Optimierung – Dynamische Programmierung

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 24. Mai 2023

Vorlesungsfolien

Optimierung: Roadmap

Branch-and-Bound

Dynamische Programmierung: Dynamische Programmierung kann dann eingesetzt werden, wenn das Problem aus vielen gleichartigen Teilproblemen besteht und eine optimale Lösung sich aus optimalen Lösungen der Teilprobleme zusammensetzt.

Approximation(salgorithmen)

Heuristische Verfahren

Grundlagen

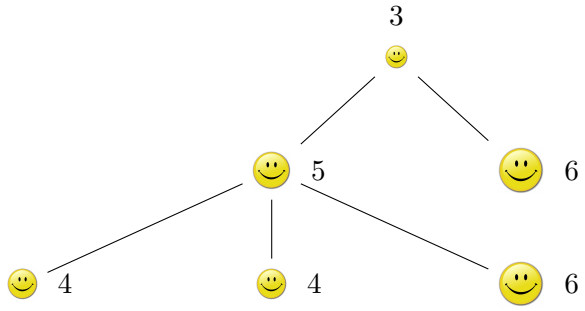
Dynamische Programmierung: Teile das Problem in eine Folge von überlappenden Teilproblemen auf und erstelle und speichere Lösungen für immer größere Teilprobleme unter Verwendung der abgespeicherten Lösungen.

Optimalitätsprinzip von Bellman: Dynamische Programmierung führt zu einem optimalen Ergebnis genau dann, wenn es sich aus den optimalen Ergebnissen der Subprobleme zusammensetzt.

Effizienz: Hängt von der Vorgehensweise bei der Aufteilung und Ermittlung der Lösungen für die einzelnen Teilprobleme ab.

Wesentlicher Aspekt: Speicherung (*memoization*) von Ergebnissen für Subprobleme zur Wiederverwendung.

Beispiel: Weighted Independent Set auf Bäumen



Geschichte der dynamischen Programmierung

Bellman: [1950er] Leistete Pionierarbeit bei der systematischen Untersuchung der dynamischen Programmierung.

Etymologie:

- Dynamische Programmierung = Zeitablauf planen.
- Verteidigungsminister war ablehnend gegenüber mathematischer Forschung.
- Bellman suchte einen eindrucksvollen Namen, um eine Konfrontation zu vermeiden.

„it's impossible to use dynamic in a pejorative sense“
„something not even a Congressman could object to“

Referenz: Bellman, R. E. Eye of the Hurricane, An Autobiography.

Anwendung der dynamischen Programmierung

Bereiche:

- Bioinformatik
- Informationstheorie
- Operations Research
- Informatik: Theorie, Grafik, Künstliche Intelligenz, Compilerbau ...

Einige bekannte Algorithmen:

- Bellman-Ford-Algorithmus für das Finden kürzester Pfade in Graphen
- Effiziente Methode für das Rucksack-Problem
- Needleman-Wunsch und Smith-Waterman Algorithmen für Genomsequenz-Alignment

Überblick

Einführendes Beispiel: Fibonacci

Gewichtetes Interval Scheduling

Segmented Least Squares

Rucksackproblem

Kürzeste Pfade

Einführendes Beispiel

Fibonacci-Zahlen

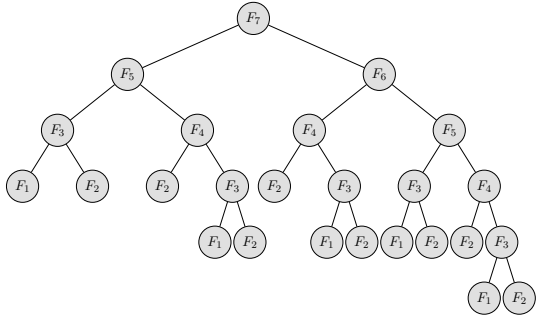
Folge von Fibonacci-Zahlen: $F_1 = F_2 = 1 \quad F_n = F_{n-1} + F_{n-2} \quad \forall n \geq 3$

Einfacher rekursiver Algorithmus:

```
Fibonacci(n):
if n = 1 oder n = 2
  return 1
else
  return Fibonacci(n − 1) + Fibonacci(n − 2)
```

Fibonacci-Zahlen: Laufzeit

- Gesamtanzahl der Aufrufe für die i-te Fibonacci-Zahl entspricht der i-ten Fibonacci-Zahl (Beispiele: $F_{10} = 55$, $F_{20} = 6765$, $F_{30} = 832040$, $F_{40} = 102334155$).
- Exponentielle Zeitkomplexität, da $F_n = F_{n-1} + F_{n-2} \geq 2F_{n-2}$ und folglich $F_n \geq 2^{\frac{n}{2}-1}$ (mit $F_2 = 2^0$) und daher $F_n \geq (\sqrt{2})^{n-2}$



Dynamische Programmierung (Rekursiv)

Speicherung: Die berechneten Fibonacci-Zahlen zwischenspeichern (z.B. in einem Array F) und in der Berechnung wiederverwenden.

```
for i ← 1 bis n
  F[i] ← leer

Fibonacci(n):
if F[n] ist leer
  if n = 1 oder n = 2
    F[n] ← 1
  else
    F[n] ← Fibonacci(n − 1) + Fibonacci(n − 2)
return F[n]
```

Laufzeit: $O(n)$ (maximal zwei rekursive Aufrufe pro Arrayeintrag)

Dynamische Programmierung (Iterativ)

Speicherung: Die berechneten Fibonacci-Zahlen zwischenspeichern (z.B. in einem Array F) und in der Berechnung wiederverwenden.

```
Linear-Fibonacci(n):
F[1] ← 1
F[2] ← 1
for i ← 3 bis n
  F[i] ← F[i − 1] + F[i − 2]
return F[n]
```

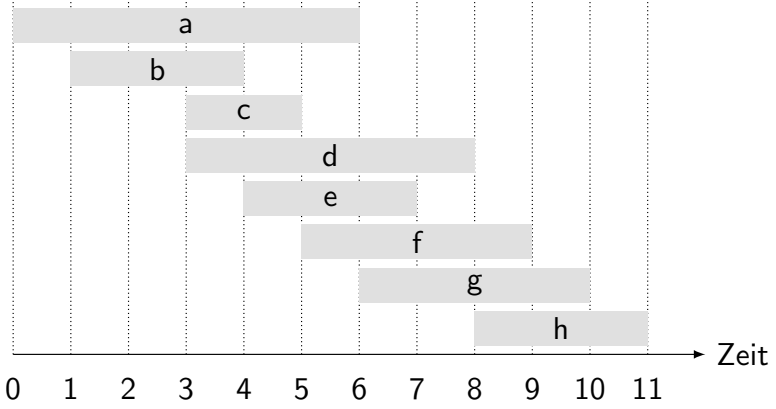
Laufzeit: $O(n)$ (konstanter Aufwand für jeden Schleifendurchlauf)

Gewichtetes Interval Scheduling

Gewichtetes Interval Scheduling

Gewichtetes Interval Scheduling:

- Job j startet zum Zeitpunkt s_j, endet zum Zeitpunkt f_j und hat ein Gewicht $w_j > 0$.
- Zwei Jobs sind kompatibel, wenn sie sich nicht überlappen.
- Ziel: Finde eine Teilmenge maximalen Gewichts von paarweise kompatiblen Jobs.

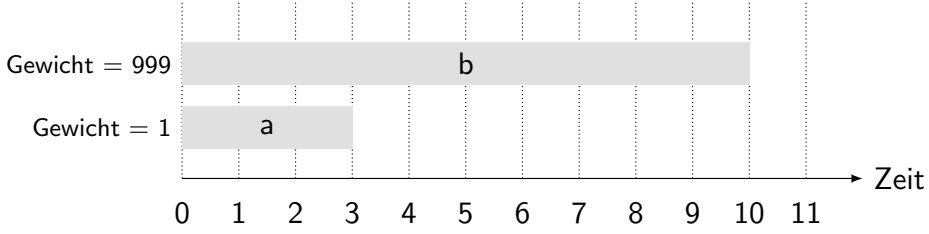


Interval Scheduling: Rückblick

Wiederholung: Greedy-Algorithmus funktioniert, wenn alle Gewichte gleich 1 sind.

- Berücksichtige Jobs in aufsteigender Reihenfolge der Beendigungszeit.
- Füge Job zur Teilmenge hinzu, wenn er kompatibel mit dem zuvor ausgewählten Job ist.

Beobachtung: Greedy-Algorithmus scheitert, wenn beliebige Gewichte erlaubt sind.

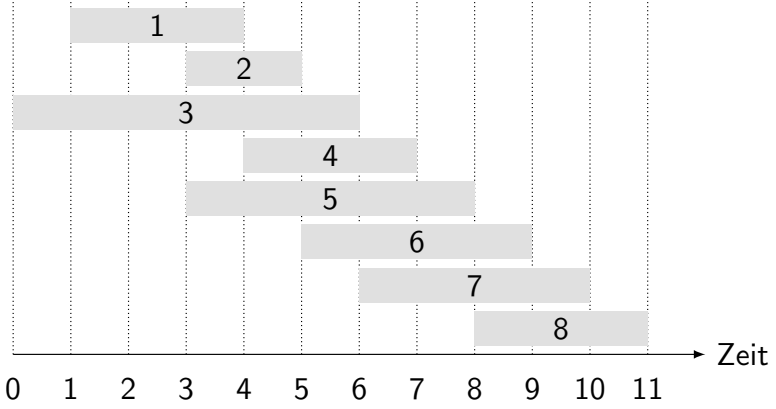


Gewichtetes Interval Scheduling

Notation: Ordne Jobs aufsteigend sortiert nach Beendigungszeit: $f_1 \leq f_2 \leq \cdots \leq f_n$.

