

# Approximationsalgorithmen

**Das Ziel:** Bearbeite schwierige Optimierungsprobleme der Form

$$\text{opt}_y f(x, y) \text{ so dass } L(x, y).$$

- Die **Zielfunktion**  $f(x, y)$  ist zu minimieren oder zu maximieren: Wir nehmen stets  $\text{opt} \in \{\text{min}, \text{max}\}$  an.
- $x$  ist die Eingabe, oder die Beschreibung des Problems.  $y$  heißt eine **Lösung** für Eingabe  $x$ , wenn das Lösungsprädikat  $L$  erfüllt ist.
- Eine Lösung  $y$  mit optimalem Wert  $f(x, y)$  ist zu bestimmen.
- Wir spezifizieren das Optimierungsproblem durch das Tripel  $(\text{opt}, f, L)$  (**Web**)

- Das Clique Problem besitzt Graphen  $G = (V, E)$  als Eingaben.
  - ▶ Genau die Cliques  $C \subseteq V$  sind Lösungen. Also ist

$$L(G, C) \Leftrightarrow C \subseteq V \text{ und je zwei Knoten in } C \\ \text{sind durch eine Kante in } G \text{ verbunden}$$

- ▶  $f(G, C) = |C|$  ist die zu maximierende Zielfunktion.
- Im Rucksackproblem sind  $n$  Objekte mit Gewichten  $g_1, \dots, g_n \in \mathbb{R}$  und Werten  $w_1, \dots, w_n \in \mathbb{N}$  gegeben. Das Ziel: Bepacke den Rucksack, so dass einerseits eine Gewichtsschranke  $G \in \mathbb{R}$  nicht überschritten wird und andererseits der Gesamtwert der eingepackten Objekte maximal ist.
  - ▶ Die Eingabe  $x$  besteht aus den Gewichten, den Werten und der Gewichtsschranke.
  - ▶ Lösungen werden durch die Teilmengen  $I \subseteq \{1, \dots, n\}$  eingepackter Objekte beschrieben, wobei  $L(x, I) \Leftrightarrow \sum_{i \in I} g_i \leq G$ .
  - ▶  $f(x, I) = \sum_{i \in I} w_i$  ist die zu maximierende Zielfunktion.

Sei  $P = (\text{opt}, f, L)$  ein Optimierungsproblem.

- $y^*$  heißt eine **optimale** Lösung für Eingabe  $x$ , falls

$$f(x, y^*) = \text{opt} \{ f(x, y) \mid y \text{ ist eine Lösung, d.h. es gilt } L(x, y) \}.$$

Wir setzen  $\text{opt}_P(x) := f(x, y^*)$ .

- Ein Approximationsalgorithmus  $A$  für  $P$  berechnet auf jeder Eingabe  $x$  eine Lösung  $A(x)$ .

Wir möchten **schnelle** Approximationsalgorithmen entwerfen, die **gute** Lösungen berechnen.

# Wie gut ist eine Approximation?

## Approximationsfaktoren

Eine Lösung  $y$  von  $P$  für Eingabe  $x$  heißt  $\delta$ -approximativ, wenn

$$f(x, y) \geq \frac{\max_P(x)}{\delta} \quad \text{für ein Maximierungsproblem}$$

beziehungsweise wenn

$$f(x, y) \leq \delta \cdot \min_P(x) \quad \text{für ein Minimierungsproblem.}$$

- Der Approximationsfaktor ist stets **mindestens** 1.
- Wenn wir den Approximationsfaktor  $\delta = 1$  erreichen, dann haben wir das Optimierungsproblem exakt gelöst.
- Je kleiner der Approximationsfaktor, umso besser ist die gefundene Lösung.

# Ein Beispiel: Das metrische Traveling Salesman Problem

- Wir haben die Spannbaum Heuristik für M-TSP entworfen.
- Wenn  $L$  die Länge der von uns berechneten Tour ist und wenn  $opt$  die minimale Länge einer Rundreise ist, dann haben wir

$$L \leq 2 \cdot opt$$

nachgewiesen.

- Die Spannbaum Heuristik ist ein **2-approximativer Approximationsalgorithmus** für M-TSP.

