

Die Entscheidungsversion von RUCKSACK ist NP-vollständig

DP 5.

Ein exakter Algorithmus für ganzzahlige

Gewichte $g_i \in \mathbb{N}$ mit dynamischer Programmierung

→ bezeichne $W(i, g)$ den größten Wert, so dass eine Bepackung aus den Objekten $\{1, 2, \dots, i\}$ mit Gewicht genau g und diesem Wert existiert

(für jede $i = 0, 1, 2, \dots, n$ und $g = 0, 1, 2, \dots, G$)

Wie berechnet man $W(i, g)$ aus früheren $W(i', g')$ Werten?

$i' \leq i$ und $g' \leq g$

→ Wir stellen eine Rekursionsgleichung für

$W(i, g)$ auf:

$$W(i, g) = \max \{ W(i-1, g), w_i + W(i-1, g-g_i) \}$$

↓
Objekt i
nicht hinzugenommen

↓
Objekt i
hinzugenommen

die Basisfälle sind

$$W(0, 0) = 0$$

$$W(0, g) = -\infty \text{ wenn } g \neq 0$$

$$W(i, g) = -\infty \text{ für } g < 0$$

(setze $-\infty$ für unmögliche Fälle bei

Maximierungsproblemen)

DP 6.

$i \backslash g$	1	2	3	4	G
1								
2								
3								
...								
i								
...								
n								

Der größte $W(n, g)$ Wert in der Zeile n ist optimal.

→ die Tabelle von $W(i, g)$ Werten berechnet und speichert der Algorithmus von oben nach unten

~~und speichert sie in der Tabelle~~

- initialisiere

- FOR $i = 1$ to n DO

FOR $g = 0$ to G DO

$$W(i, g) = \max \{ W(i-1, g), w_i + W(i-1, g-g_i) \}$$

→ Laufzeit: $O(n \cdot G)$

→ Wie wird eine optimale Bepackung berechnet?

Wenn mit jedem $W(i, g)$ noch ein Zeiger auf $W(i-1, g)$ oder auf $W(i-1, g-g_i)$ (um $W(i, g)$ zu realisieren) gesetzt wird, dann kann der Algorithmus am Ende vom maximalen $W(n, g)$ ausgehend, in $O(n)$ Schritten mit Backtracking die Objekte in einer optimalen Bepackung finden.

Beachte, dass erst am Ende klar wird, welche Zeiger/Objekte nützlich sind!

Warum sind wir mit diesem Algorithmus nicht ganz zufrieden? Ist dieser Algorithmus polynomiell?

→ falls W relativ klein, ist der Alg. praktikabel
wenn aber $w_{\max}^G = 2^{100}$, dann kann die Eingabelänge $O(100n)$ sein, und die Laufzeit $\mathcal{O}(n \cdot 2^{100})$. Allgemein, $W \rightarrow w_{\max}^G$ ist eine exponentielle Funktion von $\log w_{\max}^G$ dem entsprechenden Term in der Eingabelänge

Anders: $O(n \cdot W)$ wäre polynomiell, falls die Eingabelänge (abgesehen von der g_i) $W = \sum_i w_i^{g_i}$ und nicht $\sum_i \log w_i^{g_i}$ wäre.
z.B. wenn die Eingabe unär kodiert wäre.

$$10 = \underbrace{1111111111}_{10} \textcircled{1}$$

$$10 = 1010 \textcircled{2}$$

$\underbrace{\hspace{2cm}}_{\lceil \log 10 \rceil}$

Definition: Ein Algorithmus heißt pseudopolynomiell wenn seine Laufzeit $O(\text{Poly}(n, Z))$ ist, wobei n die Eingabelänge, und Z in Absolutwert die größte Zahl in der Eingabe ist.
 \Leftrightarrow die Laufzeit ist polynomiell in der Eingabelänge, wenn jede Instanz unär kodiert ist.

Dynamische Programmierung (Zusammenfassung)

Das Problem (die Instanz) wird in kleinere Teilprobleme aufgebrochen. Elementare (die kleinsten) Teilprobleme sind einfach zu lösen. Die Optimumwerte der Teilprobleme werden in einer 'Tabelle' gespeichert, und für die Lösung schwierigerer Teilprobleme immer wieder verwendet, mit Hilfe einer Rekursionsgleichung (bzw. rekursiven Definition).

Wann wird dynamische Programmierung verwendet?

- eine optimale Lösung eines Problems enthält optimale Lösungen für seine Teilprobleme; (optimale Teilstruktur)
- relativ "wenige" Teilprobleme, die von einem rekursiven Algorithmus immer wieder berechnet wären; (überlappende Teilprobleme)

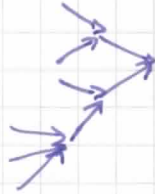
Vergleich: Dynamische Programmierung vs.

