

Näherungsalgorithmen (Approximationsalgorithmen)

WiSe 2024/25 in Trier

Henning Fernau

Universität Trier

fernau@uni-trier.de

7. November 2024

Näherungsalgorithmen

Gesamtübersicht

- Organisatorisches
- Einführung / Motivation
- Grundtechniken für Näherungsalgorithmen
- Approximationsklassen (Approximationstheorie)

Zusammenfassung des bisher Beobachteten:

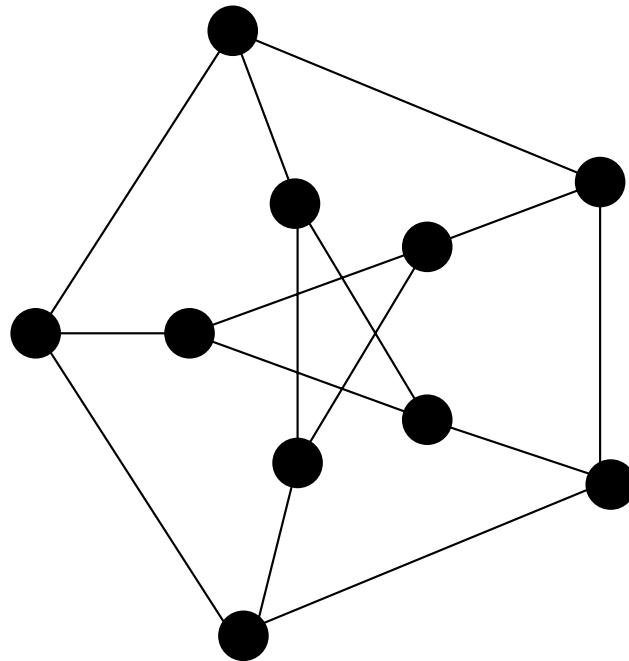
- Bisher kennengelernte Näherungsalgorithmen sind oft **sehr kurz**
- Schwierigkeit: Analyse, betreffend die **Güte** der Näherung!
- Warum sind die Gütegarantien schwierig zu beweisen?
Grundsätzlich (und intuitiv) lässt sich dies genau so begründen wie die vermutete Ungleichheit von P und NP.

Knotenüberdeckungsproblem

- + Es ist leicht, eine vorgegebene Lösung insofern zu verifizieren, als dass die Knotenüberdeckungseigenschaft überprüft wird.
- Es ist schwierig nachzuweisen, dass eine gefundene Knotenüberdeckung kleinstmöglich (optimal) ist.

Konkret: Betrachten Sie den Petersen-Graph.

Der Petersen-Graph



Ein *Optimierungsproblem* \mathcal{P} wird beschrieben durch ein Quadrupel $(I_{\mathcal{P}}, S_{\mathcal{P}}, m_{\mathcal{P}}, \text{opt}_{\mathcal{P}})$:

1. $I_{\mathcal{P}}$ ist die Menge der möglichen Eingaben (*Instanzen*),
2. $S_{\mathcal{P}} : I_{\mathcal{P}} \rightarrow$ Menge der *zulässigen Lösungen* (feasible solutions),
3. $m_{\mathcal{P}} : (x, y) \mapsto m_{\mathcal{P}}(x, y) \in \mathbb{N}$ (oder \mathbb{Q}, \dots) für $x \in I_{\mathcal{P}}, y \in S_{\mathcal{P}}(x)$ liefert den *Wert* der zulässigen Lösung y und
4. $\text{opt}_{\mathcal{P}} \in \{\min, \max\}$: \mathcal{P} Minimierungs- oder Maximierungsproblem?

$I_{\mathcal{P}}$ und $S_{\mathcal{P}}(x)$ sind —geeignet codierte— formale Sprachen über Alphabet $\{0, 1\}$.

Weitere Bezeichnungen

$S_{\mathcal{P}}^* : I_{\mathcal{P}} \rightarrow$ Menge der *bestmöglichen Lösungen* (optimum solution), d.h.

$$\forall x \in I_{\mathcal{P}} \ \forall y^*(x) \in S_{\mathcal{P}}^*(x) : m_{\mathcal{P}}(x, y^*(x)) = \text{opt}_{\mathcal{P}}\{m_{\mathcal{P}}(x, z) \mid z \in S_{\mathcal{P}}(x)\}.$$

Der Wert einer bestmöglichen Lösung wird auch $m_{\mathcal{P}}^*(x)$ notiert.

Ist \mathcal{P} aus dem Zusammenhang klar, so schreiben wir kurz —unter Fortlassung des Indexes \mathcal{P} — $I, S, m, \text{opt}, S^*, m^*$.

Hinweis: Wir benutzten schon früher C^* für eine kleinstmögliche Überdeckung.