Definition: $p(j) =$ größter Index $i < j$, sodass Job i kompatibel zu Job j ist.

Beispiel: $p(8) = 5$, $p(7) = 3$, $p(2) = 0$.



Dynamische Programmierung: Binäre Auswahl

Notation: $OPT(j) =$ Wert der optimalen Lösung für das Problem, bestehend aus den Jobs $1, 2, \ldots, j$.

Wir unterscheiden zwei Fälle:

- Fall 1: $OPT(j)$ wird erreicht mit einer Lösung, die den Job j enthält.
- Fall 2: $OPT(j)$ wird erreicht mit einer Lösung, die den Job j nicht enthält.

Konsequenz:

- Fall 1: Die Lösung kann nicht die inkompatiblen Jobs $\{p(j)+1, p(j)+2, \ldots, j-1\}$ enthalten. Daher gilt dann $OPT(j) = w_j + OPT(p(j))$.
- Fall 2: Es gilt $OPT(j) = OPT(j-1)$, da wir wissen, dass die Lösung den Job j nicht enthält. Also gilt:

$$OPT(j) = \begin{cases} 0 & \text{wenn } j = 0 \\ \max\{w_j + OPT(p(j)), OPT(j-1)\} & \text{sonst} \end{cases}$$

Gewichtetes Interval Scheduling: Brute-Force-Ansatz

Brute-Force-Algorithmus:

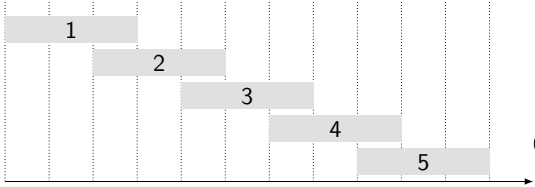
- Eingabe: $n, s_1, \ldots, s_n, f_1, \ldots, f_n, w_1, \ldots, w_n$
- Sortiere Jobs nach Beendigungszeit, sodass $f_1 \leq f_2 \leq \cdots \leq f_n$.
- Berechne $p(1), p(2), \ldots, p(n)$

```
Compute-Opt(j):
if j = 0
  return 0
else
  return max(w_j+Compute-Opt(p(j)), Compute-Opt(j − 1))
```

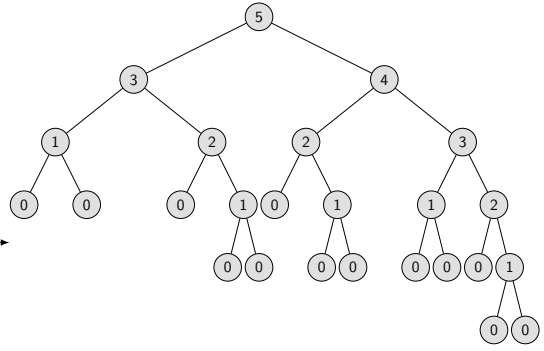
Gewichtetes Interval Scheduling: Brute-Force-Ansatz

Beobachtung: Rekursiver Algorithmus ist ineffizient wegen redundanter Subprobleme $\Rightarrow$ exponentieller Algorithmus.

Beispiel: Anzahl der rekursiven Aufrufe für eine Gruppe von schichtweise angeordneten Instanzen wächst wie eine Fibonacci-Folge.



$p(1) = 0,\ p(j) = j - 2$



Gewichtetes Interval Scheduling: Speicherung

Speicherung: Speichere Ergebnisse jedes Teilproblems in einem Cache. Berechnung nur, wenn noch nicht gespeichert.

Allgemein:

- Eingabe: n, $s_1, \ldots, s_n$, $f_1, \ldots, f_n$, $w_1, \ldots, w_n$
- Sortiere Jobs nach Beendigungszeit, sodass $f_1 \leq f_2 \leq \cdots \leq f_n$.
- Berechne $p(1), p(2), \ldots, p(n)$

```
for j ← 1 bis n
    M[j] ← leer
M[0] ← 0

M-Compute-Opt(j):
if  M[j] ist leer
    M[j] ← max(w_j + M-Compute-Opt(p(j)), M-Compute-Opt(j − 1))
return M[j]
```

- *globales Array*

Gewichtetes Interval Scheduling: Laufzeit

Behauptung: Version mit Speicherung benötigt $O(n \log n)$ Zeit.

- Sortiere nach Beendigungszeit: $O(n \log n)$.
- Berechne $p(\cdot)$: $O(n \log n)$ mittels binärer Suche (für jedes Interval) auf der nach Beendigungszeit sortierten Folge.
- `M-Compute-Opt(j)`: Jeder Aufruf benötigt $O(1)$ Zeit (ohne die Rekursion) und
  - (i) liefert entweder einen existierenden Wert `M[j]`
  - (ii) oder berechnet einen neuen Eintrag `M[j]` und macht zwei rekursive Aufrufe.
- Maß für den Fortschritt $\varphi =$ die Anzahl der nicht leeren Einträge in `M[]`.
  - Am Anfang gilt $\varphi = 0$, danach $\varphi \leq n$.
  - (ii) Erhöht φ um 1.
- Die gesamte Laufzeit von `M-Compute-Opt(n)` ist $O(n)$.

Gewichtetes Interval Scheduling: Bottom-up

Bottom-up dynamische Programmierung: Iterative Lösung.

Allgemein:

- Eingabe: n, $s_1, \ldots, s_n$, $f_1, \ldots, f_n$, $w_1, \ldots, w_n$.
- Sortiere Jobs nach Beendigungszeit, sodass $f_1 \leq f_2 \leq \cdots \leq f_n$.
- Berechne $p(1), p(2), \ldots, p(n)$

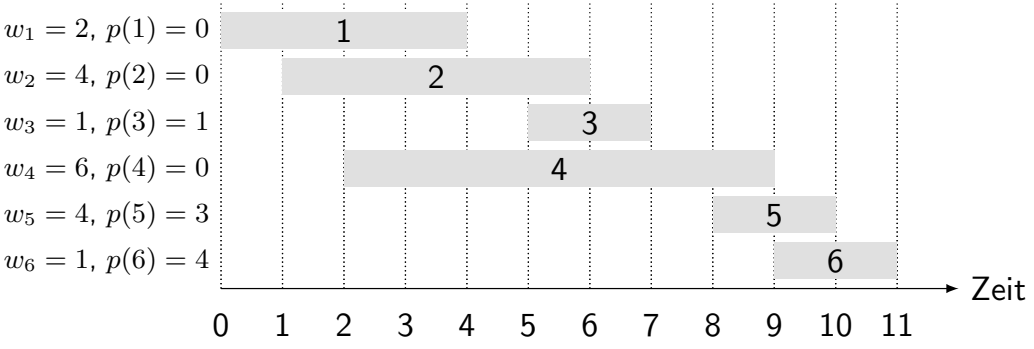
```
Iterative-Compute-Opt():
M[0] ← 0
for j ← 1 bis n
    M[j] ← max(w_j + M[p(j)], M[j − 1])
```

Laufzeit: Die Laufzeit von `Iterative-Compute-Opt` liegt in $O(n)$ (Schleife von 1 bis n).

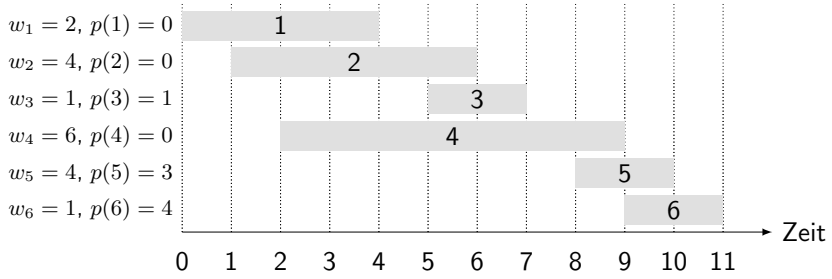
Gewichtetes Interval Scheduling: Beispiel

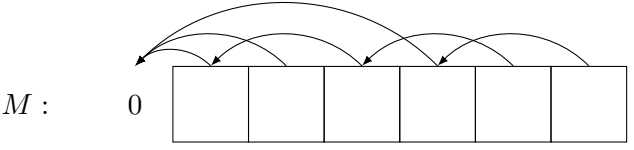
Gegeben:

- $n = 6$ Jobs mit Gewichten $w_i, i = 1 \ldots n$.
- Jobs sind schon sortiert nach Beendigungszeit.