Divide & Conquer

Modelliere die Lösung von Teilproblemen durch einen gerichteten azyklischen Graphen (DAG - directed acyclic graph)

Eine Kante $P_i \rightarrow P_j$ bedeutet, dass die Lösung von P_i zur Lösung von P_j unmittelbar verwendet wird.

- in D&C ist der Graph ein Baum (Rekursionsbaum), die Teillösungen werden oft rekursiv (top-down) berechnet und nicht gespeichert



- in Dyn Prog ist der Graph ein DAG. Die Teillösungen werden mehrfach verwendet und deshalb gespeichert. Sie werden bottom up berechnet.



Die Minimierung des Speicherplatzbedarfs spielt eine wichtige Rolle.

(Tendenzziel: in der Rekursionsgleichung sind die Größen der Teilprobleme nicht um einen Faktor (wie bei D&C), sondern um eine additive Konstante kleiner.)

Ein FPTAS für RUCKSACK

für n Objekte mit Gewichten $g_1, g_2, \dots, g_n \in \mathbb{R}_+$
 und Werten $w_1, w_2, \dots, w_n \in \mathbb{R}_+$
 und Gewichtsschranke $G \geq g_i \forall i$

wird eine Teilmenge der Objekte mit Gesamtgewicht $\leq G$
 und Gesamtwert $\geq \frac{OPT}{1+2\epsilon}$ ausgegeben

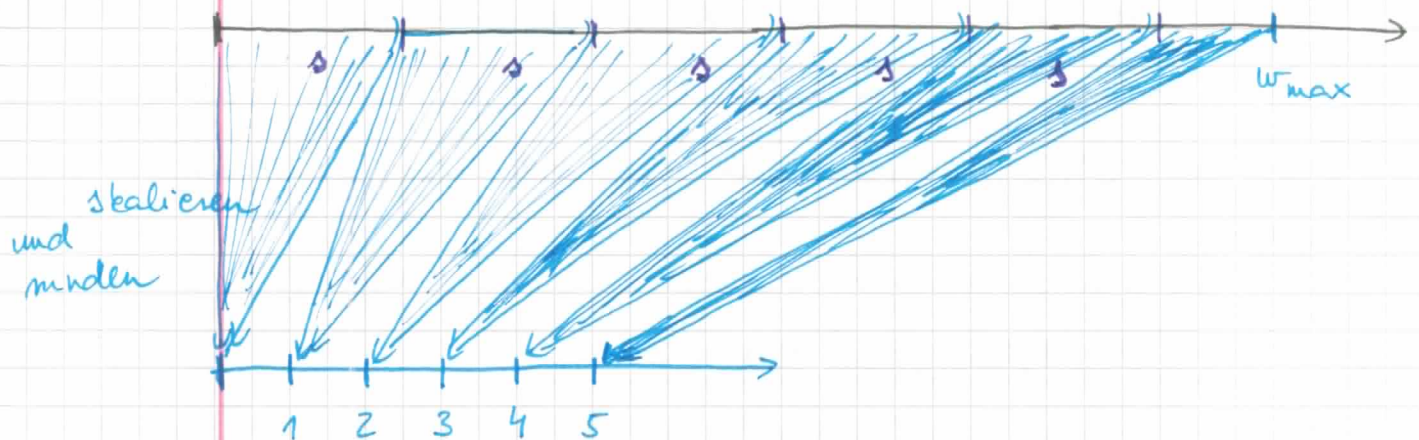
Ausgangspunkt: es gibt einen Algorithmus mit

dynamischer Programmierung für RUCKSACK mit Laufzeit $O(n \cdot W)$ für $W = \sum_i w_i$ bei ganzzahligen Werten w_i .

(also einen schnelleren Algorithmus, wenn es insgesamt wenige mögliche verschiedene Werte gibt)

Wir nutzen diesen Algorithmus um ein FPTAS für beliebige (insb. beliebig große) Werte zu entwickeln.

Idee: jeder Wert w_i wird auf das nächste Vielfache eines geeigneten s abgerundet, damit es wenige verschiedene Werte gibt.



w_i wird skaliert auf $\frac{w_i}{s}$ und abgerundet $\lfloor \frac{w_i}{s} \rfloor$

Wir berechnen eine optimale Lösung für die

Werte $\underline{w}_i = \lfloor \frac{w_i}{s} \rfloor$ (z.B. $\underline{w}_{\max} = 5$ weil $w_{\max} = 5.6s$)

Wir müssen mit der Wahl von δ vorsichtig sein:

- δ soll klein genug sein um eine gute Approximation zu ermöglichen
- δ soll groß genug sein um einen schnellen Algorithmus (durch nur wenige mögliche \underline{w}_i Werte) zu erhalten
- um δ richtig zu setzen, brauchen wir, dass der Fehler wegen Abmündungen, in der Lösung höchstens $\epsilon \cdot \text{OPT}(I)$ ist. Wir brauchen dafür eine Schätzung von $\text{OPT}(I)$ und vom Fehler
- wir nutzen $\text{OPT}(I) \geq w_{\max}$ als untere Schranke von $\text{OPT}(I)$
- der Fehler wegen Abmündung ist $\leq \delta$ für ein Objekt, und $\leq \delta \cdot n$ für alle Objekte im Rucksack
- ⇒ δ sollte so gewählt werden, dass

$$(\text{Fehler} \leq) \delta \cdot n \leq \epsilon \cdot w_{\max} (\leq \epsilon \cdot \text{OPT}(I))$$

$$\text{also, sei } \delta = \frac{\epsilon \cdot w_{\max}}{n}$$

FPTAS für RUCKSACK (ϵ sei fixiert)

