

12. Dynamische Programmierung

Wir haben in [11. Branch and Bound](#) schon gelernt wie man mit bestimmten Optimierungsproblemen umgehen kann, jetzt machen wir weiter bei der Optimierung von Dynamischer Programmierung, mit einer art Divde-and-Conquer Optimierung.

Einleitung

Grundlagen Dynamische Programmierung

- **Dynamische Programmierung:**
 - Zerlege ein Problem in eine Folge von **überlappenden Teilproblemen**.
 - Erstelle und speichere Lösungen für diese Teilprobleme.
 - Verwende die gespeicherten Lösungen, um Lösungen für immer größere Teilprobleme zu konstruieren.
- **Optimalitätsprinzip von Bellman:**
 - Dynamische Programmierung führt zu einem optimalen Ergebnis, **genau dann**, wenn sich die optimale Lösung des Gesamtproblems aus den optimalen Lösungen seiner Subprobleme zusammensetzt.
- **Effizienz:**
 - Hängt stark von der Vorgehensweise bei der Aufteilung des Problems und der Ermittlung der Lösungen für die einzelnen Teilprobleme ab.
- **Wesentlicher Aspekt:**
 - **Speicherung (Memoization)** von Ergebnissen für bereits gelöste Subprobleme zur Wiederverwendung. Dies vermeidet redundante Berechnungen und trägt zur Effizienz bei.

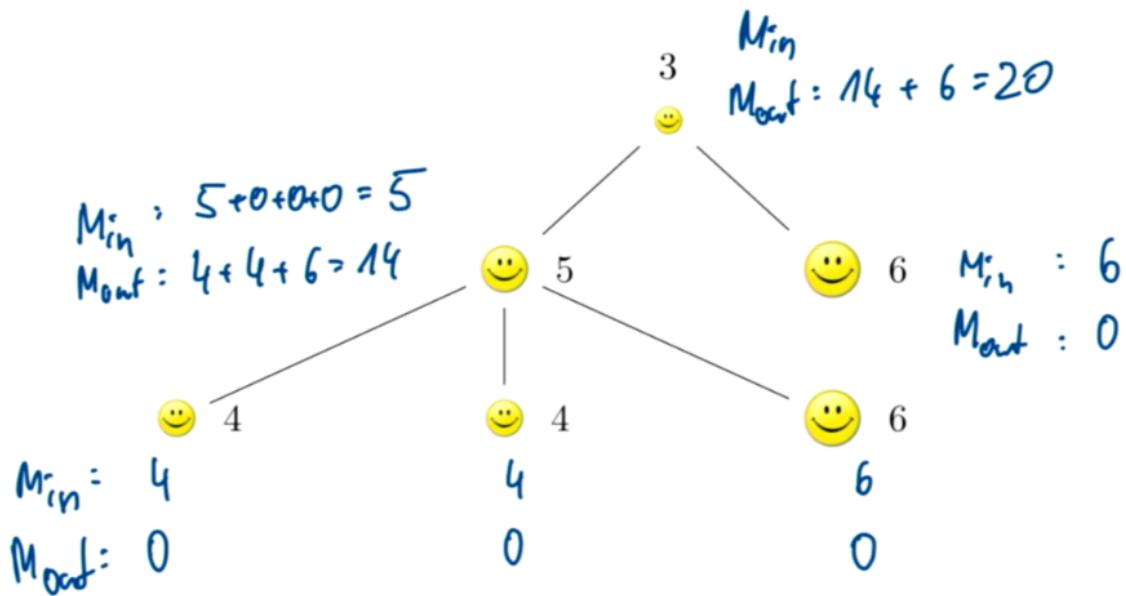
--> Es ist ähnlich zu [Divide and Conquer](#) Techniken

--> Oft exponentiell große Suchräume, aber wir müssen nicht immer alles betrachten

Weighted Independent Set auf Bäumen

 **Beispiel**

[siehe hier](#)



- Für jeden Knoten sind zwei Werte notiert:
 - M_{in} : Das maximale Gewicht eines Independent Sets im Unterbaum, **inklusive** des aktuellen Knotens. Wenn der aktuelle Knoten im Independent Set ist, dürfen seine direkten Kinder nicht im Set sein.
 - M_{out} : Das maximale Gewicht eines Independent Sets im Unterbaum, **ohne** den aktuellen Knoten. In diesem Fall können die direkten Kinder des aktuellen Knotens entweder im Independent Set sein oder nicht, je nachdem, was das maximale Gewicht ergibt.
- **Blattknoten (Gewicht w):**
 - $M_{in} = w$ (Der Knoten selbst bildet ein Independent Set)
 - $M_{out} = 0$ (Wenn der Knoten nicht im Set ist, trägt er kein Gewicht bei)
- **Innere Knoten (am Beispiel des Knotens mit Gewicht 5):**
 - Um M_{in} für den Knoten mit Gewicht 5 zu berechnen:
 - Wir nehmen das Gewicht des Knotens selbst (5).
 - Da der Knoten im Independent Set ist, dürfen seine direkten Kinder (mit Gewichten 4, 4, 6) nicht im Set sein.
 - Daher addieren wir die M_{out} -Werte der Kinder: $0 + 0 + 0 = 0$.
 - $M_{in} = 5 + 0 + 0 + 0 = 5$.
 - Um M_{out} für den Knoten mit Gewicht 5 zu berechnen:
 - Der Knoten selbst ist nicht im Independent Set.
 - Für jedes Kind wählen wir den maximalen Wert zwischen M_{in} und M_{out} des Kindes, da das Kind entweder im Independent Set sein kann oder nicht.
 - Kinder mit Gewicht 4: $\max(M_{in} = 4, M_{out} = 0) = 4$
 - Kind mit Gewicht 4: $\max(M_{in} = 4, M_{out} = 0) = 4$
 - Kind mit Gewicht 6: $\max(M_{in} = 6, M_{out} = 0) = 6$

- $M_{out} = 4 + 4 + 6 = 14.$
- **Wurzelknoten (Gewicht 3):**
 - $M_{in} = 3 + M_{out}(\text{linkes Kind}) + M_{out}(\text{rechtes Kind}) = 3 + 14 + 0 = 17$
(Anmerkung: Im Screenshot steht hier fälschlicherweise $M_{in} = 6$, dies sollte $3 + 14 = 17$ sein, da das rechte Kind mit Gewicht 6 ein M_{out} von 0 hat, wenn der Wurzelknoten im Set ist)
 - $M_{out} = \max(M_{in}(\text{linkes Kind}), M_{out}(\text{linkes Kind})) + \max(M_{in}(\text{rechtes Kind}), M_{out}(\text{rechtes Kind}))$
 - $M_{out} = \max(5, 14) + \max(6, 0) = 14 + 6 = 20.$
- Das maximale Gewicht des Weighted Independent Set für den gesamten Baum ist $\max(M_{in}(\text{Wurzel}), M_{out}(\text{Wurzel})) = \max(17, 20) = 20.$

Einführendes Beispiel

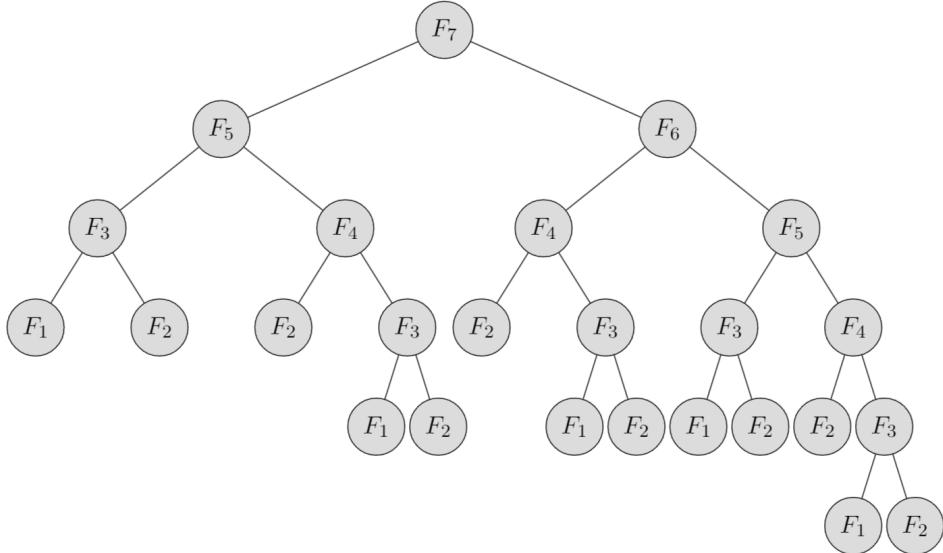
Fibonacci-Zahlen

- **Folge von Fibonacci-Zahlen:**
 - Definition: $F_1 = F_2 = 1$, $F_n = F_{n-1} + F_{n-2}$ für alle $n \geq 3$.
 - Die Folge beginnt mit: 1, 1, 2, 3, 5, 8, 13, ...
- **Einfacher rekursiver Algorithmus:**