Gewichtetes Interval Scheduling: Beispiel

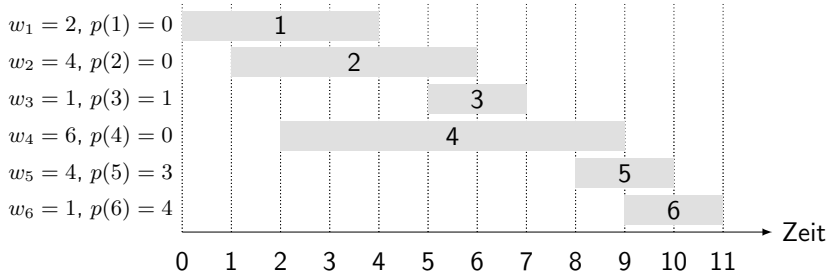


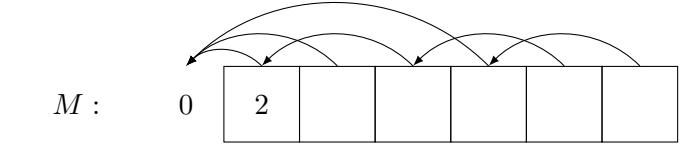


$w_j + M[p(j)]$:

$M[j-1]$:

Gewichtetes Interval Scheduling: Beispiel

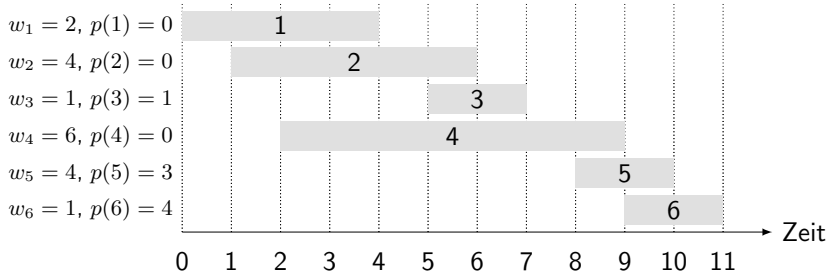


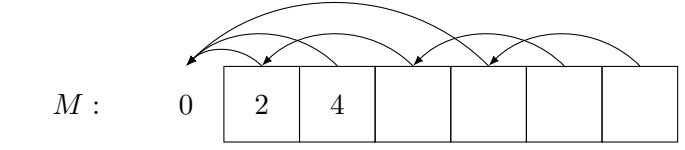


$w_j + M[p(j)]$: 2

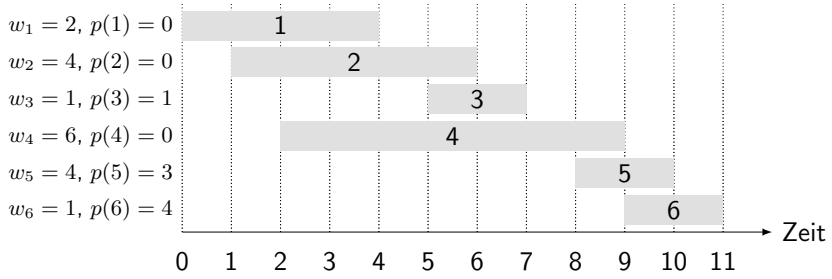
$M[j-1]$: 0

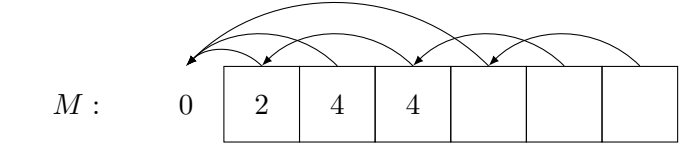
Gewichtetes Interval Scheduling: Beispiel





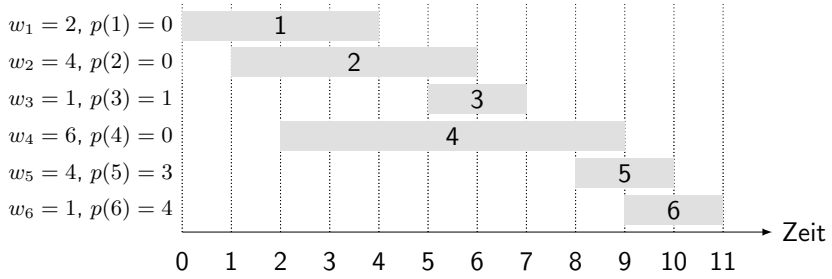
Gewichtetes Interval Scheduling: Beispiel

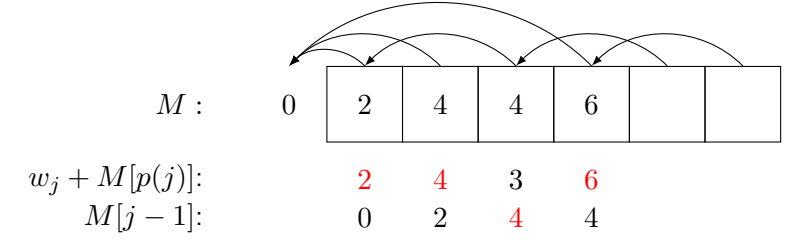




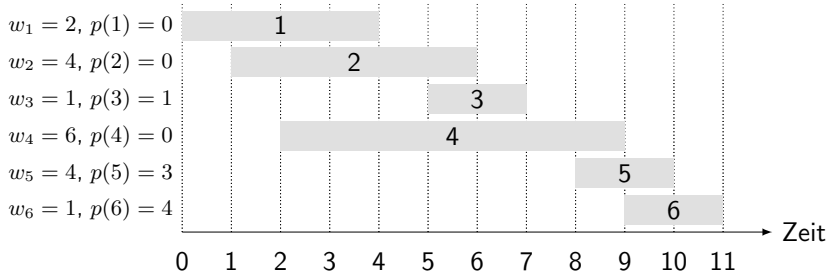
$w_j + M[p(j)]$:	2	4	3
$M[j-1]$:	0	2	4

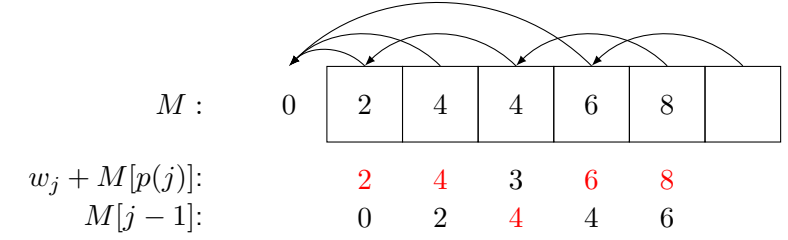
Gewichtetes Interval Scheduling: Beispiel



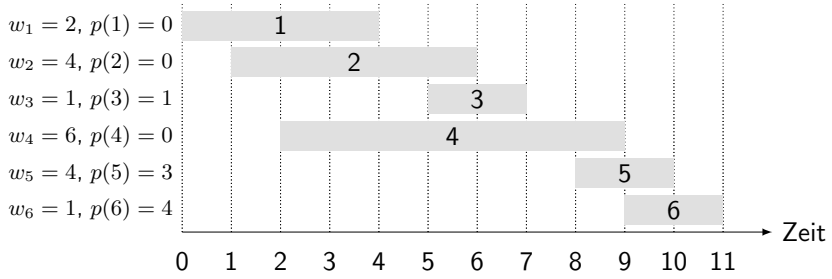


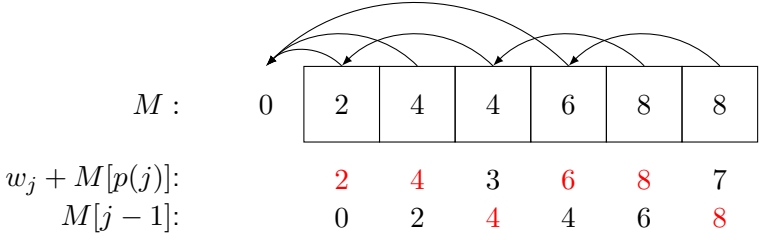
Gewichtetes Interval Scheduling: Beispiel





Gewichtetes Interval Scheduling: Beispiel





Gewichtetes Interval Scheduling: Finden einer Lösung

Frage: Algorithmus berechnet den optimalen Wert. Wie bekommen wir aber die Lösung?
Antwort: Durch eine Nachbearbeitung (Backtracking).

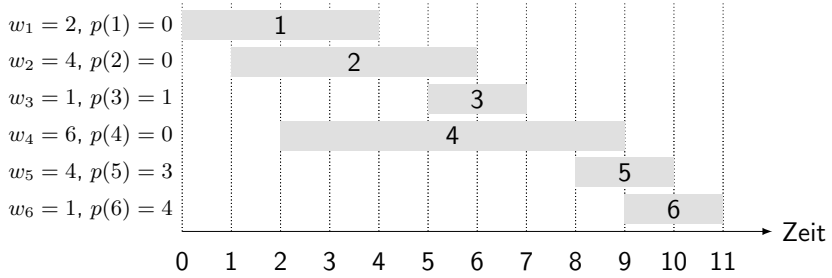
Ablauf:

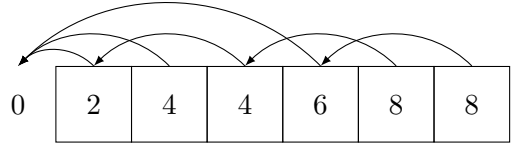
- `M-Compute-Opt(n)` oder `Iterative-Compute-Opt(n)` ausführen
- `Find-Solution(n)` ausführen

```
Find-Solution(j):
if j = 0
    Keine Ausgabe
elseif w_j + M[p(j)] > M[j − 1]
    Gib j aus
    Find-Solution(p(j))
else
    Find-Solution(j − 1)
```

- Anzahl der rekursiven Aufrufe $\leq n \Rightarrow$ Laufzeit $O(n)$.

