Karlsruher Institut für Technologie — Prof. Dr. Jörn Müller-Quade
Institut für Theoretische Informatik — Björn Kaidel, Sebastian Schlag, Sascha Witt

11. Übungsblatt zu Algorithmen I im SoSe 2017

http://crypto.iti.kit.edu/index.php?id=799
{bjoern.kaidel,sascha.witt}@kit.edu

Musterlösungen

Aufgabe 1 (*Neues Rucksackproblem,* $0+4+4$ *Punkte*)

Betrachten Sie folgendes Rucksackproblem: Sie haben eine Liste von n Gegenständen mit Volumina $c_1,\ldots,c_n \in \mathbb{N}$ und Nutzwerten $a_1,\ldots,a_n \in \mathbb{N}$ ($n \in \mathbb{N}$). Ihr Rucksack hat eine Kapazität $C \in \mathbb{N}$ und soll so gepackt werden, dass die Summe der Nutzwerte der mitgenommenen Gegenstände maximal ist. Es gelte für $1 \leq i \leq n : c_i \leq C$ (d.h. jeder Gegenstand passt auch wirklich in den Rucksack). Formal ist eine Menge $I \subseteq \{1,\ldots,n\}$ gesucht, die

$$\left\{ \sum_{i \in I} a_i \;\middle|\; \sum_{i \in I} c_i \leq C \right\} \quad \text{maximiert.}$$

a) Wiederholen Sie das in der Vorlesung vorgestellte dynamische Programm, dass in $\mathcal{O}(nC)$ eine solche optimale Menge I berechnet. (Keine Abgabe notwendig)

b) Sei nun *Opt* der maximale Nutzen eines zulässigen Rucksacks. Entwickeln Sie ein dynamisches Programm, das in $\mathcal{O}(n \cdot Opt)$ eine optimale Lösungsmenge I berechnet. Begründen Sie die Laufzeit ihres Programms!

c) Ein Algorithmus heißt ein *fully polynomial time approximation scheme* (FPTAS), wenn er in Zeit $p(n,\frac{1}{\varepsilon})$, wobei n die Eingabegröße (in diesem Fall die Anzahl Gegenstände) und p ein Polynom in zwei Variablen ist, eine zulässige (approximierende) Lösung I_1 berechnet, die gegenüber einer optimalen Lösung I^* einen nur geringfügig kleineren Nutzwert garantiert, genauer:

$$\sum_{i \in I_1} a_i \geq (1-\varepsilon) \sum_{i \in I^*} a_i \,.$$

Entwickeln Sie ein FPTAS für das Rucksackproblem und argumentieren sie, dass es sich wirklich um ein FTPAS handelt (d.h. die Laufzeit und Qualität der Lösung entsprechend den obigen Anforderungen).

Hinweis: Setzen Sie $A := \max\limits_{i=1,\ldots,n} a_i$, skalieren Sie die Nutzwerte mit $\frac{n}{A\varepsilon}$, runden Sie ab und verwenden Sie (b).

Verwenden Sie Pseudocode oder eine andere hinreichend präzise Formulierung und begründen Sie die Laufzeit Ihrer Lösungen.

Musterlösung:

a) Wir verwenden dynamische Programmierung und betrachten für $i = 1,\ldots,n$ und $W = 0,\ldots,C$ Matrixeinträge $N[i,W]$, die den maximalen Nutzen eines gepackten Rucksacks der Kapazität W enthalten, der nur mit Gegenständen mit Index kleiner oder gleich i gepackt wurde.

Für $i = 1$ ist $N[1,W] = \begin{cases} 0 & \text{falls } W < c_1 \,, \\ a_1 & \text{falls } W \geq c_1 \,. \end{cases}$

Ist die erste Zeile der Matrix berechnet, kann man für $i > 1$ den Rest bestimmten durch

$$N[i,W] = \begin{cases} \max\{N[i-1,W], N[i-1,W-c_i] + a_i\} & \text{falls } W \geq c_i\,, \\ N[i-1,W] & \text{sonst,} \end{cases}$$

denn entweder enthält die optimale Rucksack-Packung den „neuen“ Gegenstand i oder nicht.

Da jeder Matrixeintrag $N[i,W]$ in konstanter Zeit berechnet werden kann und die Matrix nC Einträge enthält, kann $N[n,C]$ in $\mathcal{O}(nC)$ Schritten berechnet werden.