Problemvarianten

Konstruktionsproblem (construction problem):

\mathcal{P}_C : Ggb. $x \in I_{\mathcal{P}}$, liefere ein $y^*(x) \in S_{\mathcal{P}}^*(x)$ sowie ihren Wert $m_{\mathcal{P}}^*(x)$

Auswertungsproblem (evaluation problem):

\mathcal{P}_E : Ggb. $x \in I_{\mathcal{P}}$, liefere $m_{\mathcal{P}}^*(x)$

Entscheidungsproblem (decision problem):

\mathcal{P}_D : Ggb. $x \in I_{\mathcal{P}}$ und Parameter $k \in \mathbb{N}$, entscheide ob

$$m_{\mathcal{P}}^*(x) \geq k, \quad (\text{falls } \text{opt}_{\mathcal{P}} = \max)$$

bzw. ob

$$m_{\mathcal{P}}^*(x) \leq k, \quad (\text{falls } \text{opt}_{\mathcal{P}} = \min).$$

Ein Beispiel: das Knotenüberdeckungsproblem VC

1. $I = \{G = (V, E) \mid G \text{ ist Graph} \}$
2. $S(G) = \{U \subseteq V \mid \forall \{x, y\} \in E : x \in U\}$ (Knotenüberdeckungseigenschaft)
3. $m = |U|$
4. $\text{opt} = \min$

VC_D ist das entsprechende „parametrisierte Problem“, d.h., ggb. Graph $G = (V, E)$ und Parameter k , gibt es eine Knotenüberdeckung $U \subseteq V$ mit $|U| \leq k$?

Die Klassen PO und NPO

$\mathcal{P} = (I, S, m, \text{opt})$ gehört zu NPO, gdw:

1. $x \stackrel{?}{\in} I$ ist in Polynomialzeit entscheidbar.
2. Es gibt ein Polynom q derart, dass $\forall x \in I \ \forall y \in S(x) : |y| \leq q(|x|)$ und für alle y mit $|y| \leq q(|x|)$ ist die Frage $y \stackrel{?}{\in} S(x)$ in Polynomialzeit entscheidbar.
3. m ist in Polynomialzeit berechenbar.

Beispiel VC: zu Punkt 2: Jede Knotenüberdeckung ist in ihrer Größe trivialerweise durch die Mächtigkeit der Knotenmenge beschränkt.

Satz 1: Ist $\mathcal{P} \in \text{NPO}$, so ist $\mathcal{P}_D \in \text{NP}$.

Beweis: (nur für $\text{opt}_{\mathcal{P}} = \max$), \mathcal{P}_D lässt sich bei Eingabe von $x \in I$ und k wie folgt lösen (in nichtdeterministischer Weise):

1. Rate y mit $|y| \leq q(|x|)$ in Zeit $\mathcal{O}(q(|x|))$. ($q(|x|)$ Bits sind zu raten)
2. Teste $y \in S(x)$ in Polynomialzeit.
3. Falls $y \in S(x)$, berechne $m(x, y)$ in Polynomialzeit.
4. Falls $y \in S(x)$ und $m(x, y) \geq k$, antworte JA.
5. Falls $y \notin S(x)$ oder $m(x, y) < k$, antworte NEIN.

Die Klasse PO

Ein Optimierungsproblem \mathcal{P} gehört zur Klasse PO gdw. \mathcal{P}_C in Polynomialzeit gelöst werden kann.

Beachte: PO über das zugehörige Konstruktionsproblem definiert!

Erinnerung: (polynomielle) **Turing-Reduzierbarkeit**

$\mathcal{P}_1 \leq_T^{(p)} \mathcal{P}_2$ gdw.

es gibt eine Turing-Maschine, die eine Instanz x von \mathcal{P}_1 (in Polynomialzeit) mit Hilfe von Orakelanfragen bearbeiten kann.

Orakelanfragen sind generierte Instanzen von \mathcal{P}_2 , die auf ein „Orakelband“ geschrieben werden und angenommenerweise in konstanter Zeit beantwortet werden.

Wir schreiben $\mathcal{P}_1 \equiv_T^p \mathcal{P}_2$, falls sowohl $\mathcal{P}_1 \leq_T^p \mathcal{P}_2$ als auch $\mathcal{P}_2 \leq_T^p \mathcal{P}_1$ gelten.

Satz 2: $\forall \mathcal{P} \in \text{NPO} : \mathcal{P}_D \equiv_T^p \mathcal{P}_E \leq_T^p \mathcal{P}_C$

Beweis: Klar: $\mathcal{P}_D \leq_T^p \mathcal{P}_E \leq_T^p \mathcal{P}_C$.

Z.z: $\mathcal{P}_E \leq_T^p \mathcal{P}_D$.

Dazu überlegen wir uns:

$$\{m_{\mathcal{P}}(x, y) \mid y \in S_{\mathcal{P}}(x)\} \subseteq 0 \dots 2^{p(|x|)} \quad (1)$$

für ein Polynom p .