Eingabe: Gewichte $g_1, g_2, \dots, g_n \leq G$
Werte w_1, w_2, \dots, w_n

1. setze $\delta = \frac{\epsilon w_{\max}}{n}$

2. sei $\underline{w}_i = \left\lfloor \frac{w_i}{\delta} \right\rfloor$ für jedes Objekt i

3. berechne die exakt optimale Lösung für die ^(gemündete) Instanz $(g_1, g_2, \dots, g_n, \underline{w}_1, \underline{w}_2, \dots, \underline{w}_n)$ mit dynamischer Programmierung

4. sei $B \subseteq \{1, 2, \dots, n\}$ die gewählte Objektmenge

gib B aus (mit Wert $\sum_{i \in B} w_i$)

Wir zeigen, dass wir tatsächlich ein FPTAS angegeben haben:

DP12. Theorem 1: Dieser Algorithmus hat Laufzeit $O\left(n^3 \cdot \frac{1}{\epsilon}\right)$

(polynomiell auch in $\frac{1}{\epsilon}$)

Wann?

Die Laufzeit des pseudopolynomiellen Algorithmus (Schritt 3.) ist $O(n \cdot W_{\text{skaliert}})$, wobei

Gesamtwert der skalierten Objekte $\leftarrow W_{\text{skaliert}} \leq n \cdot w_{\text{max}} \leq n \cdot \frac{w_{\text{max}}}{\delta} = n \cdot \frac{w_{\text{max}} \cdot n}{\epsilon \cdot w_{\text{max}}} = \frac{1}{\epsilon} \cdot n^2$

$$\Rightarrow n \cdot W_{\text{skaliert}} \leq n^3 \cdot \frac{1}{\epsilon}$$

□

Theorem 2: Der Algorithmus ist $(1+2\epsilon)$ -approximativ.

Beweis: Sei $B \subseteq \{1, 2, \dots, n\}$ die Bepackung ausgegeben vom FPTAS in Schritt 3., und B_{opt} eine optimale Bepackung

$$\text{OPT(I)} \sum_{i \in B_{\text{opt}}} w_i = \sum_{i \in B_{\text{opt}}} \delta \cdot \frac{w_i}{\delta} \leq \sum_{i \in B_{\text{opt}}} \delta \left(\left\lfloor \frac{w_i}{\delta} \right\rfloor + 1 \right) \leq$$

$$\leq \delta \cdot \sum_{i \in B_{\text{opt}}} \left\lfloor \frac{w_i}{\delta} \right\rfloor + \delta \cdot n \leq \delta \cdot \sum_{i \in B} \left\lfloor \frac{w_i}{\delta} \right\rfloor + \delta \cdot n \leq$$

\downarrow
weil B optimal für die gerundeten Werte w_i ist

$$\leq \delta \cdot \sum_{i \in B} \frac{w_i}{\delta} + \delta \cdot n = \sum_{i \in B} w_i + \epsilon \cdot w_{\text{max}} \leq \text{PTAS(I)} + \epsilon \cdot \text{OPT(I)}$$

$$\Rightarrow (1-\epsilon) \text{OPT(I)} \leq \text{PTAS(I)}$$

$$\frac{\text{OPT(I)}}{1+2\epsilon} \quad \text{für } \epsilon < \frac{1}{2}$$

□

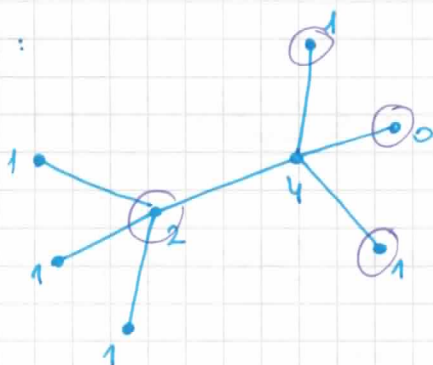
Dynamische Programmierung auf Bäumen

Beispiel: gewichtetes Vertex COVER auf Bäumen.

