rundreise R zu berechnen.
IF |R| ≤ c|V| THEN
    "G besitzt einen Hamiltonschen Kreis"
ELSE
    "G besitzt keinen Hamiltonschen Kreis"
END
```

Da A ein polynomieller Algorithmus ist, ist dieses ebenfalls ein polynomieller Algorithmus. Wir müssen noch zeigen, dass die Ausgabe stets korrekt ist.

Besitzt G einen Hamiltonschen Kreis, so gibt es in G' eine Rundreise mit Gesamtlänge $|V|$. Nach Voraussetzung liefert A dann eine Rundreise mit Gesamtlänge höchstens $c|V|$. In diesem Fall wird die Ausgabe also korrekt sein. Besitzt G hingegen keinen Hamiltonschen Kreis, so muss jede Rundreise in G' eine Kante mit Gewicht $c|V|$ benutzen. Dann ist die Länge der optimalen Rundreise mindestens $|V| - 1 + c|V| = (c+1)|V| - 1 > c|V|$. Daher wird auch die von A gefundene Rundreise Länge echt grösser als $c|V|$ haben. Auch in diesem Fall ist die Ausgabe korrekt.

$\square$

Der Algorithms, den wir im Beweis konstruiert haben, ist eine spezielle Reduktion. Die Graphen mit Hamiltonschen Kreis werden auf Graphen abgebildet mit kurzer Rundreise. Graphen ohne Hamiltonschen Kreis dagegen werden auf Graphen mit langer Rundreise abgebildet. Zwischen den beiden möglichen Fällen ist eine grosse Lücke. Daher genügt ein Approximationsalgorithmus für TSP, um zu entscheiden, ob der Graph G' durch einen Graphen G mit oder ohne Hamiltonschen Kreis erzeugt wurde. Dieses Erzeugen einer Lücke ist die einzige bekannte Technik, um zu beweisen, dass es für ein Optimierungsproblem Approximationsalgorithmen mit bestimmter Approximationsgüte nur geben kann, falls P=NP, siehe Kapitel 8.

3.2 Job-Scheduling

Wir betrachten wieder unser Job-Scheduling Problem (siehe Problem 1.1.1). Der neue Approximationsalgorithmus ist eine Variante des Algorithmus LS (list schedule) (siehe Seite 6). Der einzige Unterschied ist, dass der folgende Algorithmus SLS (sorted list schedule) die Jobs vorher nach absteigenden Laufzeiten sortiert.

Algorithmus 3.2.1

SLS (sorted list schedule):
  sortiere Jobs $J_1, \ldots, J_n$ nach absteigenden Laufzeiten $d_1, \ldots, d_n$
  (* Jetzt gelte $d_1 \geq d_2 \geq \cdots \geq d_n$ *)
  Berechne einen Schedule mittels Algorithmus LS

Nun können wir den folgenden Satz beweisen.

Satz 3.2.2 *Algorithmus* SLS *hat Approximationsgüte* 4/3.

Beweis: Sei C der von SLS erzeugte Makespan, opt der optimale Makespan, sowie s_k der Zeitpunkt, zu dem Job J_k startet (im von SLS erzeugten Schedule). Nun betrachten wir den Job J_ℓ, der als letzter fertig wird und unterscheiden zwei Fälle.

Fall 1. Es gilt

$$d_\ell \leq \frac{\text{opt}}{3}.$$

Dann erhalten wir

$$C = s_\ell + d_\ell \leq \text{opt} + \frac{\text{opt}}{3} = \frac{4}{3}\text{opt},$$

denn bis zum Zeitpunkt s_ℓ sind ja alle Maschinen durchgehend belegt, so dass $s_\ell \leq \text{opt}$ gelten muss. In diesem Fall sind wir also schon fertig.

Fall 2. Es gilt

$$d_\ell > \frac{\text{opt}}{3}.$$

Zunächst können wir annehmen, dass $\ell = n$ gilt, J_ℓ also auch der letzte Job in der Sortierung ist. Falls nämlich nicht, so können wir die Jobs $J_{\ell+1}, \ldots, J_n$ einfach weglassen, ohne dass sich der von SLS erzeugte Makespan ändert (J_ℓ war ja der Job, der als letzter terminiert). Der optimale Makespan wird dadurch nicht schlechter, so dass die Güte von SLS nicht besser werden kann. Können wir die Güte 4/3 also unter der Annahme zeigen, dass $\ell = n$ gilt, so haben wir auch Güte 4/3 im ursprünglichen Problem.

Sei nun also $\ell = n$. Dann haben alle Jobs Laufzeiten von mindestens opt/3, woraus folgt, dass in jedem optimalen Schedule höchstens zwei Jobs pro Maschine laufen können. Insbesondere gibt es dann höchstens $2m$ Jobs. Sei $n = 2m - h$ die wirkliche Anzahl der Jobs, $h \geq 0$.

Falls der optimale Schedule alle Maschinen benutzt (was man offenbar ohne Beschränkung der Allgemeinheit annehmen kann), so gilt, dass genau h Jobs alleine auf 'ihren' Maschinen laufen, während die $2(m-h)$ übrigen Jobs in Paaren den übrigen $m - h$ Maschinen zugeordnet werden. Wir haben dann die folgende

Beobachtung 3.2.3 *Jeder Schedule mit den folgenden Eigenschaften ist optimal:*

(i) die h längsten Jobs $J_1, \ldots, J_h$ laufen alleine auf ihren Maschinen, und

(ii) die übrigen Jobs sind zu Paaren $(J_{h+1}, J_n), (J_{h+2}, J_{n-1}), \ldots, (J_m, J_{m+1})$ zusammengefasst.

Wir beweisen hier nur (i), Eigenschaft (ii) ist eine Uebungsaufgabe (Uebung 3.4). Nehmen wir an, Job $J_k, k > h$ läuft in einem optimalen Schedule alleine auf einer Maschine, während $J_s, s \leq h$ mit einem anderen Job J_t gepaart ist. Dann werden diese drei Jobs zum Zeitpunkt

$$max(d_k, d_s + d_t) = d_s + d_t$$

fertig (beachte $d_s \geq d_k$). Aendern wir den Schedule so, dass wir J_s alleine laufen lassen und dafür J_k mit J_t paaren, so ergibt sich für diese drei Jobs ein Makespan von

$$max(d_s, d_k + d_t).$$

Da sowohl $d_s \leq d_s + d_t$ als auch $d_k + d_t \leq d_s + d_t$ gilt, ist dieser neue Schedule mindestens so gut wie der alte (und da dieser bereits optimal war, ebenfalls optimal). Wir haben also die Anzahl der Jobs $J_s, s \leq h$, die alleine laufen, unter Aufrechterhaltung der Optimalität vergrössert. Damit können wir nun fortfahren, bis alle Jobs $J_s, s \leq h$ alleine auf einer Maschine laufen. Daraus folgt dann Teil (i) der Beobachtung.

Nun bleibt nur noch zu beobachten, dass SLS einen Schedule mit den Eigenschaften (i) und (ii) erzeugt, denn zunächst werden die Jobs $J_1, \ldots, J_m$ auf die Maschinen verteilt. Da diese in umgekehrter Reihenfolge fertig werden, ergeben sich die Paare $(J_m, J_{m+1}), \ldots, (J_{h+1}, J_n)$. (Falls einige der Jobs $J_1, \ldots, J_m$ gleiche Laufzeiten haben, so erzeugt SLS möglicherweise einen anderen Schedule, von dem man sich aber leicht überzeugt, dass er ebenfalls optimal ist).

Im Fall 2 gilt also sogar $C = \text{opt}$, so dass SLS in jedem Fall einen Schedule mit Länge höchstens

$$\frac{4}{3}\text{opt}$$

erzeugt. Daraus folgt der Satz. □

3.3 Uebungen

Uebung 3.1 Beweise, dass jeder Graph eine gerade Anzahl von Knoten ungeraden Grades besitzt.

Uebung 3.2 Beweise, dass ein zusammenhängender Graph genau dann einen Eulerkreis besitzt, wenn jeder Knoten geraden Grad hat.

Uebung 3.3 Betrachte ein euklidisches TSP-Problem in der Ebene. Zeige, dass jede optimale Rundreise selbstüberschneidungsfrei ist.

Uebung 3.4 Seien $J_1, \ldots, J_{2m}$ $2m$ Jobs mit Laufzeiten $d_1 \geq \cdots \geq d_{2m}$. Nimm an, dass in einem optimalen Schedule auf jeder Maschine höchstens zwei Jobs laufen. Beweise, dass es einen optimalen Schedule gibt, der die Jobs zu Paaren $(J_1, J_{2m}), (J_2, J_{2m-1}), \ldots, (J_m, J_{m+1})$ zusammenfasst, wobei jedes Paar einer der m Maschinen zugeordnet wird.

Uebung 3.5 Gegeben Jobs $J_1, \ldots, J_n$ mit Laufzeiten $d_i = i, i = 1, \ldots, n$, und $m \leq n$ Maschinen. Welchen Makespan liefert der Algorithmus SLS?

Kapitel 4

Approximationsschemata

Die Approximationsalgorithmen, die wir bisher kennengelernt haben, hatten die Eigenschaft, dass sie die optimale Lösung bis auf einen festen konstanten Faktor approximieren können Im Folgenden werden wir Approximationsalgorithmen für Probleme diskutieren, die die optimale Lösung beliebig gut approximieren können. 'Beliebig gut' soll dabei heissen, dass für jedes gegebene $\varepsilon > 0$ Güte $1+\varepsilon$ (Minimierungsproblem) bzw. $1-\varepsilon$ (Maximierungsproblem) erreichbar ist. Wir werden das später noch formalisieren.

4.1 Das Rucksack-Problem

Ein Dieb findet in einem Lager n Gegenstände mit *Werten* $w_1, \dots, w_n$ und *Gewichten* $g_1, \dots, g_n$ vor. Dabei nehmen wir an, dass die Werte w_j sowie die Gewichte g_j ganzzahlig und positiv sind. Der Dieb hat einen Rucksack, der Gegenstände im Gesamtgewicht von b (ganzzahlig) aufnehmen kann. Welche Teilmenge von Gegenständen muss der Dieb in den Rucksack packen, um den Gesamtwert der gestohlenen Gegenstände zu maximieren? Etwas formaler also

Problem 4.1.1 (RUCKSACKPROBLEM) *Gegeben sind positive Werte $w_1, \dots, w_n$ und positive Gewichte $g_1, \dots, g_n$ sowie eine Schranke b. Zulässige Lösungen sind Teilmengen $I \subseteq \{1, \dots, n\}$ mit $\sum_{j \in I} g_j \leq b$. Gesucht ist eine zulässige Lösung $I \subseteq \{1, \dots, n\}$, die den Gesamtwert $\sum_{j \in I} w_j$ maximiert.*

Wir nehmen noch ohne Beschränkung der Allgemeinheit an, dass $g_j \leq b$ gilt für alle j, denn Gegenstände, die zu schwer für den Rucksack sind, können von vornherein von der Betrachtung ausgenommen werden.
Wie üblich gilt, dass das Auffinden einer optimalen Lösung für das Rucksack-Problem NP-schwer ist. Trotzdem wollen wir mit einem exakten Algorithmus für das Problem beginnen. Dieser wird später in geeigneter Weise als Baustein für den Approximationsalgorithmus verwendet. Die Idee ist *Dynamisches Programmieren.*

Hierzu definieren wir für $j = 1, \ldots, n$ und i ganzzahlig den Wert

$$F_j(i)$$

als das minimale Gewicht einer Teilmenge der ersten j Gegenstände mit Gesamtwert mindestens i (falls es keine solche Teilmenge gibt, setzen wir $F_j(i)$ auf ∞). $F_n(i)$ gibt also an, wie schwer der Rucksack des Diebes mindestens sein muss, damit die Diebesbeute den Wert i erreicht.
Daraus bekommen wir sofort folgende

Beobachtung 4.1.2 *Sei* opt *die optimale Lösung des* Rucksackproblems *(d.h. der Wert einer optimalen Teilmenge von Gegenständen). Dann gilt*

$$\mathrm{opt} = \max\{i \mid F_n(i) \leq b\}.$$

Zum Beweis überlegt man sich folgendes: sei i_0 das Maximum. Dann gilt $F_n(i_0) \leq b$, d.h. nach Definition, dass es eine Teilmenge von Gegenständen gibt, die in den Rucksack passt und mindestens Wert i_0 hat. Es gilt also $\mathrm{opt} \geq i_0$. Andererseits ist $F_n(i_0 + 1) > b$, d.h. um Werte grösser als i_0 zu erreichen, müsste der Dieb seinen Rucksack überladen, was nicht erlaubt ist. Also gilt $\mathrm{opt} = i_0$.
Nach der Beobachtung reicht es also aus, das entsprechende Maximum zu berechnen. Dabei hilft folgendes

Lemma 4.1.3

(i) $F_j(i) = 0$ *für* $i \leq 0$.

(ii) $F_0(i) = \infty$ *für* $i > 0$.

(iii) $F_j(i) = \min(F_{j-1}(i), g_j + F_{j-1}(i - w_j))$ *für* $i, j > 0$.

Beweis: (i) Wenn der Wert des Diebesgutes nicht positiv sein muss, ist die gewichtsminimale Teilmenge, die dies erreicht, offenbar die leere Menge, und die hat Gewicht 0.
(ii) Wenn der Wert des Diebesgutes positiv sein, die Auswahl der Gegenstände aber aus der leeren Menge erfolgen soll, so gibt es keine Lösung und $F_0(i) = \infty$ gilt nach Definition.
(iii) Um eine gewichtsminimale Teilmenge der ersten j Gegenstände mit Wert mindestens i zu finden, kann man wie folgt vorgehen. Man bestimmt einerseits die gewichtsminimale Menge S_1, die den j-ten Gegenstand nicht enthält, andererseits die gewichtsminimale Menge S_2, die ihn enthält. Dann wählt man diejenige mit dem kleineren Gewicht aus. S_1 hat Gewicht $F_{j-1}(i)$, während S_2 Gewicht $g_j + F_{j-1}(i - w_j)$ hat, denn in diesem Fall muss aus den ersten $j-1$ Gegenständen nur noch Wert $i - w_j$ erreicht werden, weil der j-te Gegenstand schon Wert w_j

beiträgt. Daraus folgt sofort die Formel. □

Zum Berechnen von opt werten wir nun solange Funktionswerte von F aus, bis wir das kleinste i mit $F_n(i) > b$ gefunden haben. $i - 1$ ist dann das gesuchte Optimum. Mit Hilfe des Lemmas kann diese Auswertung so geschehen, dass man zur Berechnung des jeweils aktuellen Funktionswerts nur auf bereits berechnete (und in einem zweidimensionalen, mit j und i parametrisierten Feld gespeicherte) Funktionswerte zurückgreifen muss.

Algorithmus 4.1.4 *Wir nehmen an, dass die Werte $F_j(i)$ für $j = 0$ und $i \leq 0$ (siehe Teil (i) und (ii) des Lemmas) bereits bekannt sind.*

ExactKnapsack:

```
i := 0
REPEAT
    i := i + 1
    FOR j := 1 TO n DO
        F_j(i) = min(F_{j-1}(i), g_j + F_{j-1}(i - w_j))
    END
UNTIL F_n(i) > b
Gib i - 1 aus
```

Um die Laufzeit des Algorithmus abzuschätzen, beobachten wir, dass es opt + 1 Durchläufe der REPEAT-Schleife gibt, wobei jeder Durchlauf Zeit $O(n)$ kostet. Insgesamt ergibt sich also

Satz 4.1.5 *Algorithmus* ExactKnapsack *hat Laufzeit* $O(n\ \text{opt})$.

Auf den ersten Blick mag dies wie ein polynomieller Algorithmus aussehen, das ist aber nicht der Fall, denn opt ist im allgemeinen exponentiell in der Eingabegrösse. Diese ist von der Grössenordnung

$$O\left(\sum_{j=1}^{n}(\log w_j + \log g_j) + \log b\right),$$

denn die Eingabezahlen liegen im Standardmodell binär codiert vor. opt hingegen kann Werte bis zu

$$\sum_{j=1}^{n} w_j$$

annehmen, was exponentiell in der Eingabegrösse ist, falls die Gewichte g_j und die Rucksackkapazität b nicht sehr viel grösser als die Werte w_j sind.
Die Idee des folgenden Approximationsalgorithmus ScaledKnapsack ist dann auch, aus dem ursprünglichen Problem ein 'skaliertes' Problem zu konstruieren, bei dem

die optimale Lösung so klein ist, dass sie mit Hilfe von ExactKnapsack in polynomieller Zeit bestimmt werden kann. Man muss dann nur noch zeigen, dass daraus dann auch eine vernünftige Lösung für das ursprünglichen Problem konstruiert werden kann.
Der Algorithmus hat ausser einer Instanz des Rucksackproblems noch einen Eingabeparameter $\varepsilon > 0$, mit dem die Approximationsgüte kontrolliert wird. Ziel wird es sein, Approximationsgüte $1 - \varepsilon$ zu erreichen.

Algorithmus 4.1.6 *Wir definieren* $w_{max} := \max\{w_j, j = 1, \ldots, n\}$.

ScaledKnapsack(ε):
  Wähle $k := \max(1, \lfloor \varepsilon w_{max}/n \rfloor)$
  `FOR` $j := 1$ `TO` n `DO`
    $w_j(k) := \lfloor w_j/k \rfloor$
  `END`
  Berechne $\text{opt}(k)$ und $S(k)$, die optimale Lösung und optimale Teilmenge des Rucksackproblems mit Werten $w_j(k)$ (und unveränderten g_j sowie b) mit Hilfe von ExactKnapsack;
  Gib $\text{opt}^* := \sum_{j \in S(k)} w_j$ als Lösung aus

Es ist klar, dass die Menge $S(k)$ eine zulässige Lösung auch für das ursprüngliche Problem ist, denn die Gewichte und die Rucksackkapazität haben sich ja nicht verändert.
Nun können wir die Güte von ScaledKnapsack abschätzen.

Satz 4.1.7 ScaledKnapsack(ε) *hat Approximationsgüte* $1 - \varepsilon$.

Beweis: Sei S die optimale Teilmenge für das ursprüngliche Problem. Dann gilt

$$\begin{aligned} \text{opt}^* &= \sum_{j \in S(k)} w_j \\ &= k \sum_{j \in S(k)} \frac{w_j}{k} \\ &\geq k \sum_{j \in S(k)} \lfloor \frac{w_j}{k} \rfloor \\ &\geq k \sum_{j \in S} \lfloor \frac{w_j}{k} \rfloor, \end{aligned}$$

denn $S(k)$ ist ja optimal (insbesondere also besser als S) bezüglich der skalierten Werte $w_j(k) = \lfloor w_j/k \rfloor$. Es gilt dann weiter

$$\text{opt}^* \geq k \sum_{j \in S} \lfloor \frac{w_j}{k} \rfloor$$

$$
\begin{aligned}
&\geq k \sum_{j \in S} (\frac{w_j}{k} - 1) \\
&= \sum_{j \in S} (w_j - k) \\
&= \text{opt} - k|S| \\
&= \text{opt} \left(1 - \frac{k|S|}{\text{opt}}\right) \geq \text{opt} \left(1 - \frac{kn}{w_{max}}\right),
\end{aligned}
$$

denn $|S| \leq n$ und opt $\geq w_{max}$, weil nach der Annahme $g_j \leq b$ für alle j der wertvollste Gegenstand eine untere Schranke für den optimalen Rucksackwert darstellt.
Es bleibt noch zu zeigen, dass aus dieser Herleitung $\text{opt}^* \geq \text{opt}(1-\varepsilon)$ folgt. Dazu machen wir eine Fallunterscheidung. Falls $k = 1$, so wurde das ursprüngliche Problem gar nicht skaliert, und es gilt sogar $\text{opt}^* = \text{opt}$. Falls $k = \lfloor \varepsilon w_{max}/n \rfloor$, so folgt daraus sofort

$$
\frac{kn}{w_{max}} \leq \varepsilon,
$$

was die Behauptung impliziert. $\square$

Die gewünschte Approximationsgüte haben wir also erreicht; es bleibt noch zu zeigen, dass der Algorithmus für festes ε auch wirklich polynomielle Laufzeit hat (sonst wäre er gar kein Approximationsalgorithmus in unserem Sinne).

Satz 4.1.8 ScaledKnapsack(ε) *hat Laufzeit*

$$
O\left(n^3 \frac{1}{\varepsilon}\right).
$$

Beweis: Der dominierende Term in der Laufzeit ist der Aufruf von ExactKnapsack für das skalierte Problem; alle weiteren Anweisungen zusammen verursachen nur Kosten $O(n)$.
Die Laufzeit von ExactKnapsack ist dann

$$
O\left(n \text{ opt}(k)\right) = O(n^2 \frac{w_{max}}{k}),
$$

denn $\text{opt}(k)$ kann nicht grösser sein als die Summe aller skalierten Werte, insbesondere nicht grösser als n-mal der maximale skalierte Wert.
Nun machen wir wieder die Fallunterscheidung nach dem Wert von k. Falls $k = 1$, so bedeutet das nach Konstruktion von k, dass

$$
\frac{\varepsilon w_{max}}{n} < 2
$$

gilt. Das impliziert

$$\frac{w_{max}}{k} = w_{max} \leq \frac{2n}{\varepsilon},$$

woraus der Satz in diesem Fall folgt. Falls $k = \lfloor \varepsilon w_{max}/n \rfloor)$ gilt, so kann man daraus leicht

$$\frac{w_{max}}{k} \leq \frac{n}{\varepsilon}\left(1 + \frac{1}{k}\right) \leq \frac{2n}{\varepsilon}$$

folgern, und die behauptete Laufzeit gilt auch in diesem Fall. ◻

Es ist natürlich nicht überraschend, dass die Laufzeit von ε abhängt und mit $\varepsilon \mapsto 0$ beliebig gross wird. Wichtig ist, dass sich für *festes* ε ein polynomieller Algorithmus ergibt – in diesem Fall ein $O(n^3)$-Algorithmus. Die Abhängigkeit von $1/\varepsilon$ ist polynomiell, sie kann aber in einem anderen Fall genausogut exponentiell sein, etwa von der Grössenordnung $O(n^{1/\varepsilon})$. Auch dann gilt noch, dass man für festes ε einen polynomiellen Algorithmus hat.

4.2 Approximationsschemata

Wenn sich ein Problem – so wie das Rucksackproblem hier – beliebig gut approximieren lässt, so sagen wir, das Problem besitzt ein Approximationsschema. Formal definieren wir dieses wie folgt.

Definition 4.2.1 *Ein* Approximationsschema *für ein Optimierungsproblem ist eine Menge*

$$\{A(\varepsilon) \mid \varepsilon > 0\}$$

von Approximationsalgorithmen für das Problem, mit der Eigenschaft, dass Algorithmus $A(\varepsilon)$ Approximationsgüte

$$1 - \epsilon \quad \textit{bei Maximierungsproblemen}$$

bzw.

$$1 + \epsilon \quad \textit{bei Minimierungsproblemen}$$

hat.

Die polynomielle Laufzeit ist implizit dadurch gefordert, dass $A(\varepsilon)$ ein Approximationsalgorithmus ist. Wichtig ist, dass der Wert von ε global für das Optimierungsproblem gewählt werden muss. ε darf nicht von der speziellen Instanz des Problems abhängen, auf die der Algorithmus angewendet wird.
Bei NP-schweren Problemen sind Approximationsschemata das Beste, was man sich erhoffen kann. Es gibt keine Lücke zwischen der optimalen und der besten in

polynomieller Zeit erreichbaren Lösung. Wir werden allerdings im weiteren Verlauf der Vorlesung (Kapitel 8) sehen, dass es Probleme gibt, für die beweisbar keine Approximationsschemata existieren, die man also nicht beliebig gut approximieren kann. In dieser Hinsicht gibt es auch unter den NP-schweren Problemen 'leichte' und 'schwierige' Probleme. Das Rucksack-Problem ist dabei eines der leichtesten NP-schweren Probleme überhaupt.

4.3 Approximationsschema für Job-Scheduling

Wir wollen das Konzept des Approximationsschemas an dem Job-Scheduling Problem (siehe Problem 1.1.1) noch einmal erläutern. Wir machen allerdings die zusätzliche Voraussetzung, dass die Anzahl m der Maschinen konstant ist. Es gibt auch ein Approximationsschema für den Fall variabler Maschinenanzahl, dieses ist aber wesentlich komplizierter.
Für gegebenes $\varepsilon > 0$ erreicht der folgende Algorithmus SLS(k) für geeignetes k eine Approximationsgüte von $1 + \varepsilon$.

Algorithmus 4.3.1

SLS(k):
sortiere Jobs $J_1, \dots, J_n$ nach absteigenden Laufzeiten $d_1, \dots, d_n$
(* Jetzt gelte $d_1 \geq d_2 \geq \cdots \geq d_n$ *)
Berechne einen optimalen Schedule für die ersten k Jobs
Verteile die weiteren Jobs mittels Algorithmus LS (siehe Seite 6)

Güte-Abschätzung für SLS(k). Sei J_ℓ der Job, der als letzter fertig wird, C der erzeugte Makespan. Wir unterscheiden wieder zwei Fälle.

Fall 1. $\ell \leq k$, d.h. Job J_ℓ ist unter den optimal verteilten Jobs. Dann gilt $C = \text{opt}(J_1, \dots, J_k) \leq \text{opt}$, wobei $\text{opt}(J_1, \dots, J_k)$ der Makespan ist, den die ersten k Jobs induzieren. Da natürlich auch $C \geq \text{opt}$ gilt, folgt in diesem Fall $C = \text{opt}$.

Fall 2. Einerseits gilt nun nach Lemma 1.1.5 (3) und (4)

$$C \leq \text{opt} + \frac{m-1}{m} d_\ell. \tag{4.1}$$

Andererseits haben wir (Teil (3) von Lemma 1.1.5)

$$\text{opt} \geq \frac{1}{m} \sum_i d_i \geq \frac{1}{m} \sum_{i=1}^{k} d_i \geq \frac{1}{m} \sum_{i=1}^{k} d_\ell = \frac{k}{m} d_\ell, \tag{4.2}$$

was sofort aus der Sortierung der Jobs nach absteigenden Laufzeiten folgt. (4.2) impliziert also

$$d_\ell \leq \frac{m}{k}\text{opt}. \tag{4.3}$$

Fügen wir (4.1) und (4.3) zusammen, so erhalten wir

$$C \leq \text{opt} + \frac{m-1}{k}\text{opt} = \left(1 + \frac{m-1}{k}\right)\text{opt}.$$

Um diese Schranke auf $(1+\varepsilon)$opt zu drücken, setzen wir

$$k = \left\lceil \frac{m-1}{\varepsilon} \right\rceil = O(m).$$

(Falls $k > n$, so gibt es nur $O(m)$ Jobs; in diesem Fall berechnen wir einfach den optimalen Schedule für alle Jobs.)
Es bleibt noch zu diskutieren, wie für $k = O(m)$ Jobs der optimale Schedule berechnet werden kann. Offenbar kann jeder Job unabhängig voneinander einer der Maschinen zugeordnet werden, und durch Ausprobieren aller dieser Zuordnungen kann die beste gefunden werden. Es gibt

$$m^k = m^{O(m)}$$

solche Zuordnungen, und da m konstant war, ist dies eine konstante Anzahl. Insbesondere ist die Laufzeit von SLS(k) damit polynomiell, und wir erhalten folgenden Satz, der das Ergebnis zusammenfasst.

Satz 4.3.2 *Job-Scheduling mit konstanter Maschinenanzahl besitzt ein Approximationsschema.*

Es sollte allerdings klar sein, dass dieses Approximationsschema für grösseres m keinen praktikablen Algorithmus darstellt. Wenn etwa $m = 5$ und $\varepsilon = 0.1$, so müssen nach diesem Algorithmus etwa

$$5^{50}$$

Schedules durchprobiert werden, womit man in der Praxis niemals fertig wird. Hier lohnt es sich dann, über die Laufzeiten *exakter* Scheduling-Algorithmen nachzudenken. Die NP-Schwere eines Problems impliziert noch lange nicht, dass der beste Lösungsalgorithmus im vollständigen Durchsuchen der Menge aller zulässigen Lösungen besteht.
Das Ergebnis hier ist von theoretischer Natur, indem es die Existenz eines Approximationsschemas zeigt. In der Praxis muss ein solches allerdings wesentlich anders aussehen.

4.4 Uebungen

Uebung 4.1 Wie kann man den Algorithmus ExactKnapsack so anpassen, dass er in Zeit $O(n\text{opt})$ auch die Menge der Gegenstände bestimmt, die den optimalen Wert definieren?

Uebung 4.2 Zeige, dass es für das Rucksackproblem keinen Approximationsalgorithmus mit konstantem additiven Fehler geben kann. D.h. es gibt keinen Algorithmus A, so dass bei jeder Instanz I gilt $|A(I) - \text{opt}(I)| \leq k$, für ein festes $k > 0$.

Uebung 4.3 Gegeben sei ein Optimierungsproblem, bei dem für alle Instanzen I alle zulässigen Lösungen nur ganzzahlige Werte annehmen. Ausserdem sei das Optimum jeder Instanz eine Zahl zwischen 1 und einer Konstanten B. Zeige, dass es dann kein Approximationsschema für das Problem geben kann, vorausgesetzt, das Problem ist NP-schwer und P$\neq$NP.

Kapitel 5

Max-SAT und Randomisierung

Wir werden im folgenden *randomisierte* Algorithmen kennenlernen, das sind Algorithmen, die einen Zufallsgenerator verwenden. Es stellt sich heraus, dass solche Algorithmen oft sehr einfach sind, aber verblüffend gute Ergebnisse liefern.
Das erstes Problem, für das wir einen randomisierten Approximationsalgorithmus betrachten möchten, ist

Problem 5.0.1 (Max-$\geq k$-SAT) *Eine Instanz dieses Problems ist durch eine Menge $x_1, \ldots, x_n$ von Booleschen Variablen sowie durch eine Menge $C_1, \ldots, C_m$ von Klauseln über diesen Variablen gegeben. Jede Klausel ist dabei eine Disjunktion*

$$C_i = \ell_{i,1} \vee \cdots \vee \ell_{i,k_i}, k_i \geq k$$

von mindestens k Literalen *ℓ_{ij}. Gesucht ist eine Belegung der Variablen, die möglichst viele der Klauseln erfüllt.*

Wir nehmen in diesem Kapitel an, dass keine Klausel zwei Literale enthält, die von der gleichen Variablen kommen.

Beispiel 5.0.2 *Das Problem über drei Variablen x_1, x_2, x_3 mit Klauseln*

$$C_1 = x_1 \vee \neg x_2 \vee x_3, \quad C_2 = x_1 \vee \neg x_3, \quad C_3 = x_2 \vee \neg x_3$$

ist eine Instanz von Max-$\geq$ 2-SAT. Die Belegung $x_1 = \texttt{true}, x_2 = \texttt{true}, x_3 = \texttt{false}$ ist eine optimale Belegung, weil sie alle drei Klauseln erfüllt.