# Wie gut können schwierige Optimierungsprobleme approximiert werden?

- Optimierungsprobleme mit  $NP$ -vollständigen Entscheidungsproblemen zeigen starke Variationen.
  - ▶ Einige Probleme, wie zum Beispiel das **Rucksackproblem**, sind sehr scharf durch effiziente Algorithmen approximierbar.
  - ▶ Andere, wie zum Beispiel das Problem der **Last-Verteilung** sind schon etwas schwieriger.
  - ▶ Dann folgt das **Vertex Cover Problem** und das noch schwierigere **Clique Problem**.
  - ▶ Zu den ganz harten Brocken gehören die **0-1 Programmierung** und die **ganzzahlige Programmierung**.
- Was weiss man über die Approximationskomplexität schwieriger Probleme? (**Web**)

Wir haben bereits die folgenden Scheduling Probleme untersucht.

- Im (gewichteten) **Intervall Scheduling**
  - ▶ haben wir eine wertvollste Auswahl von Aufgaben bei vorgeschriebenen Start- und Terminierungszeiten auf **einem** Prozessor ausgeführt.
- Für das **Scheduling mit minimaler Verspätung** haben wir die maximale Verspätung bei gegebenen Fristen und Laufzeiten mit einem Greedy Algorithmus minimiert.

**Das Lastverteilungsproblem:** Nur die Rechenzeiten der Aufgaben sind bekannt. Alle Aufgaben können sofort abgearbeitet werden.

**Das Ziel:** Verteile die Aufgaben so auf  $m$  identische Prozessoren, dass der **Makespan**, also der Zeitraum in dem **alle** Aufgaben abgearbeitet werden, **minimal** ist.



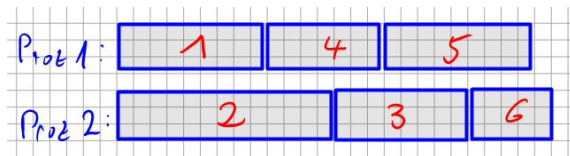
In einem realistischen Szenario muss die Last verteilt werden, „wenn sie anfällt“. Insbesondere kann die Zuweisung weiterer Aufgaben nicht abgewartet werden: Ein **On-line Algorithmus** muss Entscheidungen treffen ohne die Zukunft zu kennen. (Web)

## Ein On-line Scheduling Algorithmus:

- Aufgaben  $1, \dots, n$  mit Rechenzeiten  $t_1, \dots, t_n$  sind gegeben.
- Die Aufgaben werden „der Reihe nach“ abgearbeitet. Wenn Aufgabe  $i$  „dran“ ist, dann

weise Aufgabe  $i$  dem Prozessor mit der bisher geringsten Last zu

Aufgaben für zwei Prozessoren entstehen in der Reihenfolge 1, 2, .... Welche ist die erste Aufgabe, die *nicht* gemäß Greedy-Online Scheduling zugewiesen wurde?



Auflösung: (3)

Die On-line Strategie ist 2-approximativ.

- Angenommen, Prozessor  $i$  trägt die größte Last. Der Makespan  $T$  ist dann die Gesamtlaufzeit von Prozessor  $i$ .
- Prozessor  $i$  möge die Aufgabe  $j$  als letzte Aufgabe ausführen.
  - ▶ Zum Zeitpunkt  $T - t_j$  sind alle Prozessoren „busy“: Ansonsten hätte eine freie Maschine die Aufgabe  $j$  früher übernommen. Also ist

$$\sum_{k=1}^n t_k \geq (m-1) \cdot (T - t_j) + T = mT - (m-1) \cdot t_j \geq mT - m \cdot t_j.$$

- ▶ Dividiere durch  $m$  und  $T - t_j \leq \frac{1}{m} \cdot \sum_{k=1}^n t_k$  folgt. ◀
- ▶ Sei  $\text{opt}$  der optimale Makespan. Dann ist
$$T - t_j \leq \frac{1}{m} \cdot \sum_{k=1}^n t_k \leq \text{opt} \text{ und } t_j \leq \text{opt}.$$
- ▶ Also ist  $T = (T - t_j) + t_j \leq 2 \cdot \text{opt}$ .