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

Laufzeit

- **Gesamtzahl der Aufrufe für die i -te Fibonacci-Zahl:**
 - Entspricht der i -ten Fibonacci-Zahl selbst.
 - Beispiele:
 - $F_{10} = 55$ (ca. 55 rekursive Aufrufe)
 - $F_{20} = 6765$
 - $F_{30} = 832040$
 - $F_{40} = 102334155$
- **Exponentielle Zeitkomplexität:**
 - Begründung:
 - $F_n = F_{n-1} + F_{n-2}$
 - Daraus folgt $F_n \geq 2F_{n-2}$ (da $F_{n-1} \geq F_{n-2}$ für $n \geq 3$)
 - Weiterhin gilt $F_2 = 1 = 2^0$ und $F_3 = 2 \geq 2^1$.
 - Allgemein lässt sich zeigen, dass $F_n \geq 2^{\lfloor (n-1)/2 \rfloor}$.
 - Eine genauere Analyse zeigt, dass $F_n \geq (\sqrt{2})^{n-2}$.



- Der Rekursionsbaum verzweigt sich exponentiell. Viele Teilprobleme werden redundant berechnet (z.B. `Fibonacci(3)` wird mehrfach aufgerufen bei der Berechnung von `Fibonacci(5)`).
- Der Rekursionsbaum hat fast gleich viele Blätter wie die Aufrufhöhe

Dynamische Programmierung (Rekursiv mit Memoization)

⌚ Speicherung - Memoization

- Die bereits berechneten Fibonacci-Zahlen werden in einem Array (z.B. F) zwischengespeichert.
- Vor jeder rekursiven Berechnung wird geprüft, ob das Ergebnis für das aktuelle n bereits im Array gespeichert ist. Wenn ja, wird der gespeicherte Wert direkt zurückgegeben.
- Heißt der rekursive Aufruf wird nur berechnet, wenn er noch nie berechnet wurde, also nur ein mal.

Algorithmus:

Initialisiere ein Array F der Größe $n+1$ mit leeren Werten.

```
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: $O(n)$

- Jeder Wert von $F[i]$ wird maximal einmal rekursiv berechnet.
- Es gibt n mögliche Werte für i (von 1 bis n).
- Die Überprüfung und der Zugriff auf das Array erfolgen in konstanter Zeit.

Dynamische Programmierung (Iterativ)

- **Speicherung:**
 - Die berechneten Fibonacci-Zahlen werden in einem Array (z.B. F) gespeichert und in der Berechnung wiederverwendet.
 - Die Berechnung erfolgt bottom-up, beginnend mit den Basisfällen.
- **Algorithmus:**