Das Problem Max-$\geq k$-SAT ist NP-schwer, wir interessieren uns also für eine gute Approximation. Der folgende Algorithmus RandomSat liefert im Erwartungswert eine solche.

Algorithmus 5.0.3
RandomSat:
    FOR $i := 1$ TO n DO

```
        Wähle ein Zufallsbit z ∈ {0,1}
        IF z = 0 THEN
            belege Variable x_i mit false
        ELSE
            belege Variable x_i mit true
        END
    END
    Gib die erhaltene Belegung aus.
```

Dieser Algorithmus sieht sich die Klauseln überhaupt nicht an; er wirft für jede Variable eine Münze und setzt sie damit auf einen zufälligen Wert. Trotzdem liefert er ein gutes Ergebnis.

Satz 5.0.4 *Die erwartete Anzahl erfüllter Klauseln unter der Belegung, die von* RandomSat *berechnet wird, beträgt mindestens*

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

Der Satz besagt nicht, dass *immer* mindestens $(1-1/2^k)m$ Klauseln erfüllt werden; der Erwartungswert der Anzahl erfüllter Klauseln hat aber mindestens diese Grösse.

Beweis: Wir definieren Zufallsvariablen X und $X_i, i = 1, \ldots, n$. X bezeichnet dabei die Gesamtanzahl erfüllter Klauseln bei einer Belegung von RandomSat. Für X_i setzen wir

$$X_i := \begin{cases} 1, & \text{falls } C_i \text{ erfüllt wird} \\ 0, & \text{sonst} \end{cases}.$$

Es gilt offensichtlich

$$X = \sum_{i=1}^{n} X_i.$$

Um den Satz zu beweisen, müssen wir nun zeigen, dass $E(X)$, der Erwartungswert von X, mindestens die behauptete Grösse hat. Dazu argumentieren wir wie folgt.

$$E(X) = E(\sum_{i=1}^{n} X_i) = \sum_{i=1}^{n} E(X_i) = \sum_{i=1}^{n} \text{prob}(X_i = 1).$$

Die zweite Gleichheit ist einfach die Linearität des Erwartungswertes, die dritte Gleichheit nutzt aus, dass der Erwartungswert einer 0-1-Zufallsvariable gleich der Wahrscheinlichkeit ist, dass die Zufallsvariable den Wert 1 annimmt. Was ist nun diese Wahrscheinlichkeit für die Zufallsvariable X_i? Klar ist, dass C_i genau dann nicht erfüllt wird, wenn alle Literale in C_i den Wert `false` haben. Da dies für

alle Literale unabhängig voneinander mit Wahrscheinlichkeit 1/2 passiert und es k_i Literale in C_i gibt, erhalten wir

$$\text{prob}(X_i = 0) = \frac{1}{2^{k_i}}.$$

Für dieses Argument brauchen wir die zu Beginn gemachte Voraussetzung, dass keine zwei Literale in C_i von der gleichen Variablen kommen. Mit $k_i \geq k$ folgt nun

$$\text{prob}(X_i = 1) = 1 - \frac{1}{2^{k_i}} \geq 1 - \frac{1}{2^k},$$

und der Satz folgt durch Aufsummieren dieser Wahrscheinlichkeiten. □

Nachdem wir nun eine Vorstellung davon haben, was ein randomisierter Approximationsalgorithmus ist, wollen wir eine formale Definition aufstellen.

Definition 5.0.5 *Ein* randomisierter Approximationsalgorithmus *für ein Problem ist ein Approximationsalgorithmus für das Problem, der zusätzlich Zufallsentscheidungen treffen darf. Das übliche Modell ist, dass in polynomieller Zeit eine Zufallszahl im Bereich* $\{1, \ldots, n\}$ *gezogen werden kann, wobei* n *eine natürliche Zahl ist, deren Codierungslänge polynomiell in der Grösse der Instanz sein muss (sonst kann das Ergebnis der Zufallsauswahl ja gar nicht in polynomieller Zeit verwertet werden).*

Wir haben es zwar implizit mit der Definition gesagt, wollen aber nocheinmal explizit darauf hinweisen, dass die Laufzeit des Algorithmus durch ein *festes* Polynom in der Instanzgrösse beschränkt sein muss – die Laufzeit muss also *immer* polynomiell sein, unabhängig von den Ausgängen der Zufallsexperimente.
Nun kommen wir noch zur Definition der erwarteten Approximationsgüte, die ganz ähnlich aussieht wie im deterministischen Fall.

Definition 5.0.6 *Sei* Π *ein Optimierungsproblem mit Instanzenmenge* $\mathcal{I}$. *Die* erwartete Approximationsgüte *eines randomisierten Approximationsalgorithmus* A *bei Eingabe von Instanz* $I \in \mathcal{I}$ *ist definiert als*

$$\delta_A(I) := \frac{E(w(A(I)))}{\text{opt}(I)},$$

wobei $w(A(I))$ *der Wert der von* A *berechneten Lösung ist. Sei* $\delta : \mathcal{I} \mapsto \mathbf{R}^+$ *eine Funktion.* A *hat* erwartete Güte δ*, falls für alle Instanzen* I *gilt:*

$$\delta_A(I) \leq \delta(I) \text{ (Minimierungsproblem)}$$

bzw.

$$\delta_A(I) \geq \delta(I) \text{ (Maximierungsproblem).}$$

Diese Definition funktioniert übrigens auch für deterministische Algorithmen (die wir als randomisierte Algorithmen ohne Zufallsauswahlen auffassen können). In diesem Fall gilt immer $E(w(A(I))) = w(A(I))$, und wir bekommen die uns schon bekannte Definition des Approximationsfaktors.
Nach dieser Definition ist Algorithmus RandomSat offenbar ein randomisierter Approximationsalgorithmus für das Max-$\geq k$-SAT Problem mit erwarteter Approximationsgüte $(1 - 1/2^k)$, denn für jede Instanz mit m Klauseln gilt

$$\delta_A(I) \geq \frac{\left(1 - \frac{1}{2^k}\right) m}{\mathrm{opt}(I)} \geq \frac{\left(1 - \frac{1}{2^k}\right) m}{m} = \left(1 - \frac{1}{2^k}\right).$$

Eine Frage, die sich bei der erwarteten Güte ergibt, ist, mit welcher Wahrscheinlichkeit der Algorithmus diese Güte wirklich erreicht. Mit anderen Worten: was sagt der Erwartungswert über das wirkliche Verhalten des Algorithmus aus? Wir werden auf dieses Thema später noch ausführlicher eingehen. Im Fall von Max-$\geq k$-SAT sind wir allerdings in der glücklichen Situation, dass es auch einen deterministischen Approximationsalgorithmus für das Problem gibt, der Güte $(1 - 1/2^k)$ hat, und der damit immer eine gute Lösung garantiert. Dieser Algorithmus beruht allerdings entscheidend auf dem Algorithmus RandomSat – genauer gesagt entsteht er durch *Derandomisierung* desselben. Dies ist eine allgemeine Technik die oft anwendbar ist, um einem randomisierten Algorithmus nachträglich den Zufallsgenerator wieder 'herauszuoperieren'. Hier ist die derandomisierte Version. Sie belegt wie zuvor die Variablen nacheinander mit Werten $b_1, \ldots, b_n$, diesmal aber nicht zufällig, sondern nach einer festen Regel.

Algorithmus 5.0.7 *Wie zuvor bezeichnet X die Zufallsvariable für die Gesamtzahl erfüllter Klauseln bei einer Belegung. Der Algorithmus berechnet bedingte Erwartungswerte von X unter der Voraussetzung, dass gewisse Variablen schon gewisse Werte haben. Die bedingte Erwartung von X unter Voraussetzung B wird mit $E(X|B)$ bezeichnet; wie bedingte Erwartungswerte definiert sind und wie man sie in unserem Fall berechnen kann, ist Teil der Uebungen (Uebung 5.2).*

DetSat:

```
FOR i := 1 TO n DO
    berechne E_0 = E(X|x_j = b_j, j = 1...i-1, x_i = false)
    berechne E_1 = E(X|x_j = b_j, j = 1...i-1, x_i = true)
    IF E_0 ≥ E_1 THEN
        b_i := false
    ELSE
        b_i := true
    END
END
Gib die Belegung {x_i = b_i, i = 1,...,n} aus.
```

Satz 5.0.8 *Algorithmus* DetSat *liefert eine Belegung, die mindestens* $E(X)$ *Klauseln erfüllt, also mindestens* $(1 - 1/2^k)m$ *viele.*

Beweis: Definiere

$$E^i = E(X|x_j = b_j, j = 1, \ldots, i).$$

Es gilt $E^0 = E(X)$, und E^n ist genau die Anzahl der Klauseln, die durch die von DetSat gefundene Belegung erfüllt werden. Wenn wir nun zeigen können, dass $E^i \geq E^{i-1}$ für alle $i = 1, \ldots, n$ gilt, folgt der Satz. Hierzu verwenden wir den Partitionssatz für bedingte Erwartungen (siehe Uebung 5.1). Seien E_0 und E_1 die bedingten Erwartungswerte, die der Algorithmus in Iteration i berechnet. Dann gilt

$$\begin{aligned}
E^{i-1} &= E(X|x_j = b_j, j = 1, \ldots, i-1) \\
&= \frac{1}{2}E(X|x_j = b_j, j = 1, \ldots, i-1, x_i = \texttt{false}) + \\
&\quad \frac{1}{2}E(X|x_j = b_j, j = 1, \ldots, i-1, x_i = \texttt{true}) \\
&= \frac{1}{2}E_0 + \frac{1}{2}E_1 \\
&\leq \max(E_0, E_1) \\
&= E(X|x_j = b_j, j = 1, \ldots, i) \\
&= E^i.
\end{aligned}$$

□

Dieser Algorithmus ist also mindestens so gut wie der randomisierte Algorithmus, von dem er abgeleitet ist. Ist er vielleicht sogar besser? Dies kann nicht sein, sofern P≠NP ist, denn unter dieser Voraussetzung kann gezeigt werden, dass es keinen Approximationsalgorithmus für Max-$\geq k$-SAT gibt, der einen Approximationsfaktor von

$$1 - \frac{1}{2^k} + \varepsilon$$

erreicht, für irgendein $\varepsilon > 0$. Wir haben hier also das Phänomen, das man mit einem sehr einfachen Algorithmus bereits das bestmögliche Resultat erhält.

5.1 Uebungen

Uebung 5.1 Sind A, B zwei Ereignisse über einem Wahrscheinlichkeitsraum, so ist die *bedingte Wahrscheinlichkeit* $\text{prob}(A|B)$ (Wahrscheinlichkeit, dass A eintritt, unter der Voraussetzung, dass B eintritt) durch

$$\text{prob}(A|B) := \frac{\text{prob}(A \cap B)}{\text{prob}(B)}$$

definiert. Der *bedingte Erwartungswert* $E(X|B)$ einer Zufallsvariablen X bzgl. des Ereignisses B ist dann über die Formel

$$E(X|B) = \sum_x x \operatorname{prob}(\{X = x\}|B)$$

erklärt. Beweise den folgenden *Partitionssatz* für eine Zufallsvariable X, ein Ereignis B und eine Partition $B_1, \ldots, B_n$ von B.

$$E(X|B) = \sum_{i=1}^{n} E(X|B_i) \operatorname{prob}(B_i|B).$$

Uebung 5.2 Gib einen polynomiellen Algorithmus an, der für gegebene Klauseln $C_1, \ldots, C_m$ über den Variablen $x_1, \ldots, x_n$ und boolesche Werte $b_1, \ldots, b_i, i \leq n$ den bedingten Erwartungswert

$$E_i := E(X|\{x_j = b_j, j = 1 \ldots i\})$$

berechnet, wobei X die Zufallsvariable ist, die die Anzahl erfüllter Klauseln angibt.

Uebung 5.3 Gib einen randomisierten Approximationsalgorithmus für das Max-Cut Problem an, der einen erwarteten Approximationsfaktor von $1/2$ erreicht. Kann der Algorithmus derandomisiert werden? Wenn ja, formuliere den resultierenden deterministischen Algorithmus ohne Bezugnahme auf Wahrscheinlichkeiten und Erwartungswerte.

Kapitel 6

LP-Relaxierungen

In diesem Kapitel werden wir eine allgemeine und häufig sehr erfolgreiche Technik für Approximationsalgorithmen kennenlernen, die sogenannten *LP-Relaxierungen*. Oft wird diese Technik mit einer anderen Technik, dem *randomisierten Runden*, verknüpft, um randomisierte Approximationsalgorithmen zu erhalten. Wir wollen diese Techniken am Beispiel des Problems Max-SAT einführen.

6.1 Max-SAT und LP-Relaxierungen

Das Problem Max-SAT ist einfach das schon definierte Problem Max-$\geq$ 1-SAT, d.h. wir haben eine Boolesche Formel in konjunktiver Normalform und sollen eine Belegung der Variablen finden, die möglichst viele Klauseln erfüllt. Die Algorithmen RandomSAT und DetSAT sind für diesen Spezialfall Approximationsalgorithmen mit Güte 1/2. Wir werden einen Algorithmus beschreiben, der Güte 3/4 erreicht.
Gegeben ist eine Boolesche Formel φ über den Variablen $x_1, \ldots, x_n$ und mit Klauseln $C_1, \ldots, C_m$. Für jede Klausel C_j definieren wir

$$\begin{aligned} V_j^+ &:= \text{ Menge der unnegierten Variablen in } C_j \\ V_j^- &:= \text{ Menge der negierten Variablen in } C_j \end{aligned}$$

Als ersten Schritt zur Entwicklung eines Approximationsalgorithmus dient uns hier eine Umformulierung des Problems als *Mathematisches Programm*. Wir führen dazu für jede Boolesche Variable x_i eine Variable y_i ein. Die Bedeutung der y_i ist wie folgt.

$$y_i = \begin{cases} 1, & \text{falls } x_i = \texttt{true} \\ 0, & \text{falls } x_i = \texttt{false} \end{cases} .$$

Analog haben wir für jede Klausel C_j eine Variable z_j. Hier soll $z_j = 1$ bedeuten, dass die Klausel C_j erfüllt ist. Entsprechend bedeutet $z_j = 0$, dass C_j nicht erfüllt ist. Wir können dann Max-SAT als das folgende Optimierungsproblem

beschreiben.

(Max-SAT)

$$
\begin{array}{lll}
\text{maximiere} & \displaystyle\sum_{j=1}^{m} z_j & \\
\text{unter} & & \\
& \displaystyle\sum_{i:x_i\in V_j^+} y_i + \sum_{i:x_i\in V_j^-} (1-y_i) \geq z_j, & j=1,\ldots,m \\
& y_i, z_j \in \{0,1\}, & i=1,\ldots,n, j=1,\ldots,m.
\end{array}
$$

Jede Belegung der Variablen y_i mit Werten aus $\{0,1\}$ definiert in natürlicher Weise eine Belegung der Booleschen Variablen x_i mit `true` oder `false`. Die erste Nebenbedingung besagt, dass in jeder Klausel C_j, die zu der zu maximierenden Funktion den Wert 1 beiträgt, mindestens ein Literal erfüllt sein muss, d.h. die Klausel erfüllt sein muss. Damit entspricht die Summe $\sum_{j=1}^{m} z_j$ der Anzahl der Klauseln, die erfüllt werden. Somit ist klar, dass es sich hier um eine äquivalent Formulierung des Max-SAT Problems handelt.
In dieser Formulierung haben wir *ganzzahlige* Variablen, eine *lineare* Zielfunktion in den Variablen und *lineare* Nebenbedingungen. Ein mathematisches Programm mit diesen Eigenschaften heisst *ganzzahliges lineares Programm* (ILP). Offenbar sind ILPs im allgemeinen NP-schwer, denn wir haben das Max-SAT Problem auf eine Instanz von ILP reduziert. Die Umformulierung als ILP bringt zunächst noch nichts. Wir betrachten nun aber die sogenannte *LP-Relaxierung* des Problems.

(Relaxiertes Max-SAT)

$$
\begin{array}{lll}
\text{maximiere} & \displaystyle\sum_{j=1}^{m} z_j & \\
\text{unter} & & \\
& \displaystyle\sum_{i:x_i\in V_j^+} y_i + \sum_{i:x_i\in V_j^-} (1-y_i) \geq z_j, & j=1,\ldots,m \\
& y_i, z_j \in [0,1], & i=1,\ldots,n, \\
& & j=1,\ldots,m.
\end{array}
$$

Hier verlangen wir nicht mehr Ganzzahligkeit der Variablen, sondern lassen zu, dass diese auch Werte zwischen 0 und 1 annehmen können. Wir haben also nur noch eine lineare Zielfunktion und lineare Nebenbedingungen (die Forderung $x_i \in [0,1]$ kann ja als lineare Ungleichung $0 \leq x_i \leq 1$ geschrieben werden). Ein Problem dieses Typs bezeichnen wir als *lineares Programm* (LP). Der entscheidene Unterschied zwischen ILP und LP besteht nun darin, dass letztere sich in polynomieller Zeit lösen lassen.

Satz 6.1.1 *(Khachyian 1980) LP ist in* P*, d.h. es existiert ein polynomieller Algorithmus für jedes lineare Programm. Die Ausgabe des Algorithmus sind optimale Werte für die Variablen des Programms.*

Dies bedeutet im Hinblick auf Approximationsalgorithmen, dass lineare Programme dort als Bausteine verwendet werden können. Für Max-SAT wird die Idee sein, die Relaxierung zu lösen und dann die erhaltenen (nicht notwendigerweise) ganzzahligen Werte für die y_i geschickt zu runden, so dass man eine zulässige Lösung für das Problem bekommt, die nicht zu weit vom Optimum entfernt ist. Dass sich das ILP und das relaxierte LP dabei sehr ähnlich sind, ist natürlich die Grundlage für eine gute Approximation.
Bevor wir den Approximationsalgorithmus für Max-SAT beschreiben, soll noch kurz eine Motivation gegeben werden, warum LP leichter ist als ILP. Wir werden dann im weiteren die polynomielle Lösbarkeit von LP als 'black box' verwenden. Betrachte das Problem SUBSET SUM. Dabei ist eine Menge $q_1, \ldots, q_n$ von natürlichen Zahlen und ein Zielwert $k \leq Q := \sum_{i=1}^{n} q_i$ gegeben, und die Frage ist, ob es eine Teilmenge der Zahlen gibt, deren Summe genau k ist. Dieses Problem ist NP-vollständig und kann als ILP wie folgt geschrieben werden (wir brauchen dabei nicht einmal eine Zielfunktion).

$$\begin{array}{lll} \text{(Subset-Sum)} & \text{Finde} & x_1, \ldots, x_n \\ & \text{mit} & \\ & & \displaystyle\sum_{i=1}^{n} q_i x_i = k, \\ & & x_i \in \{0,1\}, \quad i = 1, \ldots, n. \end{array}$$

Während dieses Problem also NP-vollständig ist, ist die relaxierte Version

$$\begin{array}{lll} \text{(Relaxiertes Subset Sum)} & \text{Finde} & x_1, \ldots, x_n \\ & \text{mit} & \\ & & \displaystyle\sum_{i=1}^{n} q_i x_i = k, \\ & & x_i \in [0,1], \quad i = 1, \ldots, n \end{array}$$

trivial zu lösen: man setze einfach $x_i = k/Q$ für alle i.
Dieses Beispiel ist natürlich kein formales Argument dafür, dass LPs im allgemeinen leicht zu lösen sind (die bekannten polynomiellen Algorithmen sind übrigens auch relativ kompliziert); es soll einfach nur eine gewisse Intuition für das Verhältnis zwischen ILP und LP geben.
Nach diesem kleinen Exkurs nun zurück zu unserem Problem Max-SAT. Wir werden nun einen Algorithmus zur Lösung des Problems Max-SAT beschreiben, der zunächst das Relaxierte Max-SAT optimal löst, und dann die so erhaltene Lösung für Relaxiertes Max-SAT in eine zulässige Lösung von Max-SAT zu transformieren versucht. Der Approximationsalgorithmus für Max-SAT sieht dann folgendermassen aus.

Algorithmus 6.1.2

RR Max-SAT *(Randomized Rounding Max-SAT):*

Finde eine optimale Lösung $(\hat{y}_1, \ldots, \hat{y}_n), (\hat{z}_1, \ldots, \hat{z}_n)$
des Relaxierten Max-SAT.

```
FOR i := 1 TO n DO
    Wähle z ∈ {0,1} zufällig, so dass
            z = { 1,  mit Wahrscheinlichkeit ŷ_i
                { 0,  mit Wahrscheinlichkeit 1 − ŷ_i
    IF z = 1 THEN
        x_i := false
    ELSE
        x_i := true
    END
END
Gib die erhaltene Belegung aus.
```

$$z = \begin{cases} 1, & \text{mit Wahrscheinlichkeit } \hat{y}_i \\ 0, & \text{mit Wahrscheinlichkeit } 1 - \hat{y}_i \end{cases}$$

Der Algorithmus interpretiert die Lösung $\hat{y}_i$ des Relaxierten Max-SAT als Wahrscheinlichkeiten, und rundet dann die einzelnen Variablen y_i gemäss diesen Wahrscheinlichkeiten zu 0 und 1, bzw. der Algorithmus "rundet" die Variablen x_i zu `true` und `false`. Diese Technik aus einer zulässigen Lösung eines linearen Programms eine zulässigen Lösung des ganzzahligen linearen Programms zu erhalten, nennt man *randomisiertes Runden.*

Dieser Algorithmus hat sicherlich polynomielle Laufzeit. Für die Güte des Algorithmus erhalten wir den folgenden Satz.

Satz 6.1.3 *RR Max-SAT hat erwartete Approximationsgüte* $1 - 1/\mathrm{e}$.

Da $1 - 1/\mathrm{e} \geq 1/2$ ist dieses schon eine Verbesserung gegenüber den Algorithmen RandomSAT und DetSAT, jedoch noch nicht der angekündigte Algorithmus mit Güte 3/4.

Beweis von Satz 6.1.3: Sei die Boolesche Formel φ eine Instanz von Max-SAT. Dann ist die maximale Anzahl gleichzeitig erfüllbarer Klauseln $\mathrm{opt}(\varphi)$ in φ beschränkt durch $\sum_{j=1}^m \hat{z}_j$, denn das Optimum von Relaxiertem Max-SAT ist sicher so gross wie das Optimum von Max-SAT. Für $j = 1, \ldots, m$, sei $X_j = 1$, falls die von RR Max-SAT gefundene Belegung die Klausel C_j erfüllt. Sonst sei $X_j = 0$. Dann gilt

$$\mathrm{prob}(X_j = 0) = \prod_{i: x_i \in V_j^+} (1 - \hat{y}_i) \prod_{i: x_i \in V_j^-} (1 - (1 - \hat{y}_i)).$$

Nach Definition von Relaxiertem Max-SAT gilt

$$\sum_{i: x_i \in V_j^+} \hat{y}_i + \sum_{i: x_i \in V_j^-} 1 - \hat{y}_i \geq \hat{z}_j.$$

Wir wenden nun folgendes Lemma an, dass wir im Anschluss an diesen Beweis herleiten werden.

Lemma 6.1.4 *Seien $y_i, i = 1, \ldots, k$, positive reelle Zahlen mit $\sum_{i=1}^{k} y_i \geq y$. Dann gilt:*

$$\prod_{i=1}^{k} (1 - y_i) \leq (1 - y/k)^k.$$

Wir erhalten

$$\text{prob}(X_j = 0) \leq (1 - \hat{z}_j/k_j)^{k_j},$$

wobei k_j die Anzahl der Literale in C_j ist. Eine Uebungsaufgabe (Uebung 6.1) ist es, das folgende Lemma zu zeigen.

Lemma 6.1.5 *Für alle $k \in \mathbf{N}$ und alle $0 \leq x \leq 1$ gilt*

$$1 - \left(1 - \frac{x}{k}\right)^k \geq \left[1 - \left(1 - \frac{1}{k}\right)^k\right] x.$$

Anwendung des Lemmas liefert nun

$$\text{prob}(X_j = 1) \geq 1 - (1 - \hat{z}_j/k_j)^{k_j} \geq (1 - (1 - 1/k_j)^{k_j})\hat{z}_j \geq (1 - 1/\text{e})\hat{z}_j.$$

Damit ist die erwartete Anzahl der Klauseln, die durch die Belegung von RR Max-SAT erfüllt werden, nach unten durch

$$(1 - 1/\text{e}) \sum_{j=1}^{m} \hat{z}_j \geq (1 - 1/\text{e})\text{opt}(\varphi).$$

beschränkt. □

Nun der noch fehlende Beweis von Lemma 6.1.4.

Beweis von Lemma 6.1.4: Die Ungleichung vom arithmetischen und geometrischen Mittel besagt, dass für positive reelle Zahlen $a_i, i = 1, \ldots, k$,

$$\frac{1}{k} \sum_{i=1}^{k} a_i \geq \left(\prod_{i=1}^{k} a_i\right)^{1/k}.$$

Angewandt mit $a_i = (1 - y_i)$ zeigt diese Ungleichung das Lemma. □

Wir wollen nun den versprochenen 3/4-Approximationsalgorithmus herleiten.

Satz 6.1.6 *Wendet man* RandomSAT *und* RR Max-SAT *beide auf eine Boolesche Formel φ an und nimmt dann von den beiden zulässigen Lösungen diejenige, die mehr Klauseln erfüllt, so liefert dieses einen Approximationsalgorithmus mit Güte 3/4.*

Beweis: Der kombinierte Algorithmus hat immer noch polynomielle Laufzeit. Um die Güte abzuschätzen verwenden wir folgendes Lemma, das wir im Anschluss an diesen Beweis herleiten werden.

Lemma 6.1.7 *Für alle $k \in \mathbf{N}$ gilt*

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

Zum Beweis des Satzes sei n_1 die erwartete Anzahl von Klauseln, die von der durch RandomSAT gefundenen Belegung erfüllt werden. Für RR Max-SAT sei dieser Erwartungswert mit n_2 bezeichnet. Wir müssen $\max\{n_1, n_2\} \geq 3/2\text{opt}(\varphi)$ zeigen.
Mit k_j bezeichnen wir die Anzahl der Literale in der Klausel C_j von φ. Aus dem Beweis von Satz 6.1.3 folgt

$$n_2 \geq \sum_{j=1}^{m} (1 - (1 - 1/k_j)^{k_j}) \hat{z}_j,$$

wobei $\hat{z}_j$ wie im Beweis von Satz 6.1.3 definiert ist. Für n_1 hatten wir bei der Analyse von RandomSAT gezeigt

$$n_1 = \sum_{j=1}^{m} (1 - 1/2^{k_j}).$$

Da $\hat{z}_j \leq 1$ gilt

$$n_1 \geq \sum_{j=1}^{m} (1 - 1/2^{k_j}) \hat{z}_j.$$

Jetzt erhalten wir

$$\max\{n_1, n_2\} \geq \frac{n_1 + n_2}{2} \geq \sum_{j=1}^{m} \left[\left(1 - \left(1 - \frac{1}{k_j}\right)^{k_j}\right) + \left(1 - \frac{1}{2^{k_j}}\right) \right] \hat{z}_j.$$

Mit Lemma 6.1.7 und $\sum_{j=1}^{m} \hat{z}_j \geq \text{opt}(\varphi)$ gilt dann

$$\max\{n_1, n_2\} \geq \frac{3}{4} \sum_{j=1}^{m} \hat{z}_j \geq \frac{3}{4}\text{opt}(\varphi).$$

□

Jetzt noch der
Beweis von Lemma 6.1.7: Die Aussage des Lemmas ist äquivalent zu

$$\left(1 - \frac{1}{k}\right)^k \leq \frac{1}{2} - \frac{1}{2^k}.$$

Man prüft nun leicht nach, dass

$$\binom{k}{i}\frac{1}{k^i} \geq \binom{k}{i+1}\frac{1}{k^{i+1}}$$

für alle i. Daher gilt

$$(1-1/k)^k = \sum_{i=1}^{k}\binom{k}{i}(-1/k)^i \leq 1 - \binom{k}{1}\frac{1}{k} + \binom{k}{2}\frac{1}{k^2},$$

denn nach der obigen Ungleichung können die weggelassenen Terme zu Paaren zusammengefasst werden, so dass die Summe der beiden Terme eines Paares negativ ist. Ist k ungerade, so gibt es für den letzten Term keinen Partner. Aber der letzte Term ist dann negativ. Den Beweis abschliessen können wir durch

$$1 - \binom{k}{1}\frac{1}{k} + \binom{k}{2}\frac{1}{k^2} = \frac{1}{2} - \frac{1}{2k}.$$

□

6.2 Set Cover und LP-Relaxierungen

Als nächste Anwendung von LP-Relaxierungen und randomisiertem Runden wollen wir uns das Problem Set Cover anschauen.

Problem 6.2.1 (SET-COVER) *Eine Instanz von Set Cover besteht aus einer Grundmenge $V = \{v_1, \ldots, v_n\}$ sowie einer Menge $S_1, \ldots, S_m$ von Teilmengen von V. Gesucht ist dann die kleinste Teilmenge $H \subseteq V$ mit der Eigenschaft*

$$H \cap S_i \neq \emptyset, i = 1, \ldots, m.$$

Eine Menge H mit dieser Eigenschaft wird auch als *hitting set* bezeichnet, weil sie alle Teilmengen S_i 'trifft'. Das Set Cover Problem ist damit offenbar eine Verallgemeinerung des Vertex Cover Problems, bei dem wir $|S_i| = 2$ für alle i haben. Daraus folgt dann auch, dass Set Cover NP-schwer ist.
Um wie bei Max-SAT das Problem Set Cover als mathematisches Programm zu formulieren, führen wir Variablen $x_1, \ldots, x_n$ ein, mit folgender Bedeutung:

$$x_i = \begin{cases} 1, & \text{falls } v_i \in H_{opt} \\ 0, & \text{falls } v \notin H_{opt} \end{cases}.$$

Dabei bezeichnet H_{opt} eine feste optimale Lösungsmenge für das Set Cover Problem. Wir können nun Set Cover als folgendes Optimierungsproblem beschreiben.

$$\text{(Set Cover)} \quad \text{minimiere} \quad \sum_{i=1}^{n} x_i$$
$$\text{unter} \quad \sum_{i:v_i \in S_j} x_i \geq 1, \quad j = 1, \ldots, m$$
$$x_i \in \{0,1\}, \quad i = 1, \ldots, n.$$

Die relaxierte Version sieht folgendermassen aus.

$$\text{(Relaxiertes Set Cover)} \quad \text{minimiere} \quad \sum_{i=1}^{n} x_i$$
$$\text{unter} \quad \sum_{i:v_i \in S_j} x_i \geq 1, \quad j = 1, \ldots, m$$
$$x_i \in [0,1], \quad i = 1, \ldots, n.$$

Dieses führt zu dem folgenden Approximationsalgorithmus für Set Cover.