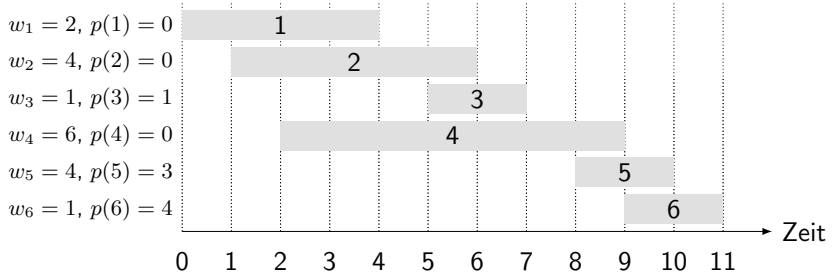
Gewichtetes Interval Scheduling: Beispiel

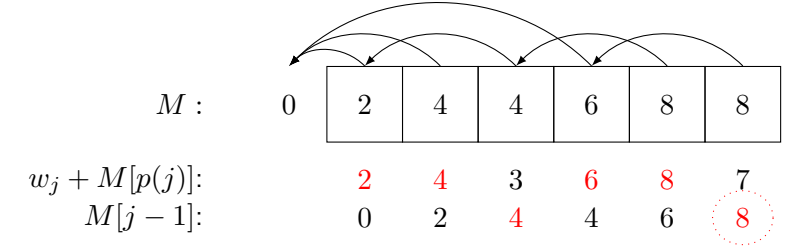




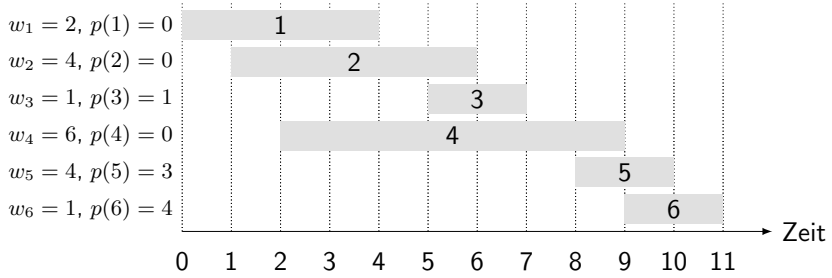
$w_j + M[p(j)]$:	2	4	3	6	8	7
$M[j-1]$:	0	2	4	4	6	8

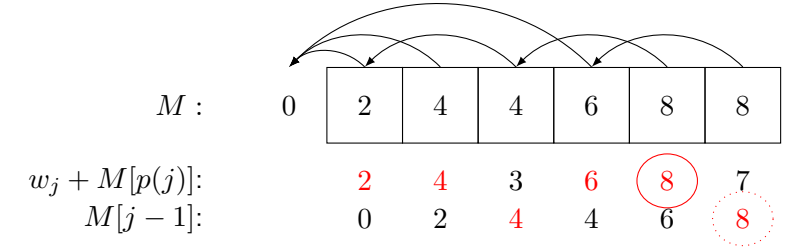
Gewichtetes Interval Scheduling: Beispiel



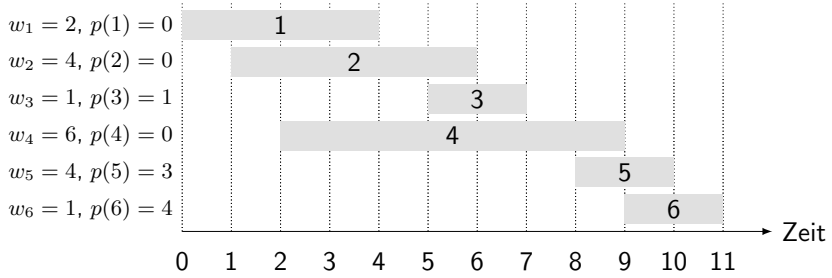


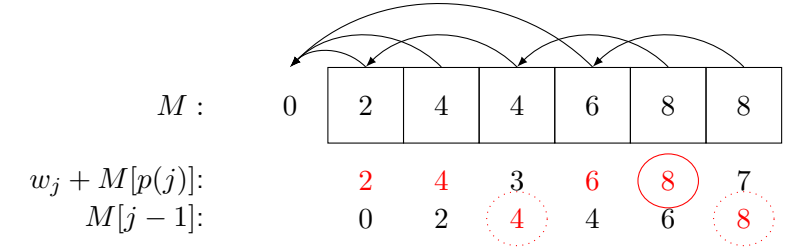
Gewichtetes Interval Scheduling: Beispiel



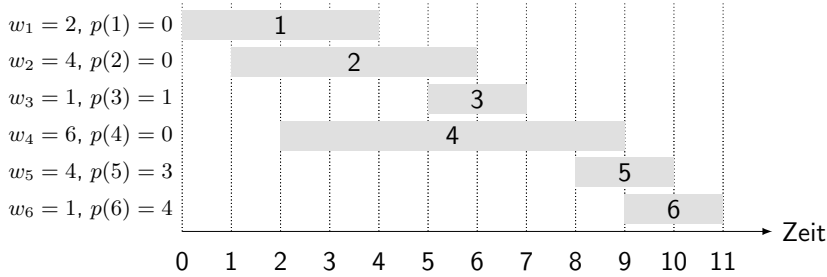


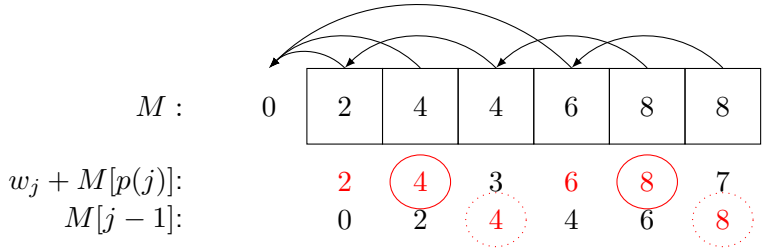
Gewichtetes Interval Scheduling: Beispiel



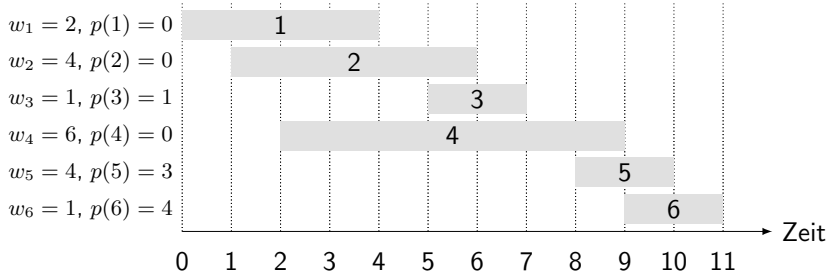


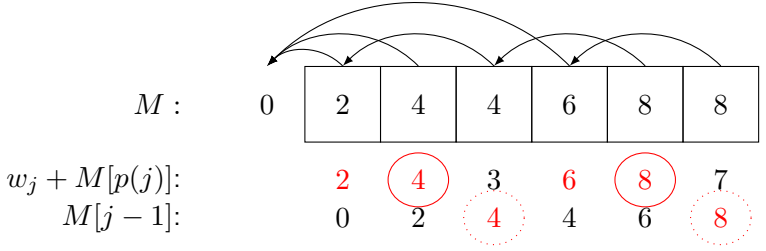
Gewichtetes Interval Scheduling: Beispiel





Gewichtetes Interval Scheduling: Beispiel





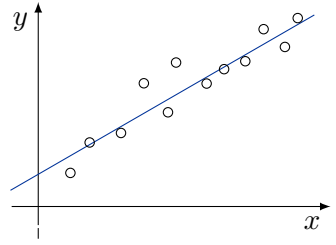
Ergebnis: 2 und 5.

Segmented Least Squares

Least Squares

- Fundamentales Problem in der Statistik und der Numerischen Analyse.
- Gegeben: n Punkte in der Ebene: $(x_1, y_1), \ldots, (x_n, y_n)$.
- Finde eine Gerade $y = ax + b$, welche die Summe des quadrierten Fehlers minimiert:

$$\text{Err} = \sum_{i=1}^{n} (y_i - (ax_i + b))^2$$



Analytische Lösung: der minimale Fehler ist erreicht, wenn

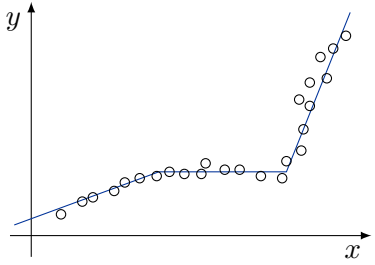
$$a = \frac{n \sum_i x_i y_i - \left(\sum_i x_i\right)\left(\sum_i y_i\right)}{n \sum_i x_i^2 - \left(\sum_i x_i\right)^2}, \; b = \frac{\sum_i y_i - a \sum_i x_i}{n}$$

Segmented Least Squares

- Punkte durch eine Folge von Geradensegmenten annähern.
- Gegeben: n Punkte in der Ebene $(x_1, y_1), \ldots, (x_n, y_n)$ so dass
- $x_1 < x_2 < \cdots < x_n$, finde eine Folge von Geraden welche eine bestimmte Funktion $f(x)$ minimiert.

