Karlsruher Institut für Technologie

Algorithmen I
Tutorium 33

Woche 12 | 13. Juli 2018

Daniel Jungkind (daniel.jungkind@student.kit.edu)

INSTITUT FÜR THEORETISCHE INFORMATIK

Inhalt

Schwarzes Brett + Klausurinfos!

- **Klausur** findet statt am **04.09.2018** von **8–10 Uhr**
- **Erlaubt**: Stifte, 4-Gänge-Menü, **Cheatsheet** (1 DIN-A4-Blatt beidseitig beliebig beschrieben)
- Klausur**anmeldung** bis 28.08.18, 12 Uhr.
  Klausur**abmeldung** bis 28.08.18, 12 Uhr, danach nur **direkt** vor Klausur im HS!

OPTIMIERUNGSPROBLEME

First World Problems

Optimierungsprobleme

Mehr Effizienz

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

...

⇒ Alles **Optimierungsprobleme**

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

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
  $\sum\limits_{i \in X} p_i$ **maximal** wird, aber $\sum\limits_{i \in X} w_i \leqslant M$ bleibt
- **Nicht alle** Gegenstände passen in den Rucksack

Lösungsansätze?

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.: $M = 10,\ (p_i, w_i) = (8,6), (5,5), (5,5)$ ↯

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

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.: $M = 10$, $(p_i, w_i) = (8, 6), (5, 5), (5, 5)$ ↯

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

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.: $M = 10$, $(p_i, w_i) = (8, 6), (5, 5), (5, 5)$ ↯

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

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.: $M = 10$, $(p_i, w_i) = (8, 6), (5, 5), (5, 5)$ ↯

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

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.: $M = 10,\ (p_i, w_i) = (8, 6), (5, 5), (5, 5)$ ↯

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

Optimierung – DP

Rekursion rückwärts: Dynamic Programming (DP)

- **Rekursion** aka **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

Rekursion rückwärts: Dynamic Programming (DP)

- **Rekursion** aka **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

Rekursion rückwärts: Dynamic Programming (DP)

- **Rekursion** aka **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

Karlsruher Institut für Technologie

Rekursion rückwärts: Dynamic Programming (DP)

- **Rekursion** aka **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

Rekursion rückwärts: Dynamic Programming (DP)

- **Rekursion** aka **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

Beispiel: Fibonacci-Zahlen

- Definiere $\mathrm{fib}(0) := \mathrm{fib}(1) := 1$,
  $$\mathrm{fib}(n) := \mathrm{fib}(n-1) + \mathrm{fib}(n-2) \quad \forall n \geqslant 2$$
- Berechne fib(40):
  $\mathrm{fib}(40) = \mathrm{fib}(39) + \mathrm{fib}(38)$,
  $\mathrm{fib}(39) = \mathrm{fib}(38) + \mathrm{fib}(37) \quad \Rightarrow \mathrm{fib}(38)$ wird zweimal berechnet

$\Rightarrow$ fib(5) wird ~ *15-Mio.-mal* berechnet: **Performance-Katastrophe!**

- Besser: Von **unten nach oben** rechnen und **Tabelle** ausfüllen
  $\mathrm{fib} = (1, 1, 0 \ldots 0) : \mathbf{array}[0..40]\ \mathbf{of}\ \mathbb{Z}$
  for $i := 2$ **to** 40 **do**
  $\quad \mathrm{fib}[i] := \mathrm{fib}[i-1] + \mathrm{fib}[i-2]$

i	0	1	2	3	...
fib[i]	1	1	2	3	...