Algorithmus 6.2.2
RRSC *(Randomized Rounding Set Cover):*

$H := \emptyset$
Finde eine optimale Lösung $(\hat{x}_1, \ldots, \hat{x}_n)$ des Relaxierten Set Cover.
FOR $i := 1$ TO $\lceil \log(2m) \rceil$ DO
  FOR $j := 1$ TO n DO
    Wähle $z \in \{0,1\}$ zufällig, so dass
$$z = \begin{cases} 1, & \text{mit Wahrscheinlichkeit } \hat{x}_j \\ 0, & \text{mit Wahrscheinlichkeit } 1 - \hat{x}_j \end{cases}$$
    IF $z = 1$ THEN
      $H := H \cup \{v_j\}$
    END
  END
END
Gib H aus.

Der Algorithmus RRSC hat polynomielle Laufzeit. Ausserdem gilt der folgende Satz.

Satz 6.2.3 *Bei jeder Instanz I von Set Cover gilt*

(1) *RRSC findet mit Wahrscheinlichkeit mindestens $1/2$ eine zulässige Lösung.*

(2) *Ist* $\text{opt}(I)$ *die Grösse einer optimalen Lösung, so ist*

$$E(|\text{RRSC(I)}|) \leq \lceil \log(2\text{m}) \rceil \text{opt(I)}.$$

Da RRSC gelegentlich keine zulässige Lösung findet, ist dieser Algorithmus also nach unserem bisherigen Verständnis kein Approximationsalgorithmus. Wir werden jedoch in Kürze unsere Definition eines Approximationsalgorithmus ändern, um auch Algorithmen wie RRSC zuzulassen. Man beachte, dass wir immer in polynomieller Zeit überprüfen können, ob RRSC eine zulässige Lösung gefunden hat, denn die Entscheidungsvariante von Set Cover liegt in NP. Dieses bedeutet ja gerade, dass überprüft werden kann, ob eine zulässige Lösung vorliegt. Doch nun der

Beweis von Satz 6.2.3: Sei I eine Instanz von Set Cover. Wir betrachten RRSC bei Eingabe I. Es gilt

$$\text{opt}(I) \geq \sum_{j=1}^{n} \hat{x}_j,$$

denn das Optimum der Relaxierung von Set Cover ist sicherlich kleiner als das des ursprünglichen Problems. Weiter sei H_i die Menge der Knoten, die RRSC im i-ten Durchlauf der äusseren Schleife der Menge H hinzufügt. Hierbei zählen wir auch die v_j, die zu Beginn des i-ten Schleifendurchlaufs schon in H sind, die RRSC aber auch im i-ten Durchlauf noch einmal zu H hinzuzufügen versucht.

$$E(|H_i|) = \sum_{j=1}^{n} \hat{x}_j \leq \text{opt}(I).$$

Hieraus folgt

$$E(|\text{RRSC(I)}|) = E(|H|) \leq \sum_{i=1}^{\lceil \log(2m) \rceil} E(|H_i|) \leq \lceil \log(2m) \rceil \text{opt}(I).$$

Damit ist Teil (2) des Satzes bereits bewiesen.
Um Teil (1) zu beweisen, setze $|S_j| = k_j, j = 1, \ldots, m$. Nun gilt

$$\text{prob}(S_j \cap H_i = \emptyset) = \prod_{l:v_l \in S_j} (1 - \hat{x}_l).$$

Mit Lemma 6.1.4 und der Ungleichung $\sum_{l:v_l \in S_j} \hat{x}_l \geq 1$, die aus der Definition des Relaxierten Set Cover folgt, erhalten wir

$$\text{prob}(S_j \cap H_i = \emptyset) \leq (1 - 1/k_j)^{k_j} \leq 1/\text{e}.$$

Dann gilt

$$\text{prob}(S_j \cap H = \emptyset) = \text{prob}(S_j \cap H_i = \emptyset \text{ für alle } i) \leq (1/\text{e})^{\lceil \log(2m) \rceil} \leq 1/(2m).$$

Somit erhalten wir schliesslich

$$\text{prob(Es existiert ein } S_j \text{ mit } S_j \cap H = \emptyset) \leq \frac{m}{2m} = \frac{1}{2}.$$

Dann aber ist die Wahrscheinlichkeit, dass $S_j \cap H \neq \emptyset$ für alle $j = 1, \ldots, m$, mindestens 1/2. Dieses zeigt Teil (1) des Satzes. □

Wir wollen nun die Definition von Approximationsalgorithmen so verallgemeinern, dass sie auch den gerade behandelten Algorithmus RRSC miteinschliesst. Im Anschluss daran werden wir ein weiteres Beispiele für LP-Relaxierungen kennenlernen.

6.3 Eine verallgemeinerte Definition von Approximationsalgorithmen

Der 'Approximationsalgorithmus' für Set Cover, den wir kennegelernt haben, ist strenggenommen kein Approximationsalgorithmus im Sinne unserer bisherigen Definitionen. Der Algorithmus berechnete ja nur mit einer gewissen Wahrscheinlichkeit eine zulässige Lösung. Wir wollen bei einem Approximationsalgorithmus in Zukunft genau solch ein Verhalten zulassen. Der folgende Satz zeigt, dass diese Erweiterung Sinn macht, sofern die 'Erfolgswahrscheinlichkeit' für das Finden einer zulässigen Lösung nicht zu klein ist.

Satz 6.3.1 *Sei p ein festes Polynom und A ein polynomieller Algorithmus, der für jede Instanz eines Optimierungsproblems mit Wahrscheinlichkeit mindestens $1/p(|I|)$ eine zulässige Lösung liefert. Dann gibt es für jedes $\varepsilon > 0$ einen polynomiellen Algorithmus A_ε, der mit Wahrscheinlichkeit mindestens $1 - \varepsilon$ eine zulässige Lösung liefert.*

Dies bedeutet, dass die Erfolgswahrscheinlichkeit beliebig nahe an 1 gebracht werden kann – in diesem Fall ist der Algorithmus sicher geeignet, um fast immer eine Approximation zu berechnen.

Beweis: Die Idee zur Konstruktion von A_ε ist einfach: A wird einfach immer wieder aufgerufen, solange bis eine zulässige Lösung gefunden wurde (für Probleme in NP kann man das immer prüfen) oder eine Maximalzahl von Iterationen erreicht ist. Wählt man diese Maximalzahl geeignet, ergibt sich die gewünschte Erfolgswahrscheinlichkeit. Hier ist der Algorithmus A_ε.

Algorithmus $A_\varepsilon(I)$:
  `FOR` $i := 1$ `TO` $\lceil p(|I|) \ln(1/\varepsilon) \rceil$ `DO`
    berechne eine Lösung S mittels $A(I)$
    `IF` S ist zulässig `THEN`
      gib S aus und brich den Algorithmus ab

```
    END
  END
```

Die Wahrscheinlichkeit, dass A_ε versagt, also in keiner der $k := \lceil p(|I|)\ln(1/\varepsilon)\rceil$ Iterationen eine zulässige Lösung findet, ist höchstens

$$\left(1 - \frac{1}{p(|I|)}\right)^k,$$

denn $1 - 1/p(|I|)$ ist ja eine obere Schranke für die Wahrscheinlichkeit des Versagens in einer festen Iteration. Mit der Ungleichung $1 + x \leq \exp(x)$ gilt nun weiter

$$\begin{aligned}\left(1 - \frac{1}{p(|I|)}\right)^k &\leq \exp(-k/p(|I|)) \\ &= \exp(-\lceil p(|I|)\ln(1/\varepsilon)\rceil/p(|I|)) \\ &\leq \exp(-p(|I|)\ln(1/\varepsilon)/p(|I|)) \\ &= \exp(-\ln(1/\varepsilon)) \\ &= \exp(\ln\varepsilon) = \varepsilon.\end{aligned}$$

Die Erfolgswahrscheinlichkeit ist also mindestens $1 - \varepsilon$, was zu zeigen war. □

Offenbar hat A_ε auch polynomielle Laufzeit. Hier geht ein, dass die Erfolgswahrscheinlichkeit von A nicht beliebig klein sein darf. Wäre sie etwa von der Ordnung $1/\exp(|I|)$, so könnte man auf die hier gezeigte Weise keinen polynomiellen Approximationsalgorithmus mit hoher Erfolgswahrscheinlichkeit konstruieren.
Der Satz führt uns nun zu einer erweiterten Definition des Begriffs des Approximationsalgorithmus, bei dem wir gewisse Fehler erlauben.

Definition 6.3.2 *Ein* probabilistischer Approximationsalgorithmus *für ein Optimierungsproblem ist ein polynomieller Algorithmus, der mit Wahrscheinlichkeit mindestens* $1/2$ *eine zulässige Lösung ausgibt.*

Die Konstante $1/2$ ist natürlich willkürlich gewählt, ist nach Satz 6.3.1 aber ebenso gut wie $1 - \varepsilon$ oder $1/p(|I|)$.
Eine weitere Frage, die wir uns schon im Zusammenhang mit randomisierten Approximationsalgorithmen gestellt haben, ist die folgende: Was sagt die erwartete Güte eines Algorithmus über dessen 'typisches' Verhalten aus? Es ist ja vorstellbar, dass ein Algorithmus zwar eine gewisse erwartete Güte besitzt, diese aber nur mit sehr kleiner Wahrscheinlichkeit wirklich erreicht wird. Das typische Verhalten könnte dann in der Berechnung einer sehr viel schlechteren Approximation bestehen. Mit einem ähnlichen Trick wie oben können wir aber argumentieren, dass uns die erwartete Güte doch einen vernünftigen Anhaltspunkt bietet, zumindest im Fall von Minimierungsproblemen.

Satz 6.3.3 *Sei A ein randomisierter Approximationsalgorithmus für ein Minimierungsproblem mit erwarteter Güte δ. Dann gibt es für jedes $\varepsilon > 0$ und jedes $p < 1$ einen Approximationsalgorithmus $A_{\varepsilon,p}$, der für jede Instanz mit Wahrscheinlichkeit mindestens p eine Lösung mit Wert höchstens $(1+\varepsilon)\delta(I)\text{opt}(I)$ liefert.*

Das bedeutet, wir können aus A einen Algorithmus konstruieren, der fast immer eine Lösung liefert, die fast so gut ist wie man es erwartet.

Beweis: Wir nehmen zur Vereinfachung an, dass A immer eine zulässige Lösung liefert (dies ist aber keine Beschränkung der Allgemeinheit, wie man sich überlegen kann). Sei X die Zufallsvariable für den Wert der von A berechneten Lösung bei fester Instanz I. Nach der Markov-Ungleichung gilt

$$\text{prob}(X \geq (1+\varepsilon)E(X)) \leq \frac{1}{1+\varepsilon}.$$

Wähle nun $k = k(\epsilon, p)$ so, dass

$$\left(\frac{1}{1+\varepsilon}\right)^k \leq 1 - p.$$

(Den minimalen Wert für k kann man sich anhand des Beweises von Satz 6.3.1 oben überlegen.) Dann sieht der Algorithmus $A_{\varepsilon,p}$ wie folgt aus.

Algorithmus $A_{\varepsilon,p}(I)$:

FOR $i := 1$ TO k DO
    $w_i := w(A(I))$
END
Gib $w := min_{i=1}^k w_i$ aus

Wir wissen, dass

$$\text{prob}(w_i \geq (1+\varepsilon)E(X)) \leq \frac{1}{1+\varepsilon}$$

gilt, für $i = 1, \ldots, k$. Ferner gilt $w \geq (1+\varepsilon)E(X)$ genau dann, wenn dies für alle w_i gilt. Daraus folgt mit der Konstruktion von k, dass

$$\text{prob}(w \geq (1+\varepsilon)E(X)) \leq 1 - p$$

gilt, also

$$\text{prob}(w \leq (1+\varepsilon)E(X)) \geq p.$$

Mit Wahrscheinlichkeit mindestens p gilt also

$$w = w(A_{\varepsilon,p}(I)) \leq (1+\varepsilon)E(w(A(I))) \leq (1+\varepsilon)\delta(I)\text{opt}(I).$$

Die Laufzeit von $A_{\varepsilon,p}$ ist natürlich immer noch polynomiell, weil k eine Konstante ist, die nur von ε und p abhängt. ◻

Für Maximierungsprobleme gilt ein ähnlicher Satz, allerdings nur unter Zusatzvoraussetzungen über δ (siehe Übungen). Das Problem ist, dass wir in diesem Fall keine Markov-Ungleichung haben: für eine allgemeine nichtnegative Zufallsvariable X gibt es keine Schranke für

$$\text{prob}(X \leq (1-\varepsilon)E(X)),$$

die nur von ε abhängt.

6.4 Ein LP-gestützter Approximationsalgorithmus für Job-Scheduling

Wir schliessen die Behandlung der LP-Relaxierungen zunächst ab, indem wir diese Technik verwenden, um einen Approximationsalgorithmus für das schon in der Einleitung behandelte Job-Scheduling Problem zu entwickeln. Dieser wird sich ganz im 'traditionellen' Rahmen bewegen: er verwendet keinen Zufallsgenerator (funktioniert also *nicht* mit randomisiertem Runden wie die anderen zwei LP-basierten Algorithmen, die wir kennengelernt haben).

Erinnern wir uns (siehe Problem 1.1.1): Beim Scheduling geht es darum, n Jobs mit Dauern $d_i, i = 1, \ldots, n$ auf m identische Maschinen so zu verteilen, dass der *Makespan* – der Zeitpunkt, zu dem der letzte Job fertig wird – minimiert wird. Wir geben zunächst eine ILP-Formulierung dieses Problems an. Dazu betrachten wir Variablen $x_{ij} \in \{0,1\}$ mit der Bedeutung, dass $x_{ij} = 1$ genau dann gilt, wenn Job i auf Maschine j läuft. Da die Reihenfolge, in der die Jobs auf einer einzelnen Maschine abgearbeitet werden, keine Rolle spielt, gibt uns das eine ganze Familie von Schedules mit dem gleichen Makespan. Hier ist nun die ILP Formulierung. t bezeichnet den Makespan.

$$\begin{array}{lll} \text{(Scheduling)} & \text{minimiere} & t \\ & \text{unter} & \sum_{j=1}^{m} x_{ij} = 1, \quad i = 1, \ldots, n \\ & & \sum_{i=1}^{n} d_i x_{ij} \leq t, \quad j = 1, \ldots, m \\ & & x_{ij} \in \{0,1\}, \quad i = 1, \ldots, n; j = 1, \ldots, m. \end{array}$$

Die ersten n Gleichungen kodieren die Bedingung, dass jeder Job auf genau einer Maschine läuft, die folgenden m Ungleichungen garantieren, dass jede Maschine zum Zeitpunkt t fertig ist. Dann ist klar, dass ein minimales t unter diesen Bedingungen dem optimalen Makespan entspricht.

Nun betrachten wir wie üblich die LP-Relaxierung dieses ILP:

$$
\begin{array}{llll}
\text{(Relaxiertes Scheduling)} & \text{minimiere} & t & \\
& \text{unter} & \sum_{j=1}^{m} x_{ij} = 1, & i = 1,\ldots,n \\
& & \sum_{i=1}^{n} d_i x_{ij} \leq t, & j = 1,\ldots,m \\
& & x_{ij} \geq 0, & i = 1,\ldots,n; \\
& & & j = 1,\ldots,m.
\end{array}
$$

Wegen der ersten n Gleichungen sind die Bedingungen $x_{ij} \leq 1$ redundant.
Nun lösen wir die Relaxierung, allerdings sind wir an einer speziellen optimalen Lösung interessiert, deren Existenz von folgendem Lemma garantiert wird. (Es gibt auch polynomielle Algorithmen zur Lösung von LP, die solche speziellen Lösungen (*Basislösungen*) berechnen).

Lemma 6.4.1 *Es gibt eine optimale Lösung* $\{\hat{x}_{ij}, i = 1,\ldots,n; j = 1,\ldots,m\}$ *für das Problem (Relaxiertes Scheduling), mit der Eigenschaft, dass höchstens* $n+m$ *Werte* $\hat{x}_{ij}$ *von Null verschieden sind.*

Der Beweis benötigt etwas lineare Algebra und ist eine Uebungsaufgabe (Uebung 6.4). Sei $\{\hat{x}_{ij}, i = 1,\ldots,n; j = 1,\ldots,m\}$ nun eine Lösung mit höchstens $n+m$ Nichtnullen. Wegen der ersten n Gleichungen muss es für jedes $i = 1,\ldots,n$ auf jeden Fall ein j geben, so dass $\hat{x}_{ij} > 0$ gilt. Dann kann es aber für höchstens m Werte von i noch ein zweites j' geben, so dass $\hat{x}_{ij'} > 0$ gilt. Mit anderen Worten: für mindestens $n-m$ der möglichen Werte von i gilt $\hat{x}_{ij} = 1$ für genau ein j (und $\hat{x}_{ij'} = 0$ für $j' \neq j$).
Das bedeutet, die relaxierte Lösung gibt uns für $n-m$ der Jobs schon eine kanonische Zuordnung zu Maschinen vor. Die restlichen höchstens m Jobs verteilen wir nun noch so, dass jede Maschine höchstens einen von ihnen bekommt. Formal können wir den resultierenden Algorithmus LP Schedule wie folgt aufschreiben.

Algorithmus 6.4.2
LP Schedule:

```
k := 1
FOR i := 1 TO n DO
    IF x̃_ij = 1 für ein j THEN
        ordne Job i Maschine j zu
    ELSE
        ordne Job i Maschine k zu
    k := k + 1 END
END
```

Satz 6.4.3 *LP Schedule hat Approximationsgüte 2.*

Beweis: Sei t^* die optimale Lösung des Relaxierten Scheduling. Dann gilt, dass der von LPS erzeugte Schedule einen Makespan von höchstens $t^* + \max_{i=1}^n d_i$ hat, denn wenn nur Jobs i, für die ein j mit $\hat{x}_{ij} = 1$ existiert, den Maschinen zugeordnet werden, erhalten wir nach Konstruktion des LP einen Schedule mit Makespan höchstens t^*; durch die zusätzlichen höchstens m Jobs (maximal einer pro Maschine) kann sich der Makespan nur um die Dauer des längsten unter diesen Jobs erhöhen. Offenbar gilt

$$t^* \leq \text{opt},$$

wobei opt die optimale Lösung von (Scheduling) ist, sowie

$$\max_{i=1}^{n} d_i \leq \text{opt}.$$

Daraus folgt sofort

$$t^* + \max_{i=1}^{n} d_i \leq 2\text{opt},$$

LP Schedule liefert also wie der Algorithmus LS aus der Einführung (siehe Seite 6) einen Schedule, der höchstens zweimal so lang ist wie der optimale. □

6.5 Uebungen

Uebung 6.1 Zeige, dass für alle $k \in \mathbf{N}$ und alle $0 \leq x \leq 1$ gilt

$$1 - \left(1 - \frac{x}{k}\right)^k \geq \left[1 - \left(1 - \frac{1}{k}\right)^k\right] x.$$

Uebung 6.2 Beim Problem Node-Cover sind eine Menge $V = \{v_1, \ldots, v_n\}$ und eine Menge $S = \{S_1, \ldots, S_m\}$ von Teilmengen von V gegeben. Gesucht ist eine minimale Teilmenge $T \subseteq S$, so dass

$$\bigcup_{i: S_i \in T} S_i = V.$$

Zeige, wie Node Cover auf Set Cover reduziert werden kann, so dass die Grössen der jeweiligen optimalen Mengen identisch sind.

Uebung 6.3 Betrachte folgenden Approximationsalgorithmus für Set Cover: Sei $f := \max\{|S_1|, \ldots, |S_m|\}$. Zunächst bestimme eine optimale Lösung $p_1, \ldots, p_n$ für das Relaxierte Set Cover. Dann nimm in die Lösungsmenge H alle v_i mit $p_i \geq 1/f$ auf.
Zeige, dass dieser Algorithmus Approximationsgüte f hat.

Uebung 6.4 Gegeben sei ein lineares Programm in n Variablen und m Ungleichungen

$$\begin{array}{lll} \text{(LP)} & \text{minimiere} & \sum_{i=1}^{n} c_i x_i \\ & \text{unter} & \\ & & \sum_{i=1}^{n} a_{ij} x_i \left\{ \begin{array}{c} = \\ \leq \end{array} \right\} b_j, \quad j = 1, \ldots, m \\ & & x_i \geq 0, \qquad\qquad\quad i = 1, \ldots, n. \end{array}$$

Beweise, dass es eine optimale Lösung für das Problem gibt, bei der höchstens m der Variablen von Null verschiedene Werte haben (falls es überhaupt eine optimale Lösung gibt).

Hinweis: Seien $\hat{x}_1, \ldots, \hat{x}_n$ die Werte irgendeiner optimalen Lösung, wobei mehr als m Werte von Null verschieden sind. Definiere $\hat{b}_j = \sum_{i=1}^{n} a_{ij} \hat{x}_i$ und betrachte die affinen Unterräume

$$U_1 := \{(x_1, \ldots, x_n) \mid x_i = 0 \text{ für alle } i \text{ mit } \tilde{x}_i = 0\}$$

und

$$U_2 := \{(x_1, \ldots, x_n) \mid \sum_{i=1}^{n} a_{ij} x_i = \hat{b}_j, j = 1, \ldots, m\}.$$

Zeige, dass sich U_1 und U_2 mindestens in einer Geraden schneiden und folgere daraus die Existenz einer optimalen Lösung mit weniger von Null verschiedenen Werten als $(\hat{x}_1, \ldots, \hat{x}_n)$.

Uebung 6.5 Das Problem TREE-MULTI-CUT ist wie folgt definiert. Gegeben ist ein Baum $T = (V, E)$ und eine Teilmenge $\{(s_1, t_1), \ldots, (s_k, , t_k)\}$ von $T \times T$. Gesucht ist eine möglichst kleine Teilmenge F der Kanten E, so dass in $T' = (V, E \backslash F)$ für keines der Paare (s_i, t_i) der Knoten s_i in derselben Zusammenhangskomponente liegt wie t_i. Formuliere dieses Problem als ein ganzzahliges lineares Programm. Die Anzahl der Variablen und die Anzahl der Nebenbedingungen des Programms sollen polynomiell in $|V|$ sein.

Uebung 6.6 Beweise den folgenden Satz: gegeben sei ein randomisierter Approximationsalgorithmus A für ein Maximierungsproblem, mit *konstanter* erwarteter Güte δ. Dann gibt es für jedes $\varepsilon > 0$ und jedes $p < 1$ einen randomisierten Approximationsalgorithmus $A_{\varepsilon,p}$ für das Problem, der für jede Instanz I mit Wahrscheinlichkeit mindestens p eine Lösung mit Güte mindestens $(1 - \varepsilon)\delta$ berechnet.

Kapitel 7

Semidefinites Programmieren und Approximationsalgorithmen

In diesem Kapitel werden wir eine neue Technik für Approximationsalgorithmen kennenlernen, das sogenannte *semidefinite Programmieren.* Relaxierungen, die auf semidefinitem Programmieren basieren (SDP-Relaxierungen) sind eine Verallgemeinerung von LP-Relaxierungen. Wir werden die Technik zunächst am Beispiel Max-Cut erläutern.

7.1 Semidefinites Programmieren und Max-Cut

Hier noch einmal zur Erinnerung die Definition von Max-Cut (siehe Problem 1.3.1). Gegeben ist ein Graph $G = (V, E)$, wobei wir annehmen dass $V = \{1, \ldots, n\}$. Gesucht ist eine Teilmenge $S \subset V$, so dass die Anzahl der Kanten, die von einem Element in S zu einem Element aus $V \backslash S$ führen, maximal ist. Wir haben für dieses Problem schon einen Approximationsalgorithmus mit erwarteter Güte $1/2$ kennengelernt. Dieser Algorithmus hat die Menge S zufällig konstruiert. Um eine Verbesserung zu erreichen, werden wir Max-Cut als mathematisches Programm formulieren.
Hierzu setzen wir $a_{ij} = 1$, falls die Kante $\{i, j\} \in E$ und $a_{ij} = 0$ sonst $i, j = 1 \ldots, n$. Weiter führen wir Variablen $x_i, i = 1, \ldots, n$, ein, die die Werte $1, -1$ annehmen können. Die Bedeutung dieser Variablen ist wie folgt: $x_i = 1$ genau dann, wenn $i \in S$. Dann können wir Max-Cut folgendermassen definieren.

$$
\begin{array}{lll}
\text{(Max-Cut)} & \text{maximiere} & \displaystyle\sum_{i<j} a_{ij} \frac{1 - x_i x_j}{2} \\
& \text{unter} & \\
& & x_i \in \{-1, 1\}, \qquad i = 1, \ldots, n.
\end{array}
$$

Nun ist dieses Programm zwar äquivalent zum Max-Cut Problem, aber das Programm ist kein ganzzahliges *lineares* Programm. Wir können dieses Programm

also nicht zu einem linearen Programm relaxieren.
Der erste Ansatz, dieses Problem zu umgehen, könnte darin bestehen, die Variablenprodukte $x_i x_j, i < j$, durch neue Variablen y_{ij} zu ersetzen. Dadurch würde das obige Programm zu einem ganzzahligen linearen Programm, da die y_{ij} ebenfalls Werte in $\{-1, 1\}$ annehmen würden. Allerdings ist dann völlig unklar, wie eine optimale Lösung des relaxierten Programms eine zulässige Lösung von Max-Cut liefern soll. Man beachte, dass das relaxierte Programm uns $\binom{n}{2}$ viele Werte y_{ij} liefert, während im Max-Cut nur n Werte für die x_i benötigt werden. Insbesondere ist nicht klar, warum eine optimale Lösung $\hat{y}_{ij}$ für das relaxierte Programm die Form $\hat{y}_{ij} = \hat{x}_i \hat{x}_j, i < j$, haben sollte.
Daher werden wir nun für jeden Knoten $i \in V$ einen Vektor v_i bestehend aus n Variablen einführen. Wir definieren weiter für zwei Vektoren $u = (u_1, \dots, u_n)$ und $v = (v_1, \dots, v_n) \in \mathbf{R}^n$ ihr Produkt uv als $\sum_{i=1}^n u_i v_i$. Schliesslich sei $S_n := \{u \in \mathbf{R}^n | uu = 1\}$. Wir erhalten dann mit den a_{ij} von vorher das sogenannte Problem Vektor-Max-Cut.

$$\begin{array}{lll} \text{(Vektor-Max-Cut)} & \text{maximiere} & \displaystyle\sum_{i<j} a_{ij} \frac{1 - v_i v_j}{2} \\ & \text{unter} & \\ & & v_i \in S_n, \qquad i = 1, \dots, n. \end{array}$$

Die Bedingung $v_i \in S_n$ hat also die Bedingung $x_i \in \{-1, 1\}$ ersetzt. Da $S_1 = \{-1, 1\}$ scheint dieses eine natürliche Verallgemeinerung zu sein. Doch der eigentliche Grund für diese Setzung ist das folgende Lemma.

Lemma 7.1.1 *Sei opt der Wert einer optimalen Lösung einer Instanz von Max-Cut und* opt* *der Wert einer optimalen Lösung der entsprechenden Instanz von Vektor-Max-Cut. Dann gilt* $\text{opt} \leq \text{opt}^*$.

Beweis: Sei $S \subset V$ eine Lösung mit optimalem Wert opt. Setze $\hat{v}_i = (1, 0, \dots, 0)$ falls $i \in S$ und $\hat{v}_i = (-1, 0, \dots, 0)$ falls $i \notin S$. Da $\hat{v}_i \in S_n$, ist dies eine zulässige Lösung von Vektor-Max-Cut mit Wert opt. Das Lemma folgt. □

Die Bedingung $v_i \in S_n$ stellt also sicher, dass eine zulässige Lösung von Max-Cut zu einer zulässigen Lösung von Vektor-Max-Cut transformiert werden kann.
Jetzt werden wir Vektor-Max-Cut noch anders, und wie sich herausstellen wird, äquivalent umformulieren. Hierzu führen wir eine Variable v_{ij} für jedes der Produkte $v_i v_j$ ein. Damit ist die Zielfunktion von Vektor-Max-Cut eine lineare Funktion der v_{ij}. Weiter sehen wir, dass nun die Anzahl der v_{ij} mit der Anzahl der Variablen in den Vektoren v_i übereinstimmt. Wir sollten aber auch noch sicherstellen, dass sich die v_{ij} als Produkte von Vektoren $v_1, \dots, v_n$ schreiben lassen.

Wir werden dieses explizit in das folgende Programm aufnehmen.

(Relaxiertes-Max-Cut)

maximiere $\sum_{i<j} a_{ij} \frac{1 - v_{ij}}{2}$

unter

$$
\begin{array}{ll}
v_{ii} = 1 & i = 1, \ldots, n \\
(v_{ij})_{i,j=1,\ldots,n} = VV^T & \text{für eine Matrix} \\
 & V \in \mathbf{R}^{n \times n}
\end{array}
$$

Die Bedingung v_{ii} entspricht der Bedingung $v_i \in S_n$, denn $v_{ii} = v_i v_i$. Die zweite Bedingung drückt aus, dass die v_{ij} Produkte von n Vektoren $v_1, \ldots, v_n$ sein sollen. Diese Vektoren können nämlich als die Zeilen von V gewählt werden.
Nach diesen Beobachtungen ist es klar, dass der optimale Wert einer Lösung des Relaxierten Max-Cut der optimale Wert einer Lösung von Vektor-Max-Cut ist. Nun gibt es einen polynomiellen Algorithmus der Programme wie Relaxiertes-Max-Cut, fast optimal löst. Ausserdem gibt es einen polynomiellen Algorithmus, der aus den v_{ij} die Vektoren v_i bestimmt. Diese beiden Algorithmen werden wir benutzen, um Vektor-Max-Cut fast optimal zu lösen. Wir werden dann noch zeigen, dass aus dieser Lösung von Vektor-Max-Cut eine gute Lösung von Max-Cut konstruiert werden kann.
Eine Matrix $Y \in \mathbf{R}^{n \times n}$, für die eine Matrix $V \in \mathbf{R}^{n \times n}$ existiert mit $Y = VV^T$, heisst *symmetrisch positiv semidefinit*. Ein *semidefinites Programm (SDP)* ist ein Programm der folgenden Form.

(SDP)

maximiere $\sum_{i<j} c_{ij} v_{ij}$

unter

$$
\begin{array}{ll}
(v_{ij})_{i,j=1,\ldots,n} & \text{ist symmetrisch} \\
 & \text{positiv semidefinit} \\
l_k(v_{11}, \ldots, v_{nn}) \leq b_k, & k = 1, \ldots, m,
\end{array}
$$

wobei die c_{ij}, b_k beliebige ganze Zahlen sind. Weiter sind die l_k lineare Funktionen in den v_{ij} mit ganzzahligen Koeffizienten. Es gilt nun der folgende Satz.

