Julius-Maximilians-Universität Würzburg
Fakultät für Mathematik und Informatik

SS14, Dr. Spoerhase
auf Basis von SS12

Exakte Algorithmen

Nils Wisiol, Andre Löffler

8. April 2014

Inhaltsverzeichnis

1 Einführung

1.1 Motivation

- effiziente Algorithmen vs. ineffiziente Algorithmen (also: polynomiell vs. super-polynomiell)
- Warum beschäftigen wir uns mit ineffizienten Algorithmen? Damit wir NP-schwere Probleme angehen können.
- Um mit NP-schweren Problemen umgehen zu können, bedient man sich:
  - Näherungsverfahren
    - Approximationsalgorithmen
    - Heuristiken
  - Exakte Verfahren
    - exakte exponentielle Algorithmen
    - parametrisierte Algorithmen, also Laufzeiten der Form $f(k) \cdot poly(n)$
- Warum haben wir Interesse an exakten Verfahren?
  - (Laufzeit nicht entscheidend)
  - nicht-approximierbare Probleme
  - moderate Eingabegröße
    - z.B.: $n^3 > 1,0941^n \cdot n$ für $n \leq 100$
    - TSP exakt lösbar für $n \leq 2000$ (für euklidsche Instanzen sogar bis 15000)
    - $2^{100} \cdot n > 2^n$ für $n \leq 100$
  - Entscheidungsprobleme (z.B.: Hamiltonkreis)
  - theoretisches Interesse
- Thema: geht es intelligenter und schneller als Brute-Force?
- typisches Resultat:

BF:	$O(2^n)$	$O^*(2^n)$
Alg1:	$O(n \cdot 1,5^n)$	$O^*(1,5^n)$
Alg2:	$O(n^2 \cdot 1,4^n)$	$O^*(1,4^n)$

- Wozu solche Ergebnisse?

$$\begin{aligned} a^{n_0'} &= c \cdot a^{n_0} \\ n_0' \cdot \log a &= \log c + n_0 \log a \\ n_0' &= \frac{\log c}{\log a} + n_0 \end{aligned}$$

$\Rightarrow$ mehr Rechenleistung bringt nicht immer die gewünschte Laufzeitverbesserung, aber:

$$\begin{aligned} a^n, b^n, a &< b \\ a^{n_0'} &= b^{n_0} \\ n_0' &= \frac{\log b}{\log a} n_0 \end{aligned}$$

- Literatur: *Exact Exponential Algorithms* von Fomin/Kratsch.

Definition 1 (O^*-Notation)**.**

$$f(n) = O^*(g(n)) \Leftrightarrow f(n) = O(g(n) \cdot poly(n))$$

z.B: $O(1,498^n \cdot n^{100}) \subsetneq O(1,499^n)$

1.2 Der Held-Karp-Algorithmus für TSP

Gegeben seien n Städte $c_1, ..., c_n$ und für jedes Paar $c_i \neq c_j$ eine Distanz $d(c_i, c_j)$. Wir suchen eine Permutation (Rundtour) Π auf $\{1, ..., n\}$, sodass die Gesamtlänge der Tour

$$\left(\sum_{i=1}^{n-1} d(c_{\Pi(i)}, c_{\Pi(i+1)}) \right) + d(c_{\Pi(n)}, c_{\Pi(1)})$$

minimal ist.

Ein brute-force-Algorithmus würde alle Permutation ausprobieren und damit Laufzeit $O(n! \cdot n)$ haben. Dies ist eine exponentielle Laufzeit, denn es gilt

$$\begin{aligned} \Theta(n! \cdot n) &= n \cdot 2^{\Theta(n \cdot \log_2 n)} \\ 2^{O(n \log n)} = (\frac{n}{2})^{\frac{n}{2}} &\leq n! \leq n^n = 2^{n \log n} \end{aligned}$$

Dynamisches Programm

Für den Algorithmus von Held-Karp definieren wir $\text{OPT}[S, c_i]$ mit der nichtleeren Menge $S \subseteq \{c_2, ..., c_n\}$ und $c_i \in S$ als Länge des kürzestens Weges von c_1 nach c_i, der genau die Städte $S \cup \{c_1\}$ besucht. Damit ist für $|S| = 1$ gerade $\text{OPT}[\{c_i\}, c_i] = d(c_1, c_i)$ und für $|S| \geq 2$:

$$\text{OPT}[S, c_i] = \min_{c_j \in S \setminus \{c_i\}} \{\text{OPT}[S \setminus \{c_i\}, c_j] + d(c_j, c_i)\}.$$

Name: *Algorithmus von Held-Karp*
Data: Städte $c_1, ..., c_n$ und Distanzfunktion d
Result: Länge der kürzesten Rundreise durch alle c_i
for $i = 1$ *bis* n **do**
  $\text{OPT}[\{c_i\}, c_i] \leftarrow d(c_1, c_i)$
end
for $j = 2$ *bis* n **do**
  for $S \subseteq \{c_2, ..., c_n\}$ *mit* $|S| = j$ **do**
    for $c_i \in S$ **do**
      $\text{OPT}[S, c_i] \leftarrow \min_{c_j \in S \setminus \{c_i\}} \{\text{OPT}[S \setminus \{c_i\}, c_j] + d(c_j, c_i)\}$
    end
  end
end
return $\min_{i \in \{2,...,n\}} \{\text{OPT}[\{c_2, ..., c_n\}, c_i] + d(c_i, c_1)\}$

Algorithm 1: Algorithmus von Held-Karp zum Lösen von TSP

Die Laufzeit von Held-Karp liegt in $O(2^n \cdot n^2) \subseteq O^*(2^n)$. Warum n^2?

1.3 Ein Branching-Algorithmus für Independent Set

Es sei ein Graph $G = (V, E)$ gegeben. Gesucht ist ein $U \subseteq V$, so dass keine zwei Knoten aus U adjazent in G sind.

Ein brute-force-Algorithmus würde in $O^*(2^n)$ alle möglichen Teilmengen ausprobieren. Stattdessen wollen wir einen Algorithmus für independent set angeben, der in $O^*(3^{n/3}) = O^*(1.4423^n)$ liegt. Wir beobachten zunächst: Ist U eine maximale unabhängige Menge, so gilt

(i) $v \in U \implies N(v) \cap U = \emptyset$,

(ii) $v \notin U \implies |N(v) \cap U| \geq 1$.

