Algorithmen I
Tutorium 32

Eine Lehrveranstaltung im SS 2017 (mit Folien von Christopher Hommel)

Daniel Jungkind (ufesa@kit.edu) | 21. Juli 2017

INSTITUT FÜR THEORETISCHE INFORMATIK

Schwarzes Brett + Klausurinfos!

- **{ }** vs. **( )** – Ungerichtete/gerichtete Kanten
- **Klausur** findet statt am **04.09.2017** von **11–13 Uhr**
- **Erlaubt**: Stifte, 4-Gänge-Menü, **Cheatsheet** (1 DIN-A4-Blatt beidseitig beliebig beschrieben)
- Klausur**anmeldung** bis 28.08.17.
  Klausur**abmeldung** bis 28.08.17, danach nur **direkt** vor Klausur im HS!

OPTIMIERUNGSPROBLEME

First World Problems

Optimierungsprobleme

Mehr Effizienz

- Dijkstra: **Kürzeste** Pfade
- Jarník-Prim bzw. Kruskal: **Minimale** Spannbäume

...

$\Rightarrow$ Alles **Optimierungsprobleme**

- **Heute**: Optimierungsprobleme **allgemein** – und wie man sie **löst**

Optimierungsprobleme – KNAPSACK

Beispiel: Ich nehme meinen Rucksack und packe ein...

- **Rucksackproblem** (KNAPSACK):
  Gegeben:
  Rucksackplatz M,
  n Gegenstände mit **Gewicht** w_i und **Profit** p_i
  Gesucht: Teilmenge X der Gegenstände, sodass
  $\sum_{i \in X} p_i$ **maximal** wird, aber $\sum_{i \in X} w_i \leqslant M$ bleibt
- **Nicht alle** Gegenstände passen in den Rucksack

Lösungsansätze?

Optimierung – Greedy

Wir bereuen nichts: Greedy-Algorithmen

- Prinzip: Reine **Gier**, never step back!
  Was **grad** am **Besten** scheint: **Direkt** nehmen!

$\Rightarrow$ Kann in **Sackgasse** führen

$\Rightarrow$ Auf die **Spitze** geht's manchmal nur durchs **Tal**

- Kann aber auch funktionieren:
  Dijkstra, Jarník-Prim, Kruskal – alles **greedy** und läuft ✓

Ein **Greedy-Algo** für **KNAPSACK**:

- Schmeiße der Reihe nach Gegenstände mit **bestem** Profit-/Gewicht-**Verhältnis** $\frac{p_i}{w_i}$ rein, bis voll

$\Rightarrow$ Aber: **nicht optimal**, Bsp.: $M = 10,\ (p_i, w_i) = (8,6),(5,5),(5,5)$ ϟ

$\Rightarrow$ Greedy-Algorithmus für KNAPSACK **ungeeignet**, kann sich eine optimale Lösung **verbauen**

Optimierung – DP

Rekursion rückwärts: Dynamic Programming (DP)

- **Teile-und-Herrsche**:
  Löse großes Problem durch **Zerlegung** in Kleinere
- **Dynamic Programming**:
  Löse **Kleinere** zuerst, setze dann zu **Größeren** zusammen:
  Konstruiere optimale „**Minimallösungen**“ $\rightsquigarrow$ zu größeren Optimallösungen **erweitern** $\rightsquigarrow$ bis zum urspr. Problem.
- Meistens **zweidimensional**: Tabelle mit **Rekursionsformel** ausfüllen. (siehe Beispiel)
- Formal: DP ist anwendbar $\Leftrightarrow$ die optimale Lösung besteht aus optimalen Lösungen von **Teilproblemen**.