$\longrightarrow$

$\Rightarrow$ „Rekursion rückwärts“

Optimierung – DP

Beispiel: Fibonacci-Zahlen

- Definiere $\mathrm{fib}(0) := \mathrm{fib}(1) := 1$,
  $$\mathrm{fib}(n) := \mathrm{fib}(n-1) + \mathrm{fib}(n-2) \quad \forall n \geqslant 2$$
- Berechne fib(40):
  $\mathrm{fib}(40) = \mathrm{fib}(39) + \mathrm{fib}(38)$,
  $\mathrm{fib}(39) = \mathrm{fib}(38) + \mathrm{fib}(37) \quad \Rightarrow \mathrm{fib}(38)$ wird zweimal berechnet

$\Rightarrow$ fib(5) wird $\sim$ *15-Mio.-mal* berechnet: **Performance-Katastrophe!**

- Besser: Von **unten nach oben** rechnen und **Tabelle** ausfüllen
  $\mathrm{fib} = (1, 1, 0 \ldots 0) : \mathrm{array}[0..40]$ of $\mathbb{Z}$
  for $i := 2$ to 40 do
  $\quad \mathrm{fib}[i] := \mathrm{fib}[i-1] + \mathrm{fib}[i-2]$

i	0	1	2	3	...
fib[i]	1	1	2	3	...

$\longrightarrow$

$\Rightarrow$ „Rekursion rückwärts“

Optimierung – DP

Beispiel: Fibonacci-Zahlen

- Definiere $\text{fib}(0) := \text{fib}(1) := 1,$
  $$\text{fib}(n) := \text{fib}(n-1) + \text{fib}(n-2) \quad \forall n \geqslant 2$$
- Berechne $\text{fib}(40)$:
  $\text{fib}(40) = \text{fib}(39) + \text{fib}(38),$
  $\text{fib}(39) = \text{fib}(38) + \text{fib}(37) \quad \Rightarrow \text{fib}(38)$ wird zweimal berechnet

$\Rightarrow$ $\text{fib}(5)$ wird ~ *15-Mio.-mal* berechnet: **Performance-Katastrophe**!

- Besser: Von **unten nach oben** rechnen und **Tabelle** ausfüllen
  $\text{fib} = (1, 1, 0 \dots 0) : \text{array}[0..40] \text{ of } \mathbb{Z}$
  for $i := 2$ to 40 do
  $\quad \text{fib}[i] := \text{fib}[i-1] + \text{fib}[i-2]$

i	0	1	2	3	...
$\text{fib}[i]$	1	1	2	3	...

$\Longrightarrow$

$\Rightarrow$ „Rekursion rückwärts"

Optimierung – DP

Beispiel: Fibonacci-Zahlen

- Definiere $\text{fib}(0) := \text{fib}(1) := 1$,
  $$\text{fib}(n) := \text{fib}(n-1) + \text{fib}(n-2) \quad \forall n \geqslant 2$$
- Berechne $\text{fib}(40)$:
  $\text{fib}(40) = \text{fib}(39) + \text{fib}(38)$,
  $\text{fib}(39) = \text{fib}(38) + \text{fib}(37) \quad \Rightarrow \text{fib}(38)$ wird zweimal berechnet

$\Rightarrow$ $\text{fib}(5)$ wird ~ *15-Mio.-mal* berechnet: **Performance-Katastrophe**!

- Besser: Von **unten nach oben** rechnen und **Tabelle** ausfüllen
  $\text{fib} = (1, 1, 0 \dots 0)$: **array**$[0..40]$ **of** $\mathbb{Z}$
  for $i := 2$ **to** 40 **do**
  $\text{fib}[i] := \text{fib}[i-1] + \text{fib}[i-2]$

i	0	1	2	3	...
$\text{fib}[i]$	1	1	2	3	...