Wir definieren außerdem die Menge $N[v] =_{\text{def}} N(v) \cup \{v\}$.

Name: *MIS*
Data: Graph $G = (V, E)$
Result: Mächtigkeit einer größten unabhängigen Menge
if $|V| = 0$ **then**
  return 0
else
  $v \leftarrow$ Knoten mit minimalem Grad ;
  return $1 + \max_{y \in N[v]} \{MIS(G \setminus N[y])\}$
end

Algorithm 2: Algorithmus zur Berechnung der Mächtigkeit einer größten unabhängigen Menge

Dieser Algorithmus hat Laufzeit $O^*(1.4423^n)$.

1.4 Weitere Branching-Algorithmen

Ein Branching-Algorithmus besteht aus Branching- und Reduktionsregeln. Führt eine Branching-Regel auf einer Eingabe der Größe n zu $r \geq 2$ Teilinstanzen der Größe $n - t_1, ..., n - t_r$, so heißt $(t_1, ..., t_r)$ *Branching-Vektor*. Es folgt für die Laufzeit $T(n)$ dass $T(n) \leq \sum_{i=1}^{r} T(n - t_i)$.

Satz 1. *Sei b eine Branching-Regel mit Branching-Vektor $(t_1, ..., t_r)$. Dann führt die ausschließliche Anwendung von b zur Laufzeit $O^*(\alpha^n)$, wobei α die eindeutige positive Lösung der Gleichung*

$$x^t - x^{t-t_1} - x^{t-t_2} - ... - x^{t-t_r} = 0$$

mit $t = \max t_i$ ist.

1.4.1 SAT

Eingabe ist eine aussagenlogische Formel in konjunktiver Normalform. Existiert eine erfüllende Belegung der Variablen? Ein brute-force-Algorithmus hat Laufzeit $O^*(2^n)$; ob ein Algorithmus mit Laufzeit $O^*((2 - \varepsilon)^n)$ existiert ist ungeklärt.

1.4.2 k-SAT

Eingabe ist eine aussagenlogische Formel in konjunktiver Normalform, in der jede Klausel Länge kleiner oder gleich k besitzt. Für eine Formel F in konjunktiver Normalform mit einer partiellen Vorbelegung t definieren wir die *bereinigte Formel* $F[t]$ durch Einsetzen von t und Vereinfachen von F. Es folgt, dass t sich genau dann zu einer erfüllenden Belegung für F erweitern lässt, wenn $F[t]$ erfüllbar ist.

Idee. Für den folgenden Algorithmus seien die partiellen Vorbelegungen t_i definiert als

$$t_i : l_1 = ... = l_{i-1} = \text{false}, \quad l_i = \text{true}.$$

Name: *k-SAT1*
Data: Formel F in konjunktiver Normalform und maximaler Klausellänge k
Result: Erfüllbarkeit von F
if *F enthällt leere Klausel* **then**
  return false
else if *F ist die leere Formel* **then**
  return true
end
wähle Klausel $l = (l_1 \vee l_2 \vee ... \vee l_q)$;
return k-SAT1($F[t_1]$)) $\vee$ k-SAT1($F[t_2]$)) $\vee$ ... $\vee$ k-SAT1($F[t_q]$));

Algorithm 3: Algorithmus zur Entscheidung einer k-CNF-Formel F

Für die Laufzeit ergibt sich die rekursive Formel $T(n) \leq T(n-1) + T(n-2) + ... + T(n-q)$ mit einem $q \leq k$. Die Laufzeit ergibt sich dann nach Satz 1, für 3-SAT beispielsweise $O^*(1.8393^n)$.

Verbesserung. Wir modifizieren den Algorithmus zu k-SAT2, indem wir immer die kleinste Klausel wählen. Wir nennen eine partielle Belegung t einer Formel *autark*, wenn jede Klausel, die mindestens ein von t belegtes Literal enthält, auch ein von t mit wahr belegtes Literal enthält. Dies

bedeutet, dass die Klausel unabhängig von allen anderen Literalen der Klausel wahr ist. Daraus ergeben sich die folgenden Beobachtungen.

(i) Wenn t autark ist, dann ist F genau dann erfüllbar, wenn $F[t]$ erfüllbar. Das bedeutet, dass wir für autarke t das Ergebnis von k-SAT2($F[t]$) zurückgeben können.

(ii) Wenn t nicht autark ist, dann enthält $F[t]$ eine Klausel von Länge höchstens $k-1$.

(iii) Sei r die Wurzel des Suchbaumes. Wenn $v \neq r$ ein (Branching-)Knoten mit k Kindern im Suchbaum ist, so ist der Vaterknoten von v ein Reduce-Knoten.

Mit diesen Verbesserungen erreicht man, dass k-SAT2 die selbe Laufzeit hat wie $(k-1)$-SAT1.

2 Dynamisches Programmieren

2.1 Dominating Set

Sei ein Graph $G = (V, E)$ gegeben. Wir nennen eine Menge $D \subseteq V$ *dominierend*, falls jeder Knoten in $V \setminus D$ adjazent zu einem Knoten aus D ist.

Ein brute-force-Algorithmus für das kleinste dominating set läuft in $O^*(2^n)$ durch Ausprobieren aller Teilmengen von V. Zur Verbesserung dieser Laufzeit betrachten wir nicht erweiterbare unabhängige Mengen (NEUM) und verwenden dynamische Progammierung. Wir beobachten, dass sich eine NEUM effizient ermitteln lässt und jede NEUM dominierend ist.

Idee. Zunächst bestimmen wir ein NEUM I. Falls dieses klein ist, können wir alle Knotenmengen D mit $|D| \leq |I|$ ausprobieren. Falls I groß ist, wenden wir das folgende Verfahren an.

Lemma 2. *Sei G ein Graph und I ein zugehöriges NEUM. Dann lässt sich eine kleinste dominierende Menge in $O^*(2^{n-|I|})$ ermitteln.*

Beweis fehlt. 15.5.12

Lemma 3. *Sei $\alpha \leq 1/2$. Dann gilt*

$$\sum_{i=0}^{\alpha \cdot n} \binom{n}{i} = O^*(2^{h(\alpha) \cdot n}),$$

wobei $h(\alpha) = -\alpha \log_2 \alpha - (1-\alpha) \log_2 (1-\alpha)$.