Dann kann \mathcal{P}_E durch binäre Suche auf dem Intervall $0 \dots 2^{p(|x|)}$ mit $p(|x|)$ vielen Orakelanfragen an \mathcal{P}_D gelöst werden.

Da $\mathcal{P} \in \text{NPO}$, ist $m_{\mathcal{P}}(x, y)$ in Zeit $r(|x|, |y|)$ für ein Polynom r berechenbar. \leadsto

$$0 \leq m_{\mathcal{P}}(x, y) \leq 2^{r(|x|, |y|)}$$

Da $\mathcal{P} \in \text{NPO}$, ist $|y| \leq q(|x|)$ für alle $y \in S_{\mathcal{P}}(x)$ für ein Polynom q .

Daher gilt für das Polynom $p(n) := r(n, q(n))$ die Beziehung (1).

Das Beispiel MAXCLIQUE

Ein *vollständiger Graph* mit n Knoten ist (isomorph zu, i.Z. \cong)

$$K_n = (\{1, \dots, n\}, \{\{i, j\} \mid 1 \leq i < j \leq n\}).$$

Eine *Clique* in einem Graphen ist eine Knotenteilmenge, die einen vollständigen Graphen induziert. Das MAXCLIQUE-Problem fragt in einem Graphen nach größtmöglichen (maximum) Cliques.

Unter Benutzung des oben eingeführten Formalismus zur Spezifizierung von Optimierungsproblemen stellt sich MAXCLIQUE wie folgt dar:

I : alle Graphen $G = (V, E)$

S : alle Cliques in G , d.h. $U \subseteq V : G[U] \cong K_{|U|}$ (vollständiger Graph mit $|U|$ Knoten)

$m = |U|$

opt = max

Algorithmus für $MAXCLIQUE_C \leq_T^p MAXCLIQUE_E$ (Selbstreduktion)

Eingabe: Graph $G = (V, E)$

Ausgabe: eine größtmögliche Clique in G

begin

$k := MAXCLIQUE_E(G);$

Falls $k = 1$ **liefere** irgendeinen Knoten in V

sonst finde Knoten $v \in V$ mit $k = MAXCLIQUE_E(G(N[v]))$

und **liefere** $\{v\} \cup MAXCLIQUE_C(G(N(v)))$.

end

Warum ist das Programm korrekt ?

- Die Auswahl „ $v \in V$ mit $k = MC_E(G(N[v]))$ “ garantiert, dass v in einer größtmöglichen Clique liegt.
- Es ist klar, dass $G(N(v))$ eine Clique (evtl. mehrere) der Größe $k-1$ enthält (und auch keine größere); eine dieser wird rekursiv gefunden.

Zeitkomplexität des Algorithmus ($n = \#$ Knoten):

$$\begin{aligned} T(1) &= \mathcal{O}(1) \\ T(n) &= (n+1) + T(n-1) \\ &\quad \hookrightarrow \text{Durchsuchen des Graphen} \\ &= (n+1) + n + \dots + n + \mathcal{O}(1) = \mathcal{O}(n^2) \end{aligned}$$

Satz 3: Ist $\mathcal{P} \in \text{NPO}$ und ist \mathcal{P}_D NP-schwer, so gilt: $\mathcal{P}_C \leq_T \mathcal{P}_D$.

Beweis: (für $\text{opt}_{\mathcal{P}} = \max$).

Im Beweisgang werden wir ein anderes NPO-Problem \mathcal{P}' konstruieren mit $\mathcal{P}_C \leq_T^p \mathcal{P}'_E$. Wegen Satz 1 ist \mathcal{P}_D nach Voraussetzung NP-vollständig, also gilt $\mathcal{P}'_E \leq_T^p \mathcal{P}'_D \leq_T^p \mathcal{P}_D$ wegen Satz 2, und die Transitivität der Reduktionsrelation liefert die Behauptung.

\mathcal{P}' ist wie \mathcal{P} definiert mit Ausnahme der Messfunktion $m_{\mathcal{P}'}$. Betrachte dazu ein Polynom q mit $\forall x \in I_{\mathcal{P}} \forall y \in S_{\mathcal{P}}(x), |y| \leq q(|x|)$ (s. Definition NPO). Mit anderen Worten: $S_{\mathcal{P}}(x) \subseteq \{0, 1\}^{\leq q(|x|)}$.

Setze nun $\tau(y) := 2^{q(|x|)-|y|}y$. Für jedes $y \in S_{\mathcal{P}}(x)$ bezeichne $\lambda(y)$ den Wert von $\tau(y)$, interpretiert als Ternärzahl. Daraus folgt: $0 \leq \lambda(y) \leq 3^{q(|x|)}$.

Außerdem gilt: $\forall y, y' \in S_{\mathcal{P}}(x) : y = y' \iff \tau(y) = \tau(y') \iff \lambda(y) = \lambda(y')$.