```
Linear-Fibonacci(n):
    F sei ein Array der Größe n+1.
    F[1] <- 1
    F[2] <- 1
    for i <- 3 bis n:
        F[i] <- F[i - 1] + F[i - 2]
    return F[n]
```

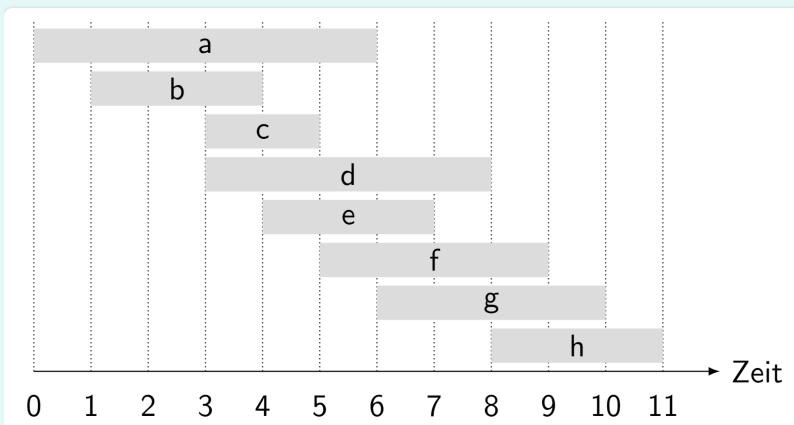
- **Laufzeit:** $O(n)$
 - Die Schleife wird $n - 2$ Mal durchlaufen.
 - Jeder Schleifendurchlauf benötigt konstanten Aufwand (Addition und Zuweisung).

Gewichtetes Intervall Scheduling

mehr dazu: [siehe hier](#)

⌚ Gewichtetes Interval Scheduling

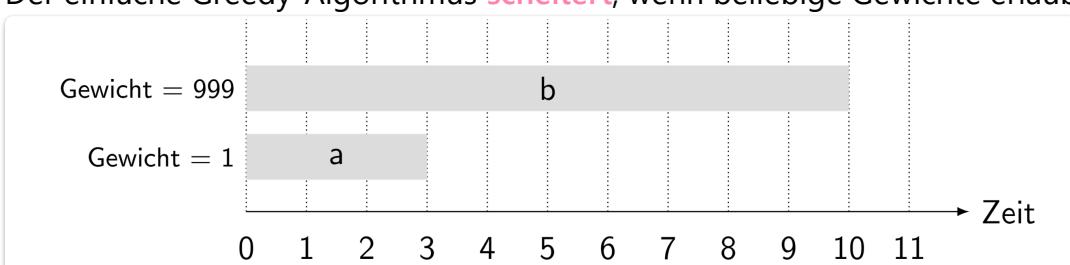
- Gegeben sind Jobs, wobei Job j startet zum Zeitpunkt s_j , endet zum Zeitpunkt f_j und hat ein Gewicht $w_j > 0$.
- Zwei Jobs sind **kompatibel**, wenn sie sich nicht überlappen.
- **Ziel:** Finde eine Teilmenge von paarweise kompatiblen Jobs mit **maximalem Gesamtgewicht**.



Da gabs die Greedy-Variante EDF (Earliest Deadline first), da wissen wir aber nicht, ob das auch das Gewicht maximiert.

Interval Scheduling: Rückblick

- **Wiederholung (einfaches Interval Scheduling):**
 - Ein Greedy-Algorithmus funktioniert, wenn alle Gewichte gleich 1 sind.
 - Vorgehensweise:
 - Berücksichtige Jobs in aufsteigender Reihenfolge der Endigungszeit.
 - Füge einen Job zur Teilmenge hinzu, wenn er kompatibel mit dem zuvor ausgewählten Job ist.
- **Beobachtung:**
 - Der einfache Greedy-Algorithmus **scheitert**, wenn beliebige Gewichte erlaubt sind.

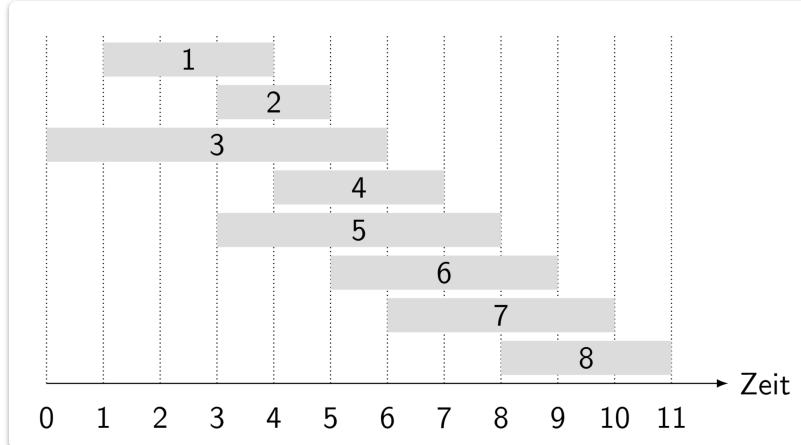


Gewichtetes Interval Scheduling

- Notation:

- Ordne die Jobs aufsteigend sortiert nach ihrer Beendigungszeit: $f_1 \leq f_2 \leq \dots \leq f_n$.
- Definition: $p(j)$ ist der größte Index $i < j$, sodass Job i kompatibel zu Job j ist (d.h., $f_i \leq s_j$). Falls kein solcher Job existiert, ist $p(j) = 0$.

- Beispiel:



- $p(8) = 5$ (Job 5 endet vor Job 8 und überlappt nicht)
- $p(7) = 3$ (Job 3 endet vor Job 7 und überlappt nicht)
- $p(2) = 0$ (Kein Job mit kleinerem Index ist mit Job 2 kompatibel)

Dynamische Programmierung: Binäre Auswahl

- Notation:

- $OPT(j)$ = Wert der optimalen Lösung für das Problem, bestehend aus den Jobs $1, 2, \dots, j$ (die nach Endzeit sortiert sind).

- Wir unterscheiden zwei Fälle für die optimale Lösung $OPT(j)$:

- 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 (Job j ist in der optimalen Lösung):

- Die Lösung kann keine inkompatiblen Jobs $\{p(j) + 1, p(j) + 2, \dots, j - 1\}$ enthalten.
- Daher besteht die optimale Lösung in diesem Fall aus Job j und der optimalen Lösung für die Jobs $1, \dots, p(j)$.
- Der Wert dieser Lösung ist $w_j + OPT(p(j))$.

- Fall 2 (Job j ist nicht in der optimalen Lösung):

- Dann ist der Wert der optimalen Lösung für die Jobs $1, \dots, j$ derselbe wie der Wert der optimalen Lösung für die Jobs $1, \dots, j - 1$.
- Der Wert dieser Lösung ist $OPT(j - 1)$.

- Rekursive Definition für $OPT(j)$:

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

- **Basisfall:** Wenn keine Jobs ($j = 0$) betrachtet werden, ist der optimale Wert 0.
- **Rekursiver Schritt:** Für $j > 0$ wählen wir das Maximum zwischen:
 - Dem Gewicht des aktuellen Jobs j plus dem optimalen Wert der kompatiblen Jobs davor ($p(j)$).
 - Dem optimalen Wert ohne den aktuellen Job j (d.h., der optimale Wert bis zum vorherigen Job $j - 1$).

Gewichtetes Interval Scheduling: Brute-Force-Ansatz

Brute-Force-Algorithmus

- **Eingabe:** n Jobs mit Startzeiten s_1, \dots, s_n , Endzeiten f_1, \dots, f_n und Gewichten w_1, \dots, w_n .
- Sortiere Jobs nach Beendigungszeit, sodass $f_1 \leq f_2 \leq \dots \leq f_n$.
- Berechne $p(1), p(2), \dots, p(n)$ (der größte Index $i < j$, sodass Job i mit Job j kompatibel ist).

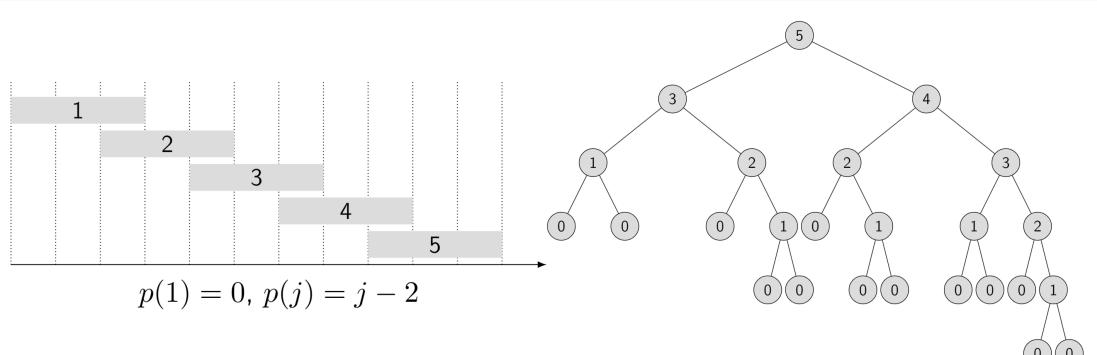
- **Rekursive Funktion zur Berechnung des optimalen Werts:**

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

- **Beobachtung:**

- Der rekursive Algorithmus ist ineffizient wegen **redundanter** Subprobleme.
- Die Anzahl der rekursiven Aufrufe wächst **exponentiell**.

Beispiel



- Für diese spezielle Anordnung gilt: $p(1) = 0$, $p(j) = j - 2$ für $j \geq 2$.
- Der Rekursionsbaum zeigt, dass viele Compute-Opt-Aufrufe mit den gleichen Argumenten wiederholt ausgeführt werden (z.B. Compute-Opt(0) und Compute-Opt(1)).
- Die Struktur des Rekursionsbaums ähnelt der eines Binärbaums, und die Anzahl der Knoten wächst exponentiell mit n . Dies führt zu einer exponentiellen Laufzeit des Brute-Force-Ansatzes.

Gewichtetes Interval Scheduling: Speicherung (Memoization)

"top-down"

- **Speicherung:**
 - Speichere die Ergebnisse jedes Teilproblems in einem Cache (z.B. einem Array M).
 - Berechnung eines Teilproblems erfolgt nur noch, wenn dessen Ergebnis noch nicht gespeichert ist.
- **Allgemein:**
 - **Eingabe:** n Jobs mit Startzeiten s_1, \dots, s_n , Endzeiten f_1, \dots, f_n und Gewichten w_1, \dots, w_n .
 - Sortiere Jobs nach Beendigungszeit, sodass $f_1 \leq f_2 \leq \dots \leq f_n$.
 - Berechne $p(1), p(2), \dots, p(n)$.
- **Algorithmus mit Memoization:**