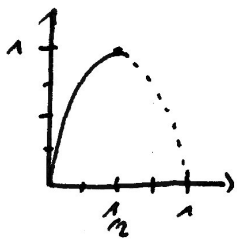


Abbildung 2.1: Graph des Binomialkoeffizienten

Beweis. Es ist $\sum_{i=0}^{\alpha n} \leq n \binom{n}{\alpha n} = O^*(\binom{n}{\alpha n})$, denn die Binomialkoeffizienten $\binom{a}{b}$ steigen für $b \leq n/2$ monoton an. Per Definition gilt

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}.$$

Die Fakultät kann abgeschätzt werden durch $\sqrt{2\pi n}(n/e)^n \leq n! \leq 2\sqrt{2\pi n}(n/e)^n$, also ist $n!$ proportional zu $(n/e)^n$. Daraus folgt, dass

$$\begin{aligned}\binom{n}{\alpha n} &= O^*\left(\frac{(n/e)^n}{(\alpha n/e)^{\alpha n}((1-\alpha)n/e)^{(1-\alpha)n}}\right) = O^*\left(\alpha^{-\alpha n}(1-\alpha)^{-(1-\alpha)n}\right) \\ &= O^*\left(2^{-\alpha \log_2 \alpha n} \cdot 2^{-(1-\alpha)\log_2(1-\alpha)n}\right),\end{aligned}$$

woraus die Behauptung folgt. □

Satz 4. *Eine kleinste dominierende Menge lässt sich in $O(1{,}7088^n)$ ermitteln.*

Beweis. Zunächst bestimmen wir eine nicht-erweiterbare unabhänge Menge I. Falls $|I| \leq \alpha n$, testen wir in $O^*(2^{h(\alpha)n})$ alle Teilmengen $D \subseteq V$ mit $|D| \leq |I|$. Falls $|I| > \alpha n$, wende Satz **??** an und berechne kleinste dominierende Menge in $O^*(2^{(1-\alpha)n})$.

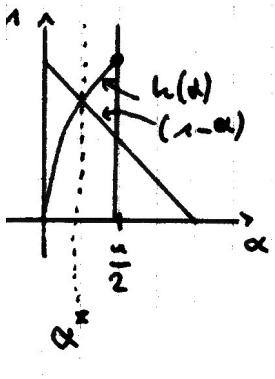
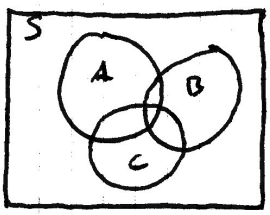


Abbildung 2.2: Bestimmung von α^* als Maximum der zwei möglichen Funktionen

Aus der Skizze ergibt sich die Laufzeit als das Maximum der beiden dargestellten Funktionen bei $\alpha^* \leq 0,22711$ bzw. $O(2^{0,7729n}) = O(1,7088^n)$. □

Der schnellste derzeit bekannte Algorithmus für dieses Problem benötigt ca. $O(1,5^n)$ und stammt aus 2010.

3 Inklusion-Exklusion



Gehe von einem Problem aus, bei es leicht ist zu zählen, welche Elemente aus dem Universum S die Eigenschaft A oder B, A und B, ... erfüllen; es aber schwer ist zu zählen, wie viele Elemente diese Eigenschaften nicht besitzen. Es ergibt sich jedoch der Zusammenhang

$$|\overline{A \cup B \cup C}| = |S| - (|A| + |B| + |C|) + (|A \cap B| + |A \cap C| + |B \cap C|) - (|A \cap B \cap C|).$$

Sind N Objekte und eine Menge $P = \{P_1, ...P_n\}$ von Eigenschaften gegeben, bezeichnen wir für jedes $S \subseteq P$ mit $N(S)$ die Anzahl der Objekte, die (mindestens) die Eigenschaften in S erfüllen. Mit $N(0)$ bezeichnen wir die Anzahl der Objekte, die keine der Eigenschaften erfüllen. Wir können oben skizzierte Formel dann verallgemeinern, es ergibt sich

Satz 5. $N(0) = \sum_{S \subseteq P} (-1)^{|S|} N(S) = N(\emptyset) + \sum_i N(P_i) + \sum_{i \leq j} N(P_i, P_j) + ...$

Beweis. Elemente des Universums, die keine der Eigenschaften besitzen, werden auf beiden Seiten der Gleichung einmal gezählt. Betrachte nun die Elemente, die genau die Eigenschaften $S = \{P_{i_1}, ..., P_{i_s}\}$, $|S| = s$. Diese werden genau in den $N(S')$ mit $S' \subseteq S$ gezählt. Daraus folgt, dass diese Objekte zur rechten Seite der Gleichung jeweils $\sum_{S' \subseteq S} (-1)^{|S'|} = \sum_{i=0}^{s} \binom{s}{i} \cdot (-1)^i = (-1+1)^s = 0$ beitragen. Objekte, die mindestens eine Eigenschaft besitzen, werden also auf der rechten Seite nicht gezählt. □

22.5.12

Korollar 6. $N(P) = \sum_{S \subseteq P} (-1)^{|S|} \overline{N}(S)$, *wobei* $\overline{N}(S)$ *die Anzahl der Objekte mit keiner der Eigenschaften in* S *ist.*

Beweis. Es sei P_i' die Eigenschaft, dass ein Objekt die Eigenschaft P_i nicht besitzt. Es ist dann $N'(0) = N(P)$ und $N'(P_1', ...P_k') = \overline{N}(P_1, ..., P_k)$. Es folgt die Behauptung. □

3.1 Hamiltonpfad

Wir betrachten nun das gerichtete Hamiltonpfadproblem. Gegeben sei ein gerichteter Graph $G = (V, E)$ sowie Knoten $s, t \in V$. Das gerichtete Hamiltonpfadproblem bezeichnet das Problem, ob ein einfacher s-t-Pfad existiert, der alle Knoten in V besucht. Dieses Problem ist NP-schwer. Brute force benötigt $O^*(n!)$ Zeit. Wir geben einen Algorithmus an, der $O^*(2^n)$ Zeit und polynomiellen Speicherplatz benötigt.

Satz 7. *Die Anzahl der* s-t*-Hamiltonpfade kann in* $O^*(2^n)$ *Zeit und mit polynomiellen Speicherplatzverbrauch ermittelt werden.*

Beweis. Wir verwenden das Inklusion-Exklusion-Prinzip. Als Objekte sehen wir alle s-t-Pfade der Länge $n-1$ an – auch solche, die nicht einfach sind. Die Eigenschaften $v \in V$ sind definiert als „Pfad besucht den Knoten v“. Also ergibt sich $N(V)$ als die Anzahl der s-t-Hamiltonpfade. Zur Berechnung von $N(V)$ bestimmen wir zunächst für $W \subseteq V$ jeweils $\overline{N}(W)$, die Anzahl der s-t-Pfade der Länge $n-1$, die W vermeiden.