Setze für jedes $x \in I_{\mathcal{P}'} = I_{\mathcal{P}}$ und jedes $y \in S_{\mathcal{P}'} = S_{\mathcal{P}}$:

$$m_{\mathcal{P}'}(x, y) = 3^{q(|x|)+1}m_{\mathcal{P}}(x, y) + \lambda(y).$$

Beachte: $\mathcal{P}' \in \text{NPO}$, denn $m_{\mathcal{P}'}$ ist in Polynomialzeit berechenbar, weil insbesondere die Exponentiation $3^{q(|x|)}$ nur $\mathcal{O}(q(|x|))$ viele Bits benötigt.

Damit: $\forall x \forall y_1, y_2 \in S_{\mathcal{P}'}(x) : y_1 \neq y_2 \rightarrow m_{\mathcal{P}'}(x, y_1) \neq m_{\mathcal{P}'}(x, y_2)$.

Also gibt es eine eindeutig bestimmte maximale zulässige Lösung $y_{\mathcal{P}'}^*(x)$ in $S_{\mathcal{P}'}^*(x)$. Nach Definition von $m_{\mathcal{P}'}$ gilt ferner:

$$(m_{\mathcal{P}'}(x, y_1) > m_{\mathcal{P}'}(x, y_2) \Rightarrow m_{\mathcal{P}}(x, y_1) \geq m_{\mathcal{P}}(x, y_2)).$$

Daraus folgt: $y_{\mathcal{P}'}^*(x) \in S_{\mathcal{P}}^*(x)$.

$y_{\mathcal{P}}^*(x)$ kann wie folgt mit einem Orakel für \mathcal{P}'_E in Polynomialzeit berechnet werden.

1. Bestimme $m_{\mathcal{P}'}^*(x)$ (Orakel!)
2. Berechne $\lambda(y_{\mathcal{P}'}^*(x)) = m_{\mathcal{P}'}^*(x) \bmod 3^{q(|x|)+1}$.
3. Bestimme y aus $\lambda(y)$.

Daher kann \mathcal{P}_C mit einem Orakel für \mathcal{P}'_E in Polynomialzeit berechnet werden, denn $m_{\mathcal{P}}^*(x) = m_{\mathcal{P}}(x, y)$ für das mit oben stehendem Algorithmus berechnete y .

Intermezzo: Reduktionen

Reduktionen werden zumeist für Schwerebeweise eingesetzt.

Da es sich dabei aber um Polynomialzeitalgorithmen handelt, kann man aber Reduktionen auch einsetzen, um positive algorithmische Ergebnisse zu erzielen. Das machen Sie alltäglich beim Benutzen von Unterprozeduren.

Klappt das auch (evtl.) für Approximationsalgorithmen?

Ein Beispiel: MinSAT

1. $I = \{(F, X) \mid F \text{ ist Boolesche Formel in KNF, Variablenmenge } X\}$

Wir können F als Menge von Klauseln und jede Klausel als Menge von Literalen auffassen.

2. $S(G) = \{0, 1\}^X$ (Belegungsfunktionen)

3. $m = |\{C \in F \mid \alpha(C) = 1\}|$

4. $\text{opt} = \min$

MinSAT_D ist das entsprechende „parametrisierte Problem“, d.h. ggb. (F, X) und Parameter k , gibt es Belegung $\alpha : X \rightarrow \{0, 1\}$ mit $|\{C \in F \mid \alpha(C) = 1\}| \leq k$?

Lemma Zu gegebener VC_D -Instanz (V, E, k) gibt es eine äquivalente MinSAT_D -Instanz (F, X, k) .

Konstruktion: Orientiere die Kanten willkürlich aber eindeutig. Setze $X = E$.
Klauseln entsprechen Mengen inzidenter Kanten (also Knoten).

Genauer:

Es gibt Bijektion f zwischen Knotenmenge V und Klauselmenge C , sodass:

$f(v)$ enthält Literal e gdw. Kante e zeigt auf v ;

$f(v)$ enthält Literal $\neg e$ gdw. Kante e zeigt von v weg.

Reduktionseigenschaft:

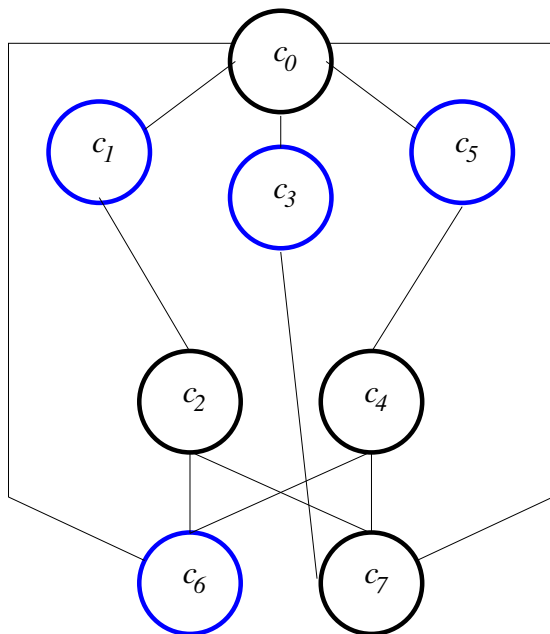
$U \subseteq V$ ist Knotenüberdeckung von (V, E) gdw.