Eingabe: Ein Baum $T(V, E)$ (ungerichtet) mit einer Gewichtung der Knoten w_v für jeden $v \in V$

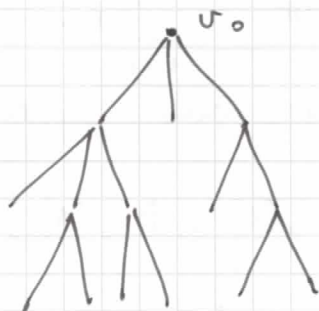
Ausgabe: Eine Knotenüberdeckung mit minimalem Gesamtgewicht

Beispiel:



Ein effizienter Algorithmus mit dynamischer Programmierung

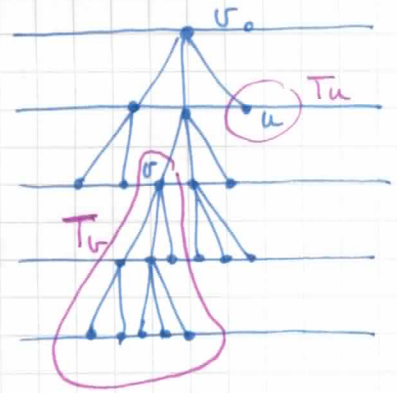
- wähle einen beliebigen Knoten $v_0 \in V$ als Wurzel;
- jetzt hat jeder Knoten außer v_0 einen Vater;
- die Knoten ohne Kinder heißen Blätter



DP. 14.

- für jeden $v \in T$ sei T_v der Teilbaum von T mit Wurzel v

- angefangen mit der tiefsten Ebene (bottom-up) berechnen wir für jeden Knoten v zwei Werte bezüglich sein Teilbaum T_v :



$D_v(0) :=$ Gewicht einer minimalen Knotenüberdeckung C_v von T_v mit $v \notin C_v$

$D_v(1) :=$ Gewicht einer minimalen Knotenüberdeckung C_v von T_v so dass $v \in C_v$

- Elementare Teilprobleme (Blätter):

für Blätter v gilt $D_v(0) = 0$ $D_v(1) = w_v$

- Rekursionsgleichung:

falls v die Kinder u_1, u_2, \dots, u_m hat, gilt

$$D_v(0) = \sum_{i=1}^m D_{u_i}(1)$$

$$D_v(1) = w_v + \sum_{i=1}^m \min \{ D_{u_i}(0), D_{u_i}(1) \}$$

- das minimale Gewicht einer Knotenüberdeckung:

$$\min \{ D_{v_0}(0), D_{v_0}(1) \}$$

Um abschließend eine Knotenüberdeckung mit minimalem Gewicht zu finden, muss der Algorithmus mit Breiten- oder Tiefensuche von der Wurzel nach unten laufen (top-down):

Sei $C \subset V$ die gesuchte Knotenüberdeckung, und sei v der Vater von u

→ falls $v \in C$ (oder $u = v_0$), dann

wir dürfen
 u hinzunehmen

$$C := \begin{cases} C \cup \{u\} & \text{falls } D_u(1) \leq D_u(0) \\ C & \text{falls } D_u(1) > D_u(0) \end{cases}$$



→ falls $v \notin C$

wir müssen
 u hinzunehmen

$$C := C \cup \{u\}$$

Auf Bäumen und Baum-ähnlichen Graphen

(b.g. Graphen mit beschränkter Baumweite)

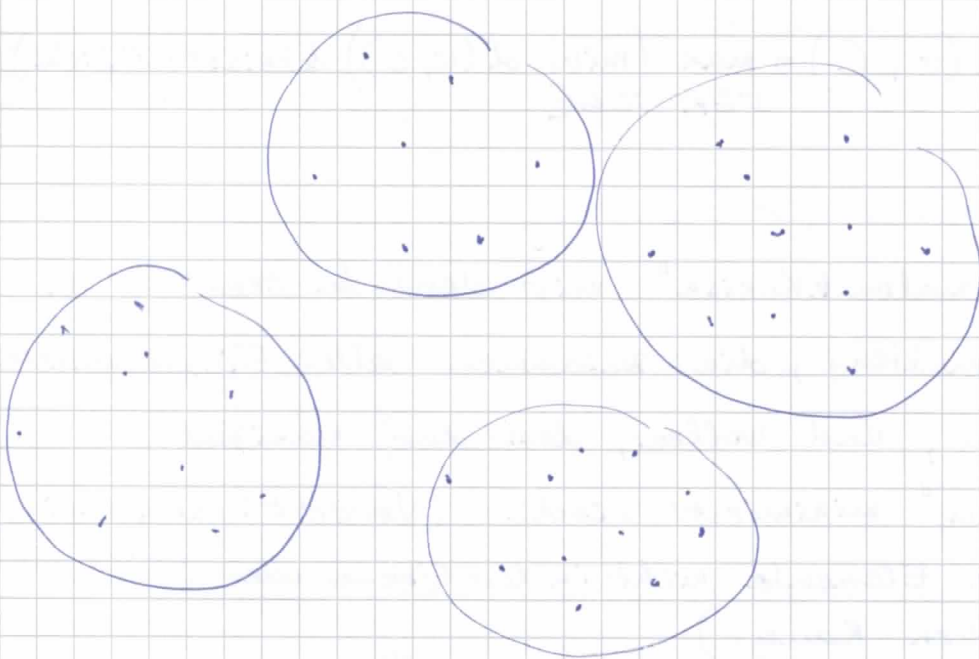
sind viele Optimierungsprobleme mit dynamischer Programmierung effizient lösbar.