Dies erreichen wir durch dynamische Programmierung. Das Programm berechnet für ein festes $W \subseteq V \setminus \{s,t\}$, ein $k = 0, ..., n-1$ und ein $u \in V \setminus W$ die Anzahl der s-u-Pfade der Länge k, die W vermeiden. Diese Zahl nennen wir $\overline{N}(W,u,k)$. Es folgt $\overline{N}(W,t,n-1) = \overline{N}(w)$. Die Berechnung erfolgt für $k = 0$ durch

$$\overline{N}(W,u,0) = \begin{cases} 1 & (u = s) \\ 0 & (\text{sonst}) \end{cases}$$

und für $k > 0$ durch

$$\overline{N}(W,u,k) = \sum_{vu \in E, v \notin W} \overline{N}(W,v,k-1)$$

Die Laufzeit des dynamischen Programms für ein festes W ist $O(n \cdot m)$ mit $m = |E|$. Durch Wiederverwendung von Speicherplatz erreicht man einen Speicherverbrauch von $O(n \cdot \log 2^m) = O(n \cdot m)$.

Zur Berechnung des Endergebnisses iterieren wir über alle $W \subseteq V \setminus \{s,t\}$ und berechnen jeweils $\overline{N}(w)$ mit dem oben angegebenen dynamischen Programm. Wir addieren den Wert des Ausdrucks $(-1)^{|W|} \cdot \overline{N}(w)$ und erhalten $N(V)$ als Wert dieser Summe. Die Gesamtlaufzeit ergibt sich damit als $O(m \cdot n \cdot 2^n)$, der Gesamtspeicherbedarf als $O(m \cdot n)$, da der Speicher wiederverwendet werden kann. □

3.2 Graphfärbung

Wir betrachten nun das Problem der Graph-Färbung. Als Eingabe sei ein Graph $G = (V,E)$ gegeben. Eine *zulässige Färbung* weißt jedem Knoten so einer Farbe zu, dass keine zwei benachbarten Knoten die selbe Farbe haben. Gesucht ist die minimale Anzahl an Farben, mit denen dies möglich ist. Brute force benötigt die Zeit $O^*(n^n)$. Wir geben einen Algorithmus an, der $O^*(2^n)$ Zeit und Speicherplatz benötigt.

Wir nennen die Menge aller Knoten einer bestimmten Farbe eine *Farbklasse*. Für korrekte Färbungen ist eine Farbklasse eine unabhängige Knotenmenge, daher können wir das Graphfärbungsproblem auch als Problem, die kleine Anzahl an unabhängigen Knotenmengen $I_1, ..., I_k$, die V partitionieren, formulieren. Dies ist verwand zum Set-Cover-Problem $(\mathcal{I}, V)$, wobei $\mathcal{I}$ die Menge aller unabhängigen Mengen ist. Verallgemeinert betrachten wir daher Set-Cover-Instanzen (S,U), bei denen U explizit gegeben ist und S in $O^*(2^n)$ mit $n = |U|$ aufgezählt werden kann.

Wir nennen ein k-Tupel $(S_1, ..., S_k) \in S^k$ ein *geordnetes k-Cover*, falls $\bigcup_{i=1}^k S_i = U$ und geben einen Algorithmus an, der die Anzahl der geordnetes k-Cover ermittelt. Für $W \subseteq U$ sei $S[W]$ definiert als $\{S_i \in S : S_i \cap W = \emptyset\}$ sowie $s[W] = |S[W]|$.

Lemma 8. *Die Anzahl der geordneten k-Cover beträgt $c_k = \sum_{W \subseteq U} (-1)^{|W|} s[W]^k$.*

5.6.12

Beweis. Wir betrachten die k-Tupel $(S_1, ..., S_k)$ als Objekte und für $v \in U$ den Ausdruck $v \in \bigcup_{i=1}^k S_i$ als Eigenschaft, und wenden darauf die Inklusions- / Exklusionsformel an. Die Eigenschaft $v \in U$ für ein k-Tupel gilt also nicht, wenn $v \notin \cup_{i=1}^k S_i$. Ist ein $W \subseteq U$ gegeben, so gelten alle Eigenschaften aus W nicht, wenn für alle $v \in W$ stets $v \notin \bigcup_i S_i$ gilt. Anders formuliert, muss für

alle i der Schnitt $S_i \cap W = \emptyset$ sein. Da es genau $s[W]$ Mengen gibt, die W vermeinden, gibt es $s[W]^k$ Möglichkeiten, ein k-Tupel aus Mengen zu bilden, die W vermeiden. Es ist damit $\overline{N}[W] = s[W]^k$. Satz 5 liefert dann

$$c_k = N[U] = \sum_{W \subseteq U} (-1)^{|W|} s[W]^k$$

als Anzahl der Tupel, für die alle Eigenschaften $v \in U$ gelten. Dies ist die Anzahl der Tupel, in denen für jedes $v \in U$ gilt dass $v \in \bigcup_i S_i$ ist. □

Satz 9. *Die Anzahl c_k aller k-Cover kann in $O^*(2^n)$ berechnet werden.*

Beweis. Es ist $c_k = \sum_{W \subseteq U} (-1)^{|W|} s[W]^k$. Zum Auswerten der Formel müssen alle Teilmengen $W \subseteq U$ aufgezählt und jeweils $s[W]$ berechnet werden. Dazu ordnen wir die Elemente von U durch $U = \{u_1, ..., u_n\}$ beliebig. Dann sei für alle $W \subseteq U$

$$g_i(W) = |\{S_i \in S[W] : \{u_1, ..., u_i\} \setminus W \subseteq S_i\}|.$$

Dies ist die Anzahl der Mengen, die W vermeiden, und alle Elemente $\{u_1, ..., u_i\} \setminus W$ enthält. Mit dieser Definition gilt $s[W] = g_0(W)$, denn $g_0(W) = |\{S_i \in S[W] : \emptyset \subseteq S\}| = |S[W]|$. Wir verwenden ein dynamisches Programm, das über $i = n, ..., 0$ iteriert und zunächst

$$g_n(W) = \begin{cases} 1 & (U \setminus W \in S) \\ 0 & (\text{sonst}) \end{cases}$$

berechnet. Für $0 \leq i < n$ ist dann

$$g_{i-1}(W) = \begin{cases} g_i(W) & (u_i \in W), \\ g_i(W \cup \{u_i\}) + g_i(W) & (u_i \notin W). \end{cases}$$