\overline{U} ist unabhängig gdw.

$\overline{f(U)} = f(\overline{U})$ wird nicht erfüllt.

Reduktion am Beispiel durch topologisches Sortieren (im Bild)

$e_0 \mapsto 0, e_1 \mapsto 0, e_2 \mapsto 0, e_3 \mapsto 0, e_4 \mapsto 1, e_5 \mapsto 1, e_6 \mapsto 1, e_7 \mapsto 0, e_8 \mapsto 0$
 e_9, e_{10}, e_{11} eigentlich überflüssig, c_1, c_3, c_5, c_6 unabhängige Menge



$$\begin{aligned}
 c_0 &: \neg e_0 \vee \neg e_1 \vee \neg e_2 \vee \neg e_3 \vee \neg e_{11} \mapsto 1 \\
 c_1 &: (e_1 \vee \neg e_4) \mapsto 0 \\
 c_2 &: (e_4 \vee \neg e_7 \vee \neg e_9) \mapsto 1 \\
 c_3 &: (e_2 \vee \neg e_5) \mapsto 0 \\
 c_4 &: (e_6 \vee \neg e_8 \vee \neg e_{10}) \mapsto 1 \\
 c_5 &: (e_3 \vee \neg e_6) \mapsto 0 \\
 c_6 &: (e_0 \vee e_7 \vee e_8) \mapsto 0 \\
 c_7 &: (e_5 \vee e_9 \vee e_{10} \vee e_{11}) \mapsto 1
 \end{aligned}$$

Lemma Zu gegebener MinSAT_D -Instanz (F, X, k) gibt es eine äquivalente VC_D -Instanz (V, E, k) .