Ist $N[n,C]$ berechnet, so findet man die Indexmenge I durch Rückwärts-Suche in der Matrix. An Eintrag $N[i,W]$ bestimmt man, ob der dort stehende Wert $N[i-1,W]$ oder $N[i-1,W-c_i]+a_i$ entspricht. Nur im zweiten Fall wird i zu I hinzugefügt, in beiden Fällen wird I rekursiv weiter an der entsprechenden Matrixstelle berechnet, bis $i = 0$.

b) Wir verwenden dynamische Programmierung und betrachten für $i = 0,\ldots,n$ und $B = 0,\ldots,Opt$ Matrixeinträge $V[i,B]$, die das minimale Volumen eines Rucksacks enthalten, der nur mit Gegenständen mit Index kleiner oder gleich i gepackt wurde und einem Gesamtnutzen von mindestens B hat.

Für $i = 0$ ist $V[0,0] = 0$ und $V[0,B] = \infty$ für $B = 1,\ldots,Opt$.

Ist die erste Zeile der Matrix berechnet, kann man für $i \geq 1$ den Rest bestimmten durch

$$V[i,B] = \begin{cases} \min\{V[i-1,B], V[i-1,B-a_i] + c_i\} & \text{falls } B \geq a_i\,, \\ V[i-1,B] & \text{sonst,} \end{cases}$$

denn entweder kann man mit Gegenstand i einen größeren Nutzen erzielen oder nicht.

Da Opt in der Regel nicht bekannt ist, rechnet man die Matrix aus, indem man B schrittweise erhöht. Zuerst wird die Spalte $V[i,1]$, dann $V[i,2]$, etc berechnet. Die Berechnung wird abgebrochen, wenn $V[n,B] > C$ ist, denn dann kann dieser Nutzwert B mit einem Rucksack der Kapazität C nicht erreicht werden und $Opt = B - 1$.

Da jeder Matrixeintrag $V[i,B]$ in konstanter Zeit berechnet werden kann und die Matrix $n \cdot Opt$ Einträge enthält, kann $V[n,Opt]$ in $\mathcal{O}(n \cdot Opt)$ Schritten berechnet werden.

Die Indexmenge I wird wie in (a) durch Ruckwärts-Suche berechnet.

c) Wie im Hinweis nahegelegt setzen wir $A := \max_{i=1,\ldots,n} a_i$, skalieren alle Nutzwerte mit $\alpha := \frac{n}{A\varepsilon}$ und runden ab: $\tilde{a}_i := \lfloor \alpha \cdot a_i \rfloor$. Hierdurch wird der maximale Nutzen Opt eines zulässigen Rucksacks ebenfalls skaliert.

Wir berechnen nun eine obere Schranke für den maximalen Nutzen $\widetilde{Opt}$ des modifizierten Problems, indem wir annehmen, alle anderen Gegenstände haben ebenfalls den größten Nutzen A. Daher ist

$$\widetilde{Opt} \leq n \cdot \alpha A = \frac{n^2}{\varepsilon}\,.$$

Wendet man den Algorithmus aus (b) auf das skalierte Problem an, braucht dieser nur Laufzeit $\mathcal{O}\big(n \cdot \widetilde{Opt}\big) = \mathcal{O}\big(\frac{1}{\varepsilon} n^3\big)$, ist also polynomial. Die berechnete Indexmenge I_1 ist eine *optimale* Lösung für das skalierte Problem, da der Algorithmus aus b) immer eine optimale Lösung für die übergebenen Parameter bestimmt.

Sei nun I^* eine Optimallösung des *unskalierten* Ausgangsproblems. Interpetieren wir I^* als Lösung des skalierten Problems, so gilt

$$\frac{1}{\alpha} \sum_{i \in I_1} \lfloor \alpha \cdot a_i \rfloor \geq \frac{1}{\alpha} \sum_{i \in I^*} \lfloor \alpha \cdot a_i \rfloor,$$

da I_1 eine optimale Lösung für das skalierte Problem ist (obwohl I^* eine optimale Lösung für das unskalierte Problem ist, muss diese Lösung nicht zwingend optimal für das skalierte Problem sein!).