Im Fall $u_i \in W$ ändert sich durch den Ausschluss von W in der Definition von g_i nichts an der Bedingung, daher bleibt die Anzahl gleich. Im anderen Fall können zwei Fälle eintreten, deren Möglichkeiten addiert werden. $g_i(W \cup \{u_i\})$ erfasst die Elemente, die u_i vermeiden, während $g_i(W)$ die Möglichkeiten enthält, die u_i enthalten. Für jede Menge $W \subseteq S$ wird der Ausdruck $g_i(W)$ n-mal ausgewertet, daher beträgt die Laufzeit $O^*(2^n)$. Der Speicherverbrauch liegt bei $O(2^n)$, wenn man zu jeder Zeit immer nur die vorhergehende Zeile in der dynamischen Tabelle speichert. Somit kann die Formel mit Hilfe von diesem Algorithmus in $O(2^n)$ Speicher und Zeit ausgewertet werden. □

4 Measure & Conquer

4.1 Maximal Independent Set

Measure and Conquer bezeichnet ein Verfahren, um Rekursonsgleichungen wie beispielsweise $T(n) \leq T(n-1) + T(n-5)$ zu lösen. Ein Algorithmus für unabhängige Mengen ist

Name: *MIS*
Data: Graph G
Result: Größte Anzahl von unabhängigen Mengen
if *ex.* v *mit* $d(v) = 0$ **then**
  return $1 + \text{MIS}(G - v)$
else if *ex.* v *mit* $d(v) = 1$ **then**
  return $1 + \text{MIS}(G - N[v])$
else if $\Delta(G) \geq 3$ **then**
  wähle v mit $d(v) \geq 3$
  return $\max\{1 + \text{MIS}(G - N[v]), \text{MIS}(G - v)\}$
else if $\Delta(G) \leq 2$ **then**
  löse diesen Spezialfall effizient
end

Algorithm 4: Algorithmus zur Berechnung der größten Anzahl unabhängiger Mengen

In Fall 3 handelt es sich um Branching, die Laufzeit wird hier durch die Rekursonsgleichung $T(n) \leq T(n-1) + T(n-4)$ beschrieben. Die Standardanalyse liefert eine Abschätzung von $O(1{,}3803^n)$. Diese Analyse kann durch die Methode Measure & Conquer verbessert werden.

4.2 Dominating Set

Wir betrachten nun einen Algorithmus für Dominating Set, der bei konventioneller Analyse Laufzeit $O(1{,}9052^n)$, bei Analyse mit Measure & Conquer jedoch $O(1{,}5259^n)$ ergibt. Dazu modellieren wir Dominating Set als Set Cover-Instanz, wobei $U = V$ und $S = \{N[v] : v \in V\}$ ist. Die Größe dieser Set Cover-Instanz ist dann $|U| + |S| = n + n = 2n$. Wir entwickeln daher einen Algorithmus für Set Cover mit $O^*(c^{|U|+|S|})$ und $c \ll 2$.

Wir nehmen ohne Einschränkung an, dass $U = \bigcup_{S_i \in S} S_i$. Damit ist eine Set Cover-Instanz durch S spezifiziert. Die *Häufigkeit* eines Elements $u \in U$ sei definiert als die Anzahl der Mengen aus S,

die u enthalten.

Name: *SC*
Data: Set Cover-Instanz S
Result:
if *(1)* $|S| = 0$ **then**
  return 0
else if *(2) es existieren* $S_1, S_2 \in S$ *mit* $S_1 \subseteq S_2$ **then**
  return $\mathrm{SC}(S - S_1)$
else if *(3) es existiert* $u \in U(S)$ *mit Häufigkeit 1, d.h. liegt in genau einem* $S^* \in S$ **then**
  return $1 + \mathrm{SC}(\mathrm{del}(S^*, S))$
end
wähle Menge $S' \in S$ mit maximaler Kardinalität ;
if *(4)* $|S'| = 2$ **then**
  löse in Polynomialzeit (vgl. Übungsaufgabe) ;
else if $|S| \geq 3$ **then**
  $S_{in} = S - S^*$;
  $S_{out} = \mathrm{del}(S, S^*)$;
  return $\min\{\mathrm{SC}(S_{in}), 1 + \mathrm{SC}(S_{out})\}$
end

Algorithm 5: Algorithmus für Set Cover

Dabei ist $\mathrm{del}(A, S) = \{T : T = R \setminus A \neq \emptyset, R \in S\}$.

Für die konventionelle Analyse sei $k'(S) = |S| + |U(S)|$. Wir können aufgrund des Algorithmus die Rekursionsungleichung

$$T(k') \leq T(k' - 1) + T(k' - 4)$$

herleiten. Die übliche Lösungsmethode ergibt $T(k') = O(1{,}3803^{k'})$. Damit ergibt sich als Laufzeit für Dominating Set $O(1{,}3803^{2n}) = O(1{,}9052^n)$.

Für die Analyse mittels Measure & Konquer definiert man n_i als Anzahl der Mengen mit Kardinalität i und m_j als Anzahl der Elemente mit Häufigkeit j. Das Maß sei

$$k(S_i) = \underbrace{\sum_{i \geq 1} w_i n_i}_{k_w(S_i)} + \sum_{j \geq 1} v_j m_j$$

mit Gewichten $w_i, v_j \in [0, 1]$. Damit ist $k(S_i) \leq |U| + |S|$. Vereinfachend nehmen wir für die Gewichte an, dass

(a) $w_i \leq w_{i+1}, v_i \leq v_{i+1}$,

(b) $w_1 = v_1 = 0$,

(c) $w_1 = v_1 = 1$ für $i \geq 6$,

(d) $\Delta w_i \geq \Delta w_{i+1}$ mit $\Delta w_i := w_i - w_{i-1}$ und Δv_i analog.

Wir bestimmen den Branching-Vektor $(\Delta k_{out}, \Delta k_{in})$ in Abhängigkeit der w_i und v_i.

(1) Im Fall, dass S_i nicht gewählt wird, verändert sich Δk_{out} wie folgt.