Übersetzung einer Min-2SAT-Instanz am Beispiel in einen Klauselgraphen

$$c_0 : (\neg w \vee \neg z)$$

$$c_1 : (w \vee \neg x)$$

$$c_2 : (x \vee y)$$

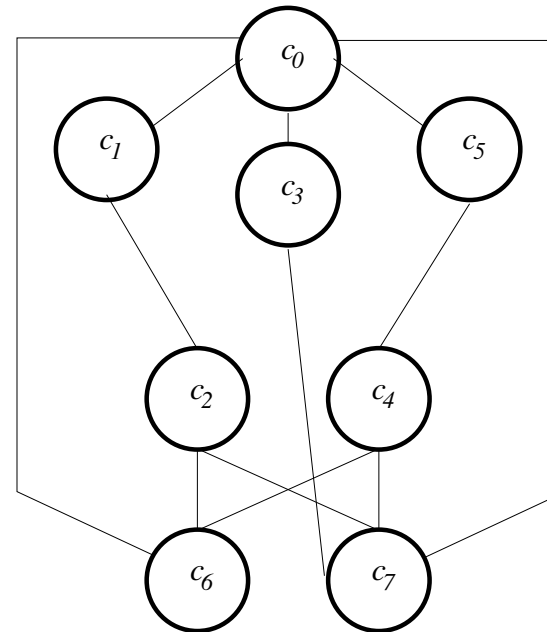
$$c_3 : (y \vee w)$$

$$c_4 : (y \vee \neg z)$$

$$c_5 : (w \vee z)$$

$$c_6 : (\neg x \vee z)$$

$$c_7 : (w \vee \neg y)$$



Eine Beispielbelegung

$w \mapsto 0, x \mapsto 1, y \mapsto 0, z \mapsto 0$

$$c_0 : (\neg w \vee \neg z) \mapsto 1$$

$$c_1 : (w \vee \neg x) \mapsto 0$$

$$c_2 : (x \vee y) \mapsto 1$$

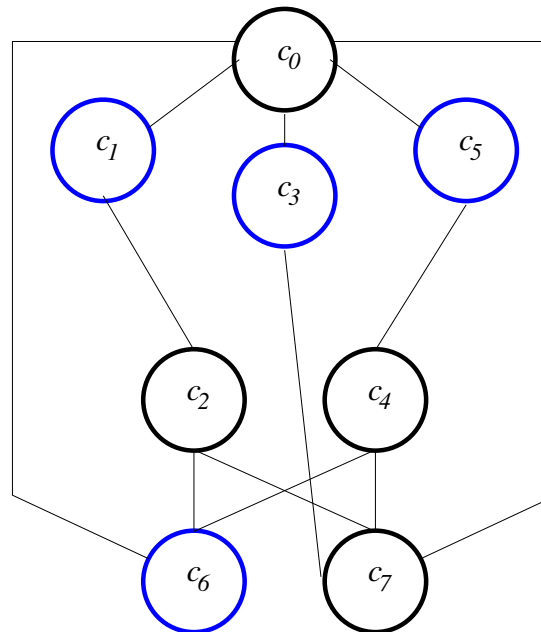
$$c_3 : (y \vee w) \mapsto 0$$

$$c_4 : (y \vee \neg z) \mapsto 1$$

$$c_5 : (w \vee z) \mapsto 0$$

$$c_6 : (\neg x \vee z) \mapsto 0$$

$$c_7 : (w \vee \neg y) \mapsto 1$$



Reduktionen für Approximationen?

- Warum ist eine Faktor- c Approximation für MinSAT auch eine gleich gute, d.h. Faktor- c , Approximation für VC?
- Gilt auch die Umkehrung?
- Lassen sich auch die Algorithmen übersetzen?
- Ist das notwendig?
- Später mehr und genauer!

Grundtechniken

- Greedy-Verfahren
- Partitionsprobleme
- Lokale Suche
- Lineares Programmieren
- Dynamisches Programmieren

Allgemeines zu Greedy-Verfahren

hier speziell bei Maximierungsverfahren

Allgemeine Aufgabe:

Aus einer Grundmenge X ist eine maximale zulässige Lösung S_{max} zu finden.

Voraussetzung: Die Menge der zulässigen Lösungen ist *monoton*, d.h., falls S zulässige Lösung ist, so auch $S' \subseteq S$ für alle $S' \subseteq S$.

Damit ist auch \emptyset eine zulässige (Ausgangs-)Lösung.

X sei nach einem geeigneten Kriterium sortiert.

Allgemeine Vorgehensweise:

Für jedes Element x der Liste X wird für die bislang gefundene zulässige Lösung S' geprüft, ob $S' \cup \{x\}$ ebenfalls zulässig ist.

Wenn ja, wird $S' := S' \cup \{x\}$ gebildet.

Daher werden stets nur zulässige Lösungen ausgegeben.

Außerdem ist klar, dass eine so gefundene zulässige Lösung S' nicht mehr durch Hinzunahme eines weiteren Elements $x \notin S'$ erweitert werden kann, denn wenn $\{x\} \cup S'$ zulässig wäre, dann auch $\{x\} \cup S''$ für alle $S'' \subseteq S'$ (Monotonie!);

speziell gälte dies für ein S'' , das als „bisherige Lösung“ an dem Punkt, als x untersucht wurde, gebildet worden war.

Allgemeines zu Greedy-Verfahren

hier speziell bei MAXCLIQUE

Allgemeine Aufgabe: Aus der Knotenmenge $X = V$ von G ist eine maximale zulässige Lösung C_{max} zu finden.

HIER: C zulässig gdw. C ist Clique.

Voraussetzung: Die Menge der zulässigen Lösungen ist monoton, d.h., falls S zulässige Lösung ist, so auch $S' \subseteq S$ für alle $S' \subseteq S$.

HIER: Jede Teilmenge einer Clique bildet eine Clique.

Damit ist auch \emptyset eine zulässige (Ausgangs-)Lösung.

X sei nach einem geeigneten Kriterium sortiert (Beispiel: absteigender Grad).

Allgemeine Vorgehensweise:

Für jedes Element x der Liste X wird für die bislang gefundene zulässige Lösung S' geprüft, ob $S' \cup \{x\}$ ebenfalls zulässig ist.

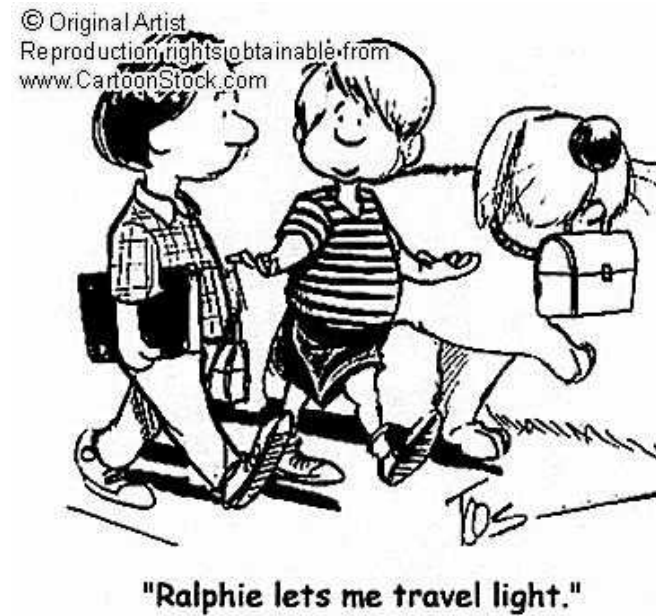
Wenn ja, wird $S' := S' \cup \{x\}$ gebildet.