# Wie gut war unsere Analyse?

## Ein worst-case Beispiel:

- Insgesamt  $(m - 1) \cdot m + 1$  Aufgaben liegen vor. Alle Aufgaben bis auf die letzte sind „kurze“ Aufgaben der Länge 1. Die letzte Aufgabe ist „lang“ und hat die Ausführungszeit  $m$ .
- Der optimale Makespan ist  $m$ :
  - ▶ Ein Prozessor führt die lange Aufgabe aus.
  - ▶ Die verbleibenden  $m - 1$  Prozessoren teilen die  $(m - 1) \cdot m$  kurzen Aufgaben unter sich auf.
- Unsere On-line Heuristik führt zuerst alle kurzen Aufgaben in der Zeit  $m - 1$  aus. Die lange Aufgabe wird zuletzt ausgeführt und kostet  $m$  zusätzliche Schritte.
- Unsere Heuristik hat Schwierigkeiten, wenn lange Aufgaben zuletzt abgearbeitet werden.

# Eine Off-line Heuristik

- (1) Aufgaben  $1, \dots, n$  mit Rechenzeiten  $t_1 \geq t_2 \geq \dots \geq t_n$  sind gegeben.
- (2) Die Aufgaben werden der Reihe nach abgearbeitet. Wenn Aufgabe  $i$  „dran“ ist, dann

weise Aufgabe  $i$  dem Prozessor mit der bisher geringsten Last zu.

## Die zentrale Beobachtung

Sei  $\text{opt}$  der minimale Makespan für  $m$  Prozessoren.


Dann ist  $2 \cdot t_{m+1} \leq \text{opt}$ .

- Es ist  $t_1 \geq \dots \geq t_m \geq t_{m+1}$ .
- Um die ersten  $m + 1$  Aufgaben abzuarbeiten, muss ein Prozessor zwei Aufgaben ausführen.
- Seine Bearbeitungszeit ist mindestens  $2 \cdot t_{m+1}$ .

Der Approximationsfaktor sinkt auf höchstens  $\frac{3}{2}$ , wenn Aufgaben gemäß fallender Bearbeitungszeit präsentiert werden.

- Wir betrachten wieder den Prozessor  $i$  mit größter Last. Prozessor  $i$  möge die Aufgabe  $j$  als Letzte ausführen.
- Es ist  $t_j \leq t_{m+1}$ , denn wir arbeiten die Aufgaben nach fallender Bearbeitungszeit ab. Also

$$2 \cdot t_j \leq 2 \cdot t_{m+1} \leq \text{opt}$$

- Wir haben  $t_j \leq \text{opt}/2$  gerade nachgewiesen und wissen, dass  $T - t_j \leq \frac{1}{m} \cdot \sum_{k=1}^n t_k \leq \text{opt}$  gilt .
- Also folgt  $T = (T - t_j) + t_j \leq \text{opt} + \text{opt}/2 = 3 \cdot \text{opt}/2$ .  
(Der tatsächliche Approximationsfaktor ist  $4/3$ .)

# Das Rucksackproblem

$n$  Objekte mit Gewichten  $g_1, \dots, g_n \in \mathbb{R}$  und Werten  $w_1, \dots, w_n \in \mathbb{N}$  sind in einen Rucksack mit Kapazität höchstens  $G$  zu packen. Der Gesamtwert der Bepackung ist zu maximieren. (Web)

- Das Rucksackproblem ist NP-vollständig.
- Wir lösen ein anderes Problem, nämlich: **Bestimme**

$\text{Gewicht}_i(w)$  = das minimale Gewicht einer Bepackung aus den ersten  $i$  Objekten mit Gesamtwert  $w$ .

- Wir lösen das Rucksackproblem, wenn wir nach dem größten Wert  $w$  mit  $\text{Gewicht}_n(w) \leq G$  suchen.
- Wir wenden dynamisches Programmieren an, um  $\text{Gewicht}_n(w)$  für alle möglichen Werte  $w \leq \sum_{i=1}^n w_i$  zu bestimmen.