  (i) Durch Wegfallen von S_i wird $k_w(S)$ um $w_{|S_i|}$ reduziert.

(ii) Sei r_j definiert als Anzahl der Elemente von S_i mit Häufigkeit j. Die Reduktion von $k_v(S)$ ist dann kleiner oder gleich $\sum_{i=2}^{6} r_i \Delta v_i$.

(iii) Angenommen $r_2 > 0$. Seien $R_1, ..., R_h$, $h \leq r_2$ die Mengen ungleich S_i, die mindestens ein Element der Häufigkeit 2 mit S_i teilen. Die Mengen R_i werden nachfolgend durch Regel (3) reduziert. Sei dazu $v_{2,i}$ die Anzahl solcher Elemente in R_i. Dann ist $|R_i| \geq v_{2,i} + 1$ und $|S_i| \geq v_{2,i} + 1$. Die Reduktion von $k_w(S)$ durch Wählen von R_i ist dann mindestens $W_{v_{2,i}+1}$. Damit existiert mindestens ein Element, das nicht in S_i liegt und aus der Instanz entfernt wird. Also wird $k_v(S)$ mindestens um v_2 reduziert.

Wir erhalten eine Reduktions von mindestens

$$\Delta k' = \begin{cases} 0 & (r_2 = 0), \\ v_2 + w_2 & (r_2 = 1), \\ v_2 + w_2 + w_3 & (r_2 = 2), \\ v_2 + w_4 & (r_2 \geq 3, |S_i| = 3), \end{cases}$$

Demnach ist in diesem Fall $\Delta k_{out} = w_{|S_i|} + \sum_{i=2}^{6} r_i \Delta v_i + \Delta k'$.

(2) Im Fall, dass S_i gewählt wird,

(i) wird $k_w(S)$ durch Wegfall von S_i um $w_{|S_i|}$ reduziert,

(ii) wird $k_v(S)$ durch Wegfall von S_i um $\sum_{i=2}^{6} r_i v_i + r_{\geq 7}$ reduziert,

(iii) wird $k_w(S)$ durch die Verkleinerung der Kardinalität derjenigen Mengen, die S_i schneiden, reduziert. Sei R eine solche Menge, d.h. $R \cap S \neq \emptyset$. Sei u also ein Element aus $R \cap S$, dann ist der Beitrag von u zur Verkleinerung der Kardinalität größer oder gleich $\Delta w_{|R|} \geq \Delta w_{|S_i|}$. Damit beträgt die gesamte Reduktion für diesen Beitrag mindestens

$$\Delta w_{|S_i|} \left(\sum_{i=2}^{6} (i-1) r_i + 6 \cdot r_{\geq 7} \right).$$

Für den Fall, dass S_i gewählt wird beträgt die Gesamtreduktion also mindestens

$$\Delta k_{in} = w_{|S_i|} + \sum_{i=2}^{6} r_i v_i + r_{\geq 7} + \Delta w_{|S_i|} \left(\sum_{i=2}^{6} (i-1) r_i + 6 \cdot r_{\geq 7} \right).$$

Sei nun für feste $(w, v) := (w_1, ..., w_6, v_1, ..., v_6)$ und $|S| \geq 3$ und $(r_i)_i$ mit $\sum_{i=2}^{6} r_i + r_{\geq 7} = |S|$ die Rekurenz $T(k) \leq T(k - \Delta k_{out}) + T(k - \Delta k_{in})$ definiert.

Wir beobachten, dass wir für jeden Branchingvektor mit $|S_i| \geq 7$ einen Branchingvektor mit $|S_i| = 7$ finden, der diesen dominiert. Dies folgt aus den Formeln für Δk_{in} und Δk_{out} zusammen mit der Tatsache, dass $\Delta w_{|S_i|} = 0$ für $|S_i| \geq 7$. Wir nehmen daher an, dass $3 \leq |S_i| \leq 7$ und damit ist die Anzahl der gültigen Belegungen für $|S_i|$ und $(r_i)_i$ endlich. Also ist für jedes feste (w, v) die Laufzeit beschränkt durch $\alpha(w, v)^k$, wobei $\alpha(w, v)$ die größte der Nullstellen der Gleichungen $x^t - x^{t-\Delta k_{out}} - x^{t-\Delta k_{in}} = 0$ für $t := \max\{\Delta k_{in}, \Delta k_{out}\}$ über alle Belegungen für $|S_i|, (r_i)_i$. Für jedes (w, v) erhalten wir so eine Laufzeitschranke $\alpha(w, v)^k$. Wir wollen nun ein (w, v) finden, sodass $\alpha(w, v)$ möglichst klein ist. Mit Methoden der quasi-konvexen Optimierung lässt sich eine Näherunglösung finden, diese liefert $\alpha(w_0, v_0) < 1,2353$.

Satz 10. *Set-Cover lässt sich in $O(1,2353^{|U|+|S|})$ lösen.*

Korollar 11. *Dominating-Set lässt sich in $O(1,5259^n)$ lösen.*

5 Tree width

5.1 Dynamisches Programm für Vertex Cover

3.7.12

Zunächst betrachten wir Vertex Cover auf Bäumen. Dazu berechnen wir, beginnend bei den Blättern, für jeden Teilbaum mit Wurzel v ein ein maximales Vertex Cover, in dem v enthalten ist, und ein Vertex Cover, in dem die Teilbaum-Wurzel v nicht enthalten ist. Wir speichern die Ergebnisse der Berechnung in den Tabellen $IN(v)$ bzw. $OUT(v)$ für Teilbäume mit Wurzel v, in denen v enthalten bzw. nicht enthalten ist.

Zur Berechnung von $OUT(v)$ für einen beliebigen Knoten v berechnen wir (in binären Bäumen) $IN(v_1) + IN(v_2)$ der beiden Kind-Knoten v_1 und v_2 von v. Wir müssen v_1 und v_2 in das Vertex Cover aufnehmen, da die Kanten (vv_1) und (vv_2) gecovert werden müssen.

Zur Berechnung von $IN(v)$ berechnen wir $\max\{IN(v_1), OUT(v_2)\} + \max\{IN(v_2), OUT(v_2)\}$, da die Kanten (vv_1) und (vv_2) bereits durch v gecovert sind und wir daher die Wahl haben, ob wir v_1 bzw. v_2 in das Vertex Cover aufnehmen.