```

Initialisiere ein Array M der Größe n+1 mit leeren Werten.
M[0] <- 0

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

```

- **Globales Array:** M ist ein globales Array, um die berechneten optimalen Werte zu speichern.
- Sehr ähnlich zu **Brute-force-Ansatz** aber das `if-statement` wurde hinzugefügt.

Gewichtetes Interval Scheduling: Laufzeit

- **Behauptung:** Die Version mit Speicherung (Memoization) benötigt $O(n \log n)$ Zeit.
- **Begründung der Laufzeit:**
 - **Sortieren nach Beendigungszeit:** $O(n \log n)$.
 - **Berechne $p(j)$:** $O(n \log n)$ mittels binärer Suche (für jedes Intervall) auf der nach Beendigungszeit sortierten Folge der Endzeiten, um den größten kompatiblen Job

mit kleinerem Index zu finden. Für jeden der n Jobs wird eine binäre Suche in $O(\log n)$ Zeit durchgeführt.

- --> liefert Werte $p(1) \dots p(n)$
- **Compute-Opt(j)** : Jeder Aufruf benötigt $O(1)$ Zeit (ohne die rekursiven Aufrufe selbst) und
 - (i) liefert entweder einen bereits existierenden Wert aus $M[j]$ zurück.
 - (ii) oder berechnet einen neuen Eintrag für $M[j]$ und macht **zwei** rekursive Aufrufe. Da jeder Eintrag in M nur einmal berechnet wird, gibt es maximal $n + 1$ rekursive Aufrufe, die tatsächlich eine Berechnung durchführen. Jeder dieser Aufrufe benötigt konstante Zeit.
- **Maß für den Fortschritt $\varphi =$ die Anzahl der nicht leeren Einträge in M .**
 - Am Anfang gilt $\varphi = 0$, danach $\varphi \leq n$.
 - Jeder rekursive Aufruf, der einen leeren Eintrag berechnet, erhöht φ um 1. Da φ maximal n erreicht, gibt es maximal n solcher Aufrufe.
- Die gesamte Laufzeit von **Compute-Opt(n)** ist somit $O(n)$.
- **Gesamlaufzeit:** Die Gesamlaufzeit des Algorithmus wird durch das Sortieren und die Berechnung der $p(j)$ -Werte dominiert, gefolgt von den maximal n konstanten Zeitaufgaben von **Compute-Opt**. Daher ist die Gesamlaufzeit $O(n \log n)$.

Gewichtetes Interval Scheduling: Bottom-up

- **Bottom-up dynamische Programmierung: Iterative Lösung.**

Allgemein

- **Eingabe:** n Jobs mit Startzeiten s_1, \dots, s_n , Endzeiten f_1, \dots, f_n und Gewichten w_1, \dots, w_n .
- Sortiere Jobs nach Beendigungszeit, sodass $f_1 \leq f_2 \leq \dots \leq f_n$.
- Berechne $p(1), p(2), \dots, p(n)$.

- **Iterativer Algorithmus:**

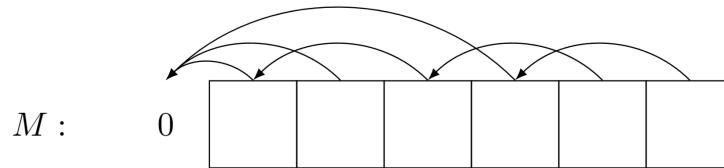
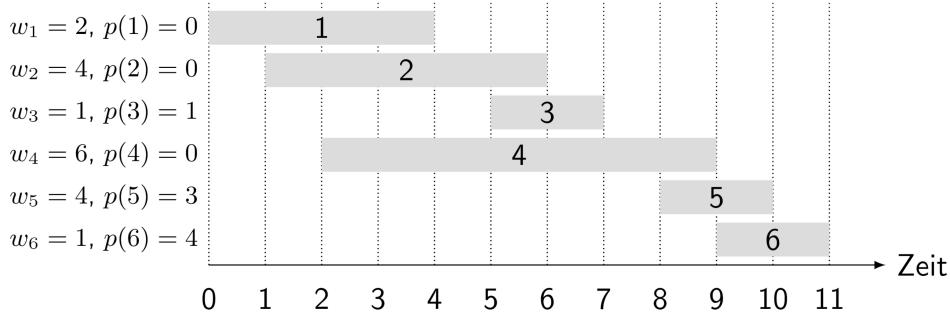
```
Iterative-Compute-Opt():
  M sei ein Array der Größe n+1.
  M[0] <- 0
  for j <- 1 bis n:
    M[j] <- max(w_j + M[p(j)], M[j - 1])
  return M[n]
```

- **Laufzeit:** Die Laufzeit von **Iterative-Compute-Opt** ist $O(n)$ (die Schleife läuft von 1 bis n). Das Sortieren der Jobs und die Vorberechnung von $p(\cdot)$ benötigen weiterhin $O(n \log n)$. Die **Gesamlaufzeit** des Bottom-up-Ansatzes beträgt somit $O(n \log n)$.

☰ Beispiel

Gegeben:

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



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

$$M[j - 1]:$$



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

$$M[j - 1]: \quad \textcolor{red}{2}$$



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

$$M[j - 1]: \quad \textcolor{red}{2} \quad \textcolor{red}{4}$$

$$\begin{array}{l}
 M : \quad \boxed{\begin{array}{|c|c|c|c|c|c|} \hline 0 & 2 & 4 & 4 & 6 & \\ \hline \end{array}}
 \\[10pt]
 w_j + M[p(j)]: \quad \begin{array}{ccccc} 2 & 4 & 3 & 6 & \end{array} \\[5pt]
 M[j-1]: \quad \begin{array}{ccccc} 0 & 2 & 4 & 4 & \end{array}
 \end{array}$$

$$\begin{array}{l}
 M : \quad \boxed{\begin{array}{|c|c|c|c|c|c|} \hline 0 & 2 & 4 & 4 & 6 & 8 \\ \hline \end{array}}
 \\[10pt]
 w_j + M[p(j)]: \quad \begin{array}{ccccc} 2 & 4 & 3 & 6 & 8 \\ \hline \end{array} \\[5pt]
 M[j-1]: \quad \begin{array}{ccccc} 0 & 2 & 4 & 4 & 6 \\ \hline \end{array}
 \end{array}$$

$$\begin{array}{l}
 M : \quad \boxed{\begin{array}{|c|c|c|c|c|c|} \hline 0 & 2 & 4 & 4 & 6 & 8 \\ \hline \end{array}}
 \\[10pt]
 w_j + M[p(j)]: \quad \begin{array}{ccccc} 2 & 4 & 3 & 6 & 8 & 7 \\ \hline \end{array} \\[5pt]
 M[j-1]: \quad \begin{array}{ccccc} 0 & 2 & 4 & 4 & 6 & 8 \\ \hline \end{array}
 \end{array}$$

$$\begin{array}{l}
 M : \quad \boxed{\begin{array}{|c|c|c|c|c|c|} \hline 0 & 2 & 4 & 4 & 6 & 8 \\ \hline \end{array}}
 \\[10pt]
 w_j + M[p(j)]: \quad \begin{array}{ccccc} 2+0 & 4+0 & 1+2 & 6+0 & 4+4 & 1+6 \\ \hline \end{array} \\[5pt]