Optimierung – DP

Lösung von KNAPSACK mit DP

- Lege zweidim. **array** $P[0..n,\ 0..M]$ **of** $\mathbb{R}$ an:
  $\Rightarrow P[i, C] =$ **optimaler Profit** für betrachtete Gegenstände $1 \dots i$
  mit benutzter Kapazität $\leqslant C$
  $\Rightarrow$ Rekursionsformel:
  $$P[i, C] = \max \left(\overbrace{P[i-1, C]}^{\text{Nehmen } i \text{ nicht}},\ \underbrace{P[i-1, C-w_i] + p_i}_{\text{Nehmen Gegenstand } i \text{ mit}} \right)$$
- **for** Items $i := 1$ **to** n **do** **for** Capacity $C := 1$ **to** M **do**
  $P[i, C] := \max\left(P[i-1, C],\ P[i-1, C-w_i] + p_i\right)$
  $Taken[i, C] :=$ **true if** links $<$ rechts
- Alternativer „Pseudo-Pseudocode“:
  for Items $i := 1$ **to** n **do** **for** Capacity $C := 1$ **to** M **do**
  if Platz langt $\wedge$ Profit(Restbestand mit i) $>$ Profit(Rest ohne i) **then**
  $Taken[i, C] :=$ **true**
  $P[i, C] :=$ besserer Profit von beiden (wird immer gesetzt)

Optimierung – DP

Lösung von KNAPSACK mit DP

$\Rightarrow$ Erinnerung:

$$P[i, C] = \max\left(\overbrace{P[i-1, C]}^{\text{Nehmen } i \text{ nicht}},\ \underbrace{P[i-1, C-w_i] + p_i}_{\text{Nehmen Gegenstand } i \text{ mit}}\right)$$

- Ausfüllen für $i = 0$: Keine Gegenstände $\Rightarrow$ Kein Profit: $P[0, \_] := 0$
- Ausfüllen für $i = 1$: Einfach (immer **rein**, sobald Platz **reicht**)

... Rest mit Formel ausfüllen...

$\Rightarrow$ Am **Ende**: $P[n, M]$ gibt **maximalen** Profit an

- Item-Menge rekonstruieren: $Taken[i, C]$ **rückwärts** laufen ab $C := M$

  for $i := n$ **downto** 1 **do**

  $\quad NowReallyTaken[i] := Taken[i, C]$

  $\quad$ **if** $Taken[i, C]$ **then** $C -= w_i$

- **Gesamt-Laufzeit**: $O(n \cdot M)$, aber **pseudo**polynomiell

Optimierung – DP: Exkurs (Nicht klausurrelevant)

Ein haarspaltender Einwurf

function InsanelyComplicated($n : \mathbb{N}$)

  $sum := 0$

  for $i := 1$ **to** n **do**

    $sum{++}$

  return sum

Welche Laufzeit hat dieser Algorithmus?

Optimierung – DP: Exkurs (Nicht klausurrelevant)

Laufzeit: Mehr Schein als Sein

- **Eigentlich** heißt „Laufzeit“: Laufzeit in Bezug auf Eingabe**größe** („*wieviel* Elemente zum Durchlaufen“ o. ä.)
- Eingabe keine Elemente, sondern ein **Wert** n?
  n wird (meist binär) **kodiert** in Größe $a := \log n$
  $\Rightarrow$ a ist **tatsächliche** Eingabegröße!
  $\Rightarrow$ die eigentliche Laufzeit: $O(n) = O(2^a)$ $\Rightarrow$ **exponentiell**
- **Aber**: Laufzeit immerhin polynomiell in Bezug auf (größten) Eingabe**wert** n $\Rightarrow$ Bezeichnung: **Pseudo**polynomiell

Optimierung – DP (Nicht klausurrelevant)

KNAPSACK mit DP: Laufzeit

- DP-Algorithmus für KNAPSACK: **Laufzeit** in $O(n \cdot M)$
- n Elemente sind „**echt da**“, aber M ist „irgendein **Wert**“
  $\Rightarrow$ Laufzeit auch **pseudopolynomiell**
- KNAPSACK ist $\mathcal{NP}$**-vollständig**, d. h. für KNAPSACK ist **kein echt polynomieller** Algo bekannt

$\Rightarrow$ So einer würde die **große ungelöste Frage** $\mathcal{P} = \mathcal{NP}$ klären (und dem Finder 1 000 000 \$ einbringen ☺).