Diese Berechnung kann in $O(\deg v)$ Zeit durchgeführt werden, wenn die Werte $IN(v_1), ..., IN(v_{\deg v})$ und $OUT(v_1), ..., OUT(v_{\deg v})$ bereits bekannt sind. Für alle Knoten ausgeführt, erhält man $O(n)$, denn die Anzahl von Kanten im Baum beträgt $n - 1$.

5.2 Definition

Sei $G = (V, E)$ ein Graph. Eine *tree decomposition* von G ist ein Paar $\widetilde{T} = (\{X_i : i \in I\}, T)$, wobei jedes X_i Teilmengen von V sind, die man *bags* nennt, und T ein Baum mit der Knotenmenge I ist. Außerdem müssen noch weitere Eigenschaften gelten:

(i) Es gilt $\bigcup_{i \in I} X_i = V$.

(ii) Für jede Kante $(uv) \in E$ gibt es ein $i \in I$ mit $\{u, v\} \subseteq X_i$.

(iii) Für alle $i, j, k \in I$ so dass j auf einem Pfad zwischen i und k in T liegt, gilt $X_i \cap X_k \subseteq X_j$.

Die *width* einer *tree decomposition* wird definiert als $\max\{|X_i| : i \in I\} - 1$. Die *tree width* $\mathrm{tw}(G)$ eines Graphen G ist die kleinste natürliche Zahl w, für die eine tree decomposition mit width w existiert.

5.3 Beispiele

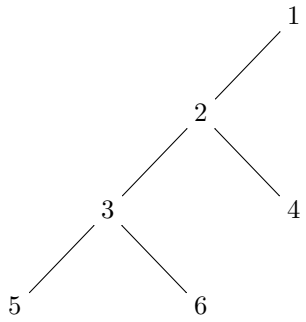


Abbildung 5.1: Beispiel (a): Eine tree decomposition dieses Graphen ist G selbst, wobei jedes bag zwei Knoten enthält. Die Menge der bags ist dann $\{\{1,2\},\{2,3\},\{2,4\},\{3,5\},\{3,6\}\}$.

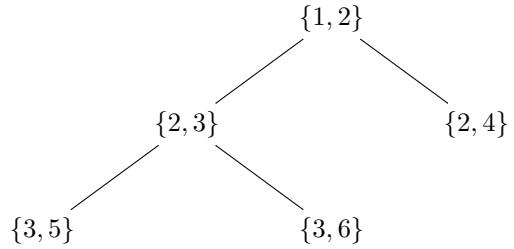


Abbildung 5.2: Die bags der tree decomposition aus G wie oben definiert.

Wie im Beispiel gesehen, haben alle Bäume stets tree width 1. Als nächstes Beispiel (b) betrachten wir einen Kreis mit n Knoten.

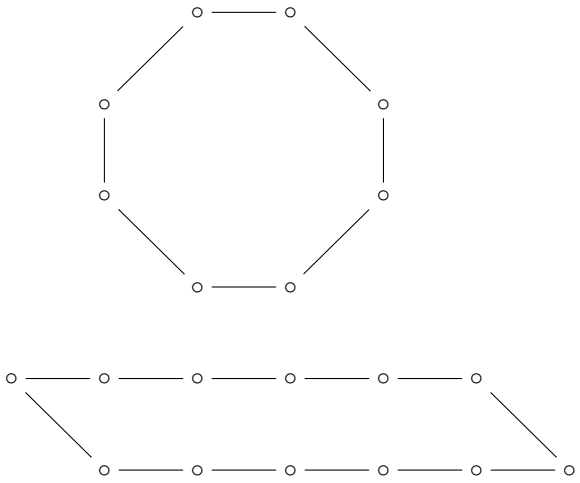


Abbildung 5.3: Beispiel (b): Zwei Bilder des selben Graphen C_n. Als bags wählt man die Mengen aus den drei linken und drei rechten Knoten, sowie jeweils 3 nebeneinanderliegende Knoten sowie einen gegenüberliegenden.

Abbildung 5.4: Bags der tree decomposition im Kreis C_n, also tree width 2.

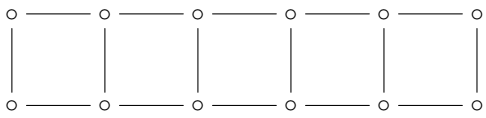


Abbildung 5.5: Beispiel (c): Eine Leiter G, die bags der tree decomposition sind wiederum 3-Mengen aus „L“ wie beim Kreis C_n

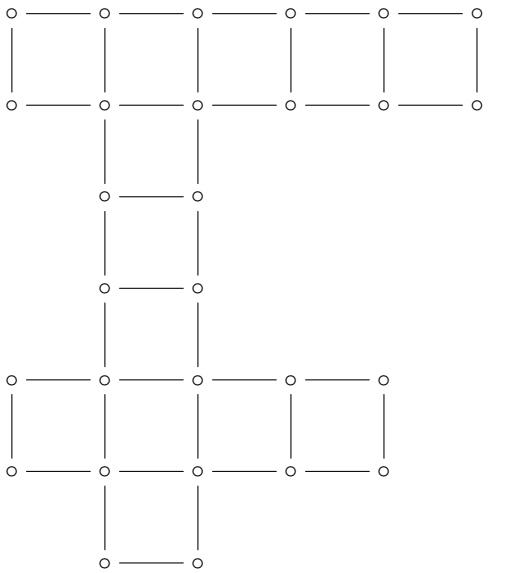


Abbildung 5.6: Beispiel (c'): Eine modifizierte Leiter G, die bags der tree decomposition sind 4-Mengen aus benachbarten Knoten (quadratisch angeordnet)

Beispiel (d): Der vollständige Graph K_n. Seine tree width beträgt $n-1$, da alle Knoten in ein bag der tree decomposition müssen.