 M[j-1]: \quad \begin{array}{ccccc} 0 & 2 & 4 & 4 & 6 & 8 \\ \hline \end{array}
 \end{array}$$

so ein Beispiel haben wir auch als ue: [ue7-A2-\(b\)](#)

Gewichtetes Interval Scheduling: Finden einer Lösung

② Frage

- Der Algorithmus berechnet den optimalen Wert. Wie bekommen wir aber die Menge der ausgewählten Jobs (die eigentliche Lösung)?

- **Antwort:** Durch eine Nachbearbeitung (Backtracking) des berechneten M -Arrays.

Ablauf

1. Führe `M-Compute-Opt(n)` oder `Iterative-Compute-Opt(n)` aus, um das Array M mit den optimalen Werten bis zu jedem Job j zu füllen.
2. Führe eine rekursive Funktion `Find-Solution(n)` aus, um die Menge der ausgewählten Jobs zu rekonstruieren.

- **Algorithmus zur Rekonstruktion der Lösung:**

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

- Die Funktion geht das Array M von hinten nach vorne durch.
- Für jeden Job j wird geprüft, ob die Einbeziehung von Job j zu einem höheren Wert führt ($w_j + M[p(j)] > M[j - 1]$).
- Wenn ja, wird Job j zur Lösung hinzugefügt und die Suche wird mit $p(j)$ fortgesetzt (da Jobs zwischen $p(j)$ und $j - 1$ nicht in der optimalen Lösung sein können, wenn j enthalten ist).
- Wenn nein, wird Job j nicht zur Lösung hinzugefügt und die Suche wird mit $j - 1$ fortgesetzt.
- **Laufzeit:** Die Anzahl der rekursiven Aufrufe von `Find-Solution` ist maximal n , da in jedem Schritt j reduziert wird. Daher ist die Laufzeit von `Find-Solution` $O(n)$.

Beispiel - Ergebnis finden

Find-Solution (6)

 $M :$

0	2	4	4	6	8	8
---	---	---	---	---	---	---

 $w_j + M[p(j)]:$ $M[j - 1]:$

2	4	3	6	8	7
0	2	4	6	8	7
	2	4	6	8	7

2

5

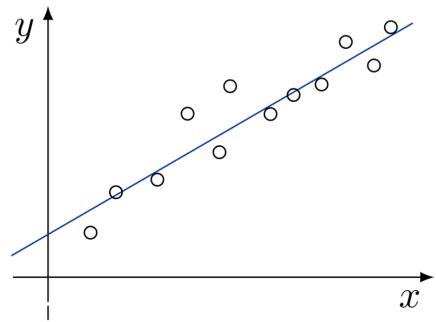
Segmented Least Squares

Least Squares

⌚ Definition

- **Fundamentales Problem in der Statistik und der Numerischen Analyse.**
- **Gegeben:** n Punkte in der Ebene: $(x_1, y_1), \dots, (x_n, y_n)$.
- **Finde:** Eine Gerade $y = ax + b$, welche die Summe der quadrierten Fehler minimiert.
- **Fehlerfunktion:** $\text{Err} = \sum_{i=1}^n (y_i - (ax_i + b))^2$

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



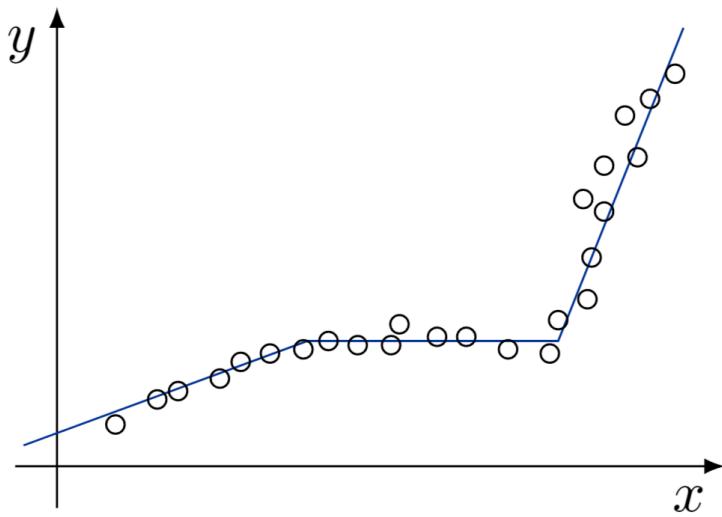
- **Analytische Lösung:** Der minimale Fehler wird erreicht, wenn:

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

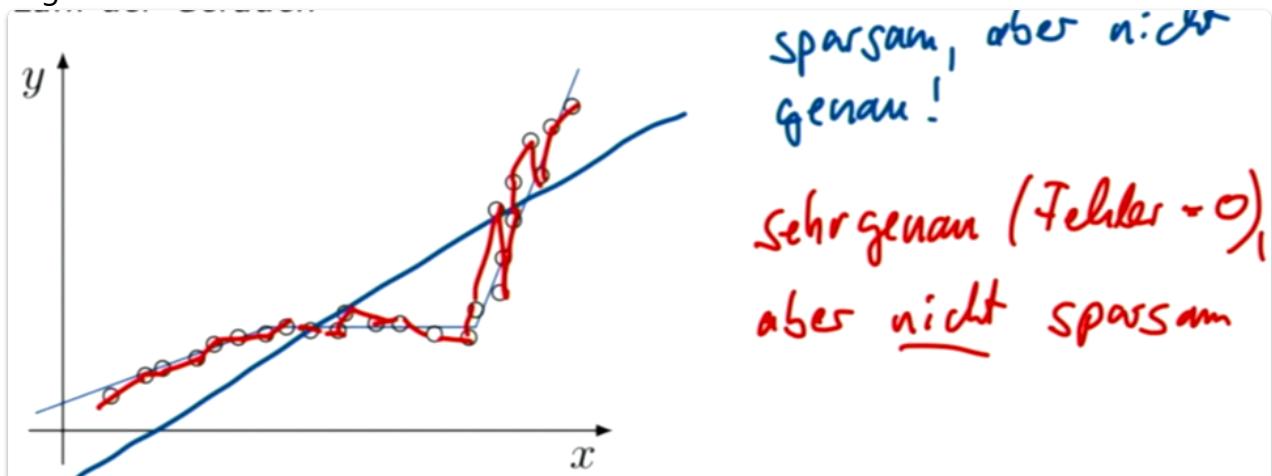
Beste Lösung (a, b) kann man in $O(n)$ berechnen

Segmented Least Squares

- **Punkte durch eine Folge von Geradensegmenten annähern.**
- **Gegeben:** n Punkte in der Ebene $(x_1, y_1), \dots, (x_n, y_n)$ so dass $x_1 < x_2 < \dots < x_n$.
- **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.
 - **Genauigkeit:** Höhe des Fehlers.
 - **Sparsamkeit:** Anzahl der Geraden (Segmente).



- **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 $c > 0$. Der Parameter c gewichtet das Verhältnis zwischen der Reduktion des Fehlers und der Hinzufügung eines neuen Segments.



Dynamischer Ansatz: Segmented Least Squares

Notation

- $OPT(j)$ = minimale Kosten für die Punkte p_1, p_2, \dots, p_j .
- $e(i, j) =$ minimale Summe des quadrierten Fehlers für die Punkte p_i, p_{i+1}, \dots, p_j , wenn diese durch eine einzige Gerade approximiert werden.

Berechnen von $OPT(j)$

- Betrachte das letzte Segment, das die Punkte p_i, p_{i+1}, \dots, p_j für ein bestimmtes i nutzt.

- Die Kosten für diese Lösung wären die optimalen Kosten für die Punkte bis p_{i-1} plus die Kosten des letzten Segments (Fehler plus Kosten für ein neues Segment).
- Kosten = $OPT(i - 1) + e(i, j) + c$.

⌚ Rekursive Definition für $OPT(j)$

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

alle Mögl. für Start des letzten Segm.

Lsg. für $1, \dots, i-1$

neuer Fehler für i, \dots, j

Kosten für ein neues Segm.

- Basisfall:** $OPT(0) = 0$ (keine Punkte, keine Kosten).
- Rekursiver Schritt:** Um die optimalen Kosten bis zum Punkt j zu finden, betrachten wir alle möglichen letzten Segmente, die bei einem Punkt i ($1 \leq i \leq j$) beginnen und bis j gehen. Für jedes solche Segment berechnen wir die Kosten als die optimalen Kosten bis zum Punkt $i - 1$ plus den Fehler des Segments von i bis j plus die Kosten c für das Hinzufügen eines neuen Segments. Wir wählen das Minimum über alle möglichen Startpunkte i .

Segmented Least Squares: Algorithmus