(Die $D_v(0)$ $D_v(1)$ Werte für alle Knoten v bilden die 'Tabelle'; man kann an $\{D_v(0), D_v(1)\}$ auch als an eine kleine Tabelle im Knoten v denken.)

Cluster Probleme (allgemein)

In der Eingabe sind eine Menge K von Punkten / Elementen mit einer Metrik (also Distanzwerten zwischen je zwei Punkten) gegeben.

Das Ziel ist, „ähnliche“ Elemente, also die sich nah aneinander befinden, zu einem sog. Cluster machen.



Anwendungen von Clustering: Finde Ähnlichkeiten / Unterschiede in großen Datenmengen, wie Webseiten, Kunden, Produkte, Wähler die gemäß Inhalt, Verhalten, Präferenzen gruppiert werden sollen.

Oder: örtliche Clustering, z.B. positioniere deine Supermärkte optimal

a.) Das k -CENTER und das k -MEDIAN Problem
(Minimierungsprobleme)

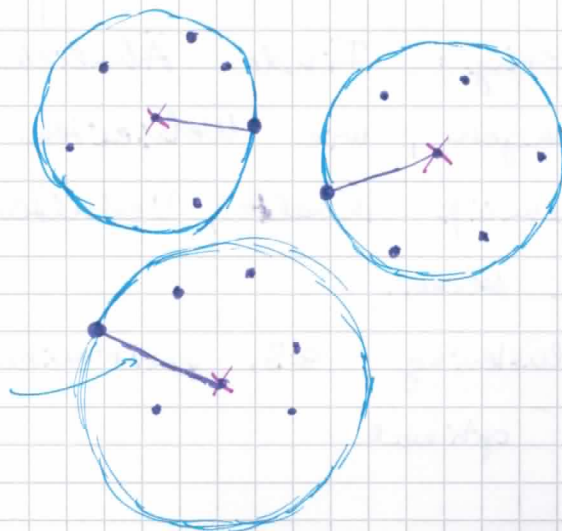
k -CENTER

Eingabe: eine Menge K von Punkten mit einer Metrik (Distanzen) $d: K \times K \rightarrow \mathbb{R}_+$ und eine Zahl k

Ausgabe: eine Menge $C \subseteq K$ von k Zentren, so dass die maximale Distanz vom nächstliegenden Zentrum über alle Punkte, minimiert wird

$$\left(\max_{v \in K} d(v, C) = \max_{v \in K} \left(\min_{c \in C} d(v, c) \right) \text{ kleinstmöglich} \right)$$

Intuition: Wir „malen k Kreise“ mit den k Zentren als Mittelpunkten, die zusammen alle Eingabepunkte decken, und wollen, dass der maximale „Radius“ minimiert wird. (Vorsicht! allgemein sind die Elemente nicht in der Ebene oder im Euklidischen Raum)



ein maximaler
Radius

Ein greedy Algorithmus für k -CENTER

→ sei $v_1 \in K$ ein beliebiger Eingabepunkt,
setze $C = \{v_1\}$

→ FOR $i=2$ TO k DO

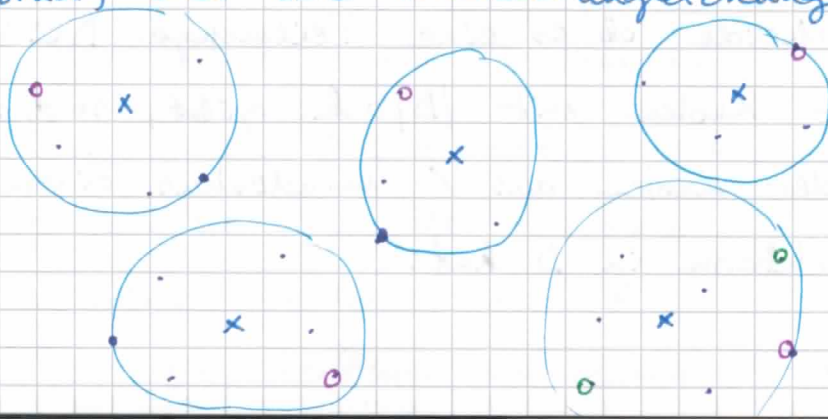
- sei $v_i \in K$ ein Punkt mit maximaler $d(v, C)$
(Mindest)Distanz vom nächstliegenden
irgendeinem Zentrum aus C
(ein Punkt der aktuell den maximalen Radius bestimmt)

- setze $C = C \cup \{v_i\}$

Theorem: Dieser Greedy Algorithmus für k -CENTER
ist 2-approximativ.