# Bestimmung von Gewicht $_i(w)$

- Wie kann das Gewicht minimiert werden, wenn Wert  $w$  zu erbringen ist?
  - ▶ Wenn das  $i$ te Objekt in den Rucksack gepackt wird, dann erhöht sich das Gewicht um  $g_i$  und der Restwert  $w - w_i$  ist von den ersten  $i - 1$  Objekten zu erbringen.
  - ▶ Oder aber das  $i$ te Objekt wird nicht in den Rucksack gepackt und der Wert  $w$  ist mit Hilfe der ersten  $i - 1$  Objekte zu erbringen.
  - ▶ Wir erhalten die Rekursionsgleichungen

$$\text{Gewicht}_i(w) := \min \{ \text{Gewicht}_{i-1}(w - w_i) + g_i, \text{Gewicht}_{i-1}(w) \}.$$

- Sei  $W = \sum_{i=1}^n w_i$ . Wir bestimmen Gewicht $_i(w)$  für alle  $w \leq W$ .
  - ▶ Es gibt insgesamt  $n \cdot W$  Teilprobleme Gewicht $_i(W)$ .
  - ▶ Da jedes Teilproblem in Zeit  $O(1)$  gelöst wird, erhalten wir die Laufzeit  $O(n \cdot W)$ .



## Schnelle Algorithmen bei kleinen Werten

Das Rucksackproblem für  $n$  Objekte und Wertesumme  $W = \sum_{i=1}^n w_i$  kann in Zeit  $O(n \cdot W)$  gelöst werden.

- Problematisch sind Objekte mit sehr großen Werten.
- Die Idee:
  - ▶ Skaliere die Werte herunter.
  - ▶ Löse das neue Rucksackproblem mit den jetzt kleinen Werten exakt.
- Wie gut ist die Approximation?

# Der Approximationsalgorithmus

- (1) Der Approximationsfaktor  $1 + \varepsilon$  sei vorgegeben. Entferne alle Objekte, deren Gewicht die Gewichtsschranke  $G$  übersteigt.
- (2) Die Werte werden nach unten skaliert, nämlich setze

$$w_i^* = \lfloor \frac{w_i}{s} \rfloor \text{ für den Skalierungsfaktor } s = \frac{\varepsilon \cdot w_{\max}}{n}.$$

- (3) Berechne eine exakte Lösung für die neuen Werte  $(w_1^*, \dots, w_n^*)$ .
- (4) Bestimme für die nach neuen Werten ausgewählten Objekte den Gesamtwert  $\tilde{w}$  gemäß alter Werte und gib  $\max\{\tilde{w}, w_{\max}\}$  aus.  
// Wir erhalten den Wert  $w_{\max}$ , wenn wir nur das Objekt mit  
// maximalen Wert einpacken. ◀

- Der größte neue Wert ist  $w_{\max}^* = \lfloor \frac{w_{\max}}{s} \rfloor = \lfloor \frac{n}{\varepsilon} \rfloor$ .
- Die Wertesumme der neuen Werte ist also höchstens  $O(\frac{n^2}{\varepsilon})$ .
- Die Laufzeit ist also durch  $O(\frac{n^3}{\varepsilon})$  beschränkt.

# Wie gut ist die Approximation?

Der Skalierungsalgorithmus ist  $(1 + \varepsilon)$ -approximativ mit Laufzeit  $O(\frac{1}{\varepsilon} \cdot n^3)$ .

- Angenommen, die optimale Bepackung für die **neuen** Werte packt genau die Objekte in der Menge  $B \subseteq \{1, \dots, n\}$  ein. Sei  $B_{\text{opt}}$  die optimale Bepackung für die **alten** Werte.