Satz 7.1.2 *Es gibt einen Algorithmus, der für jedes* SDP *und jedes* $\epsilon > 0$ *Werte* $\hat{v}_{ij} \in \mathbf{Q}$ *findet, so dass die Matrix* $(\hat{v}_{ij})_{i,j=1,\ldots,n}$ *symmetrisch positiv semidefinit ist,* $l_k(\hat{v}_{11}, \ldots, \hat{v}_{nn}) \leq b_k$, *für alle* k *und ausserdem gilt*

$$\sum_{i<j} c_{ij} \hat{v}_{ij} \geq \mathrm{opt}^* - \epsilon,$$

hierbei ist opt^* *der optimale Wert einer Lösung des semidefiniten Programms. Die Laufzeit des Algorithmus ist polynomiell in* n, $\log(1/\epsilon)$ *und den Bitgrössen der* c_{ij}, b_k, *und der Koeffizienten der* l_k.

Mit anderen Worten, der Algorithmus findet in polynomieller Zeit eine fast optimale, zulässige Lösung des Programms.

Dieses Satz ist schwer zu beweisen, und wir verzichten darauf. Schliesslich noch

Satz 7.1.3 *Es gibt einen Algorithmus, der für jede symmetrische positiv semidefinite Matrix $Y \in \mathbf{Q}^{n\times n}$ eine Matrix $V \in \mathbf{R}^{n\times n}$ berechnet, so dass $Y = VV^T$. Die Laufzeit des Algorithmus ist polynomiell in der Bitgrösse der Einträge von Y.*

Auch diesen Satz werden wir nicht beweisen. Wir erwähnen nur, dass die Zerlegung $Y = VV^T$ als *Cholesky-Zerlegung* in den meisten Büchern über Numerische Mathematik behandelt wird. In der Formulierung des Satzes haben wir einige technische Schwierigkeiten unterdrückt. Da $V \in \mathbf{R}^{n\times n}$ sein wird, können die Einträge von V nicht exakt berechnet werden—sie können irrational sein. Das bedeutet, man kann nur so etwas wie eine approximative Cholesky-Zerlegung berechnen. Allerdings können die Einträge der Matrix V beliebig genau approximiert werden. Dieses ist für unsere Zwecke ausreichend. Wir erhalten nun

Satz 7.1.4 *Es gibt einen Algorithmus, der für jedes $\epsilon > 0$ Vektoren $\hat{v}_i, i = 1,\dots,n$, berechnet, die eine zulässige Lösung für* Vektor-Max-Cut *sind. Der Wert der durch die $\hat{v}_i$ gegebenen Lösung von* Vektor-Max-Cut *weicht vom Wert* opt^* *einer optimalen Lösung höchstens um ϵ ab. Die Laufzeit des Algorithmus ist polynomiell in n und in* $\log(1/\epsilon)$.

Hier eine Beweisskizze. Zunächst berechnen wir eine fast optimale Lösung des semidefiniten Programms. Sei die Abweichung vom Optimum $\tilde{\epsilon}$ und die gefundene Matrix $\tilde{V} = (\hat{v}_{ij})$. Dann berechnen wir eine Approximation $\hat{V}$ zur Cholesky-Zerlegung von $\tilde{V}$. Die Zeilen der Matrix $\hat{V}$ sind die $\hat{v}_i$. Nun wird der Wert der $\hat{v}_i$ nicht mehr nur um $\tilde{\epsilon}$ vom Optimum abweichen. Jedoch kann man durch eine Fehleranalyse zeigen, dass die Abweichung vom Optimum nicht sehr viel grösser geworden ist. Startet man also mit einen $\tilde{\epsilon}$, das etwas besser ist als ϵ, so werden die berechneten $\hat{v}_i$ den Satz erfüllen.
Mit diesem Satz sehen wir, dass insbesondere Vektor-Max-Cut in polynomieller Zeit fast optimal gelöst werden kann. Der nächste Algorithmus zeigt nun, wie wir aus einer solchen Lösung eine Lösung für Max-Cut konstruieren können, die sehr nahe an das Optimum herankommt. Neben einem Graphen $G = (V, E)$ erwartet der Algorithmus als zusätzliche Eingabe ein $\epsilon > 0$.

Algorithmus 7.1.5
SDP-Max-Cut:
  $S := \emptyset$
  Mit Hilfe des relaxierten Max-Cut finde eine Lösung $\hat{v}_i, i = 1,\dots,n$,
  von Vektor-Max-Cut, die vom Optimum opt^* nur um ϵ abweicht.

Wähle einen zufälligen Vektor $r \in S_n$.
```
FOR i := 1 TO n DO
    IF rv_i ≥ 0 THEN
        S := S ∪ {i}
    END
END
```
Gib S aus.

Die gesuchte Lösung für Vektor-Max-Cut können wir nach dem vorangehenden Satz in polynomieller Zeit finden. Auch der Schleifendurchlauf erfordert nur polynomielle Zeit. Allerdings stellt sich die Frage, wie ein zufälliger Vektor $r \in S_n$ in polynomieller Zeit erzeugt werden kann. Dieses ist auch nicht möglich. Man kann allerdings in polynomieller Zeit einen Vektor erzeugen, der "fast" zufällig ist. Dieses wird an der Analyse des Algorithmus nicht viel ändern. Wir wollen hierauf nicht näher eingehen. Wir nehmen einfach an, dass wir einen zufälligen Vektor $r \in S_n$ erzeugen können. Dann erhalten wir folgenden Satz.

Satz 7.1.6 *Algorithmus* SDP-Max-Cut *ist ein Approximationsalgorithmus für* Max-Cut*. Bei zusätzlicher Eingabe $\epsilon > 0$ hat der Algorithmus erwartete Güte $\alpha - \epsilon$, wobei $\alpha = 0,87856$.*

Beweis: Wir haben uns bereits überlegt, dass der Algorithmus polynomielle Laufzeit besitzt. Er erzeugt sicherlich auch eine zulässige Lösung. Wir müssen also nur noch die Aussage über die erwartete Approximationsgüte beweisen.
Wir definieren Zufallsvariablen $X_{ij}, 1 \leq i \leq n, 1 \leq j < i$ wie folgt. $X_{ij} = 1$, falls $i \in S \wedge j \notin S$ oder falls $i \notin S \wedge j \in S$. Treffen beide Bedingungen nicht zu, ist $X_{ij} = 0$. Weiter definieren wir

$$X = \sum_{i<j} a_{ij} X_{ij},$$

wobei die a_{ij} definiert sind wie zu Anfang dieses Kapitels. Der vom Algorithmus berechnet Schnitt hat nun den erwarteten Wert $E(X)$. Wir erhalten

$$E(X) = \sum_{i<j} E(X_{ij}) = \sum_{i<j} \text{prob}(X_{ij} = 1) = \sum_{i<j} \text{prob}(\text{sign}(\hat{v}_i r) \neq \text{sign}(\hat{v}_j r)),$$

wobei wir mit $\text{sign}(u)$ das Vorzeichen einer reellen Zahl bezeichnen. Etwas unorthodox setzen wir $\text{sign}(0) = 1$. Die Wahrscheinlichkeiten und Erwartungswerte beziehen sich hierbei immer auf die zufällige Wahl von $r \in S_n$. Um die Wahrscheinlichkeiten in der letzten Summe abzuschätzen, betrachten wir zunächst den Fall $r, \hat{v}_i, \hat{v}_j \in \mathbf{R}^2$. Dann gilt

$$\text{prob}(\text{sign}(\hat{v}_i r) \neq \hat{v}_j r) = \frac{2\angle(\hat{v}_i, \hat{v}_j)}{2\pi} = \frac{\angle(\hat{v}_i, \hat{v}_j)}{\pi} = \frac{\arccos(\hat{v}_i \hat{v}_j)}{\pi},$$

denn die Vorzeichen von $r\hat{v}_i, r\hat{v}_j$ sind genau dann verschieden, wenn r im von $\hat{v}_i, \hat{v}_j$ eingeschlossenen (Doppel-)Winkel liegt.
Der allgemeine, n-dimensionale Fall kann nun leicht auf den zweidimensionalen Fall reduziert werden, denn wir können unsere Betrachtungen immer auf die von $\hat{v}_i, \hat{v}_j$ aufgespannte Ebene einschränken. Wir erhalten also

$$\mathrm{prob}(X_{ij} = 1) = \frac{\arccos(\hat{v}_i \hat{v}_j)}{\pi}.$$

Wir wenden jetzt folgendes Lemma an, dessen Beweis eine Uebungsaufgabe ist.

Lemma 7.1.7 *Für alle* $-1 < z < 1$ *gilt* $\frac{\arccos(z)}{\pi} \geq \alpha \frac{1-z}{2}$, *wobei* $\alpha = 0,87856$ *wie oben.*

Es gilt also

$$E(X) \geq \alpha \sum_{i<j} a_{ij} \frac{1 - \hat{v}_i \hat{v}_j}{2}.$$

Nun sind aber die Vektoren $\hat{v}_i$ fast eine optimale Lösung von Vektor-Max-Cut. Genauer, bezeichnet opt^* den Wert einer optimalen Lösung von Vektor-Max-Cut, so gilt

$$\sum_{i<j} a_{ij} \frac{1 - \hat{v}_i \hat{v}_j}{2} \geq \mathrm{opt}^* - \epsilon.$$

Mit opt bezeichnen wir nun den optimalen Wert der Max-Cut-Instanz, dann gilt mit Lemma 7.1.1

$$E(X) \geq \alpha \sum_{i<j} a_{ij} \frac{1 - \hat{v}_i \hat{v}_j}{2} \geq \alpha(\mathrm{opt}^* - \epsilon) \geq \alpha(\mathrm{opt} - \epsilon) \geq (\alpha - \epsilon)\mathrm{opt}.$$

Damit ist der Satz bewiesen.

□

Dieser Satz legt es nahe zu sagen, dass Max-Cut bis auf den Faktor α approximiert werden kann, denn wir können ja ϵ beliebig klein wählen. Deshalb führen wir nun folgende Definition ein.

Definition 7.1.8 *Sei* Π *ein Optimierungsproblem. Wir sagen,* Π *hat Approximationsgüte oder Approximationsfaktor* c, *falls für jedes* $\epsilon > 0$ *ein Approximationsalgorithmus* $A(\epsilon)$ *für* Π *existiert, der Approximationsgüte*

$$\begin{array}{ll} c - \epsilon & \text{bei Maximierungsproblemen} \\ c + \epsilon & \text{bei Minimierungsproblemen} \end{array}$$

hat.

Einen Spezialfall dieser Definition kennen wir schon. Im Fall $c = 1$ stimmt diese Definition nämlich mit der Definition von einem Approximationsschema überein. Ein Optimierungsproblem Π hat also Approximationsgüte 1 genau dann, wenn Π ein Approximationsschema besitzt. Wir können nun auch das Ergebnis über Max-Cut zusammenfassen in der Bemerkung, dass Max-Cut Approximationsgüte $\alpha = 0,87856$ besitzt.

7.2 Semidefinites Programmieren und Max-$\leq$ 2-SAT

Wir wollen nun semidefinites Programmieren anwenden, um einen besseren Approximationsalgorithmus für Max-$\leq$ 2-SAT zu erhalten. Wir haben bislang einen Approximationsalgorithmus mit Güte $3/4$ kennengelernt. Wir werden diesen auf einen Approximationsalgorithmus mit Güte $\alpha - \epsilon$ verbessern, wobei $\alpha = 0,87856$ wie im vorangegangenen Abschnitt und $\epsilon > 0$ beliebig. Nach Definition 7.1.8 zeigen wir also, dass Max-$\leq$ 2-SAT Approximationsgüte α hat.
Gegeben sei eine Max-$\leq$ 2-SAT Formel φ über den Variablen $\{x_1, \ldots, x_n\}$. D.h. φ hat die Form $\varphi = \bigwedge_{j=1}^m C_j$, und jede Klausel C_j hat die Form $C_j = l_j$ oder $C_j = l_{j1} \vee l_{j2}$ für Literale $l_j, l_{j1}, l_{j2} \in \{x_1, \ldots, x_n\} \cup \{\neg x_1, \ldots, \neg x_n\}$. Um Max-$\leq$ 2-SAT als ein mathematisches Programm zu formulieren, führen wir Variablen $y_0, y_1, \ldots, y_n$ ein. Diese Variablen können Werte in $\{-1, 1\}$ annehmen. Der Zusammenhang zwischen den Variablen x_i und den Variablen y_i wird durch folgende Äquivalenz hergestellt.

$$x_i = \texttt{true} \Leftrightarrow y_0 = y_i \tag{7.1}$$

Man beachte, dass nach dieser Setzung einer Belegung der Variablen x_i zwei Belegungen der Variablen y_i entsprechen. Es kann nämlich einmal $y_0 = 1$ gesetzt werden, und dann können die übrigen y_i entsprechend der Belegung der x_i gewählt werden. Es kann aber auch $y_0 = -1$ gesetzt werden. Auch dieses kann dann zur einer Belegung der übrigen y_i erweitert werden, die der Belegung der x_i entspricht.
Wir wollen nun für jede Klausel C_j eine Funktion u_j in den y_i definieren, so dass bei Einsetzen von Werten $y_i \in \{-1, 1\}$ die Funktion u_j nur die Werte $0, 1$ annehmen kann. Weiter soll eine Belegung der x_i die Klausel C_j genau dann erfüllen, wenn *beide* Belegungen der y_i, die dieser Belegung der x_i entsprechen, bei Einsetzen in u_j den Wert 1 ergeben.
Bestehe C_j zunächst aus einem Literal l_j. Wir setzen

$$\begin{array}{rclcrcl} u_j & = & \frac{1+y_0 y_s}{2} & \text{falls} & l_j & = & x_s \\ u_j & = & \frac{1-y_0 y_s}{2} & \text{falls} & l_j & = & \neg x_s \end{array}$$

Für Klauseln C_j mit zwei Literalen l_{j1}, l_{j2} setzen wir

$$\begin{array}{lclcrcl} u_j & = & 1 - \frac{1-y_0y_s}{2}\frac{1-y_0y_t}{2} & \text{falls} & C_j & = & x_s \vee x_t \\ u_j & = & 1 - \frac{1+y_0y_s}{2}\frac{1-y_0y_t}{2} & \text{falls} & C_j & = & \neg x_s \vee x_t \\ u_j & = & 1 - \frac{1+y_0y_s}{2}\frac{1+y_0y_t}{2} & \text{falls} & C_j & = & \neg x_s \vee \neg x_t \end{array}$$

Man prüft leicht nach, dass u_j die gewünschte Eigenschaft hat. Für Klauseln C_j mit zwei Literalen wollen wir u_j noch etwas anders schreiben. Sei z.B. $C_j = x_s \vee x_t$, dann gilt

$$u_j = 1 - \frac{1-y_0y_s}{2}\frac{1-y_0y_t}{2} = \frac{1+y_0y_s}{4} + \frac{1+y_0y_t}{4} + \frac{1-y_sy_t}{4}.$$

Allgemein können wir für alle Klauseln C_j mit zwei Variablen schreiben

$$u_j = 1 - \frac{1 \pm y_0y_s}{2}\frac{1 \pm y_0y_t}{2} = \frac{1 \pm y_0y_s}{4} + \frac{1 \pm y_0y_t}{4} + \frac{1 \pm y_sy_t}{4},$$

wobei die Vorzeichen geeignet zu wählen sind. Jetzt erhalten wir das folgende mathematische Programm, das zu Max-$\leq$ 2-SAT äquivalent ist.

$$\begin{array}{lll} (\text{Max-}\leq 2\text{-SAT}) & \text{maximiere} & \sum_{j=1}^{m} u_j = \frac{3}{4}m + \sum_{s<t} a_{st}y_sy_t \\ & \text{unter} & \\ & & y_i \in \{-1,1\}, \qquad i = 0, \ldots, n, \end{array}$$

wobei sich die a_{ij} aus den Formeln für die u_j ergeben. Nun gehen wir analog wie bei Max-Cut vor. D.h. zunächst ersetzen wir die Variablen y_i durch Vektoren v_i bestehend aus n Variablen. Bezeichnen wir mit w_j die Funktion, die aus u_j entsteht, wenn wir die y_i durch die v_i ersetzen. Wir erhalten dann

$$\begin{array}{lll} (\text{Vektor-Max-}\leq 2\text{-SAT}) & \text{maximiere} & \sum_{j=1}^{m} w_j = \frac{3}{4}m + \sum_{s<t} a_{st}v_sv_t \\ & \text{unter} & \\ & & v_i \in S_n, \qquad i = 0, \ldots, n. \end{array}$$

Wie bei Max-Cut und Vektor-Max-Cut können wir argumentieren, dass das Optimum opt* von Vektor-Max-$\leq$ 2-SAT mindestens so gross ist wie das Optimum opt von Max-$\leq$ 2-SAT selbst.
Schliesslich führen wir noch Variablen v_{st} für die Produkte v_sv_t ein. Dieses führt auf

$$\begin{array}{lll} (\text{Relaxiertes-Max-}\leq 2\text{-SAT}) & & \\ \text{maximiere} & \frac{3}{4}m + \sum_{s<t} a_{st}v_{st} & \\ \text{unter} & & \\ & v_{ss} = 1 & s = 1, \ldots, n \\ & (v_{st})_{s,t=1,\ldots,n} = VV^T & \text{für eine Matrix} \\ & & V \in \mathbf{R}^{n \times n}. \end{array}$$

Dieses ist ein semidefinites Programm. Nach Satz 7.1.2 können wir also eine Lösung dieses Programms berechnen, die bis auf beliebiges $\epsilon > 0$ an den Wert einer optimalen Lösung herankommt. Nach Satz 7.1.3 können wir diese Lösung von Relaxiertem-Max-$\leq$ 2-SAT dann in eine Lösung $\hat{v}_0, \ldots, \hat{v}_n$ von Vektor-Max-$\leq$ 2-SAT transformieren, die ebenfalls bis auf beliebiges $\epsilon > 0$ an das Optimum opt* von Vektor-Max-$\leq$ 2-SAT herankommt. Dieses führt zu folgendem Approximationsalgorithmus für Max-$\leq$ 2-SAT, der neben einer $\leq$ 2-SAT Formel φ eine zusätzliche Eingabe $\epsilon > 0$ erwartet.

Algorithmus 7.2.1

SDP-Max-$\leq$ 2-SAT:

```
S := ∅
Mit Hilfe von Relaxiertem-Max-Cut finde eine Lösung v̂_i, i = 1,...,n,
von Vektor-Max-≤ 2-SAT, die vom Optimum nur um ε abweicht.
Wähle einen zufälligen Vektor r ∈ S_n.
FOR i := 1 TO n DO
    IF sign(rv_i) = sign(rv_0) THEN
        x_i = true
    ELSE
        x_i = false
    END
END
Gib die gefundene Belegung aus.
```

Satz 7.2.2 *Bei zusätzlicher Eingabe $\epsilon > 0$ ist* SDP-Max-$\leq$ 2-SAT *ein Approximationsalgorithmus mit Güte $\alpha - \epsilon$.*

Beweis: Wie bei dem Algorithmus SDP-Max-Cut folgt, dass Algorithmus SDP-Max-$\leq$ 2-SAT ein Approximationsalgorithmus ist. Wir müssen also noch die Behauptung über die Approximationsgüte beweisen. Hierzu bezeichne X die Zufallsvariable, die die Anzahl der von einer Belegung erfüllten Klauseln zählt. Für die erwartete Anzahl der Klauseln $E(X)$, die die von Algorithmus SDP-Max-$\leq$ 2-SAT gefundene Belegung erfüllt, gilt

$$E(X) = \sum_{j=1}^{m} E(u_j),$$

wobei die Erwartungswerte der u_j über die Wahl des zufälligen Vektors $r \in S_n$ genommen wird.

Setze $\hat{y}_i = r\hat{v}_i$ und betrachte eine Klausel C_j mit $u_j = \frac{1 \pm y_0 y_s}{4} + \frac{1 \pm y_0 y_t}{4} + \frac{1 \pm y_s y_t}{4}$.

$$E(u_j) = E\left(\frac{1 \pm \hat{y}_0 \hat{y}_s}{4} + \frac{1 \pm \hat{y}_0 \hat{y}_t}{4} + \frac{1 \pm \hat{y}_s \hat{y}_t}{4}\right) =$$

$$E\left(\frac{1\pm \hat{y}_0\hat{y}_s}{4}\right)+E\left(\frac{1\pm \hat{y}_0\hat{y}_t}{4}\right)+E\left(\frac{1\pm \hat{y}_s\hat{y}_t}{4}\right).$$

Für beliebige s, t zwischen 0 und n gilt

$$\begin{array}{lclcl} E\left(\frac{1-\hat{y}_s\hat{y}_t}{4}\right) & = & \frac{1}{2}\,\mathrm{prob}(\mathrm{sign}(\hat{v}_s r)\neq \mathrm{sign}(\hat{v}_t r)) & = & \frac{1}{\pi}\arccos(\hat{v}_s\hat{v}_t) \\ E\left(\frac{1+\hat{y}_s\hat{y}_t}{4}\right) & = & \frac{1}{2}\,\mathrm{prob}(\mathrm{sign}(\hat{v}_s r)= \mathrm{sign}(\hat{v}_t r)) & = & 1-\frac{1}{\pi}\arccos(\hat{v}_s\hat{v}_t). \end{array}$$

Nach Lemma 7.1.7 gilt

$$\frac{1}{\pi}\arccos(\hat{v}_s\hat{v}_t)\geq \alpha\frac{1-\hat{v}_s\hat{v}_t}{2},$$

für $\alpha = 0,87856$. Analog kann gezeigt werden (Uebung 7.4)

$$1-\frac{1}{\pi}\arccos(\hat{v}_s\hat{v}_t)\geq \alpha\frac{1+\hat{v}_s\hat{v}_t}{2}.$$

Damit erhalten wir

$$\begin{array}{lcl} E\left(\frac{1-\hat{y}_s\hat{y}_t}{4}\right) & \geq & \alpha\frac{1-\hat{v}_s\hat{v}_t}{4} \\ E\left(\frac{1+\hat{y}_s\hat{y}_t}{4}\right) & \geq & \alpha\frac{1+\hat{v}_s\hat{v}_t}{4}. \end{array}$$

Insgesamt daher

$$E(u_j) = E\left(\frac{1\pm \hat{y}_0\hat{y}_s}{4}+\frac{1\pm \hat{y}_0\hat{y}_t}{4}+\frac{1\pm \hat{y}_s\hat{y}_t}{4}\right)$$

$$\geq \alpha\left(\frac{1\pm \hat{v}_0\hat{v}_s}{4}+\frac{1\pm \hat{v}_0\hat{v}_t}{4}+\frac{1\pm \hat{v}_s\hat{v}_t}{4}\right).$$

Daraus schliessen wir, dass $E(X)$ mindestens so gross ist wie der Wert der Lösung $\hat{v}_0, \ldots, \hat{v}_n$ für Vektor-Max-$\leq$ 2-SAT. Dieser Wert war $\mathrm{opt}^*-\epsilon$. Da opt^* mindestens so gross ist, wie das Optimum opt von Max-$\leq$ 2-SAT erhalten wir

$$E(X)\geq \alpha(\mathrm{opt}^*-\epsilon)\geq \alpha(\mathrm{opt}-\epsilon)\geq(\alpha-\epsilon)\mathrm{opt}.$$

Dieses schliesst den Beweis des Satzes ab. ◻

Wir haben also gezeigt, dass Max-$\leq$ 2-SAT Approximationsgüte $\alpha = 0,87856$ hat.

7.3 Semidefinites Programmieren und Max-SAT*

In diesem Abschnitt werden wir semidefinites Programmieren auf Max-SAT anwenden. Zur Erinnerung, bislang haben wir einen Approximationsalgorithmus für Max-SAT mit Güte 3/4. Dieses haben wir erreicht, indem wir die beiden Algorithmen Random-SAT und RR-Max-SAT miteinander kombiniert haben (siehe

Seite 50). Geht man zurück zur Analyse des kombinierten Algorithmus, so kann man recht leicht sehen, dass wir den Approximationsalgorithmus mit erwarteter Güte auch durch folgenden randomisierten Algorithmus hätten erreichen können: mit Wahrscheinlichkeit jeweils 1/2 wähle Algorithmus Random-SAT oder Algorithmus RR-Max-SAT. Wende den gewählten Algorithmus auf die Instanz von Max-SAT an und gib die erhaltene Belegung aus. Einen ähnlichen Ansatz wollen wir nun auch anwenden, allerdings mit einer Relaxierung von Max-SAT, die ein semidefinites Programm ist. Diese wird eine Kombination der semidefiniten Relaxierung von Max-$\leq$ 2-SAT und der LP-Relaxierung von Max-SAT sein.
Sei also φ eine Max-SAT Formel über den Variablen x_i mit m Klauseln C_j. Die Variablen y_i sind wie im vorherigen Abschnitt definiert. Für Klauseln C_j mit ein oder zwei Variablen definieren wir u_j ebenfalls wie bei Max-$\leq$ 2-SAT. Mit $l(C_j)$ bezeichnen wir die Anzahl der Literale in Klausel C_j. Ausserdem bezeichnen wir mit V_j^+ die Menge der unnegierten Variablen in C_j und mit V_j^- die Menge der negierten Variablen in C_j. Wir können dann das folgende mathematische Programm für Max-SAT aufstellen.

(Max-SAT)

$$\begin{array}{lll}
\text{maximiere} & \displaystyle\sum_{j=1}^{m} z_j & \\
\text{unter} & & \\
& \displaystyle\sum_{x_i \in V_j^+} \frac{1 + y_0 y_i}{2} + \sum_{x_i \in V_j^-} \frac{1 - y_0 y_i}{2} \geq z_j & j = 1, \ldots, m \\
& u_j \geq z_j & \forall C_j : l(C_j) \leq 2 \\
& z_j \in \{0,1\}, y_i \in \{-1,1\}. &
\end{array}$$

Dieses Programm können wir wie vorher in Programme Vektor-Max-SAT und Relaxiertes-Max-SAT überführen. Das Programm Vektor-Max-SAT sieht folgendermassen aus.

(Vektor-Max-SAT)

$$\begin{array}{lll}
\text{maximiere} & \displaystyle\sum_{j=1}^{m} z_j & \\
\text{unter} & & \\
& \displaystyle\sum_{x_i \in V_j^+} \frac{1 + v_0 v_i}{2} + \sum_{x_i \in V_j^-} \frac{1 - v_0 v_i}{2} \geq z_j & j = 1, \ldots, m \\
& w_j \geq z_j & \forall C_j : l(C_j) \leq 2 \\
& v_i \in S_n, & i = 0, \ldots, n \\
& z_j \in [0,1] & j = 1, \ldots, m.
\end{array}$$

Hierbei ist wie bei Max-$\leq$ 2-SAT w_j die Funktion die aus u_j durch die Ersetzung von y_i durch v_i entsteht. Das Programm Relaxiertes-Max-SAT genau aus-

zuarbeiten überlassen wir dem Leser. Das Programm (Vektor-Max-SAT) ist jedenfalls ein semidefinites Programm. Dieses können wir wiederum fast optimal lösen, um dann mit der Cholesky-Zerlegung Vektoren $\hat{v}_0, \dots, \hat{v}_n$, sowie Zahlen $\hat{z}_j, j = 1, \dots, m$, zu bestimmen, so dass diese Vektoren und Zahlen eine Lösung von Vektor-Max-SAT bestimmen, die vom Optimum opt* dieses Problem nur um ein beliebig kleines $\epsilon > 0$ abweicht.
Wir betrachten nun die folgenden randomisierten Algorithmen A_1, A_2, A_3

A_1 : Setze $x_i = \texttt{wahr}$ mit Wahrscheinlichkeit $1/2$.

A_2 : Setze $x_i = \texttt{wahr}$ mit Wahrscheinlichkeit $(1 + \hat{v}_0\hat{v}_i)/2$.

A_3 : Wähle $r \in S_n$ zufällig. Setze $x_i = \texttt{wahr}$, falls $\operatorname{sign}(\hat{v}_0 r) = \operatorname{sign}(\hat{v}_i r)$.

Die Analysen von Random-SAT, RR-Max-SAT und SDP-Max-SAT liefern nun unmittelbar oder nach leichten Aenderungen, dass

A_1 : Algorithmus A_1 erfüllt eine Klausel C_j mit $l(C_j) = k$ mit Wahrscheinlichkeit $1 - 1/2^k$.

A_2 : Algorithmus A_2 erfüllt eine Klausel C_j mit $l(C_j) = k$ mit Wahrscheinlichkeit $(1 - (1 - 1/k)^k)\hat{z}_j$.

A_3 : Algorithmus A_3 erfüllt eine Klausel C_j mit $l(C_j) \leq 2$ mit Wahrscheinlichkeit $\alpha\hat{z}_j, \alpha = 0,87856$.

Der Algorithmus SDP-Max-SAT wählt nun zufällig einen dieser drei Algorithmen, wendet den gewählten Algorithmus auf die Formel φ an und gibt die erhaltene Belegung aus. Allerdings wählt SDP-Max-SAT nicht jeden der drei Algorithmen mit gleicher Wahrscheinlichkeit. Vielmehr gilt

SDP wählt Algorithmus A_1 mit Wahrscheinlichkeit	$p_1 = 0,4785$
SDP wählt Algorithmus A_2 mit Wahrscheinlichkeit	$p_2 = 0,4785$
SDP wählt Algorithmus A_3 mit Wahrscheinlichkeit	$p_3 = 0,0430$

Man kann nun nachrechnen, dass die erwartete Anzahl der Klauseln, die SDP-Max-SAT erfüllt, gegeben ist durch

$$\sum_{j:l(C_j)=1} \left(\frac{1}{2}p_1 + (p_2 + p_3\alpha)\hat{z}_j\right) +$$
$$\sum_{j:l(C_j)=2} \left(\frac{3}{4}p_1 + \left(\frac{3}{4}p_2 + p_3\alpha\right)\hat{z}_j\right) +$$
$$\sum_{j:l(C_j)\geq 3} \left(1 - \frac{1}{2^{l(C_j)}}\right)p_1 + \left(1 - \left(1 - \frac{1}{l(C_j)}\right)^{l(C_j)}\right)p_2\hat{z}_j.$$

Man beachte, dass wir für diese Formel angenommen haben, dass A_3 keine Klausel der Länge ≥ 3 erfüllt. Es ist eine gute Uebung, sich klar zu machen, warum dieses SDP-Max-SAT nicht zu einem schlechten Approximationsalgorithmus werden lässt.