Wir interpretieren nun I_1 als Lösung für das *unskalierte* Problem und zeigen, dass es sich bei obigen Vorgehen um ein FPTAS handelt. Dazu müssen wir $\sum_{i\in I_1} a_i \geq (1-\varepsilon)\sum_{i\in I^*} a_i$. zeigen. Wir beweisen dies Schrittweise (Erklärungen zu den einzelnen Ungleichungen sind am Ende zu finden)

$$\begin{aligned}
\sum_{i\in I_1} a_i &\geq \frac{1}{\alpha}\sum_{i\in I_1}\lfloor \alpha\cdot a_i\rfloor && (1)\\
&\geq \frac{1}{\alpha}\sum_{i\in I^*}\lfloor \alpha\cdot a_i\rfloor && (2)\\
&\geq \frac{1}{\alpha}\sum_{i\in I^*}(\alpha\cdot a_i - 1) && (3)\\
&= (\sum_{i\in I^*} a_i) - \frac{|I^*|}{\alpha} && (4)\\
&\geq (\sum_{i\in I^*} a_i) - \frac{n}{\alpha} && (5)\\
&= \sum_{i\in I^*} a_i - A\varepsilon && (6)\\
&\geq \sum_{i\in I^*} a_i - \sum_{i\in I^*} a_i\varepsilon = (1-\varepsilon)\sum_{i\in I^*} a_i\,. && (7)
\end{aligned}$$

Bei (1) vergleichen wir I_1 als Lösung für das unskalierte Problem mit I_1 als Lösung für das skalierte Probleme. Da die maximale Kapazität beim Skalieren höchstens verringert wird, gilt diese Ungleichung. Bei (2) tauschen wir I_1 mit I^* (siehe dazu auch die obige Argumentation). Bei (3) schätzen wir die Abrundungsklammern ab. Von (3) zu (5) ziehen wir die „-1 “ aus der Summe und schätzen $|I^*|$ mit n ab (dies gilt, da eine optimale Lösung maximal aus allen n Elementen bestehen kann). Für (6) nutzen wir aus, dass $\alpha = \frac{n}{A\varepsilon}$. Bei (7) verwenden wir, dass alle $c_i \leq n$ sind und somit eine optimale Lösung I^* mindestens den Nutzen A haben muss (da der entsprechende Gegenstand mit diesem Wert auf jeden Fall in den Rucksack passt).

Das Vorgehen liefert also ein FPTAS, das den Nutzen einzelner Gegenstände etwas überschätzt. Der Fehler entsteht dabei nur durch das Skalieren – der verwendete Subalgorithmus aus b) gibt immer eine optimale Lösung für seine Eingabe aus. Da wir das Problem hier aber skaliert haben, muss eine optimale Lösung für die Skalierung nicht auch optimal für das unskalierte Problem sein. Durch das Skalieren beschleunigen wir den Algorithmus aber wesentlich, was wir im Endeffekt mit einer gewissen Fehlerwahrscheinlichkeit „bezahlen“.

Aufgabe 2 (*Palindrome,* 6 *Punkte*)

Ein *Palindrom* ist ein Wort w, das rückwärts geschrieben das gleiche Wort bildet. Beispiele hierfür sind ‘anna’, ‘kayak’ und ‘reittier’.
Jedes Wort w kann in eine *Sequenz von Palindromen* zerlegt werden: ‘ababba’ lässt sich in ‘a|b|abba’, ‘aba|bb|a’, ‘a|bab|b|a‘ oder ‘a|b|a|b|b|a‘ zerlegen, also in 3–6 Palindrome.

Wir bezeichnen die *minimale Anzahl* von Palindromen, in die w zerlegt werden kann, mit $p(w)$.
Entwerfen Sie einen Algorithmus, der für ein Wort w die Zahl $p(w)$ in $\mathcal{O}(n^2)$ berechnet, wobei n die Länge von w ist. Erklären Sie, wie ihr Algorithmus funktioniert. Begründen Sie außerdem kurz seine Korrektheit und analysieren Sie die Laufzeit. Die Zerlegung selbst brauchen Sie nicht ausgeben.

Hinweis: Sei $w[i..j]$ das Teilwort von w, das den i-ten bis j-ten Buchstaben enthält. Konstruieren Sie zuerst ein Array $L[i,j]$, das angibt, ob $w[i..j]$ ein Palindrom ist.