- Mehr dazu in TGI nächstes Semester...

Optimierungsprobleme – ILPs

(Integer) Linear Programming

Lineares Programm (LP)
mit n Variablen und m Constraints ($=$ Beschränkungen):

- **Lösungsvektor** $x = (x_1, \ldots, x_n) \in \mathbb{R}^n_{\geqslant 0}$ (wird **gesucht**)
- **Kosten**-/**Gewinnvektor** $c = (c_1, \ldots, c_n) \in \mathbb{R}^n$;
  $f(x) = c \cdot x = \sum c_i x_i$ soll minimiert/maximiert werden
- m **Constraints**, für $j = 1 \ldots m$:

$$a_j \cdot x \begin{Bmatrix} \leqslant \\ = \\ \geqslant \end{Bmatrix} b_j \quad \text{mit } a_j = (a_{j1}, \ldots, a_{jn}) \in \mathbb{R}^n,\ b_j \in \mathbb{R}$$

Varianten:

- **Integer LP**: LP mit allen $x_i \in \mathbb{N}_0$
  (oft durch Constraints auch $x_i \in \{0, 1\}$)
- **Mixed ILP**: LP, bei dem **einige** (aber nicht alle) $x_i \in \mathbb{N}_0$ sind

Optimierungsprobleme – ILPs

Beispiel: KNAPSACK als ILP

- **Lösungsvektor** $x \in \{0, 1\}^n$:
  $x_i = 1 \Leftrightarrow$ Gegenstand i wird eingepackt
- Profitwerte p_i bilden schon Profitvektor p:
  $\Rightarrow$ **Profitfunktion** $f(x) = p \cdot x$ soll **maximiert** werden.
- **Constraints**: Nur einen, nämlich
  $w \cdot x \leqslant M \quad$ mit $w = (w_1, .., w_n)$

- Wie **lösen** wir das jetzt?
  $\Rightarrow$ Wir **gar nicht**, aber ein *Black-Box-Solver* für ILPs schon ☺

Optimierungsprobleme – ILPs

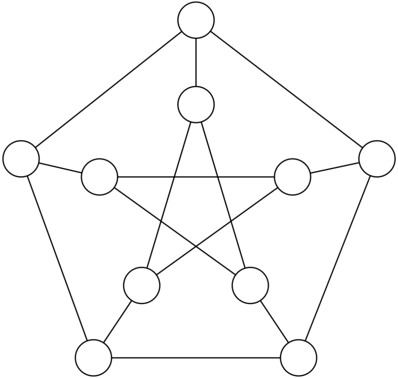
Warum dann überhaupt (M)ILPs?

- Es gibt viele **sehr effiziente** Löser für (M)ILPs
  $\Rightarrow$ **Relevantes Thema**
- **Sehr viele Probleme** können als (M)ILPs formuliert werden
- **Vorgeschmack** auf TGI (Reduktionen, $\mathcal{NP}$-Vollständigkeit)

Optimierungsprobleme – ILPs

Beispiel: VertexCover

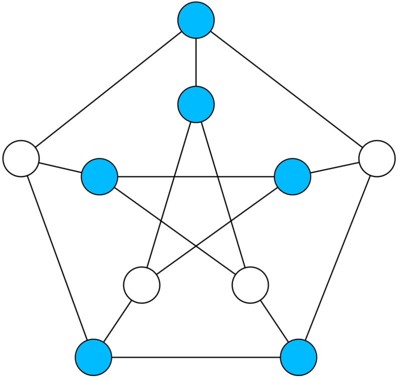
VertexCover: Haben ungerichteten, zusammenhängenden Graphen $G = (V, E)$, wollen **minimale** Teilmenge $C \subseteq V$, so dass $\forall \{u, v\} \in E : v \in C$