```

Segmented-Least-Squares(  $P = \{p_1, p_2, \dots, p_n\}$  )
   $M[0] = 0$ 
   $\underbrace{\text{for } j \leftarrow 1 \text{ bis } n}_{O(n^2) \text{ Iterat.}}$  {
     $\underbrace{\text{for } i \leftarrow 1 \text{ bis } j}_{O(n) \text{ Zeit}}$ 
      berechne Fehler  $e(i, j)$  für Punkte  $p_i, \dots, p_j$ 
    }
  }
   $\underbrace{\text{for } j \leftarrow 1 \text{ to } n}_{O(n) \text{ Iterat.}}$  {
     $M[j] = \min_{1 \leq i \leq j} (M[i-1] + e(i, j) + c)$ 
     $\underbrace{\text{return } M[n]}_{O(n) \text{ Mögl.}}$ 
  }
}

```

$O(n^3)$ Vorberechnung von $e(i, j)$

$O(n^2)$

- Laufzeit:** $O(n^3)$.
 - Flaschenhals:** Das Berechnen von $e(i, j)$ für $O(n^2)$ Paare (i, j) benötigt $O(n)$ Operationen pro Paar (um die Parameter der besten Geraden und den Fehler zu berechnen).
 - Die äußere Schleife für j läuft von 1 bis n .
 - Die innere Schleife für i läuft von 1 bis j (bis zu n Mal).
 - Innerhalb der inneren Schleife wird $e(i, j)$ berechnet und das Minimum gefunden.

- **Finden einer Lösung:** Analog zum Interval Scheduling durch Rückverfolgen der Minimierung im Array M . Wir merken uns bei der Berechnung von $M[j]$, welcher Wert von i das Minimum erzeugt hat, und rekonstruieren so die Segmente.
- **Verbesserungsmöglichkeiten:** Kann mithilfe geschickterer Vorberechnung und Wiederverwendung von Zwischenergebnissen zu $O(n^2)$ verbessert werden.

Zwischenfazit:

Zwischenfazit

• wir betrachten polynomiale Anzahl Teilprobleme (Arraygröße)

• optimale Lösung lässt sich aus opt. Lösungen geeigneter Teilprobleme zusammensetzen

• Teilprobleme lassen sich von klein nach groß anfüllen und rekursiv lösen, d.h.

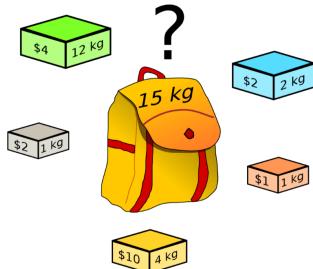
for $j = 1, \dots, n$

$M[j] = f(M[1], \dots, M[j-1])$ // z.B. binäre Auswahl oder
lineare Auswahl
return $M[n]$

• Rekonstruktion der besten Lösung (nicht nur des Wertes) durch Backtracking

Rucksackproblem

Gegeben: n Gegenstände mit positiv rationalen Gewichten g_1, \dots, g_n , Werten w_1, \dots, 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.

☰ Beispiel

#	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 and Bound wurde ein Algorithmus mit Laufzeit $O(2^n)$ vorgestellt.
- Ziel ist ein Algorithmus mit Laufzeit $O(nG)$, falls $nG < 2^n$ wesentlich effizienter ist. (G ... Eingabegröße)
- 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 $1, \dots, i$

- **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!

⌚ Richtiger Ansatz: Gewichtsbeschränkung

Definition: $OPT(i, g)$ = Maximaler Profit für die Teilmenge von den Gegenständen $1, \dots, i$, 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 und die Gewichtsbeschränkung gleich bleibt.
- **Fall 2:** $OPT(i, g)$ wird erreicht mit einer Lösung, die den Gegenstand i enthält.
 - Neue Gewichtsbeschränkung = $g - w_i$.
 - Daher gilt dann $OPT(i, g) = w_i + OPT(i - 1, g - w_i)$.

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

Fall1 Fall2

Rucksackproblem: Bottom-Up

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

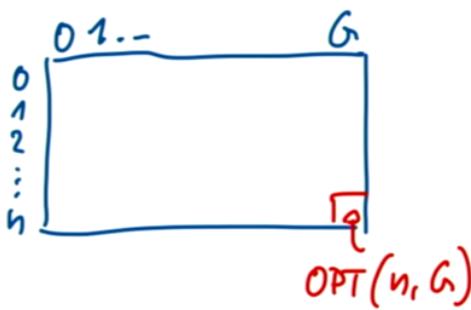
Eingabe: $n, G, g_1, \dots, g_n, w_1, \dots, w_n$