Frage: Was ist eine angemessene Wahl für $f(x)$? Die Funktion $f(x)$ sollte sowohl Genauigkeit als auch Sparsamkeit gewährleisten.

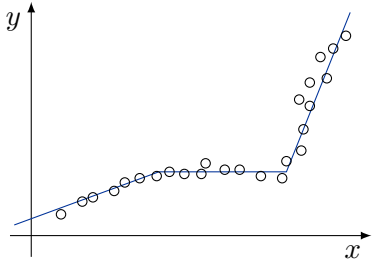
☐ *Höhe des Fehlers* ☐ *Anzahl der Geraden*



Segmented Least Squares

- Punkte durch eine Folge von Geradensegmenten annähern.
- Gegeben: n Punkte in der Ebene $(x_1, y_1), \ldots, (x_n, y_n)$ so dass
- $x_1 < x_2 < \cdots < x_n$, finde eine Folge von Geraden welche:
  - die Summe der quadrierten Fehler E in jedem Segment
  - die Anzahl der Geraden L

  minimiert.
- Tradeoff Funktion: $E + cL$, für eine Konstante $c > 0$.



Dynamischer Ansatz: Segmented Least Squares

Notation

- $OPT(j) =$ minimale Kosten für die Punkte $p_1, p_{i+1}, \ldots, p_j$
- $e(i, j) =$ minimale Summe des quadrierten Fehlers für $p_i, p_{i+1}, \ldots, p_j$

Berechnen von $OPT(j)$:

- letztes Segment nutzt die Punkte $p_i, p_{i+1}, \ldots, p_j$ für ein bestimmtes i
- Kosten $= OPT(i-1) + e(i, j) + c$

$$OPT(j) = \begin{cases} 0 & \text{falls } j = 0 \\ \min\limits_{1 \leq i \leq j} \{OPT(i-1) + e(i, j) + c\} & \text{sonst} \end{cases}$$

Segmented Least Squares: Algorithmus

```
Segmented-Least-Squares( P = {p_1, p_2, ..., p_n} )
    M[0] = 0
    for j ← 1 bis n
        for i ← 1 bis j
            berechne Fehler e(i,j) für Punkte p_i, ..., p_j

    for j ← 1 to n
        M[j] = min_{1≤i≤j} (M[i-1] + e(i,j) + c)

    return M[n]
```

Laufzeit. $O(n^3)$.

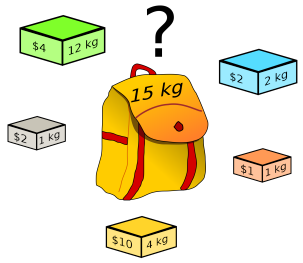
- Flaschenhals ist das Berechnen von $e(i,j)$ für $O(n^2)$ Paare, $O(n)$ pro Paar mit der Formel für Least Squares.
- Finden einer Lösung analog zu Interval Scheduling durch Rückverfolgen der Minimierung

  □ *Kann mithilfe geschickterer Vorberechnung und Wiederverwendung von Zwischenergebnissen zu $O(n^2)$ verbessert werden.*

Rucksackproblem

Rucksackproblem

Gegeben: n Gegenstände mit positiv rationalen Gewichten $g_1, \ldots, g_n$, Werten $w_1, \ldots, w_n$ und einer positiv rationalen Kapazität G.



Gesucht: Teilmenge S der Gegenstände mit Gesamtgewicht $\leq G$ und maximalem Gesamtwert.

Vereinfachung: Der Einfachheit halber nehmen wir im folgenden an, dass sowohl die Gewichte als auch die Kapazität positiv ganzzahlig sind.

Rucksackproblem: Beispiel

Beispiel: $\{3,4\}$ ergibt einen Gesamtwert von 40.

#	Wert	Gewicht	w_i/g_i
1	1	1	1
2	6	2	3
3	18	5	$3\frac{3}{5}$
4	22	6	$3\frac{2}{3}$
5	28	7	4

$G = 11$

Greedy: Füge wiederholt einen Gegenstand mit einem maximalen Verhältnis von w_i/g_i, der in den Rucksack passt, hinzu.
Beispiel: $\{5,2,1\}$ ergibt nur einen Gesamtwert von 35 $\Rightarrow$ Greedy ist nicht optimal.

Idee und Motivation

- Im Kapitel über Branch und Bound haben wir einen Algorithmus mit Laufzeit $O(2^n)$ vorgestellt.
- Hier stellen wir einen Algorithmus mit Laufzeit $O(nG)$ vor, der falls $nG < 2^n$ wesentlich effizienter ist.
- Die Intuition hinter dem Algorithmus ist, dass man nur (Teil-)lösungen mit verschiedenen Gewichten unterscheiden muss.

Dynamische Programmierung: Falscher Ansatz

Definition: $OPT(i) =$ Maximaler Profit für die Teilmenge von den Gegenständen $1, \ldots, i$.

- Fall 1: $OPT(i)$ wird erreicht mit einer Lösung, die den Gegenstand i nicht enthält.
  - Es gilt $OPT(i) = OPT(i-1)$, da wir wissen, dass die Lösung den Gegenstand i nicht enthält.
- Fall 2: $OPT(i)$ wird erreicht mit einer Lösung, die den Gegenstand i enthält.
  - Das Akzeptieren von Gegenstand i impliziert nicht, dass wir andere Gegenstände nicht aufnehmen werden.
  - Ohne zu wissen, welche anderen Gegenstände vor i ausgewählt wurden, können wir nicht sagen, ob wir für i und die nachfolgenden Gegenstände genug Platz haben.

Lösungsansatz: Die Berechnung der Teilprobleme muss die verbleibende Gesamtkapazität berücksichtigen!

Dynamische Programmierung: Gewichtsbeschränkung

Definition: $OPT(i, g)$ = Maximaler Profit für die Teilmenge von den Gegenständen $1, \ldots, i$, mit einer Gewichtsbeschränkung g.

- Fall 1: $OPT(i, g)$ wird erreicht mit einer Lösung, die den Gegenstand i nicht enthält.
  - Es gilt $OPT(i, g) = OPT(i-1, g)$, da wir wissen, dass die Lösung den Gegenstand i nicht enthält.
- Fall 2: $OPT(i, g)$ wird erreicht mit einer Lösung, die den Gegenstand i enthält.
  - Neue Gewichtsbeschränkung $= g - g_i$.
  - Daher gilt dann $OPT(i, g) = w_i + OPT(i-1, g-g_i)$.

$$OPT(i,g) = \begin{cases} 0 & \text{wenn } i = 0 \\ OPT(i-1,g) & \text{wenn } g_i > g \\ \max\{OPT(i-1,g), \quad w_i + OPT(i-1,g-g_i)\} & \text{sonst} \end{cases}$$

Rucksackproblem: Bottom-Up

Rucksack: Befülle ein $(n+1) \times (G+1)$ Array.

Eingabe: n, G, $g_1, \ldots, g_n$, $w_1, \ldots, w_n$

```
for g ← 0 bis G
    M[0,g] ← 0

for i ← 1 bis n
    for g ← 0 bis G
        if g_i > g
            M[i,g] ← M[i−1,g]
        else
            M[i,g] ← max{M[i−1,g], w_i + M[i−1, g−g_i]}
return M[n,G]
```

Laufzeit und Platzbedarf: $O(nG)$

Rucksack Algorithmus

$G + 1$ →

$n + 1$ ↓

	0	1	2	3	4	5	6	7	8	9	10	11
$\emptyset$	0	0	0	0	0	0	0	0	0	0	0	0
$\{ 1 \}$	0	1	1	1	1	1	1	1	1	1	1	1
$\{ 1, 2 \}$	0	1	6	7	7	7	7	7	7	7	7	7
$\{ 1, 2, 3 \}$	0	1	6	7	7	18	19	24	25	25	25	25
$\{ 1, 2, 3, 4 \}$	0	1	6	7	7	18	22	24	28	29	29	40
$\{ 1, 2, 3, 4, 5 \}$	0	1	6	7	7	18	22	28	29	34	35	40

OPT: $\{ 4, 3 \}$
Wert $= 22 + 18 = 40$

$G = 11$

Gegenstand	Wert	Gewicht
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Rucksackproblem: Bestimmen der Lösung

Optimale Lösung: Mit Hilfe der Werte im Array M.

```
Find-Solution(M):
i ← n
k ← G
A ← ∅
while i > 0 und k > 0
    if M[i,k] ≠ M[i-1,k]
        A ← A ∪ {i}
        k ← k − g_i
    i ← i − 1
return A
```

Rucksackproblem: Laufzeit

Laufzeit: $O(nG)$.

- Polynomiell in n.
- Aber die Laufzeit hängt auch von der Rucksackkapazität ab.
  - G ist exponentiell in der Eingabelänge, weil Zahlen binär kodiert werden.
  - Die Laufzeit ist somit nicht polynomiell in der Eingabelänge beschränkt.
- „Pseudo-polynomiell.“

Allgemein: Falls $P \neq NP$ kann das Rucksackproblem nicht in Polynomialzeit gelöst werden.

Rucksackproblem: Verbesserung

Speicherung: Man muss eigentlich nicht das gesamte Array speichern.

Verbesserung:

- Man merkt sich nur die letzte Zeile.
- Die Aktualisierung der Einträge erfolgt von rechts nach links.