Durch direktes Nachrechnen erhält man, dass für Klauseln der Länge $1, 2, 3, 4$ die erwartete Anzahl erfüllter Klauseln grösser ist als $\alpha`hatz_j, j = 1, 2, 3, 4$. Für Klauseln mit Länge mindestens 5 nutzen wir aus, dass für $k \geq 5$

$$\left(1 - \frac{1}{2^k}\right) p_1 + \left(1 - \left(1 - \frac{1}{k}\right)^k\right) p_2 \geq \left((1 - 2^{-5}) + 1 - \frac{1}{\mathrm{e}}\right) 0,4785,$$

so erhält man, dass die erwartete Anzahl von Klauseln, die SDP-Max-SAT erfüllt grösser ist als

$$0,7554 \sum_{j=1}^{m} \hat{z}_j.$$

Nun ist aber $\sum \hat{z}_j$ der Wert der Lösung $\hat{v}_i, \hat{z}_j$ von Vektor-Max-SAT. Dieser Wert ist grösser als $\text{opt}^* - \epsilon$, wobei opt^* der Wert einer optimalen Lösung von Vektor-Max-SAT ist. Dieser optimale Wert ist wie üblich grösser als opt, die maximale Anzahl von gleichzeitig erfüllbaren Klauseln der Formel φ. Damit kann dann gezeigt werden, dass im Erwartungswert mehr als $(0,7554 - \epsilon)\text{opt}$ Klauseln durch die von SDP-Max-SAT gefundene Belegung erfüllt werden.
Dieses ist natürlich nur eine marginale Verbesserung gegenüber dem Algorithmus, der Random-SAT und RR-Max-SAT kombinierte. Dieser hatte ja bereits Approximationsgüte 3/4. Das Interessante an unserem neuen Resultat ist denn auch, dass es einen Approximationsfaktor besser als 3/4 zeigt. Es wurde vorher nämlich vermutet, dass kein Algorithmus eine bessere Güte erreichen kann. Im letzten Kapitel werden wir noch etwas mehr darüber erfahren, wie gut sich die einzelnen Varianten von SAT approximieren lassen. Wir werden z.B. sehen, dass es für keine Variante von SAT ein Approximationsschema geben kann.

7.4 Approximatives Färben von Graphen

Das Färben von Graphen ist ein klassisches Problem. Eine *k-Färbung* von $G = (V, E)$ ist eine Funktion $c : V \mapsto \{1, \ldots, k\}$ mit der Eigenschaft, dass $c(v) \neq c(w)$ für alle Kanten $\{v, w\} \in E$ gilt. Mit anderen Worten: benachbarte Knoten erhalten verschiedene Farben.
Das prominenteste Färbungsproblem ist unter dem Namen 'Vier-Farben-Problem' bekannt geworden. Dabei geht es um die Vermutung, dass sich jede Landkarte so mit vier Farben färben lässt, dass benachbarte Länder verschiedene Farben bekommen. Nachdem das Problem lange offen war, konnte die Vermutung 1977 schliesslich mit Hilfe eines computergestützten Beweises gezeigt werden. (Ein einfacher Beweis existiert bis heute nicht.)

Das allgemeine Färbungsproblem ist das folgende.

Problem 7.4.1 (GRAPHENFÄRBUNG) *Gegeben ein Graph $G = (V, E)$, finde das minimale k, so dass G eine k-Färbung besitzt.*

Der optimale Wert von k heisst auch *chromatische Zahl* oder Färbungszahl von G. Die Färbungszahl eines Graphen G wird mit $\chi(G)$ bezeichnet. Es gilt der folgende Satz.

Satz 7.4.2 *Sei $k \in \mathbf{N}$. Zu entscheiden, ob ein Graph G eine k-Färbung besitzt, ist* NP-*vollständig, für alle $k \geq 3$.*

Hier wollen wir uns mit einer Variante dieses Problems befassen, bei der wir bereits wissen, dass G chromatische Zahl 3 hat, und das Ziel ist, eine Färbung von G mit möglichst wenig Farben zu finden. Es geht hier also nicht darum, die chromatische Zahl zu berechnen (das ist trivial mit der Voraussetzung), sondern eine möglichst gute Färbung unter der gegebenen Voraussetzung zu finden.

Satz 7.4.3 *Das Problem, eine 4-Färbung eines 3-färbbaren Graphen G zu finden, ist* NP-*schwer.*

Insbesondere ist damit auch das Auffinden einer (optimalen) 3-Färbbung NP-schwer, obwohl wir wissen, dass eine solche existiert. Es ist durchaus möglich (wenn auch nicht sehr wahrscheinlich), dass man in polynomieller Zeit eine 5-Färbung berechnen kann. Die beste obere Schranke, die bis 1994 bekannt war, ist folgende.

Satz 7.4.4 *Jeder 3-färbbare Graph auf n Knoten kann in polynomieller Zeit mit $O(\sqrt{n})$ Farben gefärbt werden.*

Beweis: Fixiere eine Zahl $\delta < n$. Nun betrachte einen Knoten v vom Grad grösser als δ (falls ein solcher existiert); sei $N(v)$ die Menge der Nachbarn von v in G. Da G 3-färbbar ist, muss $N(v)$ 2-färbbar sein, denn schauen wir uns eine 3-Färbung von G an, so können in $N(v)$ nur zwei Farben vorkommen (die dritte brauchen wir für v selbst). Eine solche 2-Färbung von $N(v)$ ist leicht zu finden (Greedy-Strategie). Entferne nun $N(v)$ und alle inzidenten Kanten aus G und fahre mit einem weiteren Knoten v' vom Grad grösser als δ fort, solange bis es keine solchen mehr gibt.
Dies passiert spätestens nach n/δ Runden, denn in jeder Runde werden ja mindestens δ Knoten aus G entfernt. Der verbleibende Graph hat Maximalgrad höchstens δ und kann leicht mit $\delta + 1$ Farben gefärbt werden (man färbe die Knoten einfach der Reihe nach und überlege sich, dass man zu jedem Zeitpunkt mindestens eine legale Farbe zur Verfügung hat).

In jeder der n/δ Runden vorher wurden bereits 2 Farben verwendet, insgesamt ergibt sich also eine legale Färbung mit höchstens

$$2\frac{n}{\delta} + \delta + 1$$

Farben. Wählen wir $\delta = \lceil \sqrt{2n} \rceil$, so erhalten wir eine Färbung mit höchstens $2 \lceil \sqrt{2n} \rceil + 1$ Farben, wie gewünscht. ◻

Im folgenden wird mit Hilfe von semidefinitem Programmieren gezeigt: Jeder 3-färbbare Graph auf n Knoten kann in polynomieller Zeit mit $O(n^{0.387})$ Farben gefärbt werden. Dazu ordnen wir (ähnlich wie bei Max-Cut) jedem Knoten v_i nicht eine einzelne Variable (seine Farbe), sondern einen Vektor $u_i \in \mathbf{R}^n$ zu. Die u_i definieren dann eine 'Vektor-Färbung'. Aus dieser werden wir dann später eine echte Färbung konstruieren. Zunächst definieren wir aber, was eine Vektor-Färbung sein soll. Im folgenden nehmen wir immer an, dass die Knotenmenge V von G einfach die Menge $V = \{1, \ldots, n\}$ ist.

Definition 7.4.5 *Eine* Vektor-k-Färbung *von G ist eine Abbildung $V \to \mathbf{R}^n$, $i \mapsto u_i$, wobei die u_i's Einheitsvektoren sind und*

$$u_i u_j \leq -\frac{1}{k-1}$$

für alle Kanten $\{i, j\} \in E$.

Diese Bedingung besagt, dass die Vektoren zu benachbarten Knoten einen grossen Winkel einschliessen müssen (für $k = 3$ etwa ergibt sich ein Winkel von mindestens 120^0).
Eine Vektor-Färbung ist auch für nichtganzzahliges k definiert. Das folgende Lemma zeigt, dass Vektor-Färbbarkeit eine Relaxierung der gewöhnlichen Färbbarkeit ist.

Lemma 7.4.6 *Jeder k-färbbare Graph ist Vektor-k-färbbar.*

Beweis: Uebung 7.5. Zu zeigen ist dazu, dass es k n-dimensionale Einheitsvektoren mit paarweisem Skalarprodukt höchstens $-1/(k-1)$ gibt; dann können wir die k Farben einfach auf diese Vektoren abbilden. ◻

Nun können wir Vektor-Färbungen mit semidefinitem Programmieren in Verbindung bringen. Es gilt nämlich, dass sich das Auffinden einer optimalen Vektor-Färbung (mit minimalem k) als ein semidefinites Programm schreiben lässt. Dazu betrachten wir eine Matrix $M = (m_{ij})$ mit der Bedeutung

$$m_{ij} = u_i u_j$$

und das folgende Programm.

$$
\begin{array}{lll}
\text{(SDP)} & \text{minimiere} & \alpha \\
& \text{unter} & \\
& & M = AA^T, \\
& & m_{ii} = 1, \quad \forall i, \\
& & m_{ij} \leq \alpha, \quad \forall \{i,j\} \in E.
\end{array}
$$

Sei α_{opt} die optimale Lösung dieses Programms. Der Wert k_{opt} mit

$$
\alpha_{opt} = -\frac{1}{k_{opt} - 1}
$$

heisst *vektor-chromatische Zahl* von G. Eine Vektor-k_{opt}-Färbung $u_1, \ldots, u_n$ findet man wieder mittels der Cholesky-Zerlegung der erhaltenen Matrix M (die u_i sind dann die Zeilen der Matrix A). Da G 3-färbbar ist, folgt aus Lemma 7.4.6, dass $k_{opt} \leq 3$ gilt.
Hier sind die drei Schritte zur Berechnung einer echten Färbung von G, die wir nun durchführen werden.
(I) Durch (approximatives) Lösen des Problems (SDP) und Cholesky-Zerlegung kann in polynomieller Zeit eine Vektor-$(3+\varepsilon)$-Färbung berechnet werden, für jedes $\varepsilon > 0$. Die Laufzeitabhängigkeit von ε ist von der Ordnung $\log(1/\varepsilon)$.

(II) Die in (I) erhaltene Vektor-Färbung wird randomisiert zu einer 'Halbfärbung' gerundet (d.h. mindestens die Hälfte aller Knoten hat eine Farbe, die von denen aller Nachbarn verschieden ist).

(III) Der Halbfärbungsalgorithmus wird rekursiv verwendet, um alle Knoten zu färben. Asymptotisch braucht man nicht mehr Farben als für die Halbfärbung.

Zu Schritt I muss hier nichts mehr gesagt werden; wir verfahren genauso wie bei Max-Cut und verwenden die Lösbarkeit von (SDP) und die Cholesky-Zerlegung als 'black box'.

Schritt (II). Sei $u_1, \ldots, u_n$ eine Vektor-$3+\varepsilon$-Färbung. Das bedeutet,

$$
u_i u_j \leq -\frac{1}{2} + \varepsilon
$$

für $i \neq j$, d.h. u_i und u_j schliessen einen Winkel von mindestens $(120-\varepsilon)^0$ ein. (Zur Vereinfachung bezeichnen wir im folgenden jede beliebig klein wählbare Konstante mit ε).
Eine Färbung erhalten wir nun wie folgt: wähle r zufällige Hyperebenen durch den Ursprung; diese zerlegen die Kugeloberfläche S_n in Zellen, von denen jede eine andere Farbe bekommt. Knoten i bekommt nun die Farbe der Zelle, in der 'sein' Vektor u_i liegt. Diese Färbung wird im allgemeinen nicht legal sein, weil es Kanten geben kann, deren Knoten beide in der gleichen Zelle liegen. Da die

zugehörigen Vektoren aber einen grossen Winkel einschliessen, werden (falls r gross genug gewählt wird) nur so wenige Kanten falsch gefärbt, dass wir eine Halbfärbung erhalten.

Genauer gilt: Vektoren u_i und u_j die einen Winkel θ einschliessen, werden durch eine zufällige Ursprungshyperebene mit Wahrscheinlichkeit θ/π getrennt. Das heisst, Vektoren, die zu benachbarten Knoten gehören, liegen mit Wahrscheinlichkeit höchstens

$$1 - \frac{(120-\varepsilon)^0}{180^0} = \frac{1}{3} + \varepsilon$$

auf der gleichen Seite einer zufälligen Hyperebene. Es folgt, dass sie mit Wahrscheinlichkeit höchstens

$$(\frac{1}{3} + \varepsilon)^r$$

in der gleichen Zelle des Arrangements der r Hyperebenen liegen, also fälschlicherweise die gleiche Farbe bekommen. Nun wählen wir

$$r = 2 + \lceil \log_3 \Delta \rceil,$$

wobei Δ der Maximalgrad des Graphen ist, und

$$\varepsilon = \frac{1}{3r}.$$

Dann ist $\log(1/\varepsilon) \approx \log\log\Delta$, und damit ist die Laufzeit zur Bestimmung der ursprünglichen Vektor-$3+\varepsilon$-Färbung sicher polynomiell in n. Die Wahrscheinlichkeit, dass eine spezifische Kante falsch gefärbt wird, ist höchstens

$$(\frac{1}{3} + \varepsilon)^r = (\frac{1}{3}(1 + \frac{1}{r}))^r \le \frac{e}{9}(\frac{1}{3})^{\lceil \log_3 \Delta \rceil} \le \frac{e}{9\Delta} \le \frac{1}{3\Delta}.$$

Die erwartete Anzahl der illegal gefärbten Kanten ist damit höchstens

$$m\frac{1}{3\Delta} \le \frac{n\Delta}{2} m \frac{1}{3\Delta} = \frac{n}{6}.$$

Die erwartete Anzahl der falsch gefärbten Knoten ist dann höchstens $n/3$, und sie ist grösser als $n/2$ mit Wahrscheinlichkeit höchstens $2/3$ (nach der Markov-Ungleichung). Durch Wiederholung des Experiments können wir also mit beliebig hoher Wahrscheinlichkeit $n/2$ korrekt gefärbte Knoten garantieren. Das Resultat ist eine sogenannte Halbfärbung.

Es bleibt noch zu berechnen, wieviele Farben wir benötigt haben: die r Hyperebenen unterteilen den Raum in höchstens 2^r Zellen; damit gibt es höchstens

$$2^{2+\lceil \log_3 \Delta \rceil} \le 8\ 2^{\log_3 \Delta} = 8\Delta^{\log_3 2} = O(n^{0.631})$$

Farben.

Dies ist zunächst enttäuschend, weil die Schranke sogar schlechter ist als die $O(\sqrt{n})$-Schranke aus Lemma 7.4.4. Man kann aber ausnutzen, dass die Schranke aus Schritt (II) vom Maximalgrad des Graphen G abhängt. Fixiere wie vorher eine Schranke $\delta < \Delta$. Solange G noch Knoten v vom Grad grösser als δ enthält, färbe $N(v)$ mit 2 Farben und entferne $N(v)$. Es gibt höchstens n/δ Runden, in denen wir höchstens $2n/\delta$ Farben verbrauchen. Auf den verbliebenen Graphen (vom Maximalgrad höchstens δ) wenden wir dann Schritt (II) an. Das ergibt eine Halbfärbung von G mit

$$O(\frac{n}{\delta} + \delta^{\log_3 2})$$

Farben.
Diese Schranke wird minimiert, wenn die beiden Summanden gleich sind, d.h.

$$\frac{n}{\delta} = \delta^{\log_3 2},$$

also

$$\delta = n^{1/(1+\log_3 2)}.$$

Das ergibt dann

$$O(\frac{n}{\delta}) = O(n^{1-1/(1+\log_3 2)}) = O(n^{0.387})$$

Farben.

Schritt (III). Was haben wir in den ersten zwei Schritten erreicht? Eine Halbfärbung mit $O(n^{0.387})$ Farben, d.h. höchstens $n/2$ Knoten haben noch illegal gefärbte Kanten, mit hoher Wahrscheinlichkeit. Das Vorgehen ist nun klar: Die höchstens $n/2$ Knoten werden einfach rekursiv mit neuen Farben versehen. Das ergibt dann insgesamt eine echte Färbung mit höchstens

$$O(n^{0.387} + (\frac{n}{2})^{0.387} + (\frac{n}{4})^{0.387} + \ldots = O(n^{0.387})$$

Farben. Wir erhalten also

Satz 7.4.7 *Ein 3-färbbarer Graph G auf n Knoten kann mit hoher Wahrscheinlichkeit in polynomieller Zeit mit $O(n^{1-1/(1+\log_3 2)}) = O(n^{0.387})$ Farben gefärbt werden.*

7.5 Uebungen

Uebung 7.1 Zeige, dass man das Problem, den kleinsten Eigenwert einer symmetrischen, positiv semidefiniten Matrix M zu finden, als semidefinites Programm formulieren kann.

Uebung 7.2 Beweise, dass die Menge der positiv semidefiniten Matrizen $M \in \mathbf{R}^{n\times n}$ eine konvexe Teilmenge des $\mathbf{R}^{n\times n}$ bildet, d.h. für zwei positiv semidefinite Matrizen M_1, M_2 ist auch jede Konvexkombination

$$(1-\lambda)M_1 + \lambda M_2, \quad 0 \leq \lambda \leq 1$$

positiv semidefinit.

Uebung 7.3 Zeige

$$\min_{0<t<\pi} \frac{2t}{\pi(1-\cos(t))} > 0.87856.$$

Uebung 7.4 Zeige, dass $1 - \arccos(z) \geq \alpha\frac{1}{2}(1+z)$ für $-1 < z < 1$. Hierbei ist $\alpha = 0,87856$.

Uebung 7.5 Sei $k \leq n$. Beweise, dass es k Einheitsvektoren $u_1, \ldots, u_k \in \mathbf{R}^n$ gibt, so dass

$$u_i u_j \leq -\frac{1}{k-1}$$

für alle Paare $i \neq j$ gilt.

Uebung 7.6 Beweise, dass ein Graph genau dann 2-färbbar ist, wenn er Vektor-2-färbbar ist.

Uebung 7.7 Zeige, dass das folgende ganzzahliges quadratisches Programm quadratische ganzzahlige Programm MIS eine äquivalente Formulierung des Problems Maximale Unabhängige Menge ist. Für einen Graphen $G = (V, E)$ definieren wir das Programm MIS wie folgt. Die Knotenmenge V sei $\{1, 2, \ldots, n\}$. Für jeden Knoten führen wir eine Variable x_i ein.

$$\begin{array}{lll} \text{(MIS)} & \text{maximiere} & \displaystyle\sum_{i=1}^{n} x_i - \sum_{\{i,j\}\in E} x_i x_j \\ & \text{unter} & \\ & & x_i \in \{0,1\}, \qquad i = 1, \ldots, n. \end{array}$$

Uebung 7.8 Zeige, dass das Programm MIS und das relaxierte Programm

$$\begin{array}{lll} \text{(Relaxiertes MIS)} & \text{maximiere} & \displaystyle\sum_{i=1}^{n} x_i - \sum_{\{i,j\}\in E} x_i x_j \\ & \text{unter} & \\ & & 0 \leq x_i \leq 1 \qquad i = 1, \ldots, n, \end{array}$$

stets dasselbe Optimum haben.

Kapitel 8

Nichtapproximierbarkeitsresultate

Unter einem Nichtapproximierbarkeitsresultat versteht man eine Aussage der folgenden Form: Für ein NP-schweres Optimierungsproblem kann es unter der Voraussetzung P$\neq$ NP keinen Approximationsalgorithmus mit Güte $< c$ (Minimierungsproblem) oder $> c$ (Maximierungsproblem) geben. Hier ist dann c eine Konstante oder auch eine Funktion der Eingabe. Die Voraussetzung P$\neq$ NP ist offensichtlich notwendig. Sollte nämlich P=NP gelten, so können viele NP-schwere Optimierungsprobleme exakt in polynomielle Zeit gelöst werden. Ein Nichtapproximierbarkeitsresultat haben wir bereits kennengelernt. Im Abschnitt über das Problem des Handlungsreisenden (siehe Seite 22ff.) haben wir gesehen, dass es für kein $c < 1$ einen Approximationsalgorithmus für das allgemeine TSP mit Güte $> c$ geben kann, vorausgesetzt P$\neq$NP. Zum Aufwärmen wollen wir ein weiteres relativ einfaches Nichtapproximierbarkeitsresultat für das Färben von Graphen herleiten.

8.1 Ein einfaches Nichtapproximierbarkeitsresultat

Betrachten wir wieder das Problem Graphenfärbung. Zur Erinnerung (siehe auch Abschnitt 7.4): Eine zulässige Färbung für einen Graphen $G = (V, E)$ ist eine Belegung der Knoten in V mit Farben, so dass keine zwei Knoten, die durch eine Kante in E verbunden sind, dieselbe Farbe haben. $\chi(G)$ ist die minimale Anzahl von Farben, die für eine zulässige Färbung von G benötigt wird. Ein Graph heisst *k-färbbar*, falls $\chi(G) \leq k$. Das *Färbungsproblem* besteht dann natürlich in der Berechnung von $\chi(G)$ bei Eingabe G. Wir haben bereits im Abschnitt über semidefinites Programmieren gesehen, dass das Färbungsproblem schwer ist. Insbesondere ist es auch NP-vollständig, zu entscheiden, ob ein Graph 3-färbbar ist (Satz 7.4.2) Wir werden dieses Ergebnis benutzen, um folgendes Nichtapproximierbarkeitsresultat zu zeigen.

Satz 8.1.1 *Sei $m > 0$ eine beliebige Konstante. Falls P≠NP, so gibt es keinen Approximationsalgorithmus, der für Graphen G mit $\chi(G) \geq m$ die Färbungszahl $\chi(G)$ mit Güte $< 4/3$ approximiert.*

Einige Bemerkungen, bevor wir den Satz beweisen. Die Einschränkung auf Graphen G mit $\chi(G) \geq m$ macht das Resultat natürlich stärker. Wir erlauben einem Algorithmus dadurch, auf Graphen mit kleiner Färbungszahl keine vernünftigen Antworten zu geben. Wir wollen uns überlegen, dass für $m = 1$ (alle Graphen sind erlaubt) der Satz einfach zu beweisen ist. Wir können dann nämlich mit einem Approximationsalgorithmus A der Güte $< 4/3$ entscheiden, ob ein Graph 3-färbbar ist. Dieses geschieht, indem wir A auf den Graphen G anwenden. Ist die Ausgabe < 4, so sagen wir, dass der Graph 3-färbbar ist, sonst sagen wir, dass er nicht 3-färbbar ist. Man überlegt sich leicht, dass die Antwort stets korrekt ist.
Diese Idee können wir für $m > 3$ schon nicht mehr verwenden, denn der Approximationsalgorithmus kann dann ja bei 3-färbbaren Graphen alles mögliche als Antwort liefern. Wir werden dieses Problem umgehen, indem wir zu jedem beliebigen Graphen G einen neuen Graphen G' definieren, so dass G' Färbungszahl mindestens m hat, wir aber andererseits die Färbungszahl von G leicht aus der Färbungszahl für G' berechnen können. Wir wenden dann den angenommenen Approximationsalgorithmus auf G' an. Den Graphen G' erhalten wir durch das sogenannte *Graphprodukt*.

Definition 8.1.2 *Seien $G_1 = (V_1, E_1), G_2 = (V_2, E_2)$ zwei Graphen mit disjunkten Knotenmengen. Das Graphenprodukt $G_1 \times G_2$ von G_1, G_2 ist der Graph gegeben durch die Knotenmenge $V = V_1 \times V_2$ und die Kantenmenge*

$$E = \{\{(v_1, u_1), (v_2, u_2)\} | \{v_1, v_2\} \in E_1 \text{ oder } v_1 = v_2 \text{ und } \{u_1, u_2\} \in E_2\}.$$

Das Graphenprodukt erhält man, indem man jeden Knoten in G_1 durch eine Kopie von G_2 ersetzt, ausserdem werden je zwei Knoten in verschiedenen Kopien von G_2 durch eine Kanten verbunden, wenn die den Kopien entsprechenden Knoten in G_1 durch eine Kante verbunden sind. Die Abbildung 8.1 verdeutlicht die Definition an einem Beispiel. Im Allgemeinen ist es nicht einfach, die Färbungszahl von $G_1 \times G_2$ aus der Färbungszahl von G_1 und G_2 zu bestimmen. Es gibt jedoch einen für uns interessanten Fall, in dem dieses möglich ist. Mit K_m bezeichnen wir den vollständigen Graphen auf m Knoten, d.h. K_m hat m Knoten und je zwei Knoten sind durch eine Kante verbunden. Es gilt

Lemma 8.1.3 *Sei G ein Graph mit Färbungszahl $\chi(G)$. Der Produktgraph $K_m \times G$ hat Färbungszahl $m\chi(G)$.*

Beweis: $K_m \times G$ entsteht, indem wir m Kopien von G bilden und die Knoten von je zwei Kopien paarweise miteinander durch Kanten verbinden. Zum Färben jeder Kopie von G werden $\chi(G)$ Farben benötigt. Da aber je zwei Kopien durch alle möglichen Kanten miteinander verbunden sind, müssen wir in jeder Kopie

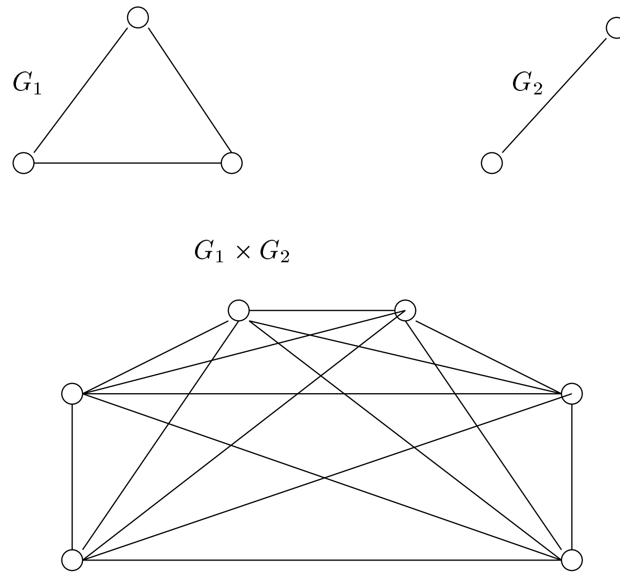

Abbildung 8.1: Beispiel eines Produktgraphen

Farben verwenden, die von den Farben in den anderen Kopien verschieden sind. Insgesamt benötigen wir also $m\chi(G)$ viele Farben. $\square$

Beweis von Satz 8.1.1: Sei A ein Approximationsalgorithmus, der für Graphen G mit $\chi(G) \geq m$ die Färbungszahl mit Güte $< 4/3$ approximiert. Wir werden daraus einen Algorithmus machen, der entscheidet, ob ein Graph 3-färbbar ist. Dieses wird den Satz beweisen.

Um zu entscheiden, ob ein gegebener Graph G 3-färbbar ist, wenden wir A auf den Graphen $K_m \times G$ an. Liefert A bei Eingabe $K_m \times G$, dass die Färbungszahl dieses Produktgraphen $< 4m$ ist, so sagen wir, dass G 3-färbbar ist, sonst sagen wir, dass der Graph G nicht 3-färbbar ist.

Zunächst einmal überlegen wir uns, dass dieses ein polynomieller Algorithmus ist. Da A ein polynomieller Algorithmus ist, müssen wir nur zeigen, dass die Grösse von $K_m \times G$ polynomiell in der Grösse von G ist. Nun ist die Anzahl der Knoten in $K_m \times G$ gerade mn, wenn n die Anzahl der Knoten in G ist. Da m eine Konstante ist, ist dieses polynomiell in der Grösse von G. Dann ist sicher auch die Anzahl der Kanten in $K_m \times G$ polynomiell in der Grösse von G.

Wir zeigen nun, dass die Antwort des Algorithmus stets korrekt ist. Zunächst einmal ist $\chi(K_m \times G) \geq m$ für alle Graphen. Denn $K_m \times G$ enthält als Teilgraphen den K_m, dessen Färbungszahl ist m. Daher liefert A bei Eingabe $K_m \times G$ also einen Wert $A(g \times K_m)$ mit $A(K_m \times G) < 4/3\chi(K_m \times G)$.

Ist G 3-färbbar, so ist nach Lemma 8.1.3 $\chi(K_m \times G) = 3m$. Die Antwort von A bei Eingabe $K_m \times G$ ist daher $< 4/3 \times 3m < 4m$. Unser Algorithmus wird G also

korrekt zu einem 3-färbbaren Graphen erklären.
Ist andererseits G nicht 3-färbbar, so ist $\chi(G) \geq 4$. Die Färbungszahl von $K_m \times G$ ist daher $\geq 4m$. Da die Antwort von A immer mindestens so gross ist wie die Färbungszahl, ist also in diesem Fall die Antwort von A mindestens $4m$, und wieder ist die Antwort des Algorithmus für 3-Färbbarkeit korrekt. □

Betrachten wir die Grundidee des Beweises. Um für ein Minimierungsproblem Π wie dem Färbungsproblem ein Nichtapproximierbarkeitsresultat zu erhalten, machen wir folgendes. Wir nehmen eine NP-vollständiges Sprache L, im Fall oben das 3-Färbbarkeitsproblem, und versuchen eine Abbildung zu konstruieren, die $x \in L$ auf Instanzen von Π mit kleinem Wert abbildet, während die Abbildung $x \notin L$ auf Instanzen von Π mit relativ grossem Wert abbildet. Auf diese Art entsteht zwischen den möglichen Werten von Bildern von $x \in L$ und den möglichen Werten von Bildern von $x \notin L$ eine *Lücke*. Um zu entscheiden, ob $x \in L$ müssen wir entscheiden, auf welcher Seite dieser Lücke der Wert des Bildes von x liegt. Hierzu genügt ein Approximationsalgorithmus. Die Grösse der Lücke bestimmt dann, wie gut die Approximationsgüte sein muss. Oder im Sinn von Nichtapproximierbarkeit ausgedrückt, welche Approximationsgüte nicht erreicht werden kann. Graphisch haben wir diese Idee in Abbildung 8.2 für das Färbbarkeitsproblem dargestellt.

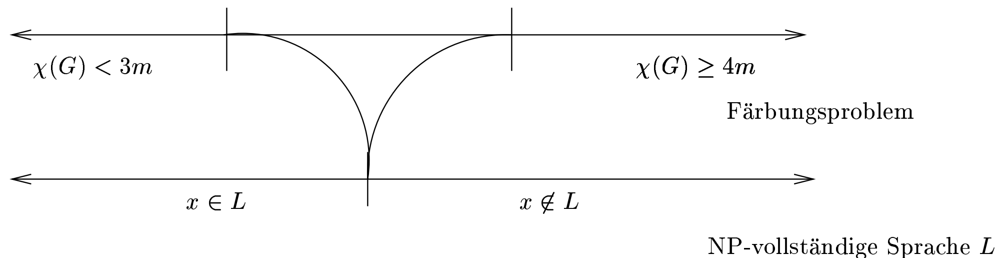

Abbildung 8.2: Erzeugen einer Lücke

8.2 Lückenerhaltende Reduktionen

Der gerade beschriebene Prozess ist aber nicht die einzige Möglichkeit Nichtappoximierbarkeitsresultate zu erzielen. Eine andere Möglichkeit besteht in sogenannten *lückenerhaltenden Reduktionen*. Diese werden wir jetzt für Maximierungsprobleme definieren und dann an einem Beispiel erläutern. Für Minimierungsprobleme kann eine analoge Definition gegeben werden.