Beweis: → Sei C_{opt} die Menge von k Zentren in einer
fixierten optimalen Lösung (markiert mit x) und R_{opt}
der optimale Radius; sei C_{alg} die k Zentren vom Greedy
Algorithmus (markiert mit \circ)

→ wir weisen jeden Punkt aus K zu einem nächstliegenden
Zentrum aus C_{opt} ; innerhalb von jedem solchen Cluster
(Kreis) ist die Distanz zweier Punkte $\leq 2 \cdot R_{opt}$
(weil beide höchstens um R_{opt} entfernt vom Cluster-Zentrum
sind, und die Dreiecksungleichung gilt).



LS 4.

FALL 1. C_{ALG} hat in jedem Cluster ein Zentrum

→ dann hat jeder Punkt höchstens Distanz $2 \cdot R_{opt}$ von einem Zentrum in C_{ALG}

FALL 2. C_{ALG} hat (mind.) 2 Zentren q und r im selben Cluster, wobei zuerst q dann später r vom Greedy Alg. ausgewählt wurde.

Dann war damals r ein Punkt mit maximaler Distanz zum ^{nächstliegenden} ~~irgendeinem~~ Zentrum aus C_{ALG} , aber diese war nicht größer als $d(q, r) \leq 2 R_{opt}$ \square

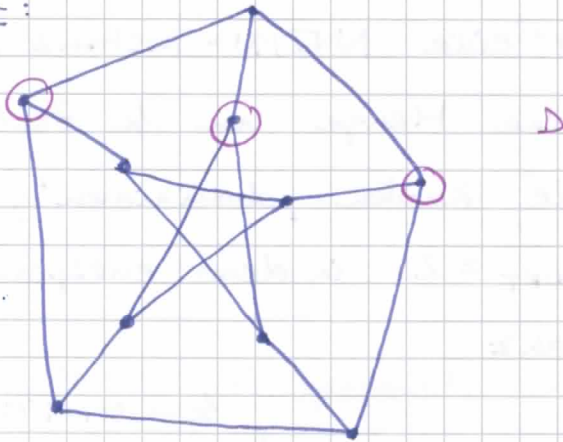
Theorem: k -CENTER ist ^{nicht} effizient um Faktor $c < 2$ approximierbar, d.h. der Greedy Algorithmus hat den bestmöglichen Approximationsfaktor unter effizienten Algorithmen.

Für den Beweis brauchen wir das folgende schwierige Problem:

DOMINATING SET (Entscheidungsversion)

Eingabe: ein Graph $G(V, E)$, und eine Zahl k

Ausgabe: entscheide ob es eine Teilmenge $D \subset V$ der Knoten mit $|D| = k$ gibt, sodass jeder Knoten aus V mindestens einen Nachbarn in D hat.

Beispiel:

Die Entscheidungsversion von DOMINATING SET ist NP-vollständig.

Wir zeigen, dass k -CENTER nicht besser als 2 approximierbar ist; durch Reduktion vom DOMINATING SET:

Angenommen, es gibt einen effizienten c -approximativen Algorithmus für k -CENTER mit $c < 2$.

Sei $G(V, E)$, k eine beliebige Eingabe für DOMINATING SET

Wir definieren eine Eingabe (eine spezielle Eingabe) für k -CENTER, und geben sie unserem (hypothetischen) Algorithmus:
Sei k dieselbe Zahl in beiden Eingaben.

Sei $K = V$ die Menge der Punkte, und die Metrik sei

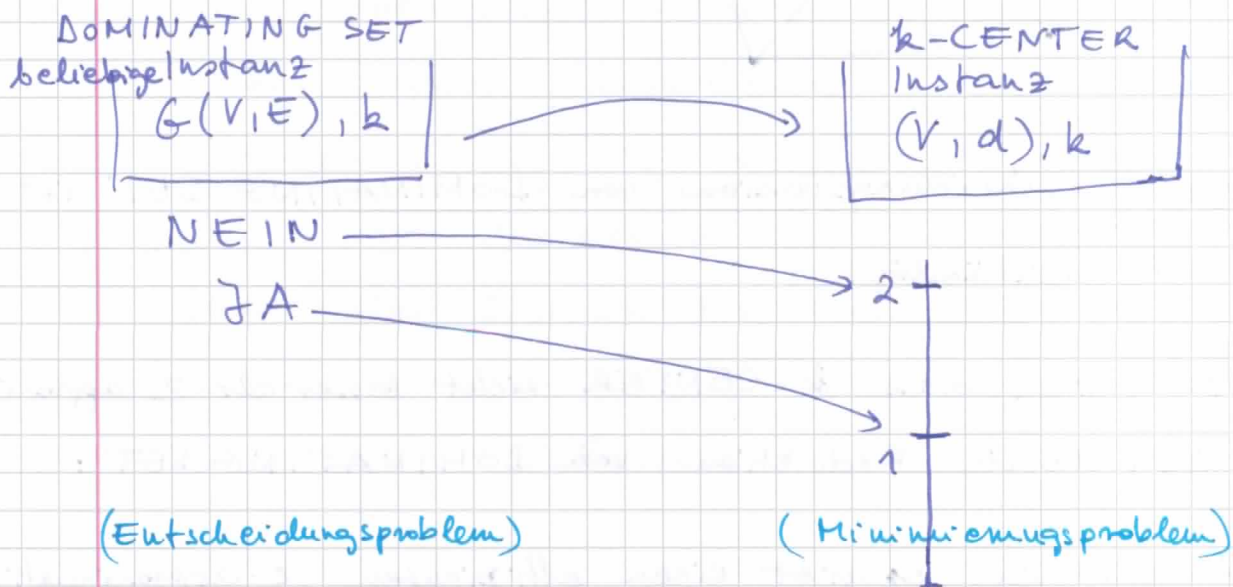
$$d(u, v) = \begin{cases} 1 & \text{falls } \{u, v\} \in E \\ 2 & \text{sonst} \end{cases}$$