```

for  $g \leftarrow 0$  bis  $G$ 
     $M[0, g] \leftarrow 0$ 

for  $i \leftarrow 1$  bis  $n$ 
    for  $g \leftarrow 0$  bis  $G$ 
        if  $g_i > g$ 
             $M[i, g] \leftarrow M[i - 1, g]$ 
        else
             $M[i, g] \leftarrow \max\{M[i - 1, g], w_i + M[i - 1, g - g_i]\}$ 
return  $M[n, G]$ 

```



Als Platz haben wir die Tabelle mit n Zeilen und g Spalten und

☰ Beispiel

Erstellen von Tabelle

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

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

$G = 11$

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

Backtracking: Bestimmen der Lösung

Nach Erstellen der Tabelle → Backtracking um Lösung zu bekommen

Bestimmen der Lösung

Optimale Lösung: Mit Hilfe der Werte im Array M :

Find-Solution(M):

```

 $i \leftarrow n$ 
 $k \leftarrow G$ 
 $A \leftarrow \emptyset$ 
while  $i > 0$  und  $k > 0$ 
    if  $M[i, k] \neq M[i-1, k]$ 
         $A \leftarrow A \cup \{i\}$ 
         $k \leftarrow k - g_i$ 
     $i \leftarrow i - 1$ 
return  $A$ 
```

Rucksackproblem: Laufzeit

Laufzeit: $O(nG)$.

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

$$G = 2^{\log(G)}$$

$$\implies O(n \cdot 2^{\log(G)})$$

- „Pseudo-polynomiell.“

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

Verbesserung

Speicherung: Man muss eigentlich nicht das gesamte Array speichern.

Verbesserung:

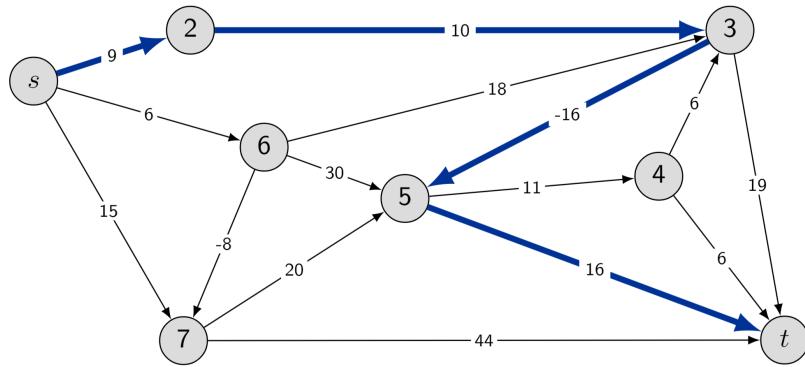
- Man merkt sich nur die letzte Zeile.
- Die Aktualisierung der Einträge erfolgt von rechts nach links.

Kürzeste Pfade

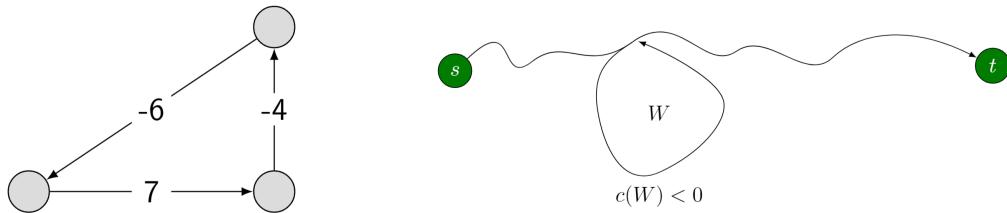
Kürzester Kantenzug: Gegeben sei ein gerichteter Graph $G = (V, E)$, mit Kantengewichten c_{vw} . Finde einen kürzesten Kantenzug zwischen Knoten s und t .

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

Beispiel:



Kreise mit negativen Kosten (negative Kreise)



Bemerkung: Im Falle von negativen Kreisen:

- keine sinnvolle Definition von kürzesten **Kantenzügen** mehr möglich.
- Definition von kürzesten **Pfaden** bleibt jedoch sinnvoll.
- **Unterschied:** In einem Pfad wird jeder Knoten maximal einmal besucht, in einem Kantenzug beliebig oft.

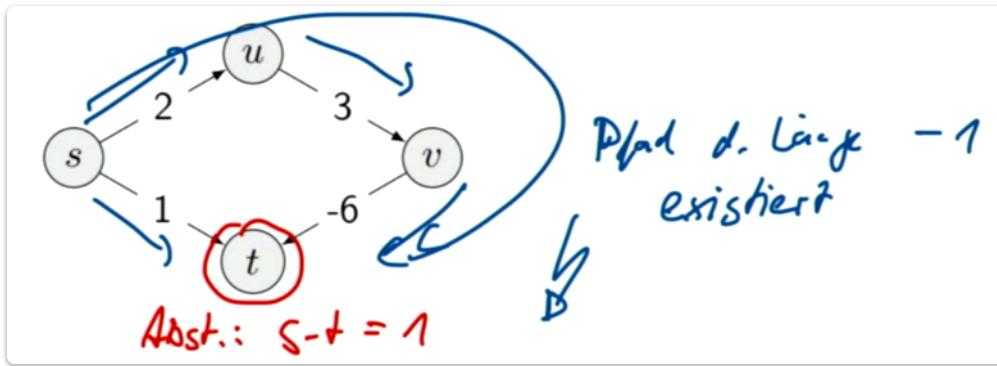
Kürzeste Pfade: Komplexität

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

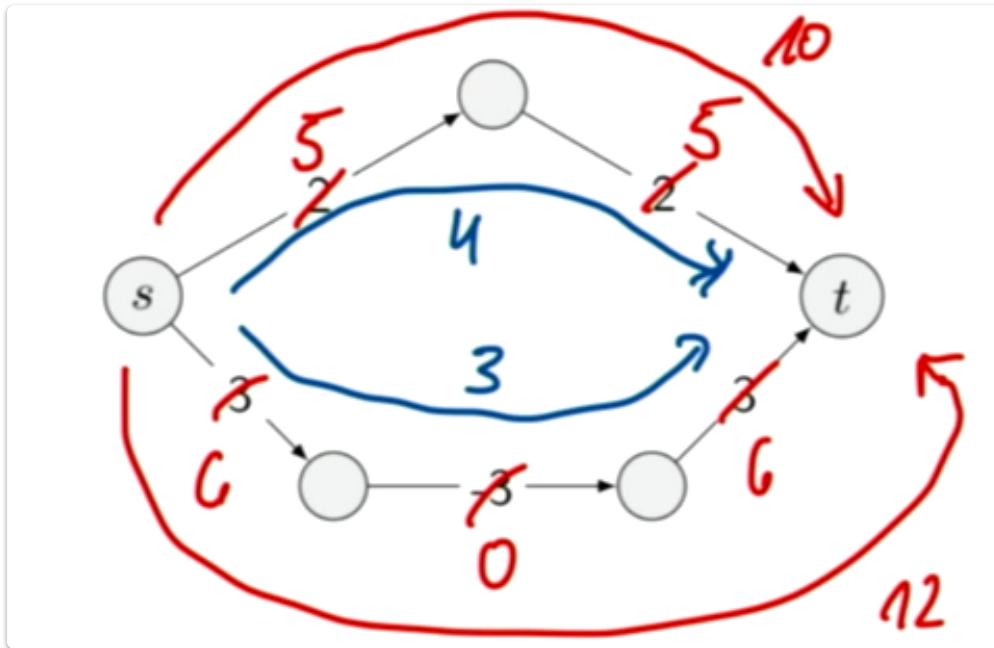
Bemerkungen: Verbietet man negative Kreise wird das Problem jedoch wieder in Polynomialzeit lösbar.

Kürzeste Pfade: Falsche Ansätze

Algorithmus von Dijkstra: Schlägt fehl bei negativen Kantengewichten.



Neugewichtung: Zu jedem Kantengewicht eine Konstante dazuzugeben schlägt fehl.



Kürzeste Pfade: Bellman's Gleichungen

Sei $G = (V, E)$ ein gerichteter Graph ohne negativen 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.

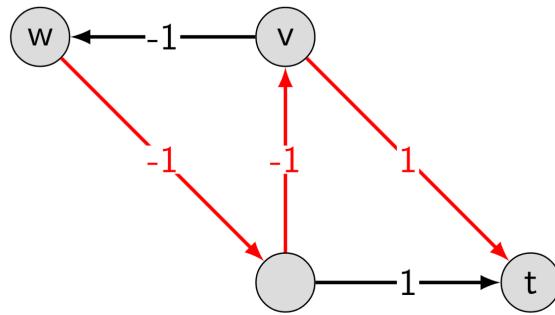
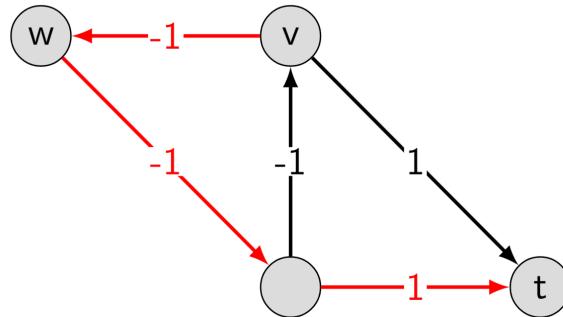
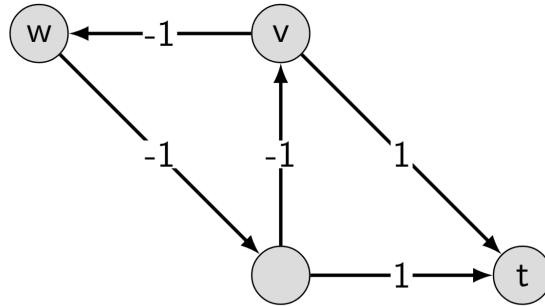
- **Basisfall:** Aussage gilt für $v = t$ mit 0 Kanten.
- **Induktionsvoraussetzung:**
 - Sei P ein kürzester $v - t$ Pfad mit ℓ Kanten und (v, w) die erste Kante von P .
 - Da es keine negativen Kreise gibt, liegt ein Knoten nie zweimal auf einem kürzesten Kantenzug.

- $P - (v, w)$ 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 P' über einen anderen Zwischenknoten $w' \rightarrow \dots \rightarrow t$. Widerspruch zur Annahme, dass P kürzester $v - t$ Pfad ist.

$$\implies OPT(v) = c_{vw} + OPT(w)$$

Gegenbeispiel

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

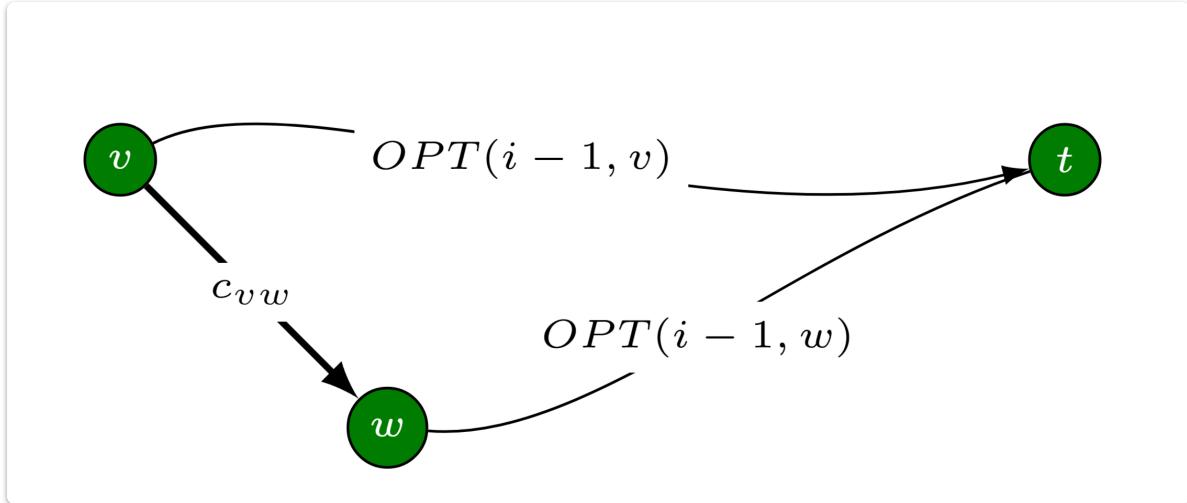


Wie man sieht, gilt hier die Bellman's Gleichung nicht mehr, da sich die Kosten durch wiederholtes Durchlaufen des negativen Kreises beliebig verringern lassen. Es existiert kein "kürzester" Pfad mehr im Sinne einer festen Länge, da man durch den negativen Kreis immer "kürzer" werden kann.

Kürzester Pfad: 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. Dann besteht der kürzeste Pfad aus einer ersten Kante (v, w) und dem besten $w - t$ Pfad, der höchstens $i - 1$ Kanten benutzt.



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

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.

Denn ein "Pfad" mit $\geq n$ Kanten muss einen Kreis enthalten.

Maximal kann die Länge $n - 1$ sein, da keine Kante doppelt vorkommt.

Kürzester Pfad: Implementierung