Definition 8.2.1 *Eine* lückenerhaltende Reduktion *eines Maximierungsproblems* Π_1 *auf ein Maximierungsproblem* Π_2 *ist ein polynomialzeit Algorithmus* R*, der als*

Eingabe Instanzen von Π_1 *akzeptiert und dessen Ausgabe Instanzen von* Π_2 *sind. Weiter existieren Funktionen* c, c' *von den natürlichen Zahlen in die natürlichen Zahlen und Funktionen* ρ, ρ' *von den natürlichen Zahlen in die reellen Zahlen* < 1*, mit den folgenden Eigenschaften*

(1) *Ist der Wert einer optimalen Lösung von Instanz* $I \in \Pi_1$ *mindestens* $c(I)$*, so ist der Wert einer optimalen Lösung der Instanz* $R(I) \in \Pi_2$ *mindestens* $c'(R(I))$*.*

(2) *Ist der Wert einer optimalen Lösung von Instanz* $I \in \Pi_1$ *echt kleiner als* $c(I)\rho(I)$*, so ist der Wert einer optimalen Lösung der Instanz* $R(I) \in \Pi_2$ *echt kleiner als* $c'(R(I))\rho'(R(I))$*.*

Wir nennen eine solche Reduktion dann eine (c, ρ, c', ρ') *Reduktion.*

Die Bedeutung dieser Definition liegt in den folgenden Beobachtungen. Angenommen wir haben einen Algorithmus A, der für eine NP-vollständige Sprache L Elemente $x \in L$ auf Instanzen I von Π_1 mit $\text{opt}(I) \geq c(I)$ abbildet. Elemente $x \notin L$ hingegen werden von A auf Instanzen I mit $\text{opt}(I) < c(I)\rho(I)$ abgebildet. Wie beim Färbungsproblem können wir dann schliessen, dass es unter P$\neq$ NP keinen Approximationsalgorithmus mit Güte $< \rho$ für Π_1 geben kann. Denn mit einem solchen könnte zwischen den Fällen $x \in L$ und $x \notin L$ entschieden werden. Die Existenz einer (c, ρ, c', ρ') Reduktion R zeigt dann aber auch, dass es keinen Approximationsalgorithmus mit Güte $< \rho'$ für Π_2 geben kann. Dies sieht man wie folgt. Mit einem Approximationsalgorithmus für Π_2 mit Güte $< \rho'$ kann zwischen den Fällen $\text{opt}(I) \geq c'(I)$ und $\text{opt}(I) < c'(I)\rho'(I)$ für Instanzen von Π_2 entschieden werden. Nach Definition einer lückenerhaltenden Reduktion sehen wir, dass dann zwischen den Fällen $\text{opt}(I) \geq c(I)$ und $\text{opt}(I) < c(I)\rho(I)$ für Instanzen von Π_1 entschieden kann. Wir wir gesehen haben, liefert dieses einen polynomiellen Algorithmus für die NP-vollständige Sprache L.
In Abbildung 8.3 ist diese Ueberlegung graphisch dargestellt. Wir werden nun eine lückenerhaltende Reduktion von Max-3-SAT nach Unabhängige Menge beschreiben. Hierbei ist Max-3-SAT die Optimierungsversion von 3-SAT. Das heisst, gegeben eine Formel φ in 3-konjunktiver Normalform, dann ist eine Belegung der Variablen in φ gesucht, die eine möglichst grosse Anzahl der Klauseln in φ erfüllt. In unserer lückenerhaltenden Reduktion wird die Funktion c die Anzahl der Klauseln in der Booleschen Formel φ zählen, c' wird einfach die Anzahl der Knoten in einem Graphen dividiert durch 3 sein. ρ wird die konstante Funktion $1 - \epsilon$ sein, $\epsilon > 0$ beliebig. ρ' wird ebenfalls die konstante Funktion $(1 - \epsilon)$ sein. Den Algorithmus, der eine Formel φ in 3-konjunktiver Normalform in einen Graphen überführt, haben wir bereits im Abschnitt über NP-Vollständigkeit kennengelernt (siehe Seite 17). Hier deshalb nur eine kurze Wiederholung.
Wir nehmen an, dass jede Klausel C_j von φ genau 3 Literale l_{j1}, l_{j2}, l_{j3} enthält. Durch Kopieren von bereits vorhandenen Literalen kann dieses immer erreicht

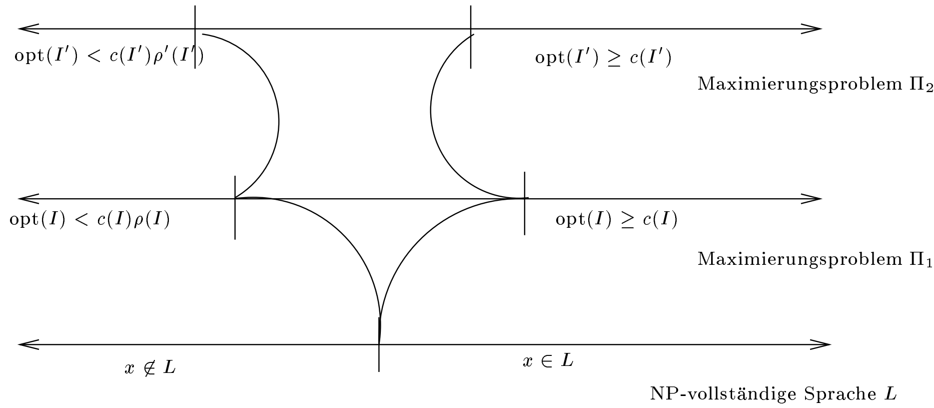

Abbildung 8.3: Erhalten einer Lücke

werden. Für jedes Literal l_{js} konstruieren wir im Graphen G einen Knoten. Die Knoten, die den Literalen in einer Klausel entsprechen, sind durch eine Kante verbunden. Ausserdem verbinden wir Knoten, die den Literalen l_{js}, l_{it} entsprechen, durch eine Kante, wenn $l_{js} = \neg l_{it}$.
Ist m die Anzahl der Klauseln in φ so ist $3m$ die Anzahl der Knoten im konstruierten Graphen G. Wir sahen bereits im Abschnitt über NP-Vollständigkeit, dass eine Formel φ für die alle m Klauseln gleichzeitig erfüllbar sind, in einen Graphen überführt wird, der eine unabhängige Menge der Gösse m besitzt. Andererseits kann man auch leicht zeigen, dass eine unabhängige Menge der Grösse $\geq (1-\epsilon)m$ zu einer Belegung der Variablen in φ führt, die $\geq (1-\epsilon)m$ viele Klauseln erfüllt. Denn eine unabhängige Menge kann aus einem Dreieck, dass einer Klausel entspricht, nur ein Element enthalten. Eine unabhängige Menge der Grösse $(1-\epsilon)$ enthält also Knoten, die Literalen aus $(1-\epsilon)m$ vielen Klauseln entsprechen. Keine zwei Literale, die den Knoten aus der unabhängigen Menge entsprechen, sind Negationen voneinander. Daher gibt es eine Belegung der Variablen, die all diese Literale erfüllt. Dann aber sind auch die $(1-\epsilon)$ vielen Klauseln, aus denen diese Literale stammen, erfüllt. Damit ist dieses also die angekündigte lückenerhaltende Reduktion, mit $\rho = \rho' = (1-\epsilon)$.
Diese Reduktion liefert uns nur dann ein Nichtapproximierbarkeitsresultat für Unabhängige Menge, wenn wir zeigen können, dass es ein $\epsilon > 0$ gibt, so dass Max-3-SAT nicht mit Güte $(1-\epsilon)$ approximiert werden kann. Dieses ist die Aussage des nächsten Satzes, den wir erst im nächsten Abschnitt beweisen werden.

Satz 8.2.2 *Es gibt eine Konstante $\epsilon > 0$ und einen polynomiellen Algorithmus A, der als Eingaben Boolesche Formeln φ akzeptiert und diese auf Boolesche Formeln $R(\varphi)$ in 3-konjunktiver Normalform abbildet, so dass*

(1) *Ist φ erfüllbar, so sind alle Klausel von $R(\varphi)$ erfüllbar.*

(2) *Ist φ nicht erfüllbar und enthält $R(\varphi)$ m Klauseln, so sind nur $(1-\epsilon)m$ dieser Klauseln gleichzeitig erfüllbar.*

Hieraus erhalten wir

Korollar 8.2.3 *Mit demselben ϵ wie in Satz 8.2.2 gilt: Vorausgesetzt* P$\neq$NP*, gibt es keinen Approximationsalgorithmus mit Güte $> (1-\epsilon)$ für* Max-3-SAT*.*

Beweis: Hier der Beweis, um noch einmal die Konzepte zu verdeutlichen. Angenommen, ein Approximationsalgorithmus A mit Güte $> (1-\epsilon)$ existiert. Um zu entscheiden, ob eine gegebene Formel φ erfüllbar ist, wenden wir A auf $R(\varphi)$ an, wobei R der Algorithmus aus Satz 8.2.2 ist. Ist m die Anzahl der Klauseln in $R(\varphi)$ und liefert A, dass mehr als $(1-\epsilon)m$ viele dieser Klauseln gleichzeitig erfüllt werden können, so geben wir aus, dass φ erfüllbar ist. Sonst geben wir aus, dass φ nicht erfüllbar ist.
Die Antwort dieses Algorithmus ist immer korrekt, denn für eine nicht erfüllbare Formal φ sind ja höchstens $(1-\epsilon)m$ der Klauseln in $R(\varphi)$ gleichzeitig erfüllbar. Andererseits liefert eine erfüllbare Formel φ eine Formel $R(\varphi)$ bei der ebenfalls alle Klausel gleichzeitig erfüllbar sind. Algorithmus A muss dann ausgeben, dass mehr als $(1-\epsilon)m$ Klauseln erfüllbar sind. ◻

Mit unserer lückenerhaltenden Reduktion von Max-3-SAT auf Unabhängige Menge sehen wir also, dass auch für Unabhängige Menge kein Approximationsalgorithmus mit Güte $(1-\epsilon)$ existiert. Eine Übungsaufgabe ist es zu zeigen, dass es für Unabhängige Menge ein Approximationsschema gibt, wenn es nur einen Approximationsalgorithmus mit konstanter Güte gibt. Wir haben aber gerade gesehen, dass es ein Approximationsschema für Unabhängige Menge nicht gegeben kann. Wir erhalten also

Satz 8.2.4 *Für keine Konstante $r < 1$ existiert ein Approximationsalgorithmus mit Güte r für* Unabhängige Menge*, es sei denn* P=NP.

Die Nichtapproximierbarkeit von Unabhängige Menge ist sogar noch dramatischer. Es kann gezeigt werden, dass es für kein $\epsilon > 0$ einen Approximationsalgorithmus mit Güte $1/n^{1-\epsilon}$ für Unabhängige Menge geben kann. Hier ist n die Anzahl der Knoten des Eingabegraphen. Diese Resultat ist allerdings sehr schwer zu beweisen.

8.3 Das Oeffnen der Lücke

In diesem Teilabschnitt werden wir den schon oben formulierten Satz 8.2.2 beweisen, der uns eine lückenerzeugende Reduktion von SAT auf Max-3-SAT liefert. Genauer, wir werden zeigen, dass es ein $\varepsilon > 0$ gibt und eine Funktion R, die SAT-Instanzen φ auf Max-3-SAT-Instanzen abbildet, mit den folgenden Eigenschaften.

1. Ist φ erfüllbar, so sind alle Klauseln in $R(\varphi)$ erfüllbar.
2. Ist φ nicht erfüllbar, so sind weniger als $(1-\varepsilon)m$ Klauseln von $R(\varphi)$ erfüllt, m die Anzahl der Klauseln in $R(\varphi)$.

Es folgt dann, dass es keinen Approximationsalgorithmus für Max-3-SAT mit Güte $1-\varepsilon$ geben kann. Insbesondere existiert also kein Approximationsschema für das Problem.
Das Haupthilfsmittel sind **PCP** (probabilistically checkable proofs), die eine Verallgemeinerung von NP definieren.

Definition 8.3.1 *Sei $L \subseteq \{0,1\}^*$ eine Sprache. Ein* $(r(n), q(n))$-Verifizierer *für L ist ein polynomieller Algorithmus V mit folgenden Eigenschaften.*

(i) V hat als Eingabe einen Binärstring x und einen 'Beweis' Π für $x \in L$. Die Ausgabe ist entweder '$x \in L$' (V akzeptiert) oder '$x \notin L$' (V akzeptiert nicht).

(ii) Falls $|x| = n$, so verwendet V $O(r(n))$ zufällige Bits und liest $O(q(n))$ Bits des Beweises Π (die Positionen der gelesenen Bits werden durch die Werte der zufälligen Bits bestimmt).

(iii) Falls $x \in L$, so existiert ein Beweis Π mit der Eigenschaft, dass V immer (mit Wahrscheinlichkeit 1) akzeptiert.

(iv) Falls $x \notin L$, so gilt für alle Beweise Π, dass V mit Wahrscheinlichkeit höchstens $1/2$ akzeptiert.

Ein solcher Verifizierer funktioniert also ganz ähnlich wie der Verifizierer, den wir für eine Sprache in NP gefordert haben (Definition 2.0.3). Er ist allerdings allgemeiner in dem Sinn, dass er Zufallsbits benutzen darf (NP-Verifizierer dürfen das nicht), '$x \notin L$' nur mit gewisser Wahrscheinlichkeit erkennen muss (NP-Verifizierer dürfen sich nie irren), und auf eine vorgegebene Zahl von Beweisbits eingeschränkt werden kann (NP-Verifizierer können auf alle Bits zugreifen). Die Fehlerwahrscheinlichkeit ist übrigens weitgehend beliebig, $1/2$ wurde nur aus Bequemlichkeit gewählt. Durch mehrmaliges Wiederholen des Verifikationsalgorithmus V kann die Akzeptanzwahrscheinlichkeit im Fall $x \notin L$ beliebig klein gemacht werden.
So wie man NP als die Menge aller Sprachen definiert, die einen Verifizierer im NP-Sinne haben, kann man auch $(r(n), q(n))$-Verifizierer als Basis nehmen.

Definition 8.3.2 PCP$(r(n), q(n))$ *ist definiert als die Menge aller Sprachen, die einen $(r(n), q(n))$-Verifizierer besitzen.*

Nun haben wir folgende Beobachtung, die ziemlich direkt aus dem oben Gesagten folgt.

Beobachtung 8.3.3 NP $= \mathrm{PCP}(0, \mathrm{poly}(n)) := \bigcup_{k=0}^{\infty} \mathrm{PCP}(0, n^k)$.

Erlauben wir also keine Zufallsbits, dafür aber polynomiell viele Beweisbits, so erhalten wir genau die NP-Verifizierer, denn die Fehlerwahrscheinlichkeit muss in diesem Fall 0 oder 1 sein; da wir wissen, dass sie höchstens 1/2 ist, kommt nur noch 0 in Frage, der Verifizierer irrt sich also nie.
Die Basis aller Nichtappropximierbarkeitsergebnisse ist nun die folgende alternative Charakterisierung von NP mit Hilfe von PCP, die 1992 von Arora et. al. gefunden wurde.

Satz 8.3.4 NP $= \mathrm{PCP}(\log n, 1)$.

Dies bedeutet, dass alle Sprachen in NP einen Verifizierer haben, der nur konstant viele Beweisbits liest, und trotzdem mit hoher Wahrscheinlichkeit zwischen '$x \in L$' und $x \notin L$' unterscheiden kann. Dazu benötigt er $O(\log n)$ Zufallsbits. Dies ist sehr überraschend und intuitiv sicher nicht klar. Die Inklusion NP $\subseteq \mathrm{PCP}(\log n, 1)$ ist auch sehr schwer zu zeigen; die andere Richtung, $\mathrm{PCP}(\log n, 1) \subseteq$ NP ist allerdings leicht, und wir werden sie hier skizzieren.

PCP$(\log n, 1) \subseteq$ **NP.** Sei V ein $(\log n, 1)$-Verifizierer für eine Sprache L. Verwendet V $c \log n$ Zufallsbits für konstantes c, so kann V durch $2^{c \log n} = n^c$ viele deterministische Algorithmen simuliert werden, von denen jeder einer möglichen Belegung der Zufallsbits entspricht. Jeder dieser Algorithmen hat polynomielle Laufzeit und liest $O(1)$ Beweisbits. Ein NP-Verifizierer für L kann nun wie folgt erhalten werden: für gegebenes x lässt man alle n^c deterministischen Ausprägungen von V nacheinander ablaufen; dies kostet nur polynomielle Zeit. Ist $x \in L$, so akzeptieren alle Ausprägungen, und die Antwort ist '$x \in L$'. Im Fall $x \notin L$ akzeptieren nach Definition von V höchstens die Hälfte aller Ausprägungen. Insbesondere gibt es also eine Ausprägung, die nicht akzeptiert, und sobald eine solche zu ihrem Ergebnis kommt, wird '$x \notin L$ ausgegeben.
Wir haben damit aus V einen deterministischen polynomiellen Verifizierer für L gebastelt, also ist $L \in$ NP.

Die Charakterisierung von NP mittels PCP wird es uns nun erlauben, die lückenerzeugende Reduktion von SAT auf Max-3-SAT durchzuführen. Als Zwischenschritt reduzieren wir dafür zunächst jede SAT-Instanz auf eine Instanz des Problems Max-k-Function-SAT, welches im Folgenden definiert wird und eine enge Beziehung zu PCP hat.

Problem 8.3.5 (MAX-k-FUNCTION-SAT) *Gegeben sind m Boolesche Funktionen $f_1, \ldots, f_m$ in jeweils k Variablen aus einer n-elementigen Variablenmenge $x_1, \ldots, x_n$ (k ist eine Konstante). Jede Funktion ist durch ihre Wahrheitstabelle der Grösse 2^k gegeben, die für jede Belegung der k Variablen den Funktionswert angibt.*

Gesucht ist eine Belegung der Variablen $x_1, \ldots, x_n$, die die Anzahl der Funktionen mit Wert `true` *maximiert. (Eine Funktion mit Wert* `true` *nennen wir auch* erfüllt*, um nahe an der SAT-Terminologie zu bleiben.)*

Hier ist nun die Reduktion SAT $\mapsto$ Max-k-Function-SAT. Sei φ eine SAT-Formel. Wir konstruieren dann eine Max-k-Function-SAT-Instanz $R_1(\varphi)$, wobei k die Anzahl der von einem festen $(\log n, 1)$-Verifizierer V für SAT gelesenen Beweisbits ist. (Damit ist k wirklich eine Konstante.) O.B.d.A. können wir annehmen, dass jeder von V verarbeitete Beweis Π Länge $N := kn^c$ für eine Konstante c hat, denn wie schon oben beobachtet, gibt es höchstens n^c deterministische Ausprägungen von V, von denen jede auf maximal k Bits zugreift. Potentiell kann damit auf höchstens N Bits des Beweises überhaupt zugegriffen werden, so dass wir den Beweis auf diese Bits komprimieren können.
Die Variablen von $R_1(\varphi)$ sind nun $x_1, \ldots, x_N$, mit der Interpretation, dass jede Belegung der Variablen einem möglichen Beweis entspricht (dabei identifizieren wir das Bit 0 mit dem Wert `false` und 1 mit dem Wert `true`).
Die Funktionen f_i werden durch die möglichen Auswahlen der $c \log n$ Zufallsbits definiert, wobei wir jede Auswahl mit einer Teilmenge $S \subseteq \{0,1\}^{c \log n}$ identifizieren können (S enthält die Positionen der mit 1 belegten Zufallsbits). Für eine solche Auswahl S seien $i_1(S), \ldots, i_k(S)$ die Positionen der von V gelesenen Bits in Π, $b(i)$ der Wert des Bits an Position i. Dann definieren wir die Funktion

$$f_S(b_1, \ldots, b_k) = \begin{cases} \texttt{true}, & \text{falls } V \text{ akzeptiert für } b(i_\ell(S)) = b_\ell, \ell = 1, \ldots, k \\ \texttt{false}, & \text{sonst} \end{cases}$$

f_S codiert also, wie sich V bei festen Zufallsbits S verhält, abhängig davon, was an den gelesenen Stellen des Beweises steht.
Die Wahrheitstabelle von f_S kann berechnet werden, indem wir V für alle 2^k möglichen Tupel $(b_1, \ldots, b_k)$ an den Stellen $i_1(S), \ldots, i_k(S)$ laufen lassen (und dabei die Zufallsbits festhalten). Da k konstant und die Laufzeit von V polynomiell ist, kann die Tabelle in polynomieller Zeit berechnet werden. Da es nur $2^{c \log n} = n^c$ Mengen S gibt, kann die Max-k-Function-SAT-Instanz $R_1(\phi)$ also in polynomieller Zeit erzeugt werden.
Nun gilt (einfach nach Konstruktion und Definition des Verifizierers) folgendes.

(i) Falls φ erfüllbar ist, so gibt es eine Belegung der $x_1, \ldots, x_N$ (einen Beweis Π), die alle Funktionen f_S in $R_1(\varphi)$ erfüllt (V akzeptiert immer).

(ii) Falls φ nicht erfüllbar ist, so gilt für alle Belegungen von $x_1, \ldots, x_N$ (alle Beweise Π), dass höchstens die Hälfte aller Funktionen f_S von $R_1(\varphi)$ erfüllt sind (V akzeptiert mit Wahrscheinlichkeit höchstens 1/2).

Wir haben also eine lückenerzeugende Reduktion, wobei die Lücke daher kommt, dass V entweder immer oder nur in der Hälfte der Fälle akzeptiert.

Um die zu Beginn angekündigte Reduktion auf Max-3-SAT zu erhalten, benötigen wir nun noch eine lückenerhaltende Reduktion R_2 von Max-k-Function-SAT auf Max-3-SAT. Die Komposition von R_1 auf R_2 liefert dann die gewünschte Reduktion R.
Sei also I eine Instanz von Max-k-Function-SAT, mit N Variablen $x_1, \ldots, x_N$ und m Funktionen $f_1, \ldots, f_m$ (in unserem Fall ist $m = n^c$).

Schritt 1. Wir erzeugen zunächst eine MAX-k-SAT Instanz, indem wir jede Funktion f_i auf eine k-SAT-Formel F_i abbilden und die Instanz I dann auf

$$R_2'(I) := \bigwedge_{i=1}^{m} F_i.$$

Dies ist dann auch eine k-SAT-Formel und damit eine MAX-k-SAT-Instanz.
Zur Konstruktion von F_i betrachten wir alle möglichen Belegungen $b_1, \ldots, b_k$ der Variablen $x_{i_1}, \ldots, x_{i_k}$ in der Funktion f_i, für die

$$f_i(b_1, \ldots, b_k) = f$$

gilt, und erzeugen für diese eine Klausel

$$C_i(b_1, \ldots, b_k) = \ell_1 \vee \cdots \vee \ell_k,$$

wobei

$$\ell_j := \begin{cases} x_{i_j}, & \text{falls } b_j = f \\ \neg x_{i_j}, & \text{falls } b_j = w \end{cases}$$

Dann gilt, dass $C_i(b_1, \ldots, b_k)$ genau dann nicht erfüllt ist, wenn $x_{i_j} = b_j$ für alle j gilt. Die Formel F_i ist nun die Konjunktion

$$F_i := \bigwedge_{b_1,\ldots,b_k, f_i(b_1,\ldots,b_k)=f} C_i(b_1, \ldots, b_k),$$

und es gilt dass F_i genau dann nicht erfüllt ist, wenn f_i nicht erfüllt ist. Für jede Belegung gilt also, dass die Anzahl erfüllter Funktionen f_i in I gleich der Anzahl erfüllter Formeln F_i in $R_2'(I)$ ist.
Zur Lückenerhaltung gilt dann folgendes.

(i) Sind in I alle Funktionen erfüllt, so sind in $R_2'(I)$ alle Klauseln $C_i(b_1, \ldots, b_k)$ erfüllt.

(ii) Falls in I mehr als $m/2$ aller Funktionen nicht erfüllt sind, so sind in $R_2'(I)$ mehr als $m/2$ der F_i und damit mehr als $m/2$ der Klauseln $C_i(b_1, \ldots, b_k)$ nicht erfüllt (denn wenn F_i nicht erfüllt ist, so gilt dies für mindestens

eine definierende Klausel). Mit anderen Worten, von den höchstens $m2^k$ Klauseln in $R_2'(I)$ sind mehr als $m/2$ nicht erfüllt, d.h. es sind höchstens

$$m2^k - m/2 = \left(1 - \frac{1}{2^{k+1}}\right) m2^k$$

Klauseln erfüllt.

Das heisst, bei der Reformulierung von Max-k-Function-SAT in MAX-k-SAT wird die Lücke zwar wesentlich kleiner, sie 'überlebt' aber: in $R_2'(I)$ ist ein Anteil von höchstens

$$1 - \frac{1}{2^{k+1}}$$

aller Klauseln erfüllbar, falls in I höchstens die Hälfte aller Funktionen erfüllbar waren.
Zur endgültigen Reduktion auf Max-3-SAT definieren wir nun noch eine (natürlich lückenerhaltende) Reduktion R_2'' von MAX-k-SAT auf Max-3-SAT.
Dazu betrachten wir eine Klausel

$$C_i(b_1, \ldots, b_k) = \ell_1 \vee \cdots \vee \ell_k$$

von $R_2'(I)$ und ersetzen diese durch eine Konjunktion von $k-2$ Klauseln mit 3 Literalen, wie folgt.

$$\begin{aligned} D_i(b_1, \ldots, b_k) \;:=\; & (\ell_1 \vee \ell_2 \vee z_1) \wedge (\neg z_1 \vee \ell_3 \vee z_2) \wedge \cdots \wedge (\neg z_{k-4} \vee \ell_{k-2} \vee z_{k-3}) \\ \wedge\; & (\neg z_{k-3} \vee \ell_{k-1} \vee \ell_k). \end{aligned}$$

Die z_j sind dabei neue Variablen, die wir exklusiv für die Klausel $C_i(b_1, \ldots, b_k)$ einführen.
Es gilt:

(i) Falls C_i erfüllt ist, so existiert eine Belegung der z_j, so dass auch D_i erfüllt ist.

(ii) Falls C_i nicht erfüllt ist, so ist auch D_i nicht erfüllt, für jede Belegung der z_j.

Es folgt, dass für feste Belegung die Anzahl der erfüllten Klauseln $C_i(b_1, \ldots, b_k)$ in $R_2'(I)$ gleich der Anzahl der erfüllten Formeln $D_i(b_1, \ldots, b_k)$ in $R_2(I)$ ist, wobei wir $R_2(I)$ aus $R_2'(I)$ erhalten, indem wir jede Klausel $C_i(b_1, \ldots, b_k)$ durch die Formel $D_i(b_1, \ldots, b_k)$ ersetzen (dies ist genau die Reduktion R_2''). Da diese aus $k-2$ Klauseln besteht, hat $R_2(I)$ nun $(k-2)m2^k$ Klauseln, und bezüglich der Lückenerhaltung gilt folgendes.

(i) Falls in I alle Funktionen erfüllt sind, so sind in $R_2(I)$ alle Klauseln erfüllt.

(ii) Falls in I mehr als $m/2$ aller Funktionen nicht erfüllt sind, so sind in $R_2(I)$ mehr als $m/2$ der D_i und damit mehr als $m/2$ der Klauseln nicht erfüllt (denn wenn D_i nicht erfüllt ist, so gilt dies für mindestens eine definierende Klausel). Mit anderen Worten, von den höchstens $(k-2)m2^k$ Klauseln in $R_2(I)$ sind mehr als $m/2$ nicht erfüllt, d.h. es sind höchstens

$$(k-2)m2^k - m/2 = \left(1 - \frac{1}{(k-2)2^{k+1}}\right)(k-2)m2^k$$

Klauseln erfüllt.

Das heisst, wenn in I höchstens die Hälfte aller Funktionen erfüllbar ist, so ist in der Max-3-SAT-Instanz $R_2(I)$ höchstens ein Anteil von

$$1 - \frac{1}{(k-2)2^{k+1}}$$

aller Klauseln erfüllt. Da k konstant ist, erhalten wir die gewünschte Lücke und können diesen Teilabschnitt in folgendem Satz zusammenfassen.

Satz 8.3.6 *Falls es für SAT einen* $(\log n, 1)$*-Verifizierer gibt, der höchstens k Beweisbits liest, so ist Max-3-SAT nicht auf einen Faktor von*

$$1 - \frac{1}{(k-2)2^{k+1}}$$

approximierbar.

Man kann zeigen, dass es einen Verifizierer gibt, der mit 11 Beweisbits auskommt, so dass sich eine explizite Schranke für die Approximierbarkeit ergibt. Diese liegt allerdings sehr knapp unter 1. Mit einer komplizierteren Anwendung der PCP-Methode kann man zeigen, dass Max-3-SAT nicht auf einen Faktor über 7/8 approximierbar ist. Dieses Ergebnis ist dann optimal, denn einen Algorithmus mit (erwartetem) Faktor 7/8 erhalten wir mittels des Algorithmus Random-Sat.

8.4 Uebungen

Uebung 8.1 Für einen Graphen G bezeichnen wir mit $i(G)$ seine *Unabhängigkeitszahl*, d.h. die Grösse einer grössten unabhängigen Menge in G. Zeige, dass für das Produkt $G = G_1 \times G_2$ zweier Graphen die Unabhäbgigkeitszahl gegeben ist durch $i(G) = i(G_1)i(G_2)$.

Uebung 8.2 Benutze Uebung 8.1, um zu zeigen, dass es für Unabhängige Menge ein Approximationsschema gibt, wenn es für Unabhängige Menge einen Approximationsalgorithmus mit konstanter Güte r gibt.

Kapitel 9

Musterlösungen zu den Uebungen

Musterlösung zu Uebung 1.1 Zu analysieren ist der Approximationsalgorithmus Next Fit zum Packen von Umzugskartons. Dieses Problem ist in der Literatur als das *Bin Packing* Problem bekannt.
Seien $K_1, \ldots, K_m$ die Umzugskartons in der Reihenfolge, wie sie von Next Fit gepackt werden. Für $j = 1, \ldots, m$ bezeichne g_j die "Füllhöhe" von K_j, also die Summe der Grössen aller in K_j gepackten Umzugsgüter. Hier ist das entscheidende Lemma.

Lemma 9.0.1 *Für $j = 1, \ldots, m-1$ gilt $g_j + g_{j+1} > 1$.*

Beweis: Angenommen, $g_j + g_{j+1} \leq 1$. Sei G_i das erste Umzugsgut, das von Next Fit in den Karton K_{j+1} gepackt wurde. Wegen $a_i \leq g_{j+1}$ gilt dann aber $g_j + a_i \leq 1$, G_i hätte also noch in K_j gepasst, Widerspruch. ◻

Nun benötigen wir noch folgende Beobachtung, die uns eine untere Schranke für die minimal mögliche Anzahl opt der Umzugskartons gibt. Es gilt nämlich

$$\text{opt} \geq \sum_{j=1}^{m} g_j,$$

weil natürlich mindestens soviele Kartons benötigt werden, wie die Gesamtgrösse der Umzugsgüter beträgt. Nun können wir wie folgt argumentieren.