(prüfe, dass diese Distanzen eine Metrik definieren!)

→ Falls $(G(V, E), k)$ eine JA-Instanz für DOMINATING SET, mit einer dominierenden Knotenmenge $D \subset V$, dann kann $D \subset V$ mit $|D| = k$ als die Menge von Zentren in der Instanz $((K, d), k)$ für k -CENTER gewählt werden, mit $R_{\text{opt}} = 1$ (weil jeder Knoten mit einem Zentrum benachbart ist also Distanz 1 von D hat).

LS 6.

→ Falls $(G(V, E), k)$ eine NEIN-Instanz ist, dann gibt es keine Menge von k Zentren, die zusammen alle Knoten 'dominieren', und deshalb $R_{opt} = 2$ in der entsprechenden k -CENTER Instanz



Ein besser als 2-approximativer Algorithmus für k -CENTER kann somit DOMINATING SET entscheiden. Dies ist effizient nicht möglich, angenommen $P \neq NP$

k -MEDIAN (ein anderer Typ von Clustering)

Eingabe: eine Menge K (von Punkten) mit einer Metrik (Distanzen) $d: K \times K \rightarrow \mathbb{R}_+$ und eine Zahl k

Ausgabe: eine Menge $M \subset K$ von k Zentren, so dass die Summe der Distanzen vom nächstliegenden Zentrum über alle Punkte minimiert wird

$$\left(\sum_{v \in K} d(v, M) = \sum_{v \in K} \left(\min_{c \in M} d(v, c) \right) \text{ kleinstmöglich} \right)$$

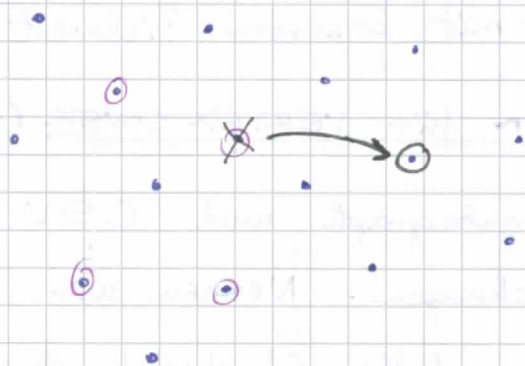
Ein Algorithmus mit lokaler Suche für k-MEDIAN

- sei $i=0$
- wähle eine Menge M^0 von beliebigen k Punkten als Zentren
- WHILE es geeignete $w \in M^i$ und $u \notin M^i$ gibt, DO:
 ersetze ein $w \in M^i$ durch einen $u \notin M^i$
 falls dies zu kleinerer Summe der
 Distanzen führt (d.h. die Zielfunktion reduziert)

$$M^{i+1} = (M^i \setminus \{w\}) \cup \{u\}$$

$$i = i+1$$

(Wir sagen, dass die Lösung M^{i+1} eine benachbarte Lösung von M^i ist weil sie nur „ein wenig unterschiedlich“ sind)



Theorem: (ohne Beweis) Dieser Algorithmus ist 5-approximativ für k-MEDIAN.

b.) Lokale Suche, DefinitionLernbeispiel: min-VERTEX COVERein Algorithmus mit lokaler Suche für VERTEX COVER

- sei $G(V, E)$ der Eingabegraph
- sei $C^0 = V$ die Knotenüberdeckung am Anfang und $i = 0$
- WHILE es gibt $v \in C^i$ sodass $C^i \setminus \{v\}$ noch eine Knotenüberdeckung

$$\text{sei } C^{i+1} = C^i \setminus \{v\}$$

$$i = i + 1$$

lokale Suche: Wechsle zu einer benachbarten (wenig veränderten) ^{besseren} Lösung, so lange es solche gibt mit besserem Zielwert

(Nachbarschaft)

Die entsprechende Definition für VERTEX COVER (für diesen Algorithmus)

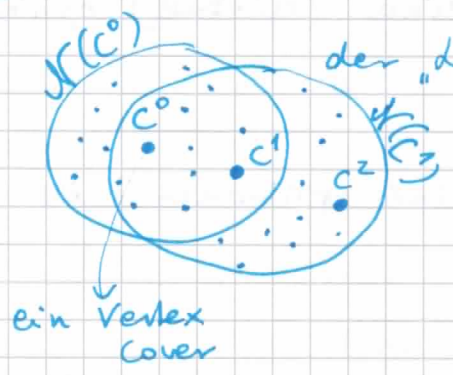
Sei $G(V, E)$ ein Eingabegraph und $C \subseteq V$ und $C' \subseteq V$ seien beide Knotenüberdeckungen. Nennen wir C und C' benachbarte Lösungen, falls C' durch das Hinzufügen oder das Entfernen eines Elements aus C entsteht.