Musterlösung:
Wir verwenden dynamische Programmierung und geben die Rekurrenzen, Reihenfolge und Sentinels an. Das Array $L[i,j]$ enthalte *true*, wenn $w[i..j]$ ein Palindrom der Länge $j-i+1$ ist. Man kann L berechnen indem man $L[i,j] := \mathit{true}$ für $i = j$ und $i - 1 = j$ festlegt und

for $j = 1,\ldots,n$ **do** −− Schleife über das Teilwort $w[1..j]$
  for $i = j-1,\ldots,1$ **do** −− Schleife über die Palindrom-Länge
    $L[i,j] := (w[i] = w[j] \text{ und } L[i+1,j-1])$

durchläuft (wir gehen davon aus, dass das Array zu Beginn komplett mit *false* initialisiert ist, insbesondere für $i > j+1$ sind die Einträge also *false*). Die Werte $i = j$ und $i - 1 = j$ sind Sentinels und bezeichnen Palindrome der Länge Eins (ein einzelner Buchstabe) und Null (kein Buchstabe). Längere Palindrome werden dadurch berechnet, dass man den ersten $w[i]$ und letzten Buchstaben $w[j]$ vergleicht und prüft ob die dazwischenliegenden Buchstaben ein Palindrom sind. In der Schleife wird schrittweise das Wort w durch Anhängen eines Buchstabens $w[j]$ aufgebaut, und dabei längere Palindrome aus kürzen konstruiert. Mit L kann man nun $p(w)$ mit dynamischer Programmierung über die Teilworte $w[1..j]$ berechnen. Man kann $M[j] = p(w[1..j])$ errechnen, in dem man $M[0] = 0$ als Sentinel festlegt und dann

for $j = 1,\ldots,n$ **do** $\quad M[j] := \min_{i=1,\ldots,j} \{M[i-1]+1 \mid L[i,j] = \mathit{true}\}$

durchläuft. Für jedes $M[j]$ durchläuft i dabei alle möglichen Zerlegungs-Schnitte in $w[1..j]$. Falls $w[i..j]$ ein Palindrom ist, kann man auf $p(w[1..(i-1)])$ des übrigen Worts zurückgreifen und Eins dazu zählen. Um $p(w[1..j])$ zu ermitteln, bestimmt man das Minimum über alle möglichen Schnitte.
Der gesamte Algorithmus liegt in $\mathcal{O}(n^2)$, da sowohl die Berechnung von L in $\mathcal{O}(n^2)$ als auch diejenige von M in $\mathcal{O}(n^2)$ liegen.

Aufgabe 3 (*Vertex Cover,* 4 *Punkte + bis zu* 4 *Bonuspunkte*)

Wir betrachten das in der Übung vorgestellte Problem *Vertex Cover*:

Gegeben ist ein ungerichteter Graph $G = (V,E)$.

Gesucht ist eine minimale Menge $C \subseteq V$, sodass $\forall\{u,v\} \in E\colon \{u,v\} \cap C \neq \emptyset$.

a) Entwerfen Sie einen effizienten Greedy-Algorithmus, der des *Vertex Cover*-Problem auf Bäumen in Linearzeit $\mathcal{O}(V)$ optimal löst. Begründen Sie, warum Ihr Algorithmus funktioniert. Die Laufzeit muss nicht argumentiert werden. Sie können davon ausgehen, dass die Datenstruktur, in der der ungerichtete Baum vorliegt, es ermöglicht in $\mathcal{O}(1)$ zu Überprüfen, ob ein Knoten ein Blatt ist und das jeder Knoten im Baum einen Zeiger auf seinen Vorgänger (Parent) im Baum enthält. Ihr Algorithmus darf den Baum beliebig modifizeren (beispielsweise Knoten löschen).

b) Der leicht verwirrte Professor Peter R. Oblem schlägt folgenden Algorithmus zur Lösung des Vertex Cover-Problems vor: Wähle immer den Knoten mit höchstem Grad (d.h. der größten Anzahl von inzidenten Kanten) aus, füge ihn zum Vertex Cover hinzu, und lösche ihn samt aller seiner Kanten. Wiederhole, bis der Graph keine Kanten mehr enthält.

   i. Zeigen Sie (z.B. mittels eines Beispiels), dass der vorgeschlagene Algorithmus nicht optimal ist.
   Hinweis: Diese Teilaufgabe gibt einen Bonuspunkt.

   ii. Zeigen Sie, dass es sich bei dem Algorithmus auch nicht um eine 1,5-Approximation handelt, d.h. dass es Fälle gibt, in denen das berechnete Vertex Cover um mehr als einen Faktor 1,5 größer ist als das optimale Vertex Cover.
   Hinweis: Diese Teilaufgabe gibt 2 Bonuspunkte.

   iii. Zeigen Sie, dass es sich bei dem Algorithmus nicht einmal um eine 2-Approximation handelt, d.h. dass es Fälle gibt, in denen das berechnete Vertex Cover um mehr als einen Faktor 2 größer ist als das optimale Vertex Cover.
   Hinweis: Diese Teilaufgabe löst offensichtlich auch die vorherige Teilaufgabe, und gibt *zusätzlich zu dieser* einen weiteren Bonuspunkt.