$$2\text{opt} \geq 2\sum_{j=1}^{m} g_j = g_1 + g_m + \sum_{j=1}^{m-1}(g_j + g_{j+1}) > m - 1,$$

woraus $m < 2\text{opt} + 1$ folgt.[1] Da m und opt ganze Zahlen sind, gilt sogar

$$m \leq 2\text{opt},$$

[1] Diese Ableitung gilt für $m > 1$. Im Fall von $m = 1$ ist die Aussage offensichtlich.

womit gezeigt ist, dass Next Fit Approximationsgüte 2 hat.

Musterlösung zu Uebung 2.1

zu (a) Wir konstruieren eine Reduktion von Unabhängige Menge auf vertex cover. Gegeben eine Instanz von Unabhängiger Menge, d. h. ein Graph $G = (V, E)$ und eine Schranke K, betrachte die Instanz von vertex cover mit demselben Graphen G und der Schranke $n - K$, wobei n die Anzahl der Knoten von G ist.

Es gilt: $W \subseteq V$ ist ein vertex cover von G genau dann, wenn $V \setminus W$ eine unabhängige Menge ist. Denn wären zwei Knoten aus $V \setminus W$ noch durch eine Kante e verbunden, wäre W kein vertex cover, die Kante e wäre nicht abgedeckt. Umgekehrt, ist $V \setminus W$ eine unabhängige Menge, müssen alle Kanten von G mindestens einen Endpunkt in W haben, also ist W ein vertex cover.

Aus dieser Beobachtung folgt, das G ein vertex cover der Grösse K besitzt genau dann, wenn G eine unabhängige Menge der Grösse $n - K$ besitzt.

Die Reduktion kann in polynomieller Zeit berechnet werden, da nur K von n abgezogen werden muss.

zu (b) Wir konstruieren wiederum eine Reduktion auf Unabhängige Menge. Ist $G = (V, E)$ gegeben, so konstruiere $\overline{G} = (V', E')$ wie folgt: $V = V'$ und $\{i, j\} \in E', i, j \in V$ genau dann, wenn $\{i, j\} \notin E$. Einer Klique der Grösse K in G entspricht eine unabhängige Menge der Grösse K in $\overline{G}$. Durch eine Schleife über alle möglichen $\binom{n}{2}$ Kanten kann $\overline{G}$ in polynomieller Zeit konstruiert werden.

Musterlösung zu Uebung 2.2 Sei A ein polynomieller Algorithmus für das Entscheidungsproblem Unabhängige Menge. Bei Eingabe $G = (V, E)$ kann die Grösse einer optimalen unabhängigen Menge bestimmt weren, indem wir mit Hilfe von Algorithmus A für $k = 1, \ldots, |V|$ bestimmen, ob es eine unabhängige Menge der Grösse $\geq k$ gibt. (Etwas besser geht es durch binäre Suche nach der Grösse der optimalen unabhängigen Menge.)
Damit haben wir einen polynomiellen Algorithmus B für die Berechnung der Grösse einer optimalen unabhängigen Menge. Wir zeigen nun noch, wie mit diesem Algorithmus auch eine optimale unabhängige Menge in $G = (V, E)$ gefunden werden kann. Wir setzen $V = \{1, \ldots, |V|\}$.

Algorithmus 9.0.2 *Bestimmung einer optimalen unabhängigen Menge*

$U = \emptyset$
Berechne mit B die Grösse K der optimalen unabhängigen Menge in G.
`WHILE` $K \neq 0$ `DO`

```
        FOR i = 1 TO |V| DO
            V = V\{i}.
            Entferne aus E die zu i inzidenten Kanten
            Mit Algorithmus B berechne die Grösse m einer optimalen
            unabhängigen Menge in G = (V, E).
            IF m = K − 1 THEN
                U := U ∪ {i}, K := K − 1
            END
        END
    END
    Gib U aus.
```

Da B eine polynomieller Algorithmus sein soll, wird auch der oben beschriebene Algorithmus polynomiell sein. Die IF-Schleife stellt sicher, dass wir nur dann den Knoten i nicht zu U hinzunehmen, wenn es auch eine optimale unabhängige Menge gibt, die i nicht enthält.

Musterlösung zu Uebung 2.3 Sei $G = (V, E)$ ein Graph. Um die Formel φ zu konstruieren, die genau dann erfüllbar ist, wenn G einen Hamiltonschen Kreis besitzt, führen wir wie in der Aufgabenstellung beschrieben die Variablen $x_{ij}, i, j = 1, \ldots, |V|$ ein. Wie ebenfalls in der Aufgabenstellung beschriebn, soll x_{ij} für die Aussage "Der j-te Knoten des Hamiltonschen Kreises ist der Knoten i" stehen. Die Formel φ wird in konjunktiver Normalform sein (allerdings nicht in 3-konjunktiver Normalform). Es gibt vier verschiedene Arten von Klauseln, die die folgenden vier notwendigen und hinreichenden Bedingungen für die Existenz eines Hamiltonschen Kreises in G ausdrücken sollen.

(1) Der Knoten i muss auf dem Kreis liegen, $i = 1, \ldots, |V|$.

(2) Der Knoten i darf nicht gleichzitig als j-ter und als k-ter Knoten auf dem Kreis liegen, $i = 1, \ldots, |V|$.

(3) Irgendein Knoten muss der j-te Knoten auf dem Kreis sein, $j = 1 \ldots, |V|$.

(4) Ist i der j-te Knoten und k der $j + 1$-te Knoten auf dem Kreis, so muss in E die Kante $\{i, k\}$ enthalten sein, $j = 1, \ldots, |V|$. Hier identifizieren wir $|V| + 1$ und 1, um den Kreis wieder zu schliessen.

Wir können (1) mit Hilfe der Variable x_{ij} durch

$$\bigwedge_{i=1}^{|V|} (x_{i1} \vee x_{i2} \vee \ldots \vee x_{in})$$

ausdrücken.

Für (2) erhalten wir

$$\bigwedge_{i=1}^{|V|} \bigwedge_{j \neq k} (\neg x_{ij} \vee \neg x_{ik}).$$

(3) wird ausgedrückt durch

$$\bigwedge_{j=1}^{|V|} (x_{1j} \vee x_{2j} \vee \ldots \vee x_{nj}).$$

Um (4) zu erhalten bilden wir

$$\bigwedge_{\{i,k\} \notin E} \bigwedge_{j=1}^{|V|} (\neg x_{ij} \vee \neg x_{k,j+1}),$$

hierbei sind $|V| + 1$ und 1 zu identifizieren.
Die Formel φ erhalten wir schliesslich, indem wir die Formeln für (1)-(4) durch $\wedge$'s verbinden. Dass diese Formel genau dann erfüllt ist, wenn G einen Hamiltonschen Kreis bestizt, folgt aus der Tatsache, dass (1)-(4) hinreichend und notwendig für die Existenz eines Hamiltonschen Kreises sind.

Musterlösung zu Uebung 3.1 Sei $G = (V, E)$ ein Graph. Für $v \in V$ bezeichne mit $\deg(v)$ den Grad des Knoten $v \in V$, also die Anzahl der zu v inzidenten Kanten. Es gilt

$$\sum_{v \in V} \deg(v) = 2|E|,$$

wobei $|E|$ die Anzahl der Kanten ist. Um die Gleichung zu beweisen, beobachte man, dass in der Summe auf der linken Seite jede Kante $e \in E$ zweimal gezählt wird, einmal für jeden der beiden Knoten zu denen e inzident ist.
Als Konsequenz dieser Gleichung erhalten wir, dass $\sum_{v \in V} \deg(v)$ eine gerade ganze Zahl ist. Da die Knoten mit geradem Grad einen geraden Anteil zur Summe $\sum_{v \in V} \deg(v)$ beitragen, muss dieses auch für die Knoten mit ungeradem Grad gelten. Daraus folgt, dass es eine gerade Anzahl von Knoten ungeraden Grades gibt.

Musterlösung zu Uebung 3.2 Wir zeigen zunächst, dass die Bedingungen hinreichend sind, dass sie also die Existenz eines Eulerkreise garantieren. Sei G also ein zusammenhängender Graph, in dem jeder Knoten geraden Grad hat. Beginne an einem beliebigen Knoten v des Graphen mit einem Pfad. Entferne jede Kante des Graphen, die durchlaufen wird. Falls dieser Pfad nicht weitergeführt werden kann, muss er in v enden (gerader Grad jedes Knoten). Falls schon alle Kanten benutzt wurden, ist dieser Pfad ein Eulerkreisreis. Sonst betrachte den

übriggebliebenen Graphen. Da der Graph zusammenhängend ist, muss es noch einen Knoten auf dem ersten Pfad geben, aus dem noch eine Kante hinausführt. Starte nun wie vorher einen Pfad an diesem Knoten, und lösche alle benutzten Kanten. Dieser Pfad muss wieder am Anfangsknoten enden. Daher können die beiden Pfade zu einem einzigen Pfad verbunden werden. Falls dieser Pfad noch kein Eulerkreis ist, kann er wieder wie vorher verlängert werden, bis ein Eulerkreis konstruiert wurde.
Wir zeigen nun, dass die Bedingungen auch notwendig sind, d.h., dass sie von der Existenz eines Eulerkreises impliziert werden. Offensichtlich ist ein Graph G mit einem Eulerkreis zusammenhängend. Wir müssen noch zeigen, dass jeder Knoten $v \in V$ geraden Grad besitzt. Sei K eine Eulerkreis. Starte mit einem Durchlauf der Kanten von G an einem Knoten $w \neq v$. Jedesmal wenn der Kreis den Knoten v über irgendeine Kante kommend besucht, muss er diesen Knoten über eine andere Kante wieder verlassen. Wir können die Kanten also auf diese Art zu Paaren zusammenfassen. Dieses zeigt, dass v geraden Grad besitzt.

Musterlösung zu Uebung 3.3 Beweis durch ein Bild. Nehmen wir an, wir haben eine selbstüberschneidende Rundreise, wie auf der linken Seite in Abbildung 9.1.

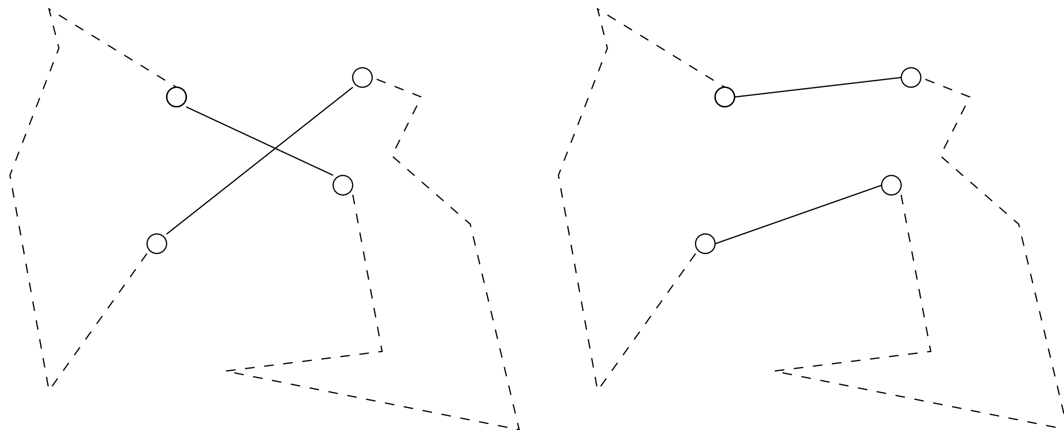

Abbildung 9.1: Kürzere Rundreise durch Flippen von Kanten

Indem wir die beiden sich schneidenden Kanten durchh die beiden Kanten auf der rechten Seite ersetzen, erhalten wir wieder eine Rundreise. Nach der Dreiecksungleichung muss diese Rundreise kürzer sein als die Rundreise auf der linken Seite. Die Rundreise auf der linken Seite kann daher auf keinen Fall die minimale Rundreise sein.

Musterlösung zu Uebung 3.4 Nehmen wir an, wir haben einen Schedule, der nicht so aussieht wie verlangt. Sei i der kleinste Index, so dass J_i nicht mit J_{2m-i+1} gepaart ist. Dann ist J_i mit J_k und J_{2m-i+1} mit J_ℓ gepaart, $i+1 \leq k, \ell \leq 2m-i$. Die vier Jobs $J_i, J_k, J_{2m-i+1}, J_\ell$ werden dann zum Zeitpunkt

$$\max(d_i + d_k, d_{2m-i+1} + d_\ell) = d_i + d_k$$

fertig. Aendern wir die Verteilung der Jobs so ab, dass J_i mit J_{2m-i+1} und J_k mit J_ℓ gepaart wird, so erhalten wir für diese vier Jobs einen Makespan von

$$\max(d_i + d_{2m-i+1}, d_k + d_\ell).$$

Nun gilt sowohl

$$d_i + d_{2m-i+1} \leq d_i + d_k$$

als auch

$$d_k + d_\ell \leq d_i + d_k,$$

also ist der Makespan des abgeänderten Schedules nicht schlechter als der des ursprünglichen. Auf diese Weise können nacheinander alle 'falschen' Paarungen beseitigt werden, ohne dass der Makespan ansteigt. Das heisst, es gibt einen optimalen Schedule, der so aussieht wie verlangt.

Musterlösung zu Uebung 3.5 Man überlegt sich leicht, dass SLS die $2km$ grössten Jobs so verteilt, dass alle Maschinen gleichzeitig fertig werden, und zwar zum Zeitpunkt

$$\frac{1}{m}\sum_{i=r+1}^{2km+r} i = \frac{1}{2m}\sum_{i=1}^{2km}(i+r) = \frac{1}{m}(\frac{2km(2km+1)}{2} + 2kmr) = k(2km+1) + 2kr.$$

Die restlichen Jobs erzeugen dann noch eine zusätzliche Laufzeit von r, so dass sich

$$\text{opt} \leq C = k(2km+1) + 2kr + r$$

ergibt. Nun können wir auch leicht die Güte von SLS in diesem speziellen Fall berechnen. Wir erhalten

$$\frac{C}{\text{opt}} \leq \frac{k(2km+1) + 2kr + r}{k(2km+1) + 2kr + \frac{r(r+1)}{2m}} = 1 + \frac{r - \frac{r(r+1)}{2m}}{k(2km+1) + 2kr + \frac{r(r+1)}{2m}}.$$

Für $n \to \infty$ geht die Güte gegen 1.

Musterlösung zu Uebung 4.1 Zusammen mit jedem Wert $F_j(i), i, j > 0$ speichern wir die Information, ob

$$F_j(i) = F_{j-1}(i)$$

oder

$$F_j(i) = g_j + F_{j-1}(i - w_j)$$

gilt. Eine der beiden Gleichungen stimmt nach Definition von $F_j(i)$ auf jeden Fall, und die gesuchte Information kann bei der Berechnung von $F_j(i)$ in Zeit $O(1)$ erhalten werden. (Falls beide Gleichungen zutreffen, entscheiden wir uns nach Belieben für eine der beiden.)
Im ersten Fall wissen wir (siehe den Beweis der Rekursionsformel für $F_j(i)$ im Skript), dass es eine gewichtsminimale Teilmenge von $\{1, \ldots, j\}$ mit Wert mindestens i gibt, die j nicht enthält, im zweiten Fall gibt es eine solche, die j enthält.
Um eine Teilmenge $S_j(i) \subseteq \{1, \ldots, j\}$ zu finden, die den Wert $F_j(i)$ realisiert, können wir dann die folgende Formel verwenden, die sich direkt aus der vorhergehenden Beobachtung ergibt.

$$S_j(i) = \begin{cases} S_{j-1}(i), & \text{falls } F_j(i) = F_{j-1}(i) \\ S_{j-1}(i - w_j) \cup \{j\}, & \text{falls } F_j(i) = g_j + F_{j-1}(i - w_j) \end{cases}.$$

Verwende diese Formel nun zur Berechnung einer global optimalen Menge $S = S_n(\text{opt})$.

Musterlösung zu Uebung 4.2 Sei A ein polynomieller Approximationsalgorithmus für das Rucksackproblem mit additivem Approximationsfaktor $k \in \mathbf{N}$. Wir werden mit Hilfe von A einen exakten polynomiellen Algorithmus für das Rucksackproblem beschreiben. Dieses wird die Behauptung beweisen.
Sei eine Instanz I des Rucksackproblems gegeben durch die Werte $w_1, \ldots, w_n$, die Gewichte $g_1, \ldots, g_n \in \mathbf{N}$ und die Gewichtsschranke b. Konstruiere eine neue Instanz I', indem die Werte w_i durch $(k+1)w_i$, für alle i ersetzt werden. Es gilt $\text{opt}(I') = (k+1)\text{opt}(I)$, und eine Teilmenge $S \subseteq \{1, \ldots, n\}$, die eine optimale Lösung für I' liefert, liefert auch eine optimale Lösung für I.
Liefert A für I' die Lösung S mit Gesamtwert W, so gilt $\text{opt}(I') - W \leq k$. Aber alle Werte in I' sind ganzzahlige Vielfache von $k+1$, damit ist auch $\text{opt}(I')$ ein ganzzahliges Vielfaches von $k+1$. Aus $\text{opt}(I') - W \leq k$ folgt daher $\text{opt}(I') = W$ und eine optimale Teilmenge S für I' ist gefunden. Wie oben schon gesagt, liefert S dann auch eine optimale Lösung für I.
Da I' leicht aus I zu konstruieren ist, liefert dieses einen exakten polynomiellen Algorithmus für das Rucksackproblem.

Musterlösung zu Uebung 4.3 Angenommen, es gibt doch ein Approximationsschema. Dann können wir das Problem mit einem Algorithmus A bis auf einen

Faktor $1+1/(B+1)$ (Minimierunsproblem) bzw. $1-1/(B+1)$ (Maximierungsproblem) approximieren. Das bedeutet, für jede Instanz eines Minimierungsproblems gilt

$$\operatorname{opt}(I) \leq A(I) \leq \left(1 + \frac{1}{B+1}\right) \operatorname{opt}(I) < \operatorname{opt}(I) + 1,$$

bei Maximierungsproblemen erhalten wir

$$\operatorname{opt}(I) \geq A(I) \geq \left(1 - \frac{1}{B+1}\right) \operatorname{opt}(I) > \operatorname{opt}(I) - 1.$$

In beiden Fällen folgt aus der Ganzzahligkeit von $A(I)$, dass $A(I) = \operatorname{opt}(I)$ gelten muss, A ist also ein Algorithmus, der das Problem in polynomieller Zeit exakt löst. Im Fall von P$\neq$NP kann das aber nicht sein, also erhalten wir einen Widerspruch.

Musterlösung zu Uebung 5.1 Der behauptete Partitionssatz ist ganz einfach zu beweisen, man muss nur die Definitionen verwenden, De Morgan's Regeln sowie die Tatsache, dass für disjunkte Ereignisse $A_1, \ldots, A_m$ stets gilt

$$\sum_{i=1}^{m} \operatorname{prob}(A_i) = \operatorname{prob}(\bigcup_{i=1}^{m} A_i).$$

$$\begin{aligned}
E(X|B) &= \sum_x x \operatorname{prob}(\{X = x\}|B) \\
&= \sum_x x \operatorname{prob}(\{X = x\} \cap B) / \operatorname{prob}(B) \\
&= \sum_x x \operatorname{prob}\left(\{X = x\} \cap \bigcup_{i=1}^{n} B_i\right) / \operatorname{prob}(B) \\
&= \sum_x x \operatorname{prob}\left(\bigcup_{i=1}^{n} (\{X = x\} \cap B_i)\right) / \operatorname{prob}(B) \\
&= \sum_x x \sum_{i=1}^{n} \operatorname{prob}(\{X = x\} \cap B_i) / \operatorname{prob}(B) \\
&= \sum_x x \sum_{i=1}^{n} \operatorname{prob}(\{X = x\}|B_i) \operatorname{prob}(B_i) / \operatorname{prob}(B) \\
&= \sum_x x \sum_{i=1}^{n} \operatorname{prob}(\{X = x\}|B_i) \operatorname{prob}(B_i|B) \\
&= \sum_{i=1}^{n} \sum_x x \operatorname{prob}(\{X = x\}|B_i) \operatorname{prob}(B_i|B)
\end{aligned}$$

$$= \sum_{i=1}^{n} E(X|B_i)\,\mathrm{prob}(B_i|B).$$

Musterlösung zu Uebung 5.2 Man überlegt sich zunächst, dass der bedingte Erwartungswert genau wie der 'normale' Erwartungswert linear ist. Dann können wir für jedes Ereignis B

$$E(X|B) = \sum_{\ell=1}^{n} E(X_\ell|B)$$

schreiben, wobei X_ℓ wieder die 0-1-Indikatorvariable ist, die angibt, ob die ℓ-te Klausel erfüllt ist. Man erhält dann nach Definition

$$E(X_\ell|B) = \mathrm{prob}(X_\ell = 1|B).$$

Was ist also die Wahrscheinlichkeit, dass Klausel C_ℓ erfüllt ist, unter der Voraussetzung $B = \{x_j = b_j, j = 1, \ldots, i\}$? Dazu müssen wir uns die Klausel anschauen. Enthält sie ein bereits mit `true` belegtes Literal x_j oder $\neg x_j, j \leq i$, so ist diese bedingte Wahrscheinlichkeit 1. Die mit `false` belegten Literale können ignoriert werden. Nun existieren möglicherweise noch 'freie' Literale x_j oder $\neg x_j, j > i$, sagen wir f_ℓ viele. Dann ist die bedingte Wahrscheinlichkeit, dass C_ℓ erfüllt wird, genau

$$1 - \frac{1}{2^{f_\ell}},$$

nach dem gleichen Argument wie in der Vorlesung. Nun muss nur noch über diese bedingten Wahrscheinlichkeiten aufsummiert werden, um den gesuchten bedingten Erwartungswert zu erhalten. Das geht offenbar in polynomieller Zeit.

Musterlösung zu Uebung 5.3 Betrachte einen Graphen $G = (V, E)$, die Knoten seien mit $v_1, \ldots, v_m$ durchnumeriert. Der folgende Algorithmus berechnet einen zufälligen Schnitt in G.

RandomCut:

```
S := ∅
FOR i := 1 TO n DO
    wähle ein Zufallsbit z ∈ {0,1}
    IF z = 1 THEN
        S := S ∪ {v_i}
    END
END
```

Zu zeigen ist, dass dieser Algorithmus erwartete Güte 1/2 hat. Dazu betrachten wir die Zufallsvariable X für die Anzahl der Schnittkanten zwischen S und

$V \setminus S$. Seien die Kanten mit $e_1, \ldots, e_m$ durchnumeriert. Dann definieren wir die Zufallsvariablen

$$X_i := \begin{cases} 1, & \text{falls } e_i \text{ Schnittkante bzgl. } S \\ 0, & \text{sonst} \end{cases} .$$

Es gilt $X = \sum_{i=1}^m X_i$, und damit auch

$$E(X) = \sum_{i=1}^m E(X_i) = \sum_{i=1}^m \text{prob}(X_i = 1).$$

Was ist nun die Wahrscheinlichkeit, dass eine feste Kante $e_i = \{v_{i1}, v_{i2}\}$ eine Schnittkante ist? Dies ist genau dann der Fall, wenn $v_{i1} \in S$ und $v_{i2} \notin S$ oder $v_{i1} \notin S$ und $v_{i2} \in S$ gilt. Dies sind zwei disjunkte Ereignisse, jedes von ihnen ist der Schnitt unabhängiger Ereignisse. Also gilt

$$\begin{aligned} \text{prob}(X_i = 1) &= \text{prob}(v_{i1} \in S)\,\text{prob}(v_{i2} \notin S) + \text{prob}(v_{i1} \notin S)\,\text{prob}(v_{i2} \in S) \\ &= \frac{1}{2} \cdot \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{2}. \end{aligned}$$

Es folgt, dass die erwartete Anzahl von Schnittkanten genau

$$\frac{1}{2}m \geq \frac{1}{2}\text{opt}$$

beträgt, wir haben also eine erwartete Güte von $1/2$, wie gefordert.

Die Derandomisierung dieses Algorithmus geht wieder mit bedingten Erwartungswerten. Der Ereignisraum besteht aus allen möglichen Schnitten S. $B(i, T)$ bezeichne das Ereignis $\{S \mid S \cap \{v_1, \ldots, v_i\} = T\}$.

DetCut:

```
T := ∅
FOR i := 1 TO n DO
    berechne E_0 := E(X|B(i,T))
    berechne E_1 := E(X|B(i,T ∪ {v_i}))
    IF E_1 ≥ E_0 THEN
        T := T ∪ {v_i}
    END
    Gib den Schnitt T aus
END
```

Bezeichne T_i die Menge T nach der i-ten Iteration, $i = 0, \ldots, n$. Dann gilt $T_0 = \emptyset$ und $E(X) = E(X|B(0, T_0))$. Ferner ist $E(X|B(n, T_n))$ genau die Grösse des von DetCut berechneten Schnittes. Wenn wir also zeigen können, dass

$$E(X|B(i, T_i)) \geq E(X|B(i-1, T_{i-1}))$$

gilt, so folgt, dass der Schnitt T mindestens $E(X)$ Kanten schneidet, also mindestens $m/2$. DetCut hat dann Approximationsgüte $1/2$.

Dazu können wir mit dem Partitionssatz argumentieren. Das Ereignis $B(i-1, T_{i-1})$ ist die disjunkte Vereinigung der Ereignisse $B(i, T_{i-1})$ und $B(i, T_{i-1} \cup \{v_i\})$. Die bedingte Wahrscheinlichkeit beider Ereignisse unter der Voraussetzung $B(i-1, T_{i-1})$ ist jeweils $1/2$, weil v_i mit gleicher Wahrscheinlichkeit in S ist wie in $V \setminus S$. Seien E_0 und E_1 die bedingten Erwartungswerte, die in Iteration i berechnet werden. Dann gilt

$$\begin{aligned}
E(X|B(i-1,T_{i-1})) &= \frac{1}{2}E(X|B(i,T_{i-1})) + \frac{1}{2}E(X|B(i,T_{i-1}\cup\{v_i\})) \\
&= \frac{1}{2}E_0 + \frac{1}{2}E_1 \\
&\leq \max(E_0, E_1) \\
&= E(X|B(i,T_i)).
\end{aligned}$$

Wie berechnet man die bedingten Erwartungen E_0 und E_1 in jeder Iteration? Wie oben ergeben sich diese durch Summation der bedingten Kantenwahrscheinlichkeiten. Es genügt also, für eine gegebene Kante e die bedingte Wahrscheinlichkeit dafür zu berechnen, dass diese Kante eine Schnittkante ist, unter der Voraussetzung $B(i, R)$, wobei $R = T_{i-1}$ oder $R = T_{i-1} \cup \{v_i\}$. Falls beide Knoten der Kante e in $\{v_1, \ldots, v_i\}$ liegen, so ist mittels R bereits festgelegt, ob e Schnittkante ist, die bedingte Wahrscheinlichkeit ist also entweder 0 oder 1. Andernfalls ist mindestens ein Knoten noch 'frei', und wie oben kann man sich leicht überlegen, dass die bedingte Wahrscheinlichkeit dann genau $1/2$ beträgt.

Aus diesen Überlegungen folgt auch, dass es für den Test '$E_1 \geq E_0$' nicht nötig ist, sämtliche Kanten zu betrachten. In Schritt i genügt es, alle zu v_i inzidenten Kanten zu behandeln. Denn für jede Kante e, die v_i nicht enthält, gilt

$$\text{prob}(e \text{ Schnittkante}|B(i,T_{i-1})) = \text{prob}(e \text{ Schnittkante}|B(i,T_{i-1}\cup\{v_i\})),$$

der Beitrag dieser bedingten Kantenwahrscheinlichkeit ist also in E_0 und E_1 gleich gross und kann demzufolge ignoriert werden. Das gleiche gilt für Kanten $e = \{v_i, v_j\}$ mit $j > i$, denn deren bedingte Wahrscheinlichkeit, Schnittkante zu sein, ist genau $1/2$ (weil v_j sich erst später für S oder $V \setminus S$ 'entscheidet'.
Es bleiben also nur noch Kanten $e = \{v_i, v_j\}$ mit $j < i$ zur Betrachtung übrig. Für diese gilt nun

$$\text{prob}(e \text{ Schnittkante}|B(i,T_{i-1})) = \begin{cases} 1, & \text{falls } v_j \in T_{i-1} \\ 0, & \text{sonst} \end{cases},$$

denn das Ereignis $B(i, T_{i-1})$ impliziert $v_i \notin S$, also wird e Schnittkante genau dann, wenn $v_j \in S$, und das ist genau dann der Fall, wenn $v_j \in T_{i-1}$ gilt. Analog gilt

$$\text{prob}(e \text{ Schnittkante}|B(i,T_{i-1}\cup\{v_j\})) = \begin{cases} 1, & \text{falls } v_j \notin T_{i-1} \\ 0, & \text{sonst} \end{cases}.$$

Es folgt, dass die erwartete Anzahl von Schnittkanten $\{v_i, v_j\}, j < i$ unter der Voraussetzung $B(i, T_{i-1})$ genau die Anzahl der Nachbarn von v_i in T_{i-1} ist, während die erwartete Anzahl unter der Voraussetzung $B(i, T_{i-1})$ gerade $i - 1 - |T_{i-1}|$ beträgt. E_1 ist also mindestens so gross wie E_0 (und v_i landet in T) genau dann, wenn v_i mindestens soviele Nachbarn in $\{v_1, \ldots, v_{i-1}\} \setminus T_{i-1}$ wie in T_{i-1} hat. Diese Aussage führt dann auch zur geforderten Umformulierung von DetCut ohne Bezugnahme auf Wahrscheinlichkeiten und Erwartungswerte.

DetCut:

```
T := ∅
FOR i := 1 TO n DO
    berechne n_0 := |{v_j | {v_i, v_j} ∈ E, v_j ∈ T}|
    berechne n_1 := |{v_j | {v_i, v_j} ∈ E, v_j ∈ {v_1, ..., v_{i-1}} \ T}|
    IF n_1 ≥ n_0 THEN
        T := T ∪ {v_i}
    END
    Gib den Schnitt T aus
END
```

Man kann nun auch sehr einfach sehen, dass diese Version einen Schnitt mit mindestens der Hälfte aller Kanten erzeugt. Dazu muss man nur per Induktion zeigen, dass der durch T definierte Schnitt nach der i-ten Iteration mindestens die Hälfte aller Kanten des von $\{v_1, \ldots, v_i\}$ induzierten Teilgraphen enthält, für alle i.

Musterlösung zu Uebung 6.1 Wir benutzen die Gleichung

$$(1-a)\sum_{i=0}^{k-1} a^i = 1 - a^k.$$

Wenden wir diese mit $a = 1 - x/k$ auf die linke Seite der zu zeigende Ungleichung an, erhalten wir

$$1 - \left(1 - \frac{x}{k}\right)^k = \frac{x}{k}\sum_{i=0}^{k-1}\left(1 - \frac{x}{k}\right)^i \geq x\left[\frac{1}{k}\sum_{i=0}^{k-1}\left(1 - \frac{1}{k}\right)^i\right].$$

Verwenden wir die obige Gleichung noch einmal mit $a = 1 - 1/k$, erhalten wir die behauptete Ungleichung.