Die Nachbarschaft $\mathcal{N}(C)$ der Lösung C besteht aus allen benachbarten Lösungen, also

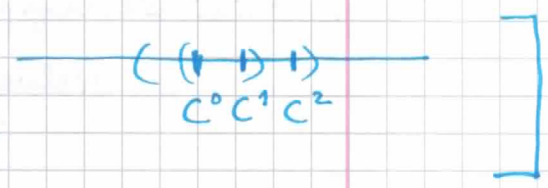
$C' \in \mathcal{N}(C)$ (und $C \in \mathcal{N}(C')$) falls sie benachbart sind.

Die Nachbarschaft einer Lösung ist also eine Menge von ähnlichen Lösungen.

[eine abstrakte Darstellung von Nachbarschaften der "Lösungsraum"

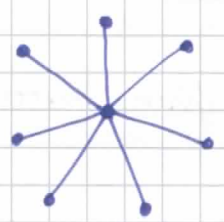


noch abstraktere Darstellung in 1D

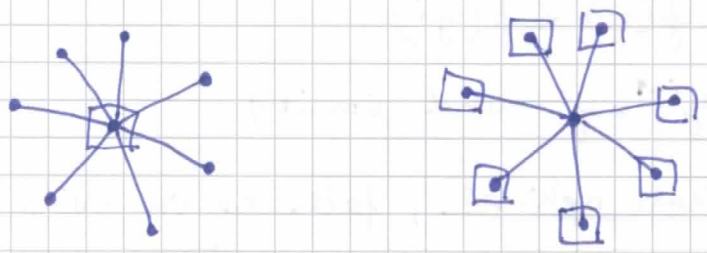


Was für eine Lösung kann so ein Algorithmus finden?
Anders: wie sind die Lösungen für die es keine bessere benachbarte Lösung gibt?

Beispiel: sei der Sterngraph die Eingabe für min-VERTEX COVER, mit den oben definierten Nachbarschaften der Knotenüberdeckungen.



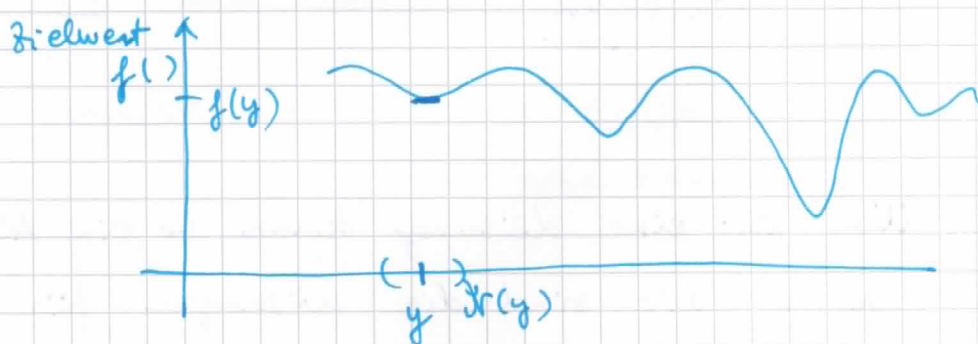
Von welchen Lösungen kann kein weiterer Knoten entfernt werden?



Eine solche Lösung heißt lokal optimal (weil es keine bessere in ihrer ^(Nachbarschaft) Umgebung gibt).

Eine lokal optimale Lösung kann, je nach Problem, viel schlechter sein als eine (global) optimale Lösung.

Abstrakte Darstellung einer lokal optimalen Lösung eines Minimierungsproblems:



Definition: Lokale Suche (allgemein)

Sei ein Minimierungsproblem zu lösen, wobei zu jeder Lösung y eine Umgebung $\mathcal{N}(y)$ benachbarter Lösungen definiert ist.

Stille lokale Suche

→ sei y^0 eine Lösung für Instanz x
 $i=0$

→ WHILE y^i nicht lokal optimal DO

- bestimme ein $y \in \mathcal{N}(y^i)$ so dass

$$f(y) < f(y^i)$$

- setze $y^{i+1} = y$ und $i := i+1$

(Eine Lösung y ist lokal optimal, falls es in $\mathcal{N}(y)$ keine Lösung mit kleinerem Zielwert gibt.)