Musterlösung:

a) Der Algorithmus geht vor wie folgt:

```
Function Tree-Vertex-Cover(T = (V,E) : Tree)
    C := {}
    L : Queue
    foreach v in T do
        if v is leaf then L.push(v)
    while L is not empty do
        v := L.pop()
        u := v. parent
        T.delete(v)
        if u ≠ ⊥ then
            C := C ∪ {u}
            p_u := u. parent
            T.delete(u)
            if p_u ≠ ⊥ then
                if p_u is leaf then L.push(p_u)
    return C
```

Der Algorithmus basiert auf folgender Beobachtung: Es existiert optimales Vertex Cover C, dass keine Blätter des Baumes enthält. Dies gilt, da man in jedem optimalen Cover C_l, welches die Blätter enthält, die Blätter durch deren Parent-Knoten ersetzen kann. Das durch diese Ersetzung entstandene Cover ist höchstens so groß wie C_l.

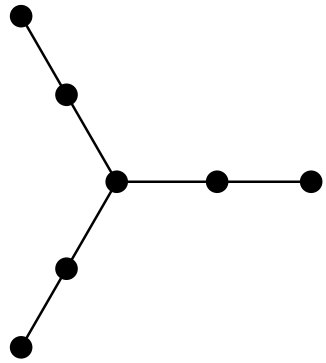
Der Algorithmus sammelt schritteweise die Elternknoten aller Blätter ein, fügt diese dem Cover hinzu, und entfernt alle bereits gecoverten Kanten (u,v), indem sowohl der Kind- als auch der Elternknoten aus dem Baum entfernt werden.

Die Korrektheit ist wie folgt zu sehen:

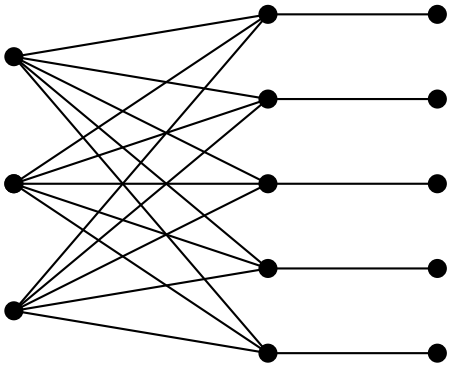
- *Überdeckung aller Kanten:* Es werden nur Kanten aus dem Baum gelöscht, die inzident zu einem der ausgewählten Elternknoten waren. Da diese Knoten Teil des Vertex Covers sind, sind gelöschte Kanten immer überdeckt. Der Algorithmus läuft so lange, bis der Baum leer ist. Damit sind am Ende alle Kanten überdeckt.
- *Minimalität:* Sei v ein Blatt und u der Elternknoten von v. Es ist klar, dass entweder u oder v Teil des minimalen Vertex Covers sein müssen. Angenommen, v sei Teil des minimales Vertex Covers. Da v keine Kante außer der zwischen u und v abdeckt, kann man problemlos v durch u ersetzen. Damit ist die Wahl der Elternknoten immer ausreichend und minimal. Da wir im Algorithmus iterativ die Blätter und ihre Eltern entfernen, kann dieses Argument wiederum iterativ auf die anderen Knoten angewendet werden, die entsprechend auch vom Algorithmus ausgewählt werden.

b) i. Betrachte den in Abbildung 1a abgebildeten Graphen. Der zentrale Knoten wird als erstes ausgewählt, und danach müssen noch drei weitere Knoten ins Vertex Cover aufgenommen werden. Drei Knoten insgesamt würden aber reichen.