Daher werden stets nur zulässige Lösungen ausgegeben.

Maximum Knapsack (Rucksackproblem)

- I : endliche Menge X bzw. „Liste“ $X = \{x_1, \dots, x_n\}$
Profitfunktion $p : X \rightarrow \mathbb{N}$ bzw. „Liste“ $\{p_1, \dots, p_n\}$
Größenfunktion $a : X \rightarrow \mathbb{N}$ bzw. „Liste“ $\{a_1, \dots, a_n\}$
Fassungsvermögen $b \in \mathbb{N}$ des Rucksacks
[O.E.: $\forall 1 \leq i \leq n : a_i \leq b$ im Folgenden]
- S : $\{Y \subseteq X \mid a(Y) = \sum_{x_i \in Y} a_i \leq b\}$
 m : $p(Y) = \sum_{x_i \in Y} p_i$
opt.: max

Mitteilung: Knapsack ist NP-vollständig.



Heuristische Idee: Es ist günstig, möglichst solche Sachen x_i in den Rucksack zu tun, die möglichst viel Profit versprechen im Verhältnis zum benötigten Platz.

\leadsto GreedyKnapsack (X, p, a, b)

1. Sortiere X absteigend nach dem Verhältnis $\frac{p_i}{a_i}$
($\{x_1, \dots, x_n\}$ sei die erhaltene sortierte Grundmenge)
2. $Y := \emptyset$ (die leere Menge ist zulässig)
3. Für $i := 1$ bis n tue
wenn $a_i \leq b$, dann $Y := Y \cup \{x_i\}$; $b := b - a_i$.
4. Liefere Y zurück.

GreedyKnapsack kann **beliebig schlechte Ergebnisse** liefern, wie folgendes Beispiel lehrt:

$$p : \{p_1 = 1, \dots, p_{n-1} = 1, p_n = b - 1\}$$

$$a : \{a_1 = 1, \dots, a_{n-1} = 1, a_n = b = k \cdot n\} \text{ für ein beliebig großes } k \in \mathbb{N}$$

Hier ist für $x = (X, p, a, b) : m^*(x) = b - 1 = k \cdot n - 1 > kn - k = k(n - 1)$ (durch Wahl des letzten Elements), aber GreedyKnapsack liefert, da

$$\frac{p_1}{a_1} = 1 = \dots = \frac{p_{n-1}}{a_{n-1}} > \frac{p_n}{a_n} = \frac{b - 1}{b}$$

$$m_{Greedy}(x) = \sum_{i=1}^{n-1} p_i = n - 1, \text{ d.h. } \frac{m^*(x)}{m_{Greedy}(x)} > k$$

Andere heuristische Idee fürs Rucksackfüllen:

X absteigend nach dem Profit sortieren. \leadsto GreedyKnapsack'(X, p, a, b)

1. Sortiere X absteigend nach dem Profit p_i .
($\{x_1, \dots, x_n\}$ sei die erhaltene sortierte Grundmenge)
2. $Y := \emptyset$ (die leere Menge ist zulässig)
3. Für $i := 1$ bis n tue
wenn $a_i \leq b$, dann $Y := Y \cup \{x_i\}$; $b := b - a_i$.
4. Liefere Y zurück.

Wähle als Beispiel $a_1 = \dots = a_{n-1} = 1$, $a_n = b - \epsilon$, $b = n - 1$, $p_1 = \dots = p_{n-1} = 1$, $p_n = 1 + \epsilon$. Wieder eine **beliebig schlechte** Lösungen möglich!

Erstaunlich: Kombinationsalgorithmus mit Gütegarantie

\leadsto GreedyKnapsackKombi(X, p, a, b)

1. $Y_1 := \text{GreedyKnapsack}(X, p, a, b)$
2. $Y_2 := \text{GreedyKnapsack}'(X, p, a, b)$
3. Wenn $p(Y_1) \geq p(Y_2)$, dann liefere Y_1 , sonst liefere Y_2 .

Eine andere (vereinfachte) Kombination lautet: GreedyKnapsack''(X, p, a, b)

1. Sortiere X absteigend nach Verhältnis $\frac{p_i}{a_i}$ ($\{x_1, \dots, x_n\}$: sortierte Grundmenge)
2. $Y := \emptyset$ (die leere Menge ist zulässig)
3. Für $i := 1$ bis n tue
wenn $a_i \leq b$, dann $Y := Y \cup \{x_i\}$; $b := b - a_i$.
4. " Suche x_{max} mit $\forall i : p_i \leq p_{max}$ (maximaler Profit!)
5. " Wenn $p(Y) \geq p_{max}$ dann liefere Y sonst liefere $\{x_{max}\}$.

