

Julius-Maximilians-Universität Würzburg  
Fakultät für Mathematik und Informatik

**SS12, Dr. Spoerhase**

# **Exakte Algorithmen**

Nils Wisiol

5. Juni 2012

# Inhaltsverzeichnis

1	Einführung	3
2	Dynamisches Programmieren	4
3	Inklusion-Exklusion	6
4	Measure & Conquer	9

# 1 Einführung

Hier fehlt noch et-  
was.

## 2 Dynamisches Programmieren

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

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

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

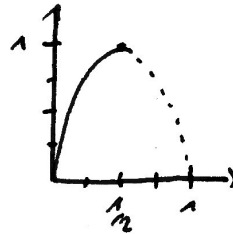


Abbildung 2.1: Graph des Binomialkoeffizienten

*Beweis.* Es ist  $\sum_{i=0}^{\alpha n} \binom{n}{i} \leq n \binom{n}{\alpha n} = O^*(\binom{n}{\alpha n})$ , denn die Binomialkoeffizienten  $\binom{n}{b}$  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 2.** Eine kleinste dominierende Menge lässt sich in  $O(1,7088^n)$  ermitteln.

*Beweis.* Zunächst bestimmen wir eine nicht-erweiterbare unabhängige 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  $O^*(2^{(1-\alpha)n})$ .

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.

Hier fehlt noch et-  
was  
15.5.12  
Hier fehlt noch et-  
was

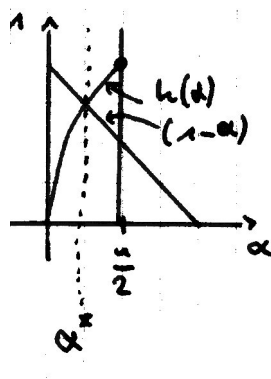
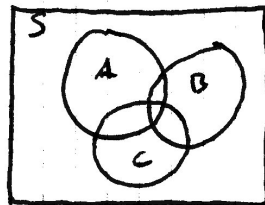


Abbildung 2.2: Bestimmung von  $\alpha^*$  als Maximum der zwei möglichen Funktionen

### 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  $P = \{P_1, \dots, P_n\}$  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  $N(\emptyset)$  bezeichnen wir die Anzahl der Objekte, die keine der Eigenschaften erfüllen. Wir können oben skizzierte Formel dann verallgemeinern, es ergibt sich

**Satz 3.**  $N(\emptyset) = \sum_{S \subseteq P} (-1)^{|S|} N(S) = N(\emptyset) + \sum_i N(P_i) + \sum_{i < j} N(P_i, P_j) + \dots$

*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  $S = \{P_{i_1}, \dots, P_{i_s}\}$ ,  $|S| = s$ . 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.  $\square$

22.5.12

**Korollar 4.**  $N(P) = \sum_{S \subseteq P} (-1)^{|S|} \overline{N}(S)$ , 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  $N'(\emptyset) = N(P)$  und  $N'(P'_1, \dots, P'_k) = \overline{N}(P_1, \dots, P_k)$ . Es folgt die Behauptung.  $\square$

**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 5.** 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  $k = 0, \dots, n - 1$  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.  $\square$

**Graph-Färbung.** Wir betrachten nun das Problem der Graph-Färbung. Als Eingabe sei ein Graph  $G = (V, E)$  gegeben. Eine *zulässige Färbung* weist 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  $I_1, \dots, I_k$ , die  $V$  partitionieren, formulieren. Dies ist verwandt 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  $(S_1, \dots, S_k) \in S^k$  ein *geordnetes  $k$ -Cover*, falls  $\bigcup_{i=1}^k S_i = U$  und geben einen Algorithmus an, der die Anzahl der geordneten  $k$ -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 6.** 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  $(S_1, \dots, S_k)$  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  $v \notin \bigcup_{i=1}^k S_i$ . 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  $W$  vermeiden, 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 3 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.  $\square$

**Satz 7.** 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  $U = \{u_1, \dots, u_n\}$  beliebig. Dann sei für alle  $W \subseteq U$

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

Dies ist die Anzahl der Mengen, die  $W$  vermeiden, und alle Elemente  $\{u_1, \dots, u_i\} \setminus W$  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  $i = n, \dots, 0$  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.  $\square$



## 4 Measure & Conquer

Measure and Conquer bezeichnet ein Verfahren, um Rekursionsgleichungen 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 1:** Algorithmus zur Berechnung der größten Anzahl unabhängiger Mengen

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