Beispiel (e): Ein Gitter-Graph G mit $k \cdot l$ Knoten. Als Bags wählen wir jeweils zwei benachbarte Spalten, damit erhalten wir eine obere Schranke von $\mathrm{tw}(G) \leq 2 \cdot \min\{k, l\}$. 17.7.12

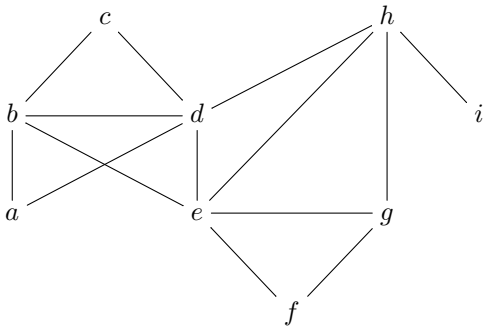


Abbildung 5.7: Beispiel (f)

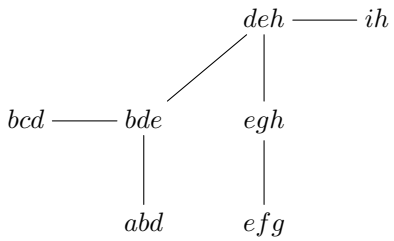


Abbildung 5.8: Tree decomposition für Beispiel (f)

Satz 12. *Planare Graphen haben tree width* $O(\sqrt{n})$.

Satz 13. *Eine tree decomposition der width k eines Graphen G mit tree width kleiner oder gleich k kann in* $2^{O(k^3)} \cdot n$ *Zeit berechnet werden.*

5.4 Verbessertes dynamisches Programm für Vertex Cover für Graphen mit beschränkter Tree Width

Satz 14. *Für einen Graphen G mit einer tree decomposition* $\widetilde{T} = (\{X_i : i \in I\}, T)$ *mit width k kann ein optimales vertex cover in* $O(2^k k|I|)$ *Zeit berechnet werden.*

Wir verwenden die folgende Idee. Für jeden bag X_i überprüfen wir für alle $2^{|X_i|}$ Möglichkeiten, ob diese ein Vertex Cover für $G[X_i]$ darstellen. Diese Information speichern wir in der Tabelle A_i. Das optimale Vertex Cover berechnen wir dann mit einem bottom-up dynamischen Programm in $\widetilde{T}$. Die Schwierigkeit dabei ist die Sicherstellung der Konsistenz durch das Aktualisieren der Tabelle beim Verarbeiten der Kanten. Dazu verwenden wir folgende Schritte:

0. **table creation.** Für jeden bag $X_i = \{x_{i1}, ..., x_{in}\}, n_i = |X_i|$ legen wir eine Tabelle A_i folgender Form mit 2^{k_i} Zeilen an:

x_{i1}	x_{i2}	$\cdots$	$x_{i_{n_i}}$	m_i
0	0	$\cdots$	0	
0	0	$\cdots$	1	
$\vdots$			$\vdots$	
1	1	$\cdots$	1	

Eine *Zuordnung* für ein bag X_i ist eine Abbildung $C_i : X_i \to \{0,1\}$, die für jeden Knoten in X_i festlegt, ob er im Vertex Cover enthalten ist. Die Tabelle A_i hat eine Zeile für jede mögliche Zuordnung von X_i. Wir setzen nun $V_i := \bigcup_{j \in T_i} X_i$. Ziel ist es, für jede Zuordnung C_i von X_i den Wert

$$m_i(C_i) := \min\{|V'| : V' \subseteq V_i \text{ ist ein Vertex Cover für } G[V_i] \text{ das } C_i \text{ respektiert}\}$$

berechnen. *C_i wird von V' respektiert*, wenn $V' \cap V_i = C_i^{-1}(1)$ gilt.

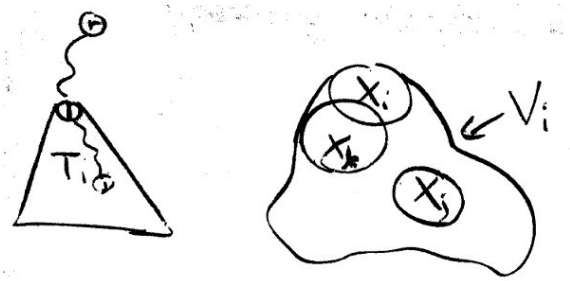


Abbildung 5.9: Skizze zum Vorgehen bei der Berechnung des Vertex Cover durch ein dynamisches Programm

1. **table init.** Sei X_i ein bag und C_i eine Zuordnung für X_i. Wir nennen C_i genau dann *gültig*, wenn $C_i{-}1(1)$ ein Vertex Cover für $G[X_i]$ ist. Wir initialisieren für jede Zuordnung C_i den Wert m_i nach der Vorschrift

$$m_i(C_i) := \begin{cases} |C_i^{-1}(1)| & (C_i \text{ ist gültig}), \\ \infty & (\text{sonst}). \end{cases}$$

2. **dynamic program.** Wir durchlaufen den Baum bottom-up und aktualisieren „adjazente“ Tabellen gegeneinander. Sei $i \in I$ der Vater von $j \in I$. Wir nehmen weiterhin an, dass

$$\begin{aligned} X_i &= \{z_1, ..., z_s, v_1, ..., v_{t_i}\}, \\ X_j &= \{z_1, ..., z_s, u_1, ..., u_{t_j}\}. \end{aligned}$$

   Es folgt unmittelbar $X_i \cap X_j = \{z_1, ..., z_s\}$. Eine *Erweiterung* einer Zuordnung $C : W \to \{0,1\}$ ist eine Zuordnung $\widetilde{C} : \widetilde{W} \to \{0,1\}$ mit $\widetilde{W} \supseteq W$ und $\widetilde{C}|_W = C$.

   Für jede Zuordnung $C : \{z_1, ..., z_s\} \to \{0,1\}$ und jede Erweiterung $C_i : X_i \to \{0,1\}$ aktualisieren wir gemäß

$$m_i(C_i) := m_i(C_i) + \min\{m_j(C_j) : C_j : X_j \to \{0,1\} \text{ ist Erweiterung von } C\} - |C^{-1}(1)|.$$

   Der Term $|C^{-1}(1)|$ vermeidet Doppelzählungen.

   Hat i mehrere Kindknoten $j_1, ..., j_l$ in T, dann wird A_i für jedes Kind analog aktualisiert. Schritt 2 stoppt, wenn die Wurzel des Baumes erreicht wurde.

3. **result.** Die Größte des minimalen Vertex Cover kann aus der Tabelle A_r der Wurzel des Baumes abgelesen werden, denn es ist gegeben durch

$$\min\{m_r(C_r) : C_r \text{ ist eine Zuordnung für } X_i\}.$$

   Durch Speichern der Vertex Cover der Teilbäume für jede Zurordnung und jeden bag kann man auch das Vertex Cover selbst berechnen.

Für die Laufzeitanalyse genügt es zu zeigen, dass die Aktualisierung in $O(2^k k)$ durchzuführen ist.