$\Longrightarrow$

$\Rightarrow$ „Rekursion rückwärts“

Optimierung – DP

Lösung von KNAPSACK mit DP

- Lege zweidim. P : **array**$[0..n,\ 0..M]$ **of** $\mathbb{R}$ an:
  $\Rightarrow P[i, C] =$ **optimaler Profit** für betrachtete Gegenstände $1 \ldots i$ 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
  $P[i, C] := \max\left(P[i-1, C],\ P[i-1, C-w_i] + p_i\right)$
  $Taken[i, C] := \left(P[i-1, C] < P[i-1, C-w_i] + p_i\right)$: Bool
- Bedeutet („Pseudo-pseudo", so NICHT in der Klausur!):
  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

- Lege zweidim. P : **array**$[0..n,\ 0..M]$ **of** $\mathbb{R}$ an:
  $\Rightarrow P[i, C] =$ **optimaler Profit** für betrachtete Gegenstände $1 \ldots i$ 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**
  $P[i, C] := \max\left(P[i-1, C],\ P[i-1, C-w_i] + p_i\right)$
  $Taken[i, C] := \left(P[i-1, C] < P[i-1, C-w_i] + p_i\right)$: Bool
- Bedeutet („Pseudo-pseudo", so NICHT in der Klausur!):
  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

- Lege zweidim. P : **array**$[0..n,\ 0..M]$ **of** $\mathbb{R}$ an:
  $\Rightarrow P[i, C] =$ **optimaler Profit** für betrachtete Gegenstände $1 \ldots i$
  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**
  $P[i, C] := \max\left(P[i-1, C],\ P[i-1, C-w_i] + p_i\right)$
  $Taken[i, C] := \left(P[i-1, C] < P[i-1, C-w_i] + p_i\right)$: Bool
- Bedeutet („Pseudo-pseudo“, so NICHT in der Klausur!):
  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

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

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

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

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
  $NowReallyTaken[i] := Taken[i, C]$
  if $Taken[i, C]$ then $C -= w_i$
- **Gesamt-Laufzeit:** $O(n \cdot M)$, aber **pseudo**polynomiell

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
  $NowReallyTaken[i] := Taken[i, C]$
  if $Taken[i, C]$ then $C -= w_i$
- **Gesamt-Laufzeit**: $O(n \cdot M)$, aber **pseudo**polynomiell

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: $\mathit{Taken}[i, C]$ **rückwärts** laufen ab $C := M$

  for $i := n$ **downto** 1 **do**
  $\quad \mathit{NowReallyTaken}[i] := \mathit{Taken}[i, C]$
  $\quad$ **if** $\mathit{Taken}[i, C]$ **then** $C \mathrel{-}= w_i$

- **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: Exkurs (Nicht klausurrelevant)

Karlsruher Institut für Technologie

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: 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** $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ klären (und dem Finder 1 000 000 $ einbringen ☺).

- Mehr dazu in TGI nächstes Semester...

Optimierung – DP (Nicht klausurrelevant)

Karlsruher Institut für Technologie

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** $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ klären
(und dem Finder 1 000 000 \$ einbringen ☺).

- Mehr dazu in TGI nächstes Semester...

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** $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ klären
(und dem Finder 1 000 000 $ einbringen ☺).

- Mehr dazu in TGI nächstes Semester...

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** $\mathcal{P} \stackrel{?}{=} \mathcal{NP}$ klären
(und dem Finder 1 000 000 \$ einbringen ☺).

- Mehr dazu in TGI nächstes Semester...

Optimierung – DP

Aufgabe 1: Ein paar Euro für den Frieden

Der ebenso berechenbare wie sterbliche Turing-Man (halb Mensch, halb Turing-Maschine) ist Doktor Meta auf den Fersen! Nachdem er dank der Hilfe einiger pfiffiger Informatik-Studenten am KIT die Pläne des Superbösewichts aufdecken konnte, will er nun die Wechselkurse gehörig aufmischen. Dafür braucht er allerdings selbst hohe Geldsummen. Glücklicherweise kennt Turing-Man ein paar alte Bekannte, die ihm genau dabei aushelfen können: Ausrangierte Geldautomaten! In Reih' und Glied stehen n solcher Maschinen auf einem Schrottplatz, wobei jeder Automat $i = 1, ..., n$ eine Menge an Münzen a_i beinhaltet. Diese gibt er allerdings nur ab, wenn Turing-Man den Automaten davor nicht um Geld gebeten hat. Der wandelnde Schreiblesekopf muss nun die Automaten der Reihe nach ablaufen und sich für jeden Automaten entscheiden, ob er ihn anpumpt oder nicht. Erfreulicherweise kennt Turing-Man seine alten Freunde in- und auswendig und weiß ihre Münzanzahl.
Helft dem mechanischen Helden und löst dieses Problem mittels DP.