$$\sum_{i \in B_{\text{opt}}} w_i \leq sn + \sum_{i \in B_{\text{opt}}} s \cdot \lfloor \frac{w_i}{s} \rfloor \leq sn + \sum_{i \in B} s \cdot \lfloor \frac{w_i}{s} \rfloor$$

denn  $B$  ist die beste Bepackung für die neuen Werte. Also ist

$$\sum_{i \in B_{\text{opt}}} w_i \leq sn + \sum_{i \in B} w_i.$$

- Es ist  $s \cdot n = \frac{\varepsilon \cdot w_{\text{max}}}{n} \cdot n = \varepsilon \cdot w_{\text{max}}$ . Wenn  $w_{\text{max}} \leq \sum_{i \in B} w_i$ , dann

$$\sum_{i \in B_{\text{opt}}} w_i \leq \varepsilon \cdot w_{\text{max}} + \sum_{i \in B} w_i \leq (1 + \varepsilon) \cdot \sum_{i \in B} w_i.$$

# Und wenn der maximale Wert sehr groß ist?

- Wir müssen nur noch den Fall  $w_{\max} > \sum_{i \in B} w_i$  betrachten.
- Jetzt gibt der Skalierungsalgorithmus den Wert  $w_{\max}$  aus.  Wie gut ist diese Bepackung im Vergleich zur optimalen Bepackung?

$$\sum_{i \in B_{\text{opt}}} w_i \leq \varepsilon \cdot w_{\max} + \sum_{i \in B} w_i \leq \varepsilon \cdot w_{\max} + w_{\max} = (1 + \varepsilon) \cdot w_{\max}.$$

Wir haben also in Zeit  $O(\frac{1}{\varepsilon} \cdot n^3)$  einen  $(1 + \varepsilon)$ -approximativen Algorithmus erhalten. Wir müssen also nur moderat mit Laufzeit bezahlen, um eine gute Approximation zu erhalten.

Man sagt deshalb auch, dass unser Approximationsalgorithmus ein **volles Approximationsschema** ist.

Wenn die approximative Lösung nur 10 Prozent schlechter als das Optimum sein darf, dann braucht der gerade vorgestellte Algorithmus folgende Laufzeit:

- (1)  $\Theta(n^2)$
- (2)  $\Theta(n^3)$
- (3)  $\Theta(n^{3+1/10})$
- (4)  $\Theta(n^{10})$

Auflösung: (2)  $\Theta(\frac{1}{\epsilon} \cdot n^3)$  mit  $\epsilon = 0.1$

# Das (ungewichtete) Vertex Cover Problem

Für einen ungerichteten Graphen  $G = (V, E)$  suchen wir eine kleinstmögliche Überdeckung, also eine Menge  $\dot{U}$  von Knoten, so dass alle Kanten einen Endpunkt in  $\dot{U}$  besitzen.

- Angenommen ein Matching  $M \subseteq E$ , also eine Menge von **knoten-disjunkten Kanten**, ist gegeben.  
Dann muss jede Überdeckung  $\dot{U}$  für jede Kante in  $M$  einen Endpunkt enthalten und  $|\dot{U}| \geq |M|$  folgt.
- **Die Idee:** Wir berechnen ein **nicht vergrößerbare** Matching  $M$  und definieren  $\dot{U}$  als die Menge **aller** Endpunkte von Kanten aus  $M$ .
  - ▶ Behauptung:  $\dot{U}$  ist eine Überdeckung.
  - ▶ Warum? Wenn die Kante  $e = \{u, v\}$  keinen Endpunkt in  $\dot{U}$  besitzt, dann ist weder  $u$  noch  $v$  Endpunkt einer Kante aus  $M$ . Also ist  $M \cup \{e\}$  ein größeres Matching.
- Wie berechnet man ein nicht vergrößerbare Matching?

# Die Matching Heuristik

- (1) Die Eingabe besteht aus einem ungerichteten Graphen  $G = (V, E)$ . Setze  $M := \emptyset$ .
- (2) while ( $E \neq \emptyset$ )
  - ▶ Wähle eine Kante  $e \in E$  und füge  $e$  zu  $M$  hinzu.
  - ▶ Entferne alle Kanten aus  $E$ , die einen Endpunkt mit  $e$  gemeinsam haben.
- (3) Gib die Knotenmenge  $\ddot{U} = \{v \in V \mid v \text{ ist Endpunkt einer Kante in } M\}$  aus.  
// Wir wissen bereits, dass  $\ddot{U}$  eine Überdeckung ist.  
// Desweiteren muss jede Überdeckung mindestens  $|M| \geq |\ddot{U}|/2$   
// Knoten besitzen.