Lemma: Für alle Instanzen x gilt:

$$m_{\text{GreedyKnapsack}''}(x) \leq m_{\text{GreedyKnapsackKombi}}(x)$$

Damit überträgt man die im folgenden Satz gezeigte Gütegarantie von Greedy-Knapsack'' auf GreedyKnapsackKombi.

Abkürzend schreiben wir:

$m_G, m_{G''}$ für $m_{\text{GreedyKnapsack}}$ bzw. $m_{\text{GreedyKnapsack}''}$.

Satz 1: Ist x eine MaximumKnapsack-Instanz, so ist

$$\frac{m^*(x)}{m_{G''}(x)} < 2.$$

Beweis: Es sei j der Index des ersten Elements, welches der Greedy-Algorithmus GreedyKnapsack *nicht* in den Rucksack tut. Es gilt:

$$\overline{p_j} := \sum_{i=1}^{j-1} p_i \leq m_G(x) \leq m_{G''}(x).$$

Ferner setzen wir:

$$\overline{a_j} := \sum_{i=1}^{j-1} a_i \leq b.$$

Behauptung: $m^*(x) < \overline{p_j} + p_j$.

Aus der Behauptung folgt die Aussage des Satzes, denn:

- Gilt $p_j \leq \overline{p_j}$, so ist $m^*(x) < 2\overline{p_j} \leq 2m_G(x) \leq 2m_{G''}(x)$.
- Gilt $p_j > \overline{p_j}$, so ist $m^*(x) < 2p_j \leq 2p_{\max} \leq 2m_{G''}(x)$.

Warum gilt die Behauptung?

Betrachte (kurzfristig) die folgende Verallgemeinerung des Rucksackproblems: Es sei nun erlaubt, „Bruchstücke“ der x_i (mit entsprechend der Größe skaliertem Gewinn) in den Rucksack zu tun. Der Wert einer optimalen Lösung für das neue Problem ist sicher eine obere Schranke für $m^*(x)$. Wie man leicht einsieht, ist

$$\overline{p}_j + (b - \overline{a}_j) \cdot \frac{p_j}{a_j}$$

der maximale Wert einer Lösung des variierten Problems, wenn die x_i nach $\frac{p_i}{a_i}$ absteigend sortiert vorliegen und j wie oben definiert ist.

$$\Rightarrow m^*(x) \leq \overline{p}_j + \underbrace{(b - \overline{a}_j)}_{< a_j} \cdot \frac{p_j}{a_j} < \overline{p}_j + p_j.$$

Tatsächlich liefert der Beweis zu Satz 1, dass der folgende (noch einfachere) Algorithmus eine $\frac{1}{2}$ -Approximation von Maximum Knapsack ist:

1. Sortiere X absteigend nach $\frac{p_i}{a_i}$.
2. $Y := \emptyset; i := 1;$
3. Solange $i \leq n$ und $a_i \leq b$, tue
 $Y := Y \cup \{x_i\}; b := b - a_i; i := i + 1.$
4. Wenn $a_i \leq b$, dann liefere Y ,
 sonst, wenn $p_i \leq p(Y)$, dann liefere Y ,
 sonst liefere $\{x_i\}.$

Hinweise: (1) Verallgemeinerung von Problemen, spezieller Relaxation des Zahlbereichs, ist oft ein Ansatz, um Schranken für Approximationsgüte zu gewinnen.

(2) Es gibt ein eigenes Buch zu diesem Thema: S. Martello, P. Toth: Knapsack Problems; Algorithms and Computer Implementations, Wiley, 1990.

Was haben wir gelernt? Sinnvolle Lektionen...

- Greedy kann sinnvoll für die Entwicklung von Näherungsverfahren sein.
- „Leichte Abänderungen“ von sinnvoll erscheinenden Heuristiken können aus einem im schlimmsten Fall schlechten Verfahren (ohne Approximationsgarantie) eines mit Gütegarantien machen.
- Zum Nachweis von Schranken für optimale Lösungen kann es sinnvoll sein, das Problem „aufzuweichen“ und z.B. nicht-ganzzahlige Lösungen zu betrachten (Relaxation; hier Transport von „Bruchstücken“).
- Argumente fokussieren gerne auf die „erste Abweichung“ von „optimaler Strategie“ (evtl. auch gegenüber der Relaxation).

Hinweise

Greedy-Verfahren können unter gewissen Voraussetzungen auch garantiert optimale Lösungen (für PO-Probleme) liefern.

Ein Beispiel hierfür ist die Berechnung eines Spannbaums mit kleinstem Kantengewicht.

Näheres hierzu im Speziellen in Algorithmen-Vorlesungen von Jun.-Prof. Kindermann bzw. von Prof. Näher.

Die Theoretische Informatik liefert *Matroide* als Grundgerüst.