Optimierung – DP

Lösung zu Aufgabe 1

- **Idee**: Frage ich den Automaten, oder frage ich ihn nicht?
- Sei $C[i] =$ Anzahl der Münzen, die Turing-Man von Automat i bis n im Best-Case einsammeln kann
- n ist letzter Automat: Setze $C[n] := a_n$ (Best-Case).
  Setze $\forall j > n : \quad C[j] := 0.$

$\Rightarrow$ Rekursionsformel:

$$C[i] = \max(\underbrace{C[i+1]}_{\text{Fragen } i \text{ nicht}}, \underbrace{C[i+2] + a_i}_{\text{Fragen } i \text{ doch}})$$

$\Rightarrow$ Von hinten nach vorne ausfüllen:

for $i := n - 1$ **downto** 1 **do**
  $C[i] := \max(C[i+1], C[i+2] + a_i)$

$Taken[\cdot]$ rekonstruieren:

for $i := 1$ **to** n **do**
  if $C[i+1] < C[i+2] + a_i$ **then** $Taken[i] :=$ **true**

Optimierung – DP

Lösung zu Aufgabe 1

- **Idee**: Frage ich den Automaten, oder frage ich ihn nicht?
- Sei $C[i]$ = Anzahl der Münzen, die Turing-Man von Automat i bis n im Best-Case einsammeln kann
- n ist letzter Automat: Setze $C[n] := a_n$ (Best-Case).
  Setze $\forall j > n: \quad C[j] := 0$.

$\Rightarrow$ Rekursionsformel:

$$C[i] = \max(\underbrace{C[i+1]}_{\text{Fragen } i \text{ nicht}}, \underbrace{C[i+2] + a_i}_{\text{Fragen } i \text{ doch}})$$

$\Rightarrow$ Von hinten nach vorne ausfüllen:

for $i := n-1$ **downto** 1 **do**
  $C[i] := \max(C[i+1], C[i+2] + a_i)$

$Taken[\cdot]$ rekonstruieren:

for $i := 1$ **to** n **do**
  if $C[i+1] < C[i+2] + a_i$ **then** $Taken[i] :=$ **true**

Optimierung – DP

Lösung zu Aufgabe 1

- **Idee**: Frage ich den Automaten, oder frage ich ihn nicht?
- Sei $C[i] =$ Anzahl der Münzen, die Turing-Man von Automat i bis n im Best-Case einsammeln kann
- n ist letzter Automat: Setze $C[n] := a_n$ (Best-Case).
  Setze $\forall j > n : \quad C[j] := 0$.

$\Rightarrow$ Rekursionsformel:

$$C[i] = \max(\underbrace{C[i+1]}_{\text{Fragen } i \text{ nicht}}, \underbrace{C[i+2] + a_i}_{\text{Fragen } i \text{ doch}})$$

$\Rightarrow$ Von hinten nach vorne ausfüllen:

for $i := n - 1$ **downto** 1 **do**
  $C[i] := \max(C[i+1], C[i+2] + a_i)$

$Taken[\cdot]$ rekonstruieren:

for $i := 1$ **to** n **do**
  if $C[i+1] < C[i+2] + a_i$ **then** $Taken[i] :=$ **true**

Optimierungsprobleme – ILPs

(Integer) Linear Programming

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

- **Lösungsvektor** $x = (x_1, \ldots, x_n) \in \mathbb{R}^n_{\geq 0}$ (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 $j = 1 \ldots m$:

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

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

(Integer) Linear Programming

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

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

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

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

(Integer) Linear Programming

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

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

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

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

(Integer) Linear Programming

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

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

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

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

(Integer) Linear Programming

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

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

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

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

(Integer) Linear Programming

Lineares Programm / Lineares Problem (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 $w = (w_1, \dots, w_n)$
- In üblicher Schreibweise:

$$\begin{aligned} \max_{x \in \{0,1\}^n} \quad & f(x) = p \cdot x \\ \text{sodass} \quad & w \cdot x \leqslant M \\ \text{mit} \quad & w = (w_1, \dots, w_n),\ p = (p_1, \dots, p_n). \end{aligned}$$

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

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 $w = (w_1, \dots, w_n)$
- In üblicher Schreibweise:

$$\begin{aligned} \max_{x \in \{0,1\}^n} \quad & f(x) = p \cdot x \\ \text{sodass} \quad & w \cdot x \leqslant M \\ \text{mit} \quad & w = (w_1, \dots, w_n),\ p = (p_1, \dots, p_n). \end{aligned}$$

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

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 $w = (w_1, .., w_n)$
- In üblicher Schreibweise:

$$\max_{x \in \{0,1\}^n} \quad f(x) = p \cdot x$$

$$\text{sodass} \quad w \cdot x \leqslant M$$

$$\text{mit} \quad w = (w_1, .., w_n),\ p = (p_1, .., p_n).$$

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

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 $w = (w_1, .., w_n)$
- In üblicher Schreibweise:

$$\begin{aligned} \max_{x \in \{0,1\}^n} \quad & f(x) = p \cdot x \\ \text{sodass} \quad & w \cdot x \leqslant M \\ \text{mit} \quad & w = (w_1, .., w_n),\ p = (p_1, .., p_n). \end{aligned}$$

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

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 $w = (w_1, .., w_n)$
- In üblicher Schreibweise:

$$\begin{aligned} \max_{x \in \{0,1\}^n} \quad & f(x) = p \cdot x \\ \text{sodass} \quad & w \cdot x \leqslant M \\ \text{mit} \quad & w = (w_1, .., w_n),\ p = (p_1, .., p_n). \end{aligned}$$

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

Optimierungsprobleme – ILPs

Warum dann überhaupt (M)ILPs?

- LPs **polynomiell** lösbar, ILPs i. A. **nicht** ($\mathcal{NP}$-schwer)
- Es gibt trotzdem 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)
- (Ein ganzes Ergänzungsfach dazu: Operations Research)

Optimierungsprobleme – ILPs

Karlsruher Institut für Technologie

Warum dann überhaupt (M)ILPs?

- LPs **polynomiell** lösbar, ILPs i. A. **nicht** ($\mathcal{NP}$-schwer)
- Es gibt trotzdem 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)
- (Ein ganzes Ergänzungsfach dazu: Operations Research)

Optimierungsprobleme – ILPs

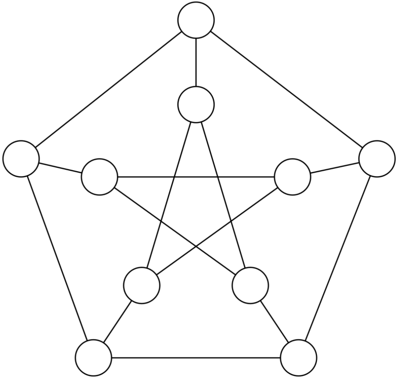
Warum dann überhaupt (M)ILPs?

- LPs **polynomiell** lösbar, ILPs i. A. **nicht** ($\mathcal{NP}$-schwer)
- Es gibt trotzdem 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)
- (Ein ganzes Ergänzungsfach dazu: Operations Research)