Optimierungsprobleme – ILPs

Beispiel: VERTEXCOVER

VERTEXCOVER: Haben ungerichteten, zusammenhängenden Graphen $G = (V, E)$, wollen **minimale** Teilmenge $C \subseteq V$, so dass $\forall\{u, v\} \in E : v \in C$ (**Blau**: Ein mögliches Vertex-Cover C)



Optimierungsprobleme – ILPs

VertexCover als ILP

- **Lösungsvektor** $x \in \{0,1\}^n$: $x_i = 1 \Leftrightarrow$ Knoten $i \in C$
- Kostenvektor $c = (1, .., 1) \in \{1\}^n$,
  minimiere $f(x) = c \cdot x = \sum x_i = |C|$
- m **Constraints**: $\forall \{u, v\} \in E$ jeweils $x_u + x_v \geqslant 1$

Optimierungsprobleme – ILPs

Aufgabe:

Dem ebenso verrückten wie vergesslichen Superbösewicht Doktor Meta ist nach langen Nächten der Schlaflosigkeit wieder eingefallen, dass er ja noch die Weltherrschaft erlangen wollte. Als ersten Schachzug zu seinem genialen Triumph möchte er die Kontrolle über seine Heimatstadt Traffalach gewinnen, um sie im Anschluss zur Welthauptstadt zu erklären. Hierzu plant er, sich ein einzigartiges Merkmal von Traffalach zu Nutze machen: Als die Stadt gegründet wurde, unterteilte man das Gebiet großflächig in Besitztümer, wobei für jedes Besitztum wiederum mehrere Besitzurkunden unter den Siedlern verteilt wurden. Aus nicht näher bekannten Gründen wurde zudem in der Traffalacher Verfassung festgehalten, dass dem, dem es gelingen sollte, für jedes Besitztum eine der Besitzurkunden zu erlangen, der Besitzanspruch für die gesamte Stadt zufällt.
Da die mächtigen Herrscherfamilien Traffalachs traditionell untereinander bis aufs Blut verfeindet sind, ist dies bis heute noch niemandem gelungen, doch mit Hilfe eines intriganten Netzwerkes von Unterhändlern konnte Doktor Meta eine Menge von n Angeboten erhalten. Ärgerlicherweise weigern sich seine Geschäftspartner, die Urkunden einzeln zu verkaufen und verlangen stattdessen jeweils eine stattliche Summe von c_i Euro für die Urkundenmenge U_i. Doktor Meta hat bereits analysiert, dass die Angebote mehr als ausreichen, um für alle k Besitztümer eine Urkunde zu bekommen. Daher möchte er nun so wenig Geld wie möglich für die Übernahme von Traffalach ausgeben (um möglichst viele Reserven für seinen weiteren Weltherrschafts-Feldzug übrig zu haben). Da er kürzlich gehört hat, was für eine tolle Sache ILPs doch sind, soll ein solches hierbei zum Einsatz kommen, um die Menge der Angebote zu bestimmen, auf die Doktor Meta eingehen sollte.

Formuliert das Problem als ILP.

Optimierungsprobleme – ILPs

Lösung:

- **Lösungsvektor** $x \in \{0,1\}^n$,
  $x_i = 1 \Leftrightarrow$ Urkundenmenge U_i wird gekauft
- **Kostenvektor** $c = (c_1, ..., c_n) \in \mathbb{R}^n_{\geqslant 0}$
  Minimiere $f(x) = c \cdot x = \sum c_i x_i$
- k **Constraints**, für $j = 1...k$:
  $$\sum_{i=1}^{n} U_{ij} \cdot x_i \geqslant 1 \quad \text{mit } U_{ij} = \begin{cases} 1, & \text{Urkunde } j \in U_i \\ 0, & \text{Urkunde } j \notin U_i \end{cases}$$

Das Problem heißt allgemein übrigens SetCover.

Schönes Wochenende noch! ☺