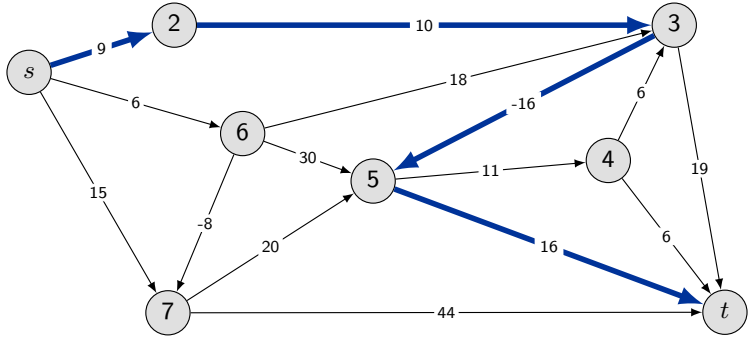
Kürzeste Pfade

Kürzeste Kantenzüge

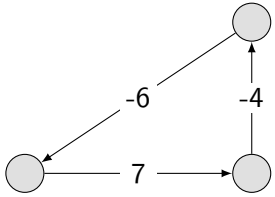
Kürzester Kantenzug: Gegeben sei ein gerichteter Graph $G = (V, E)$, mit Kantengewichten c_{vw}. Finde einen kürzesten Kantenzug zwischen Knoten s und t.

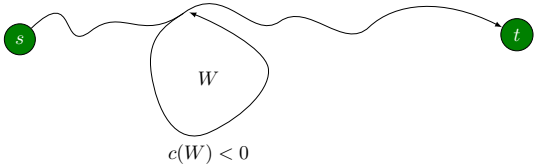
- *erlaube negative Gewichte (bei Dijkstra-Algorithmus nicht erlaubt)*

Beispiel:



Kürzeste Kantenzüge: Kreise mit negativen Kosten (negative Kreise)





Bemerkung: Im Falle von negativen Kreisen:

- keine sinnvolle Definition von kürzesten Kantenzug mehr möglich,
- Definition von kürzesten Pfaden bleibt jedoch sinnvoll.

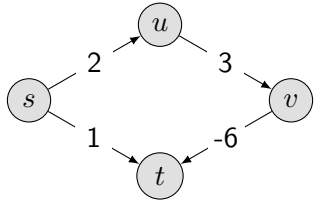
Kürzeste Pfade: Komplexität

Theorem: Das Finden eines kürzesten Pfades in einem gerichteten Graphen mit reellwertigen Kantengewichten ist **NP**-vollständig.

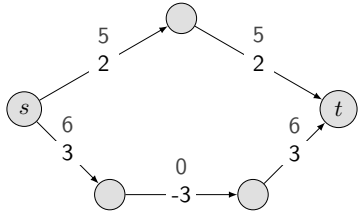
Bemerkungen: Verbietet man negative Kreise wird das Problem jedoch wieder in Polynomialzeit lösbar.

Kürzeste Pfade: Falsche Ansätze

Algorithmus von Dijkstra: Schlägt fehl bei negativen Kantengewichten.



Neugewichtung: Zu jedem Kantengewicht eine Konstante dazugeben schlägt fehl.



Kürzeste Pfade: Bellman's Gleichungen

Sei $G = (V, E)$ ein gerichteter Graph ohne negative Kreise mit Kantengewichten c_{vw} und $t \in V$, dann gilt für die Länge $OPT(v)$ eines kürzesten v-t Pfades P in G:

$$OPT(t) = 0$$

$$OPT(v) = \min_{(v,w)\in E} \{c_{vw} + OPT(w)\}, \forall v \in V, v \neq t.$$

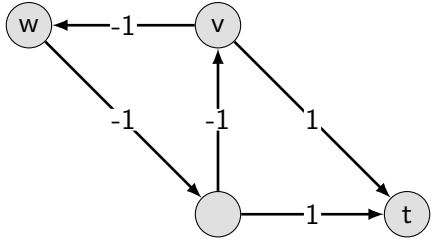
Beweis:

Wir zeigen mit Induktion über die Anzahl Kanten auf dem kürzesten v-t Pfad, dass $OPT(v)$ die Länge dieses kürzesten v-t Pfades ist.

- Aussage gilt für $v = t$ mit 0 Kanten.
- Sei P ein kürzester v-t Pfad mit ℓ Kanten und (v, w) die erste Kante von P.
- Da es keine negativen Kreise gibt, liegt ein Knoten nie zweimal auf einem kürzesten Kantenzug.
- $P - (v, w)$ ist ein kürzester w-t Pfad mit $\ell - 1$ Kanten, der v nicht enthält. Wegen der Induktionsvoraussetzung ist seine Länge $OPT(w)$.
- Damit ist aber $c_{vw} + OPT(w)$ die Länge von P und $OPT(v) \leq c_{vw} + OPT(w)$.
- Wäre $OPT(v) < c_{vw} + OPT(w)$, so gäbe es einen kürzeren v-t Pfad als P über einen anderen Zwischenknoten $w' \rightarrow$ Widerspruch

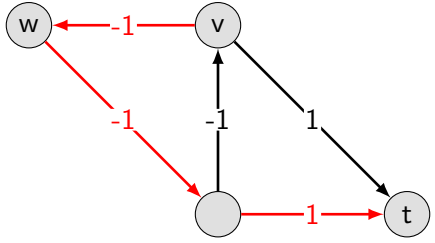
Bellman's Gleichungen: Gegenbeispiel

Falls G negative Kreise enthält, gelten Bellman's Gleichungen nicht mehr:



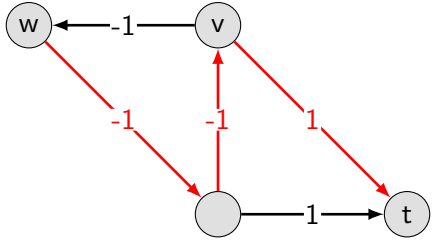
Bellman's Gleichungen: Gegenbeispiel

Falls G negative Kreise enthält, gelten Bellman's Gleichungen nicht mehr:



Bellman's Gleichungen: Gegenbeispiel

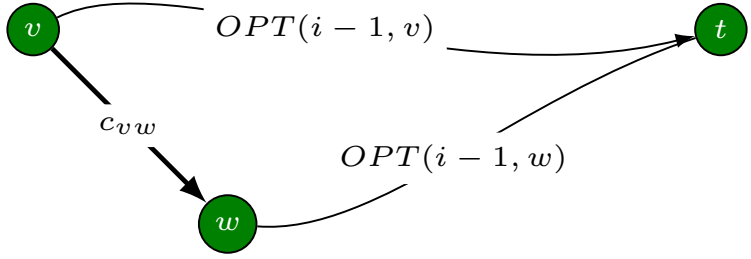
Falls G negative Kreise enthält, gelten Bellman's Gleichungen nicht mehr:



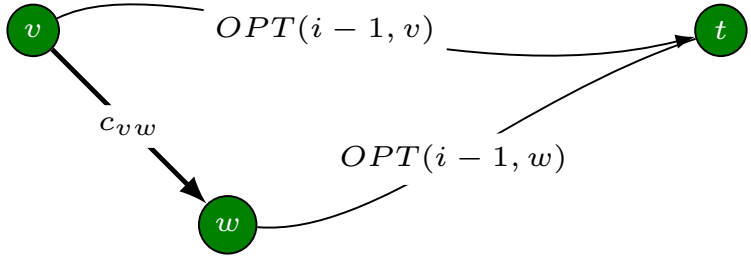
Kürzeste Pfade: Dynamische Programmierung

Definition: $OPT(i, v) =$ Länge eines kürzesten v-t Pfades P, der höchstens i Kanten benutzt.

- Fall 1: P benutzt höchstens $i - 1$ Kanten.
  - $OPT(i, v) = OPT(i - 1, v)$
- Fall 2: P benutzt genau i Kanten.
  - Sei (v, w) die erste Kante auf P. Dann besteht der kürzeste Pfad aus (v, w) und dem besten w-t Pfad, der höchstens $i - 1$ Kanten benutzt.



Kürzeste Pfade: Dynamische Programmierung

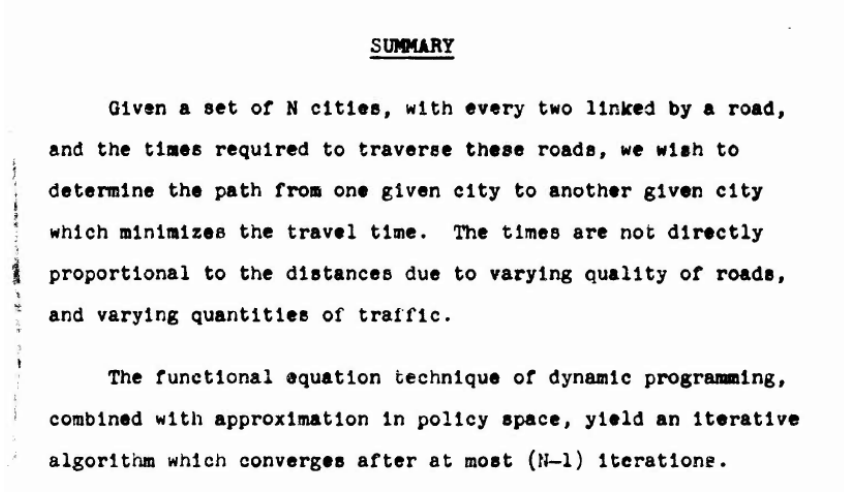


$$OPT(i,v) = \begin{cases} 0 & \text{wenn } i = 0 \text{ und } v = t \\ \infty & \text{wenn } i = 0 \text{ und } v \neq t \\ \min\left\{OPT(i-1,v), \min\limits_{(v,w)\in E}\{c_{vw} + OPT(i-1,w)\}\right\} & \text{ansonsten} \end{cases}$$

Anmerkung: Aufgrund von Bellman's Gleichungen ist $OPT(n-1,v) = OPT(v) =$ Länge eines kürzesten v-t Pfades, wenn es keine negativen Kreise gibt.