```

Simple-Shortest-Path-DP( $G, s, t$ ):
foreach node  $v \in V$ 
     $M[0, v] \leftarrow \infty$ 
 $M[0, t] \leftarrow 0$ 
for  $i \leftarrow 1$  bis  $n - 1$ 
    foreach Knoten  $v \in V$ 
         $M[i, v] \leftarrow M[i - 1, v]$ 
    foreach Kante  $(v, w) \in E$ 
         $M[i, v] \leftarrow \min(M[i, v], c_{vw} + M[i - 1, w])$ 
return  $M[n - 1, s]$ 

```

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

$$n = |V|$$

$$m = |E|$$

$$n \leq m$$

☰ Beispiel für den nicht verbesserten Algorithmus

siehe Folien: AD_12_DynamischeProgrammierung, p.52

Man schaut sich bei jeder Iteration alle Knoten an und ob sie sich seit letzter Iteration verbessert haben / verbessern können.

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.

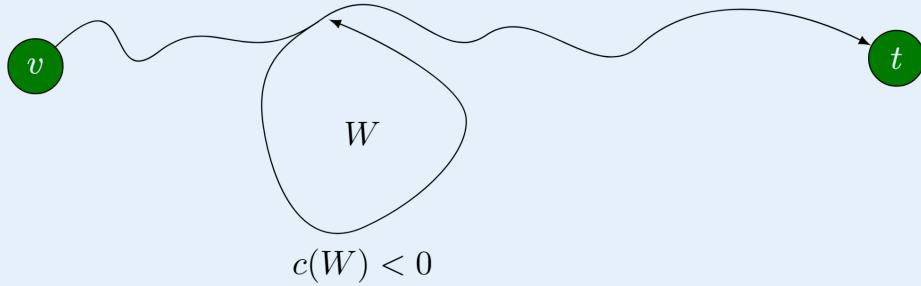
ⓘ Negative Kreise erkennen

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.

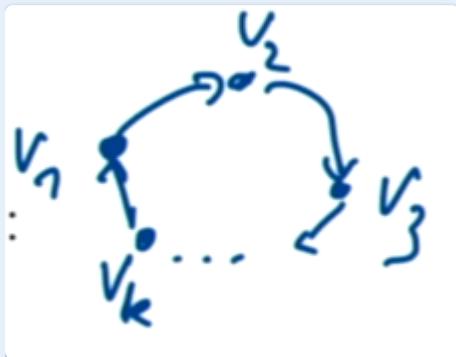
✓ Beweis: durch Widerspruch

- Da $OPT(n, v) < OPT(n - 1, v)$, wissen wir, dass K genau n Kanten hat.
- Mit Hilfe des Schubfachprinzip 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.



ⓘ Umgekehrt: Wenn es einen Kreis gibt, dann muss es eine Verbesserung geben

Lemma: Falls G einen negativen Kreis enthält, der 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, \dots, v_k) , dann gilt:

$$m = \sum_{i=1}^k [OPT(n - 1, v_i) - OPT(n - 1, v_{i+1 \bmod k})] = 0.$$

- Falls nun C ein negativer Kreis ist, ergibt sich $\sum_{i=1}^k c_{v_i v_{i+1 \bmod k}} < 0 = m$.
- D.h. es muss mindestens ein i existieren mit

$$c_{v_i v_{i+1 \bmod k}} < OPT(n-1, v_i) - OPT(n-1, v_{i+1 \bmod k})$$

und somit gilt für die Kante $(v_i, v_{i+1 \bmod k})$, dass

$$c_{v_i v_{i+1 \bmod k}} + OPT(n-1, v_{i+1 \bmod k}) < OPT(n-1, v_i).$$

Folgerung aus den Lemmas:

Aus den beiden vorherigen Lemmas folgt nun:

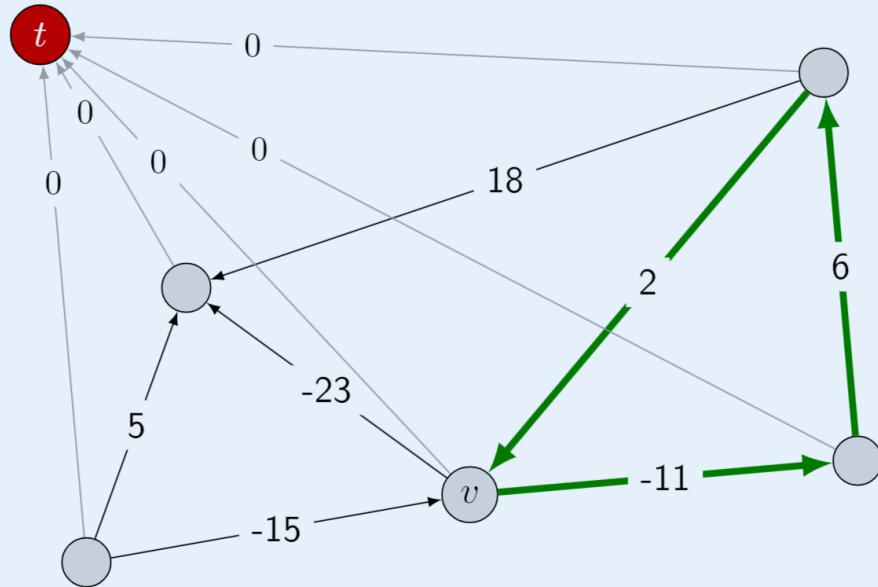
Theorem: G enthält einen negativen Kreis von dem aus t erreicht werden kann genau dann wenn ein Knoten v mit $OPT(n, v) < OPT(n-1, v)$ existiert.

① 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.
- Dijkstra hatte Zeit $O((n + m) \cdot \log(n))$, aber dafür keine Behandlung von negativen Kanten

Effiziente Implementierung von Bellman-Ford

Push-Based-Shortest-Path-DP(G, s, t):

foreach Knoten $v \in V$

$M[v] \leftarrow \infty$

 Nachfolger[v] $\leftarrow \emptyset$

$M[t] = 0$

for $i \leftarrow 1$ bis $n - 1$

foreach Kante $(v, w) \in E$

if $M[v] > M[w] + c_{vw}$

$M[v] \leftarrow M[w] + c_{vw}$

 Nachfolger[v] $\leftarrow w$

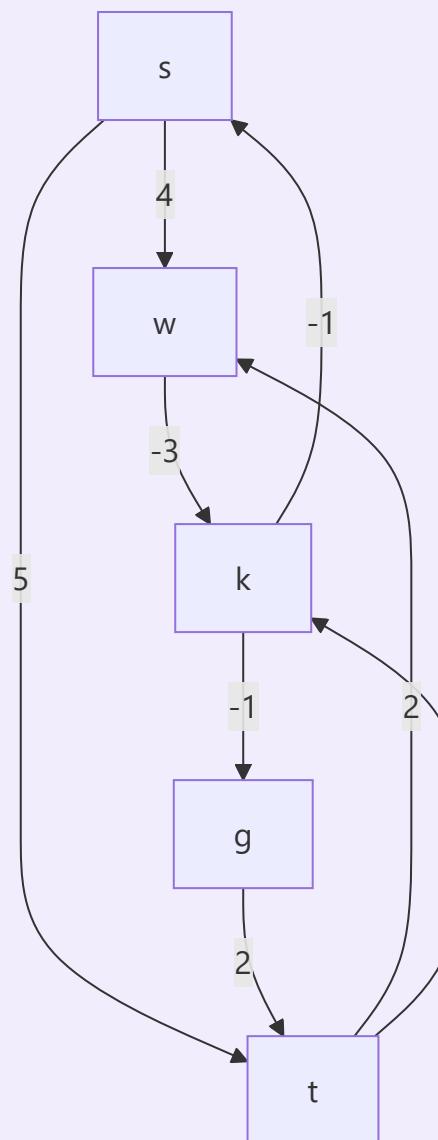
if kein $M[w]$ Wert ändert sich in Iteration i , stop.

return $M[s]$

:≡ Beispiel für den verbesserten Algorithmus

Beispiel siehe [AD_12_DynamischeProgrammierung, p.60](#)

:≡ Weiteres Beispiel aus der ue7-A4-(b)



Man schaut, mit wie viel Gewicht man zu t kommt, je nach dem wie viele Schritte man gehen kann und nimmt, falls man einen anderen Wert hat den kleineren.

	0	1	2	3	4
t	0	0	0	0	0
w	∞	∞	∞	-2	-2
k	∞	∞	1	1	1
s	∞	5	5	5	2
g	∞	2	2	2	2

Ergebnis: $s \rightarrow w \rightarrow k \rightarrow g \rightarrow t$

Finden eines kürzesten Pfades: Korrektheit

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

⌚ Definition

Ein Knoten w ist der Nachfolger eines Knotens 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: $M[v] \geq c_{vw} + M[w]$.
 → wenn der Nachfolger gesetzt wird gilt $M[v] = c_{vw} + M[w]$; später kann sich $M[w]$ aber noch verringern.

ⓘ Lemma

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

✓ Beweis

- Seien v_1, \dots, 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 $M[v_i] \geq c_{v_iv_{i+1}} + M[v_{i+1}]$ für alle i mit $1 \leq i < k$ (unmittelbar bevor die Kante (v_k, v_1) gesetzt wird) $M[v_k] > c_{v_kv_1} + M[v_1]$.
- Nach Aufsummieren aller Ungleichungen verschwinden die $M[v_i]$ -Terme und wir erhalten: $0 > \sum_{i=1}^{k-1} c_{v_iv_{i+1}} + c_{v_kv_1}$, also ist C ein negativer Kreis.

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

- Versteh 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