Optimierungsprobleme – ILPs

Beispiel: VERTEXCOVER

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

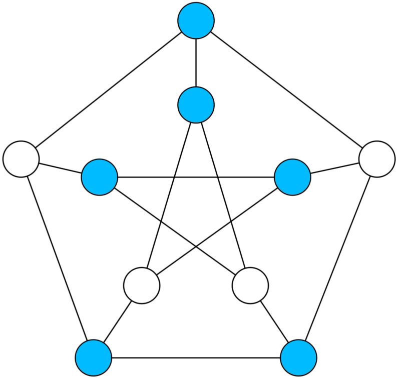


Optimierungsprobleme – ILPs

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

$$\Longrightarrow \qquad \min_{x \in \{0,1\}^n} \quad f(x) = c \cdot x$$

$$\text{sodass} \quad x_u + x_v \geqslant 1 \quad \forall \{u, v\} \in E$$

$$\text{mit} \quad c = (1, ..., 1).$$

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 $x_u + x_v \geq 1$

$$\Longrightarrow \quad \min_{x \in \{0,1\}^n} f(x) = c \cdot x$$

$$\text{sodass} \quad x_u + x_v \geq 1 \quad \forall \{u,v\} \in E$$

$$\text{mit} \quad c = (1,...,1).$$

Optimierungsprobleme – ILPs

VERTEXCOVER als ILP

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

$$\Longrightarrow \qquad \min_{x \in \{0,1\}^n} \quad f(x) = c \cdot x$$

$$\text{sodass} \quad x_u + x_v \geqslant 1 \quad \forall \{u, v\} \in E$$

$$\text{mit} \quad c = (1, ..., 1).$$

Optimierungsprobleme – ILPs

VertexCover als ILP

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

$$\Longrightarrow \qquad \min_{x \in \{0,1\}^n} f(x) = c \cdot x$$

$$\text{sodass} \quad x_u + x_v \geqslant 1 \quad \forall \{u, v\} \in E$$

$$\text{mit} \quad c = (1, .., 1).$$

Optimierungsprobleme – ILPs

Aufgabe 2: Seine Abschiedsvorstellung

Der ebenso geniale wie wahnsinnige Superbösewicht Doktor Meta lacht hysterisch: Die Weltherrschaft ist zum Greifen nah! Der verrückte Superschurke plant nämlich, die gesamte Welt in einer Nacht in Schutt und Asche zu legen, um anschließend eine neue Weltordnung aufzubauen. Dafür muss er lediglich ausreichend Militärdepots und sonstige explosionsgefährdete Lagerstätten in die Luft jagen. Genügend Schurken und Handlanger zu rekrutieren, die bei der Verlegung von Zündschnüren helfen, kostet allerdings viel Geld. Er will daher möglichst wenig Depots in die Luft sprengen und trotzdem die ganze Welt, also alle Planquadrate $1 \ldots k$, ausradieren. Auf der Welt gibt es insgesamt n Depots, wobei ein Depot i die Planquadrate $P_i \subseteq \{1, \ldots, k\}$ dem Erdboden gleichmachen kann, was allerdings c_i Euro kostet. Nun muss der Superbösewicht lediglich die billigste Menge aller Depots bestimmen, die alle Planquadrate $1 \ldots k$ abdeckt.

Helft mit, die Welt zu zerstören und formuliert das Problem als ILP.

Optimierungsprobleme – ILPs

Lösung zu Aufgabe 2

- **Lösungsvektor** $x \in \{0,1\}^n$,
  $x_i = 1 \Leftrightarrow$ Depot i wird zur Sprengung ausgewählt
  ($\hat{=}$ Planquadrate P_i sind abgedeckt)
- **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 $j = 1...k$:
  $$\sum_{i=1}^{n} P_{ij} \cdot x_i \geqslant 1 \quad \text{mit } P_{ij} = \begin{cases} 1, & \text{Planquadrat } j \in P_i \\ 0, & \text{Planquadrat } j \notin P_i \end{cases}$$

$\Rightarrow$ Andere Schreibweise:

$$\min_{x \in \{0,1\}^n} c \cdot x \quad \text{sodass} \quad P^\top \cdot x \geqslant \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix} \quad \text{mit } P = (P_{ij}),\ P_{ij} = ...,\ c = ...$$

Das Problem heißt allgemein übrigens SETCOVER.

Danke für eure Aufmerksamkeit! ☺