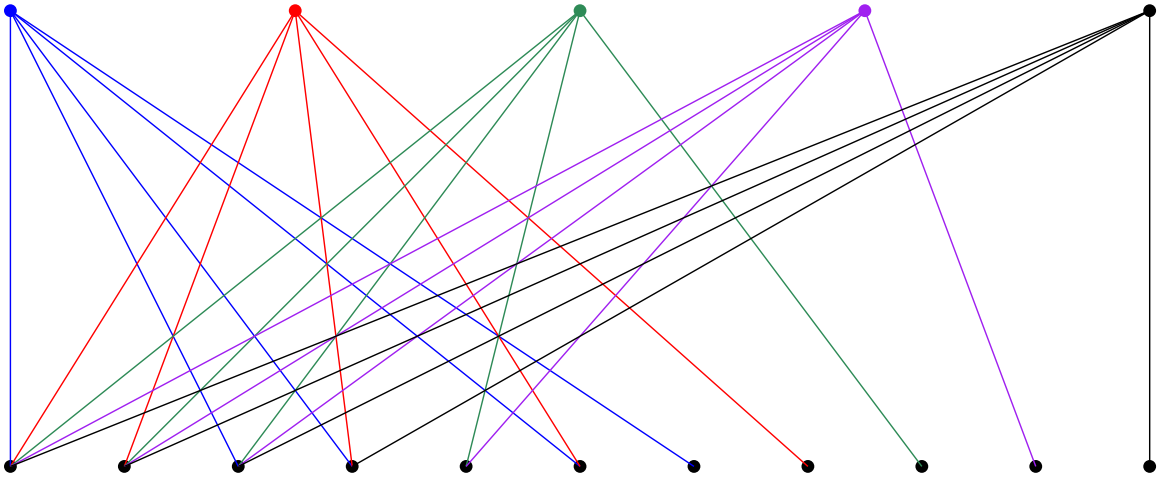
ii. Betrachte den in Abbildung 1b abgebildeten Graphen. Die Knoten lassen sich hier in drei Gruppen aufteilen: Links, Mitte und Rechts. Dabei sind alle linken Knoten mit allen mittleren Knoten verbunden, und jeweils ein mittlerer Knoten mit jeweils einem rechten Knoten. Die linken Knoten haben alle Grad 5, die mittleren Knoten Grad 4, die rechten Knoten Grad 1. Der Algorithmus wird nun zunächst die drei linken Knoten auswählen und entfernen, und muss danach noch fünf weitere Knoten ins Vertex Cover aufnehmen. Insgesamt hat das Vertex Cover also Größe 8. Ein minimales Vertex Cover bestünde nur aus den mittleren Knoten, hätte also Größe 5. Es gilt $\frac{8}{5} > 1{,}5$, damit ist die Behauptung gezeigt.



(a) Gegenbeispiel für b) i.



(b) Gegenbeispiel für b) ii.



(c) Gegenbeispiel für b) iii.

Abbildung 1: Graphen für Aufgabe 3

iii. Betrachte den in Abbildung 1c abgebildeten Graphen. Die Knoten können hier in obere und untere Knoten aufgeteilt werden. Zur besseren Übersicht haben wir die oberen Knoten sowie ihre Kanten unterschiedlich eingefärbt. Darüber hinaus haben die Farben keine Bedeutung. Die fünf oberen Knoten haben Grade (5,5,5,5,5), die unteren Knoten haben Grade (von links nach rechts) (5,4,4,3,2,2,1,1,1,1,1). Die Idee ist nun zu zeigen, dass der Algorithmus alle unteren Knoten auswählt, wobei es dazu kommen kann, dass der Knoten mit dem größten Grad nicht eindeutig ist, und der Algorithmus „unglücklich“ zwischen diesen auswählt:
Zunächst wählt der Algorithmus den unteren Knoten mit Grad 5 und entfernt diesen. Die Grade der oberen Knoten sinken damit auf (4,4,4,4,4) und der neue Maximalgrad ist 4. Jetzt wählt der Algorithmus den zweiten unteren Knoten (Grad 4) und entfernt ihn. Die Grade der oberen Knoten sinken damit auf (4,3,3,3,3). Der Algorithmus wählt nun den dritten unteren Knoten (Grad 4), womit die Grade der oberen Knoten auf (3,3,2,2,2) sinken. Der neue Maximalgrad ist 3. Als nächstes wird der vierte der unteren Knoten (mit Grad 3) gewählt, die Grade der oberen Knoten sinken auf (2,2,2,2,1), neuer Maximalgrad ist 2. Es wird der fünfte untere Knoten (Grad 2) gewählt, die oberen Grade sinken auf (2,2,1,1,1). Der sechste untere Knoten wird gewählt, die oberen Grade sinken auf (1,1,1,1,1). Nun müssen noch fünf weitere Knoten gewählt werden (entweder von oben oder unten), somit werden insgesamt 11 Knoten ausgewählt. Es ist aber klar, dass die fünf oberen Knoten alleine ein Vertex Cover bilden. Damit ist das berechnete Vertex Cover um mehr als einen Faktor zwei größer als ein minimales Vertex Cover.