Bellman-Ford Algorithmus

Der hier vorgestellte Algorithmus wurde unabhängig von Richard Bellman und Lester Ford im Jahre 1956 entwickelt und wird deshalb auch häufig als Bellman-Ford-Algorithmus bezeichnet.



SUMMARY

Given a set of N cities, with every two linked by a road, and the times required to traverse these roads, we wish to determine the path from one given city to another given city which minimizes the travel time. The times are not directly proportional to the distances due to varying quality of roads, and varying quantities of traffic.

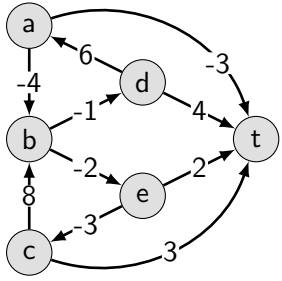
The functional equation technique of dynamic programming, combined with approximation in policy space, yield an iterative algorithm which converges after at most (N–1) iterations.

Kürzeste Pfade: Implementierung

```
Shortest-Path(G,s,t):
foreach node v ∈ V
    M[0,v] ← ∞
M[0,t] ← 0
for i ← 1 bis n − 1
    foreach Knoten v ∈ V
        M[i,v] ← M[i − 1,v]
    foreach Kante (v,w) ∈ E
        M[i,v] ← min (M[i,v],c_vw + M[i − 1,w])
return M[n − 1,s]
```

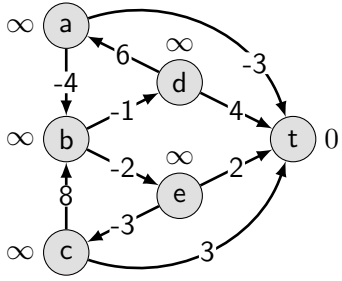
Analyse: $O(mn)$ Zeit, $O(n^2)$ Platz.

Kürzeste Pfade: Beispiel



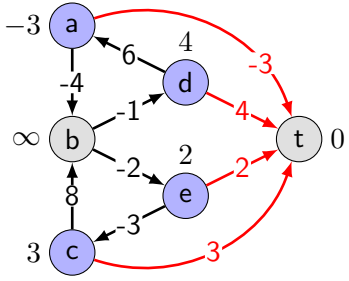
	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Kürzeste Pfade: Beispiel



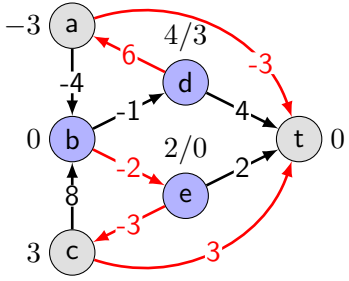
	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Kürzeste Pfade: Beispiel



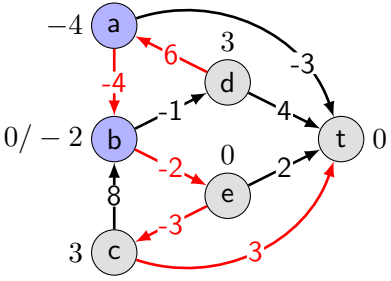
	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Kürzeste Pfade: Beispiel



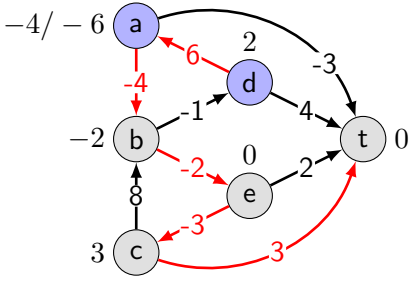
	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Kürzeste Pfade: Beispiel



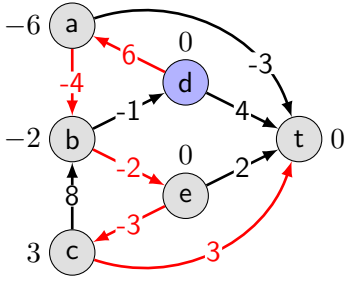
	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Kürzeste Pfade: Beispiel



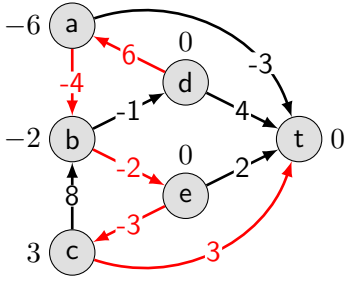
	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Kürzeste Pfade: Beispiel



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Kürzeste Pfade: Beispiel



	0	1	2	3	4	5
t	0	0	0	0	0	0
a	∞	-3	-3	-4	-6	-6
b	∞	∞	0	-2	-2	-2
c	∞	3	3	3	3	3
d	∞	4	3	3	2	0
e	∞	2	0	0	0	0

Kürzeste Pfade: Implementierung

Einen kürzesten Pfad finden: Verwalte den Nachfolger für jeden Knoten.

Korrektheit der Ergebnisse:

- Der beschriebene Algorithmus liefert immer eine korrekte Lösung, wenn keine negativen Kreise im Graphen vorhanden sind.
- Wenn negative Kreise vorhanden sind, kann der Algorithmus falsche Ergebnisse liefern.

Überprüfung auf negative Kreise:

- Nach dem Terminieren des Algorithmus.
- Für jeden Knoten v wird überprüft: Falls $OPT(n, v) < OPT(n-1, v)$, dann gibt es einen negativen Kreis.

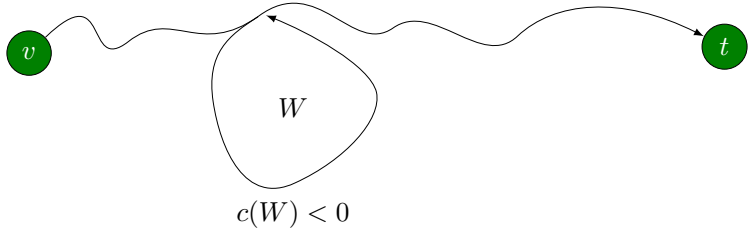
Negative Kreise erkennen

Lemma: Wenn $OPT(n, v) < OPT(n-1, v)$ für einen Knoten v, dann enthält (ein beliebiger) kürzester v-t Kantenzug K mit maximal n Kanten einen Kreis W.
Außerdem hat W negative Kosten.

Beweis: durch Widerspruch

- Da $OPT(n, v) < OPT(n-1, v)$, wissen wir, dass K genau n Kanten hat.
- Mit Hilfe des Schubfachprinzips kann man zeigen, dass K einen gerichteten Kreis W enthalten muss.
- Löschen von W ergibt einen v-t Kantenzug mit $< n$ Kanten $\Rightarrow W$ hat negative Kosten.

□



Negative Kreise erkennen

Lemma: Falls G einen negativen Kreis enthält von dem aus t erreicht werden kann, dann gibt es eine Kante (v, u), sodass $OPT(n-1, v) > c_{vu} + OPT(n-1, u)$ (und somit $OPT(n, v) < OPT(n-1, v)$).

Beweis:

- Sei C ein beliebiger Kreis in G auf den Knoten $(v_1, \ldots, v_k)$, dann gilt: $m = \sum_{i=1}^{k} [OPT(n-1, v_i) - OPT(n-1, v_{i+1 \mod k})] = 0$.
- Falls nun C ein negativer Kreis ist, ergibt sich $\sum_{i=1}^{k} c_{v_i v_{i+1 \mod k}} < 0 = m$.
- D.h. es muss mindestens ein i existieren mit $c_{v_i v_{i+1 \mod k}} < OPT(n-1, v_i) - OPT(n-1, v_{i+1 \mod k})$ und somit gilt für die Kante $(v_i, v_{i+1 \mod k})$, dass $c_{v_i v_{i+1 \mod k}} + OPT(n-1, v_{i+1 \mod k}) < OPT(n-1, v_i)$.

□

Negative Kreise erkennen

Aus den beiden vorherigen Lemmas folgt nun:

Theorem: G enthält einen negativen Kreis von dem aus t erreicht werden kann genau dann wenn ein Knoten v mit $OPT(n, v) < OPT(n-1, v)$ existiert.

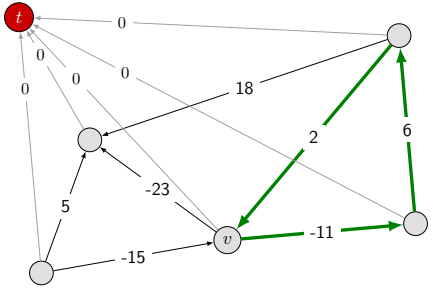
Negative Kreise erkennen

Theorem: Man kann in Zeit $O(nm)$ entscheiden, ob ein Graph einen negativen Kreis hat und diesen im positiven Fall auch ausgeben.