Musterlösung zu Uebung 6.2 Sei $V = \{v_1, \ldots, v_n\}, S = \{S_1, \ldots, S_m\}$ eine Instanz von Node Cover (NC). Wir konstruieren daraus eine Instanz von Set Cover (SC) wie folgt. Als Grundmenge nehmen wir $S = \{S_1, \ldots, S_m\}$. Als Menge

W von Teilmengen von S nehmen wir $W = \{W_1, \ldots, W_n\}$, wobei $W_i = \{S_j \in S | v_i \in S_j\}$. Für jedes $v_i \in V$ haben wir also genau ein W_i in W, bestehend nämlich aus den S_j, die v_i enthalten. Es gilt

$$v_i \in S_j \iff S_j \in W_i.$$

Wir behaupten nun, dass T ein zulässiges node cover für das Paar (V, S) ist genau dann, wenn T ein zulässiges set cover für das Paar (S, W) ist. Dieses ergibt sich aus der folgende Kette von äquivalenten Aussagen.

$$\begin{aligned} & T \text{ ist node cover für } (V,S) \\ \iff & \forall v_i \in V \exists S_j \in S : v_i \in S_j \\ \iff & \forall W_i \in W \exists S_j \in S : S_j \in W_i \\ \iff & T \text{ ist set cover für } (S,W). \end{aligned}$$

Insbesondere ergibt sich hieraus die Identität der Grösse der optimalen Lösungen. Denn zu einer Lösung einer Instanz von SC (NC) kann ja eine gleich grosse Lösung einer Instanz von NC (SC) gefunden werden.

Musterlösung zu Uebung 6.3 Für jede Teilmenge S_j enthält das relaxierte Set Cover die Ungleichung

$$\sum_{x_i : v_i \in S_j} x_i \geq 1.$$

Da $|S_j| \leq f$ muss in der zulässigen Lösung $p_1, \ldots, p_n$ mindestens ein p_i existieren mit $v_i \in S_j$ und $p_i \geq 1/f$. Dann aber wird v_i in der Menge H enthalten sein. H ist also ein zulässiges set cover.
Setze $y_i = 1$, falls v_i in H und $y_i = 0$ sonst, $i = 1, \ldots, n$. Es gilt $\sum_{i=1}^{n} y_i = |H|$. Ausserdem gilt $y_i/p_i \leq f$. Ist $y_i = 0$, so gilt diese Ungleichung offensichtlich. Ist hingegen $y_i = 1$, so $v_i \in H$ und $p_i > 1/f$. Dann gilt $y_i/p_i = f$. Schliesslich erhalten wir

$$|H| = \sum_{i=1}^{n} y_i \leq \sum_{i=1}^{n} f p_i = f \sum_{i=1}^{n} p_i \leq f \mathrm{opt},$$

denn die optimale Lösung des relaxierten Set Cover ist sicherlich besser als die des eigentlichen Problems Set Cover.

Musterlösung zu Uebung 6.4 Wir schauen uns die Dimensionen der Unterräume U_1 und U_2 des $\mathbf{R}^n$ an. Wenn in $\tilde{X}$ mehr als m Werte von Null verschieden sind, so hat U_1 mehr als m 'freie' Koordinaten, also

$$dim(U_1) > m.$$

U_2 ist durch m lineare Einschränkungen definiert. Geben wir diese nacheinander zu $\mathbf{R}^n$ hinzu, verringert sich die Dimension jedesmal höchstens um 1, so dass auf jeden Fall noch

$$dim(U_2) \geq n - m$$

gilt. Aus der linearen Algebra wissen wir, dass

$$\dim(U_1 \cap U_2) = \dim(U_1) + \dim(U_2) - \dim(U_1 \oplus U_2)$$

gilt, wobei $U_1 \oplus U_2$ die direkte Summe von U_1 und U_2 ist. Da letztere höchstens Dimension n haben kann, erhalten wir

$$\dim(U_1 \cap U_2) \geq 1,$$

$U_1 \cap U_2$ ist also mindestens eine Gerade g. Da g nach Definition den Lösungsvektor

$$\tilde{x} = (\tilde{x}_1, \ldots, \tilde{x}_n)$$

enthält, kann man g in der Form

$$\{x = \tilde{x} + \lambda(\tilde{y} - \tilde{x}) \mid \lambda \in \mathbf{R}\}$$

schreiben, wobei $\tilde{y} \neq \tilde{x}$ ein weiterer Punkt in $U_1 \cap U_2$ ist, der sich also in mindestens einer Koordinate von $\tilde{x}$ unterscheidet, sagen wir $\tilde{x}_i \neq \tilde{y}_i$. Ferner muss dann $\tilde{x}_i \neq 0$ gelten, sonst wäre $\tilde{y}_i = 0$ nach Definition von U_1 und wir hätten doch $\tilde{x}_i = \tilde{y}_i$. Wählen wir nun

$$\lambda = -\frac{\tilde{x}_i}{\tilde{y}_i - \tilde{x}_i},$$

so erhalten wir einen Vektor $x \in U_1 \cap U_2$ mit $x_i = 0$. Dieser hat mehr Nulleinträge als $\tilde{x}$ und ist ebenfalls eine optimale Lösung, falls er zulässig ist. Warum? Weil die ganze Gerade g aus optimalen Lösungen bestehen muss, denn gäbe es eine nichtoptimale Lösung auf g, so gäbe es auch eine Lösung mit noch besserem Zielfunktionswert als $\tilde{x}$ (kann man sich mit leicht ähnlich wie eben überlegen), was ein Widerspruch zur Optimalität von $\tilde{x}$ ist. Es kann nun sein, dass x unzulässig ist – das kann aber nur daran liegen, dass wir 'auf dem Weg' von $\tilde{x}$ nach x entlang g das zulässige Gebiet verlassen haben. Das bedeutet aber ebenfalls, dass zwischendurch eine Koordinate x_j Null geworden ist, und nehmen wir diesen Punkt als x, so erhalten wir auch eine zulässige (und damit optimale) Lösung mit mehr Nullen als $\tilde{x}$.

Musterlösung zu Uebung 6.5 Da $T = (V, E)$ eine Baum ist, gibt es für jedes der Paare (s_i, t_i) genau einen Pfad, der s_i und t_i miteinander verbindet. Sein E_i die Menge der Kanten, die auf dem Pfad von s_i nach t_i liegen. Diese Mengen können mit Hilfe von depth-first search in polynomieller Zeit berechnet werden.

Eine Teilmenge $F \subseteq E$ ist genau dann eine zulässige Lösung für das Problem Tree-Multi-Cut ist, wenn $F \cup E_i \neq \emptyset$ für alle i. Wir führen nun eine Variable x_e für jede Kante $e \in E$ ein. $x_e = 1$ soll bedeuten, dass e in die Menge F aufgenommen wird. Dann kann Tree-Multi-Cut wie folgt als ganzzahliges lineares Programm geschrieben werden.

$$
\begin{array}{lll}
\text{(Tree-Multi-Cut)} & \text{minimiere} & \sum_{e \in E} x_e \\
 & \text{unter} & \\
 & & \sum_{e \in E_i} x_e \geq 1, \quad j = 1, \ldots, k \\
 & & x_e \in \{0,1\}, \quad e \in E.
\end{array}
$$

Wie gefordert hat dieses Programm polynomielle Grösse.

Musterlösung zu Uebung 6.6 Wir müssen wie im Fall eines Minimierungsproblems zeigen, dass die Wahrscheinlichkeit, dass die Güte der von A berechneten Lösung stark von der erwarteten Güte abweicht, nicht zu gross ist.
Im Minimierungsfall konnten wir dazu die Markov-Ungleichung verwenden, die uns folgende Abschätzung geliefert hat (X ist die Zufallsvariable für den Wert der Lösung bei fester Instanz I).

$$\text{prob}(X \geq (1+\varepsilon)E(X)) \leq \frac{1}{1+\varepsilon}. \tag{9.1}$$

Die Idee war dann, den Algorithmus A wiederholt laufen zu lassen, und als Ausgabe die beste jemals berechnete Lösung zu nehmen. Lassen wir A k-mal laufen, erhalten wir mit Wahrscheinlichkeit mindestens

$$1 - \left(\frac{1}{1+\varepsilon}\right)^k$$

eine Lösung mit Wert höchstens $(1+\varepsilon)E(X)$. Durch geeignete Wahl von k (abhängig von ε) können wir diese Wahrscheinlichkeit beliebig nahe an 1 bringen. Um das gleiche im Maximierungsfall tun zu können, benötigen wir eine vernünftige obere Schranke für

$$\text{prob}(X \leq (1-\varepsilon)E(X)).$$

Im allgemeinen existiert dafür aber keine Schranke analog zu (9.1), die nur von ε abhängt. Um dies zu sehen, betrachte die Zufallsvariable

$$Y : \{1, \ldots, n\} \mapsto \mathbf{R} \ \text{ mit } Y(i) = 0 \text{ für } i < n \text{ und } Y(n) = 1.$$

Sind alle $i \in \{1, \ldots, n\}$ gleich wahrscheinlich, so gilt $E(Y) = 1/n$ und

$$\text{prob}(Y \leq (1-\varepsilon)E(Y)) = \frac{n-1}{n},$$

für beliebiges $\varepsilon \in [0,1)$. Das heisst, die Wahrscheinlichkeit kann nicht durch eine Funktion von ε von 1 'wegbeschränkt' werden. Der Grund besteht darin (wie wir gleich sehen werden), dass es keine konstante obere Schranke für das Verhältnis $Y/E(Y)$ gibt, der Erwartungswert also viel kleiner sein kann als der maximale Wert von Y.
Hier nun ein Lemma, das man als 'umgekehrte Markov-Ungleichung' bezeichnen könnte.

Lemma 9.0.3 *Sei X eine nichtnegative Zufallsvariable, wobei $X(\omega)/E(X) \leq \Delta$ für alle ω aus dem zugrundeliegenden Wahrscheinlichkeitsraum gelte. Dann gilt*

$$\text{prob}(X \leq (1-\varepsilon)E(X)) \leq 1 - \frac{\varepsilon}{\Delta}.$$

Beweis: Sei

$$p := \text{prob}(X \leq (1-\varepsilon)E(X)).$$

Dann gilt

$$E(X) \leq p(1-\varepsilon)E(X) + (1-p)M,$$

wobei $M := \sup_\omega X(\omega)$, denn mit Wahrscheinlichkeit p hat X höchstens Wert $(1-\varepsilon)E(X)$, mit Wahrscheinlichkeit $1-p$ höchstens Wert M. Weiter gilt dann

$$E(X) \leq (1-\varepsilon)E(X) + (1-p)M,$$

was äquivalent zu

$$p \leq 1 - \frac{\varepsilon E(X)}{M}$$

ist. Aus $X(\omega)/E(X) \leq \Delta$ für alle ω folgt nun $E(X)/M \geq \frac{1}{\Delta}$, und das Lemma folgt. □

Angewendet auf unsere Zufallsvariable X (Güte der von A berechneten Lösung) gilt $X \leq \text{opt}(I)$ und $E(X) \geq \delta\text{opt}(I)$, folglich

$$\frac{X}{E(X)} \leq \frac{1}{\delta}.$$

Wir bekommen dann

$$\text{prob}(X \leq (1-\varepsilon)E(X)) \leq 1 - \varepsilon\delta,$$

was eine Schranke in ε ist, da δ als konstant angenommen war. Nun können wir durch wiederholtes Laufenlassen von A wie im Minimierungsfall mit beliebig hoher Wahrscheinlichkeit eine Lösung bekommen, die beliebig nahe am Erwartungswert liegt.

Musterlösung zu Uebung 7.1 Zunächst erinnern wir uns dass λ genau dann ein Eigenwert von M ist, wenn $\det(M - \lambda I) = 0$ gilt, wobei I die Einheitsmatrix ist. Warum? Sei λ ein Eigenwert, dann gibt es einen nichttrivialen Vektor v mit

$$Mv = \lambda v,$$

was äquivalent ist zu $(M - \lambda I)v = 0$. Das heisst, die Matrix $M - \lambda I$ ist singulär, hat also Determinante 0. Umgekehrt können wir genauso schliessen. Wenn $\det(M - \lambda I) = 0$, so gibt es eine nichttriviale Lösung v zum Gleichungssytem $(M - \lambda I)v = 0$, und dieses v ist dann ein Eigenvektor zum Eigenwert λ.
Hier ist das semidefinite Programm zur Berechnung des kleinsten Eigenwertes einer positiv semidefiniten Matrix M. Es hat Variablen λ und a_{ij}, zusammengefasst in einer Matrix A.

$$\begin{array}{lll} \text{(SDP)} & \text{maximiere} & \lambda \\ & \text{unter} & \\ & & A \text{ positiv semidefinit} \\ & & A = M - \lambda I. \end{array}$$

Die Bedingungen $A = M - \lambda I$ sind offenbar lineare Bedingungen in λ und den Variablen a_{ij}. Zunächst überlegen wir uns, dass dieses Problem überhaupt eine maximale Lösung besitzt, λ also nicht beliebig gross werden kann. Dazu betrachten wir einen Vektor $x \neq 0$ und den Wert

$$x^T A x = x^T M x - \lambda \|x\|^2.$$

Wählen wir λ nur gross genug, so gilt $x^T A x < 0$, und A ist nicht mehr positiv semidefinit. λ ist also beschränkt. Nun gilt

Lemma 9.0.4 *Sei λ^* die optimale Lösung von SDP. Dann ist λ^* der kleinste Eigenwert von M.*

Beweis: Wir haben folgende einfache Tatsache. μ ist genau dann ein Eigenwert von M, wenn $\mu - \lambda$ ein Eigenwert vom $M - \lambda I$ ist. Dies folgt sofort aus

$$Mv = \mu v \Leftrightarrow (M - \lambda I)v = (\mu - \lambda)v.$$

Ferner ist λ^* das maximale λ, so dass alle Eigenwerte von $M - \lambda I$ nichtnegativ sind (denn dies ist eine äquivalente Charakterisierung von symmetrischen, positiv semidefiniten Matrizen).
Das heisst, λ^* ist das maximale λ, so dass $\mu - \lambda \geq 0$ für alle Eigenwerte μ von M gilt. Das heisst aber, dass λ^* der kleinste Eigenwert von M ist. $\square$

Musterlösung zu Uebung 7.2 Wir verwenden die Charakterisierung, dass eine Matrix M genau dann positiv semidefinit ist, wenn für alle Vektoren x die Eigenschaft

$$x^T M x \geq 0$$

gilt. Sei nun $M = (1 - \lambda)M_1 + \lambda M_2$. Dann können wir wie folgt schliessen.

$$\begin{aligned} x^T M x &= x^T \left((1 - \lambda)M_1 + \lambda M_2\right) x^T \\ &= (1 - \lambda)x^T M_1 x + \lambda x^T M_2 x \geq 0, \end{aligned}$$

weil sowohl $(1 - \lambda), \lambda$ als auch $x^T M_1 x, x^T M_2 x$ nach Voraussetzung nichtnegativ sind.

Musterlösung zu Uebung 7.3 Im Bereich $0 < t \leq \pi/2$ gilt

$$\frac{2t}{\pi(1 - \cos t)} \geq 1.$$

Wir können uns also auf den Bereich $\pi/2 < t < \pi$ beschränken. In diesem Bereich ist die Funktion $2t/(\pi(1 - \cos t))$ differenzierbar. Die Ableitung ist gegeben durch

$$\frac{1 - \cos t - t \sin t}{(1 - \cos t)^2}.$$

Die Ableitung besitzt im Bereich $\pi/2 < t < \pi$ nur eine Nullstelle t_0, die gegeben ist durch $\cos t_0 + t_0 \sin t_0 = 1$. Numerisch ergibt sich für t_0 der Wert $t_0 = 2,331122....$ Da auch die Ableitung im Bereich $\pi/2 < t < \pi$ stetig ist, kann man durch Einsetzen von Werten grösser und kleiner als t_0 verifizieren, dass t_0 ein Minimum ist. Nun gilt

$$\frac{2t_0}{\pi(1 - \cos t_0)} \geq 0,87856.$$

Musterlösung zu Uebung 7.4 Für den arccos gilt

$$\pi - \arccos(z) = \arccos(-z) \text{ oder } 1 - \arccos(z)/\pi = \arccos(-z)/\pi.$$

Wir wissen bereits aus der vorangegangenen Uebung, dass $\arccos(z)/\pi \geq \alpha(1 - z)/2$ für alle $-1 < z < 1$. Ersetzen wir z durch $-z$ erhalten wir

$$1 - \arccos(z)/\pi = \arccos(-z)/\pi \geq \alpha \frac{1 + z}{2}.$$

Musterlösung zu Uebung 7.5 Wir konstruieren k Vektoren mit den gewünschten Eigenschaften im $\mathbf{R}^k$, damit liegen sie insbesondere im $\mathbf{R}^n$. Die Idee ist wie im 3-dimensionalen Fall aus der Vorlesung, die Vektoren in Form eines regelmässigen Simplex zu wählen. Betrachte dazu die k Einheitsvektoren $e_1, \ldots, e_k$ und ihren Schwerpunkt $c = (1/k, \ldots, 1/k)$ und definiere

$$v_i = e_i - c.$$

Bis auf Skalierung sind die v_i bereits die gesuchten Vektoren u_i, die wir mittels

$$u_i = \frac{v_i}{\|v_i\|}$$

erhalten. Nun bleiben nur noch die geforderten Eigenschaften nachzurechnen. Die u_i sind per Definition Einheitsvektoren, und es gilt

$$u_i \cdot u_j = \frac{1}{\|v_i\|\|v_j\|} v_i \cdot v_j.$$

Nun überlegt man sich leicht, dass

$$\begin{aligned} \|v_i\| &= \sqrt{1 - \frac{1}{k}} = \sqrt{\frac{k-1}{k}} \quad \forall i, \\ v_i \cdot v_j &= -\frac{1}{k} \quad \forall i \neq j \end{aligned}$$

gilt, woraus dann

$$u_i \cdot u_j = \frac{k}{k-1}\left(-\frac{1}{k}\right) = -\frac{1}{k-1}$$

folgt, für $i \neq j$.

Musterlösung zu Uebung 7.6 Wir haben die eine Richtung schon gezeigt: wenn G 2-färbbar ist, so ist G vektor-2-färbbar. Um die andere Richtung zu sehen, betrachte eine Vektor-2-Färbung, definiert durch Vektoren $u_1, \ldots, u_n$ mit

$$u_i \cdot u_j \leq -\frac{1}{2-1} = -1$$

für alle Kanten $\{i, j\}$. Das bedeutet aber, es muss $u_i = -u_j$ gelten, sofern $\{i, j\}$ eine Kante ist, denn Skalaraprodukt -1 heisst ja, dass der von den Vektoren eingeschlossene Winkel gerade π ist. Betrachte nun eine Zusammenhangskomponente Γ des Graphen, einen Knoten $v = v_i \in \Gamma$ sowie seinen Vektor u_i. Dann muss jeder Nachbar von v Vektor $-u_i$ haben, deren Nachbarn wiederum Vektor $-(-u_i) = u_i$ usw. Das heisst, alle Knoten in Γ werden so auf zwei Vektoren $u_i, -u_i$ abgebildet, dass benachbarte Knoten verschiedene Vektoren haben. Dann definieren diese Vektoren aber eine echte 2-Färbung auf Γ. Da dieses Argument für alle Zusammenhangskomponenten gilt, ist der Graph insgesamt 2-färbbar.

Musterlösung zu Uebung 7.7 Für eine Lösung $\hat{x}_i, i = 1, \dots, n$, von MIS mit maximalem Wert sei $S = \{i \in V | \hat{x}_i = 1\}$. Sei $E' \subseteq E$ die Menge der Kanten zwischen Knoten in S. Der Wert der Lösung $\hat{x}_i$ ist nun $|S| - |E'|$. Weiter können wir durch Entfernen von höchstens $|E'|$ Knoten aus S eine unabhängige Menge U der Grösse mindestens $|S| - |E'|$ erzeugen. Daher ist das Optimum $\text{opt}^* = |S| - |E'|$ von MIS höchstens so gross wie die Grösse opt einer maximalen unabhängigen Menge in G.
Andererseits können wir für jede unabhängige Menge U in G die zulässige Lösung $\hat{x}_i = 1 \Leftrightarrow i \in U$ von MIS erzeugen. Der Wert einer solchen Lösung ist $|U|$. Daher gilt auch $\text{opt} \leq \text{opt}^*$ und somit $\text{opt} = \text{opt}^*$.

Musterlösung zu Uebung 7.8 Es genügt zu zeigen, dass es für Relaxiertes MIS eine optimale Lösung $\hat{x}_i$ gibt mit $\hat{x}_i \in \{0, 1\}$ für alle i. Um dieses zu zeigen, genügt es wiederum, zu jeder optimalen Lösung $\hat{x}_i$ mit $0 < \hat{x}_{i_0} < 1$ für ein $i_0 \in \{1, \dots, n\}$, eine Lösung zu konstruieren, die mindestens genauso grossen Wert hat, wo aber die Variable x_{i_0} auf 0 oder 1 gesetzt ist. Die $\hat{x}_j, j \neq i_0$, bleiben unverändert.
Sei also $\hat{x}_i$ eine optimale Lösung mit $0 < \hat{x}_{i_0} < 1$. Sei t ein Parameter. Wir betrachten den Wert einer Lösung, bei der $\hat{x}_{i_0}$ durch $\hat{x}_{i_0} + t$ ersetzt wird, die restlichen Werte aber unverändert bleiben. Der Wert einer solchen Lösung ist gegeben durch

$$\sum_{i=1}^{n} \hat{x}_i - \sum_{\{i,j\} \in E} \hat{x}_i \hat{x}_j + t \left(1 - \sum_{\{i_0,j\} \in E} \hat{x}_j \right).$$

Ist $\Delta = \left(1 - \sum_{\{i_0,j\} \in E} \hat{x}_j \right) = 0$, so können wir den Wert von x_{i_0} auf 0 oder 1 setzen, ohne den Wert der Lösung zu ändern.
Ist $\Delta \neq 0$, so können die $\hat{x}_{i_0}$ nicht optimal gewesen sein. Denn ist $\Delta > 0$, so wird durch Wahl von $t > 0$ der Wert der Lösung erhöht. Analog, kann bei $\Delta < 0$ durch $t < 0$ der Wert der Lösung erhöht werden.

Musterlösung zu Uebung 8.1 Wir zeigen

(1) $i(G_1 \times G_2) \geq i(G_1) i(G_2)$

(2) $i(G_1 \times G_2) \leq i(G_1) i(G_2)$

zu (1) Sei U_1 eine unabhängige Menge in G_1 und U_2 eine unabhängige Menge in G_2. Es genügt zu zeigen, dass dann $U_1 \times U_2$ eine unabhängige Menge in $G_1 \times G_2$ ist. Wären nun $(v_1, u_1), (v_2, u_2) \in U_1 \times U_2$ in $G_1 \times G_2$ durch eine Kante verbunden, so hiesse dieses, dass entweder v_1, v_2 in G_1 durch eine Kante verbunden sind, oder dass v_1, v_2 und u_1, u_2 in G_2 durch eine

Kante verbunden sind. Der erste Fall widerspricht der Annahme, dass U_1 unabhängige Menge ist. Der zweite Fall widerspricht der Tatsache, dass U_2 unabhängige Menge ist.

zu (2) Sei nun U eine unabhängige Menge in $G_1 \times G_2$ mit $|U| = i(G_1 \times G_2)$. Sei U_1 die Menge der Knoten in G_1, die als erste Komponente eines Knoten in U auftauchen. U_1 muss eine unabhängige Menge in G_1 sein, daher $|U_1| \leq i(G_1)$. Für jedes $x \in U_1$ sei U_x die Menge der Knoten $y \in G_2$ mit $(x,y) \in U$. Jedes U_x muss eine unabhängige Menge in G_2 sein. Daher gilt für alle $x \in U_1$, dass $|U_x| \leq i(G_2)$. Nun gilt

$$|U| = \sum_{x \in U_1} |U_x| \leq \sum_{x \in U_1} i(G_2) \leq i(G_1) i(G_2).$$

Musterlösung zu Uebung 8.2 Aus der Lösung zu Uebung 8.1 erhalten wir einen polynomiellen Algorithmus A mit folgenden Eigenschaften: Gegeben $G^2 = G \times G$ für einen Graphen G und eine unabhängige Menge U' der Grösse k in G^2, so berechnet A eine unabhängige Menge U der Grösse mindestens $\lceil \sqrt{k} \rceil$. A berechnet zunächst die Menge U_1 aller Knoten in G, die erste Komponente eines Elementes in U' sind. Ist diese Menge bereits der Grösse mindestens $\lceil \sqrt{k} \rceil$, so ist U_1 die Ausgabe. Sonst berechnet der Algorithmus für alle $x \in U_1$ die Menge U_x, derjenigen Knoten $y \in G$ mit $(x,y) \in U'$. Nach den Argumenten aus der Lösung zu Uebung 8.1 ist jede dieser Mengen eine unabhängige Menge in G und eine davon muss jetzt mindestens die Grösse $\lceil \sqrt{k} \rceil$ haben.
Sei nun B ein Approximationsalgorithmus für Unabhängige Menge mit Güte $c < 1$ konstant. Sei $\epsilon > 0$ beliebig, wir werden einen Approximationsalgorithmus für Unabhängige Menge konstruieren, der Güte $1 - \epsilon$ hat. Setze

$$l = \lceil \log \left(\frac{\log c}{\log(1-\epsilon)} \right) \rceil.$$

Bilde sukzessive die Graphen $G^2, G^4, \ldots, G^{2^l}$. Da l konstant ist, d.h. unabhängig von der Grösse von G, ist die Grösse all dieser Graphen polynomiell in der Grösse von G. Nach Aufgabe 1.) gilt:

$$i(G^{2^i}) = i(G)^{2^i}, i = 0, \ldots, l.$$

Wir benutzen nun Algorithmus B, um in G^{2^l} eine unabhängige Menge U_l der Grösse mindestens $ci(G^{2^l}) = ci(G)^{2^l}$ zu finden. Durch l-fache Anwendung von Algorithmus A finden wir nun sukzessive unabhängige Mengen U_i in den Graphen $G^{2^i}, i = l-1, \ldots, 0$, der Grösse mindestens

$$c^{1/2^{l-i}} i(G)^{2^i}.$$

Insbesondere gilt für U_0

$$|U_0| \geq c^{1/2^l} i(G).$$

Nach Wahl von l ist

$$c^{1/2^l} > 1 - \epsilon.$$

Damit ist die Güte dieses Algorithmus $1 - \epsilon$ wie gefordert. Nach dem bereits Gesagten ist die Laufzeit des Algorithmus polynomiell.

Index

$(r(n), q(n))$-Verifizierer, 88
$\{0,1\}^*$, 14
k-Färbung, 74
PCP$(r(n), q(n))$, Komplexitätsklasse, 88
3-SAT Problem, 16
3-konjunktive Normalform, 16

Approximationsalgorithmus, 9
Approximationsalgorithmus, probabilitischer, 56
Approximationsalgorithmus, randomisierter, 42
Approximationsfaktor, 9
Approximationsgüte, *siehe* Approximationsfaktor
Approximationsgüte, eines Problems, 67
Approximationsgüte, erwartete, 42
Approximationsschema, 36
Arithmetisches und geometrisches Mittel, Ungleichung, 50

bedingter Erwartungswert, 45
bedingte Wahrscheinlichkeit, 44

CH Algorithmus, 25
Cholesky-Zerlegung, 65
chromatische Zahl, 75
Clique, 20

Derandomisierung, 43
DetSat Algorithmus, 43
Dynamisches Programmieren, 31

Eigenwert, 79
Entscheidungsproblem, 13
ETSP, *siehe* Euklidischer Handlungsreisender
EUKLIDISCHER HANDLUNGSREISENDER Problem, 21
Eulerkreis, 23
ExactKnapsack Algorithmus, 33

ganzzahliges lineares Programm, 47
ganzzahliges quadratisches Programm, 80
GRAPHENFÄRBUNG Problem, 75, 81
Graphenprodukt, 82

Halbfärbung, 77, 78
HAMILTONSCHER KREIS Problem, 22
HANDLUNGSREISENDER Problem, 22

ILP, *siehe* ganzzahliges lineares Programm

JOB-SCHEDULING Problem, 5, 28, 37, 58

lückenerhaltende Reduktion, 84
LI Algorithmus, 11
lineares Programm, 47
LP, *siehe* lineares Programm
LP Schedule Algorithmus, 59
LP-Relaxierung, 47
LS Algorithmus, 6

Makespan, 5
Markov-Ungleichung, 57
MAX-3-SAT Problem, 85, 86
MAX-$\geq k$-SAT Problem, 40
MAX-≤ 2-SAT Problem, 68, 69
MAX-k-FUNCTION-SAT Problem, 89

MAX-CUT Problem, 10, 45, 62
MAX-SAT Problem, 46, 71
MAXIMALE UNABHÄNGIGE MENGE Problem, 80
MIS, *siehe* Maximale Unabhängige Menge
MSB Algorithmus, 22

Next-Fit Algorithmus, 12
Nichtapproximierbarkeit, 81
NODE-COVER Problem, 60
NP, Komplexitätsklasse, 15
NP-schwer, 19
NP-vollständig, 18

Optimierungsproblem, 8

P, Komplexitätsklasse, 17
PCP, 88
Produktgraph, *siehe* Graphenprodukt

randomisiertes Runden, 49
RandomSat Algorithmus, 40
Reduktion, 17
RR Max-SAT Algorithmus, 49
RRSC Algorithmus, 53
RUCKSACKPROBLEM, 31
Rundreise, 21

ScaledKnapsack Algorithmus, 34
SDP, *siehe* semidefinites Programm
SDP-Max-$\leq$ 2-SAT Algorithmus, 70
SDP-Max-Cut Algorithmus, 65
SDP-Max-SAT Algorithmus, 73
SDP-Relaxierung, 62
semidefinites Programm, 64
SET-COVER Problem, 52
SLS Algorithmus, 28
SLS(k) Algorithmus, 37
Sprache, 14
SUBSET-SUM Problem, 48
symmetrisch positiv semidefinite Matrix, 64

TREE-MULTI-CUT Problem, 61
TSP, *siehe* Handlungsreisender

UNABHÄNGIGE MENGE Problem, 13
Unabhängigkeitszahl, 93

Vektor-k-Färbung, 76
vektor-chromatische Zahl, 77
VEKTOR-MAX-$\leq$ 2-SAT Problem, 69
VEKTOR-MAX-CUT Problem, 63
VEKTOR-MAX-SAT Problem, 72
Verifizierer, 14
Vertex-Cover, 20
Vier-Farben-Problem, 74