Die Matching Heuristik ist ein 2-Approximationsalgorithmus für VC.

Welche Laufzeit hat die Matching-Heuristik?

- (1)  $\Theta(n + m)$
- (2)  $\Theta(n \log n + m)$
- (3)  $\Theta((n + m) \log n)$
- (4)  $\Theta((n + m)^2)$

Auflösung: (1), arbeite mit Adjazenzlisten.



# Besitzt die Matching Heuristik vielleicht bessere Approximationsfaktoren als 2?

Betrachte den vollständigen bipartiten Graphen mit  $n$  Knoten „auf jeder Seite“.

- Eine optimale Überdeckung wählt die  $n$  Knoten einer Seite aus.
- Jedes nicht vergrößerbare Matching besteht aus  $n$  Kanten, weil sonst mindestens ein Endpunkt auf jeder Seite frei bleibt.
- Damit wählt die Matching Heuristik alle  $2n$  Knoten als Überdeckung.

# Das gewichtete Vertex Cover Problem

Jeder Knoten  $v$  des Graphen  $G = (\{1, \dots, n\}, E)$  erhält das Gewicht  $w_v$ . Gesucht ist eine Überdeckung mit **minimalem Gewicht**.

- Wir benutzen die lineare Programmierung. Das gewichtete Vertex Cover Problem kann wie folgt formuliert werden:

$$\text{minimiere } \sum_{v \in V} w_v \cdot x_v \quad \text{so dass } x_u + x_v \geq 1 \text{ für alle } \{u, v\} \in E$$
$$\text{und } x_u \geq 0 \text{ für alle } u \in V.$$

- Wenn  $U \subseteq \{1, \dots, n\}$  eine Überdeckung ist, dann ist der Inzidenzvektor  $x_U$  von  $U$  eine Lösung. Aber es gibt unerwünschte fraktionale Lösungen.
  - ▶ Jede Überdeckung für den vollständigen Graphen (mit Gewichten  $w_v = 1$ ) benötigt mindestens  $n - 1$  Knoten, aber
  - ▶ das lineare Programm hat die fraktionale Lösung  $x = (\frac{1}{2}, \dots, \frac{1}{2})$  mit Zielwert  $n/2$ .

- **Unser Approximationsalgorithmus:**

- (1) Bestimme eine optimale Lösung  $x^*$  des linearen Programms.
- (2) Runde die im Allgemeinen fraktionale Lösung  $x^*$ , nämlich setze

$$\tilde{U} = \{v \in V \mid x_v^* \geq \frac{1}{2}\}.$$

- $\tilde{U}$  ist tatsächlich eine Überdeckung, denn für jede Kante  $\{u, v\}$  von  $G$  ist  $x_u^* + x_v^* \geq 1$ . Also ist  $x_u^* \geq \frac{1}{2}$  oder  $x_v^* \geq \frac{1}{2}$ .  
 $u \in \tilde{U}$  oder  $v \in \tilde{U}$  folgt.

- Wie gut ist die Approximation?

Wenn  $\text{opt} = \sum_{v=1}^n w_v \cdot x_v^*$ , dann hat jede Überdeckung mindestens die Größe  $\text{opt}$ . Also folgt

$$\sum_{v \in \tilde{U}} w_v \leq \sum_{v \in V} w_v \cdot (2x_v^*) = 2 \cdot \sum_{v \in V} w_v \cdot x_v^* = 2 \cdot \text{opt}$$

## Die Rundung fraktionaler Lösungen

Nach Rundung erhalten wir eine 2-approximative Lösung für das gewichtete Vertex-Cover Problem.

- Wir haben verschiedene, für Optimierungsprobleme wichtige Entwurfsmethoden eingesetzt.
  - ▶ In der Lastverteilung haben wir mit **Greedy Algorithmen** gearbeitet.
  - ▶ Für das Rucksackproblem haben wir die **dynamische Programmierung** und
  - ▶ für das gewichtete Vertex Cover Problem die **lineare Programmierung** eingesetzt.