Beweis:

- Gib einen neuen Knoten t hinzu und verbinde alle Knoten mit t mit einer Kante mit den Kosten 0.
- Überprüfe, ob $OPT(n, v) = OPT(n-1, v)$ für alle Knoten v.
  - Wenn ja, dann gibt es keine negativen Kreise
  - Wenn nein, dann extrahiere den Kreis aus dem kürzesten Kantenzug von v zu t mit maximal n Kanten

□



Kürzeste Pfade: Praktische Verbesserungen

Praktische Verbesserungen:

- Verwalte nur ein Array M[v] = kürzester v-t Pfad, den wir bisher gefunden haben.
- Brich den Algorithmus ab, sobald sich nach einer vollen Iteration kein Eintrag in M mehr geändert hat.

Theorem: Beim Ablauf des Algorithmus ist M[v] die Länge eines v-t Pfades und nach i Runden von Updates ist der Wert M[v] nicht größer als die Länge eines kürzesten v-t Pfades, der $\leq i$ Kanten benutzt.

Gesamte Auswirkung:

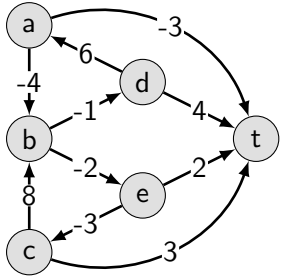
- Speicher: $O(m + n)$.
- Laufzeit: $O(mn)$ Worst-Case, aber wesentlich schneller in der Praxis.

Bellman-Ford: Effiziente Implementierung

```
Push-Based-Shortest-Path(G,s,t):
foreach Knoten v ∈ V
    M[v] ← ∞
    Nachfolger[v] ← ∅

M[t] = 0
for i ← 1 bis n − 1
    foreach Kante (v, w) ∈ E
        if M[v] > M[w] + c_vw
            M[v] ← M[w] + c_vw
            Nachfolger[v] ← w
    if kein M[w] Wert ändert sich in Iteration i, stop.
return M[s]
```

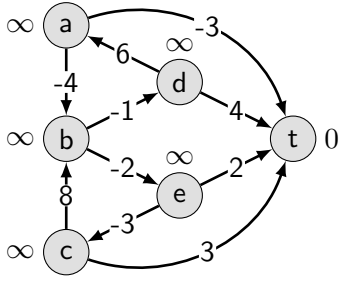
Kürzeste Pfade: Beispiel



	0	1	2
t	0	0	0
a	∞	-6	-6
b	∞	-2	-2
c	∞	3	3
d	∞	0	0
e	∞	0	0

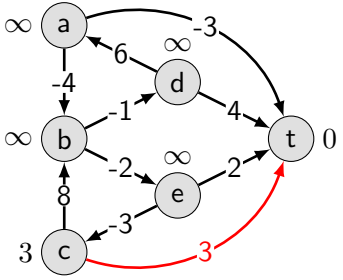
Die Anzahl der Iterationen hängt nun stark von der Reihenfolge ab in der die Kanten in einer Iteration durchlaufen werden.

Kürzeste Pfade: Beispiel



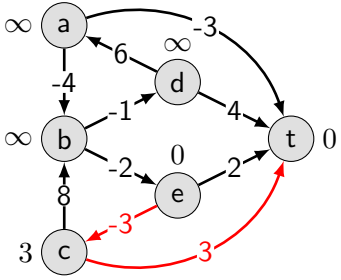
	0	1	2
t	0	0	0
a	∞	-6	-6
b	∞	-2	-2
c	∞	3	3
d	∞	0	0
e	∞	0	0

Kürzeste Pfade: Beispiel



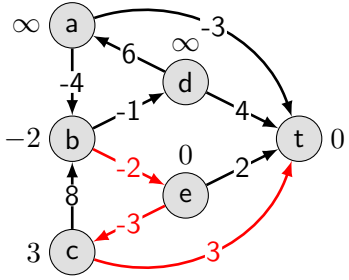
	0	1	2
t	0	0	0
a	∞	-6	-6
b	∞	-2	-2
c	∞	3	3
d	∞	0	0
e	∞	0	0

Kürzeste Pfade: Beispiel



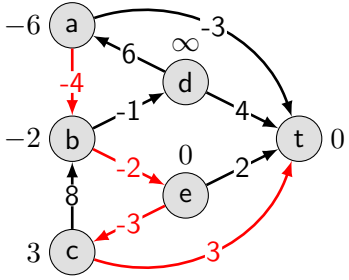
	0	1	2
t	0	0	0
a	∞	-6	-6
b	∞	-2	-2
c	∞	3	3
d	∞	0	0
e	∞	0	0

Kürzeste Pfade: Beispiel



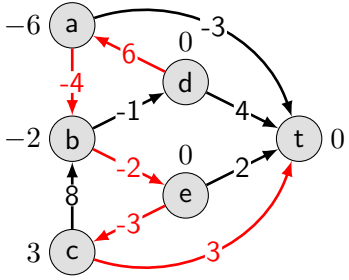
	0	1	2
t	0	0	0
a	∞	−6	−6
b	∞	−2	−2
c	∞	3	3
d	∞	0	0
e	∞	0	0

Kürzeste Pfade: Beispiel



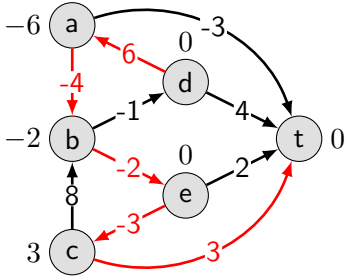
	0	1	2
t	0	0	0
a	∞	−6	−6
b	∞	−2	−2
c	∞	3	3
d	∞	0	0
e	∞	0	0

Kürzeste Pfade: Beispiel



	0	1	2
t	0	0	0
a	∞	-6	-6
b	∞	-2	-2
c	∞	3	3
d	∞	0	0
e	∞	0	0

Kürzeste Pfade: Beispiel



	0	1	2
t	0	0	0
a	∞	-6	-6
b	∞	-2	-2
c	∞	3	3
d	∞	0	0
e	∞	0	0

Finden eines kürzesten Pfades: Korrektheit

Den kürzesten Pfad finden: Verwalte den Nachfolger für jeden Knoten.

Definition: Ein Knoten w ist der Nachfolger eines Knoten v (an einer beliebigen Stelle im Algorithmus), falls $M[v]$ zuletzt auf $c_{vw} + M[w]$ geändert wurde.

Beobachtung: Falls w momentan der Nachfolger von v ist dann gilt:
$M[v] \geq c_{vw} + M[w]$.

Finden eines kürzesten Pfades: Korrektheit

Lemma: Falls der Nachfolgergraph einen Kreis hat (an einer beliebigen Stelle im Algorithmus), dann ist der Kreis negativ.

Beweis:

- Seien $v_1, \ldots, v_k$ die Knoten auf einem Kreis C im Nachfolgergraph und sei (v_k, v_1) die letzte Kante in C, die vom Algorithmus hinzugefügt wurde.
- Dann gilt $M[v_i] \geq c_{v_i v_{i+1}} + M[v_{i+1}]$ für alle i mit $1 \leq i < k$ und (unmittelbar bevor die Kante (v_k, v_1) gesetzt wird) $M[v_k] > c_{v_k v_1} + M[v_1]$.
- Nach Aufsummieren aller Ungleichungen verschwinden die $M[v_i]$-Terme und wir erhalten: $0 > \sum_{i=1}^{k-1} c_{v_i v_{i+1}} + c_{v_k v_1}$, also ist C ein negativer Kreis.

□

Finden eines kürzesten Pfades: Korrektheit

Aus dem vorherigen Lemma folgt, dass falls G keine negativen Kreise enthält, der Nachfolgergraph kreisfrei ist.

Nach Beendigung des Algorithmus auf einem Graphen G ohne negative Kreise, folgt nun:

- jeder Knoten v von dem aus t erreichbar ist hat genau einen Nachfolger w und $OPT(n-1, v) = c_{vw} + OPT(n-1, w)$.
- Also enthält der Nachfolgergraph für jeden solchen Knoten genau einen Pfad nach t und dieser Pfad ist ein kürzester Pfad.

Dynamische Programmierung Zusammenfassung

Vorgehen beim Entwurf von Dynamischen Programmen

- Verstehe und charakterisiere die Struktur des Problems und seiner optimalen Lösungen anhand von überlappenden Teilproblemen und optimalen Teillösungen
- Definiere rekursiv den Wert einer optimalen Lösung
- Berechne und speichere die Werte der optimalen (Teil-)Lösungen in einer Tabelle
- Konstruiere die optimale Lösung durch Rückverfolgung der Optimierungsentscheidungen des Algorithmus

Techniken der Dynamischen Programmierung

- Binäre Auswahl: z.B. gewichtetes Interval Scheduling
- Mehrfachauswahl: z.B. segmented least squares, kürzeste Pfade
- Einführen zusätzlicher Variablen: z.B. Knapsack

Top-down vs. bottom-up: rekursive vs. iterative Berechnung

Ergänzende Literatur

J. Kleinberg and E. Tardos. *Algorithm Design*. Pearson, 2005. Kapitel 6.