Algorithmentheorie

Daniel Neuen (Universität Bremen)

WiSe 2023/24

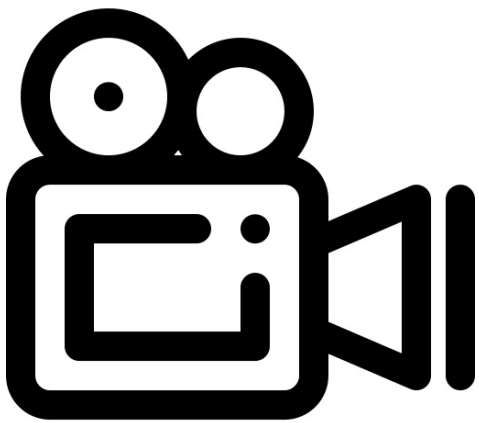
Kürzeste Wege (Teil 2) und Dynamische Programmierung

9. Vorlesung

Aufzeichnung der Vorlesung

Diese Vorlesung wird aufgezeichnet und live gestreamt.

- Aufzeichnungen nur der Lehrenden durch sich selbst.
- Bei Rückfragen aus dem Auditorium und Diskussion bitte deutlich anzeigen, falls das Mikro stumm geschaltet werden soll.



Organisatorisches

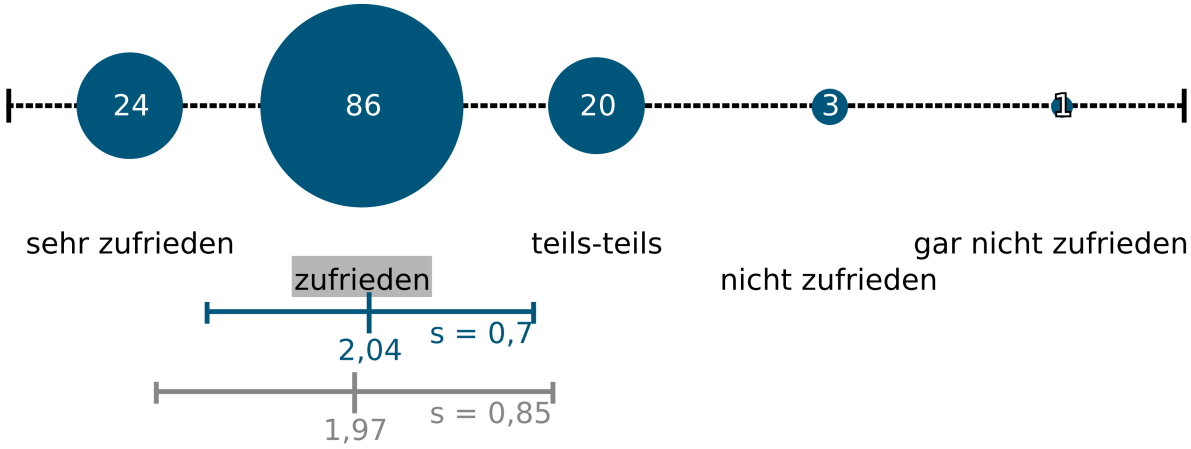
Evaluation:

- 136 Bewertungen abgegeben $\rightarrow$ 5 Bonuspunkte

Aufgaben zur Klausurvorbereitung:

- separates Blatt mit Aufgaben zur Klausurvorbereitung
- spätestens am Montag im StudIP verfügbar
- Aufgaben decken Stoff der gesamten Vorlesung ab
- Musterlösungen zum Ende der Vorlesungszeit
- Keine Abgabe bzw. Korrektur!

Evaluation



+ Gute Vorlesung und Tutorium
+ Gute Erklärungen (VL + Tut)
+ Präsenzübungen hilfreich
+ Fragen werden ausführlich beantwortet
+ Aufzeichnung der VL

- Overlays in Folien
- Mehr Interaktion in VL
- Mehr Zeit für Grundlagen in VL
- Blatt 2 zu schwer
- Hausaufgaben früher veröffentlichen
- Musterlösung hochladen
- Besprechung Hausaufgaben hilft nicht
- Hörsaal nicht gut

Kürzeste Wege

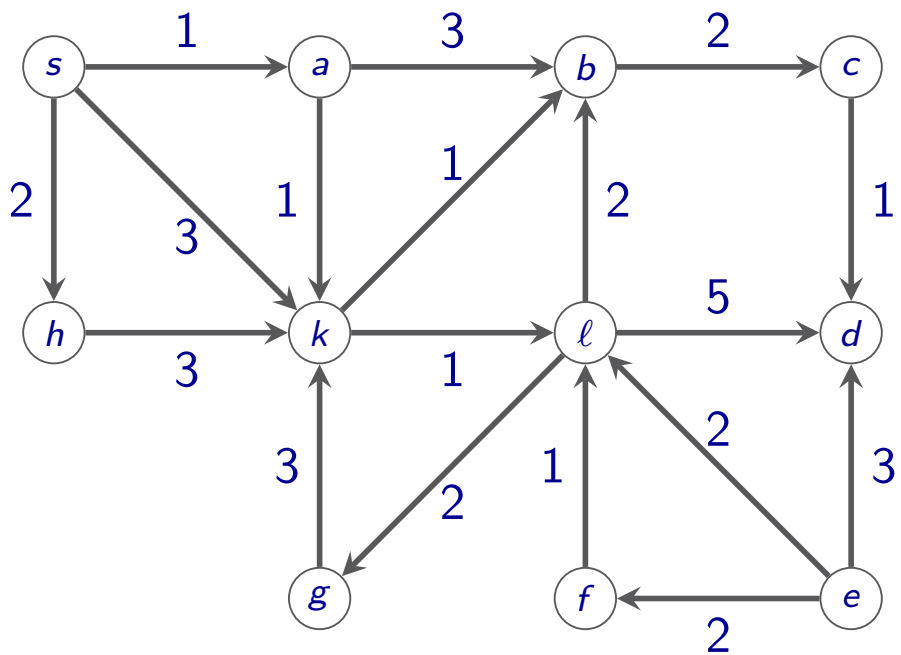
mit beliebigen Kantengewichten:

Bellmann-Ford Algorithmus

Wiederholung: Das Kürzeste-Wege-Problem

Gegeben: gewichteter Digraph $G = (V, E, c)$ und ein Startknoten $s \in V$.

Gesucht: für jedes $v \in V$ ein kürzester Weg von s nach v und die Distanz $d(s, v)$. (single-source shortest-paths)



Wiederholung: Dijkstra's Algorithmus

Dijkstra's Algorithmus

1 **Initialisiere** $S \leftarrow \emptyset$, $dist(s) \leftarrow 0$ und $\forall v \in V \setminus \{s\} : dist(v) \leftarrow \infty$
2 **while** $S \neq V$ **do**
3 Finde $u \in V \setminus S$ mit minimaler vorläufiger Distanz $dist(u)$
4 $S \leftarrow S \cup \{u\}$
5 **for** $v \in V \setminus S$ mit $(u, v) \in E$ **do**
6 $dist(v) \leftarrow \min\{dist(v), dist(u) + c(u, v)\}$
7 **return** $dist(v)$ für alle $v \in V$

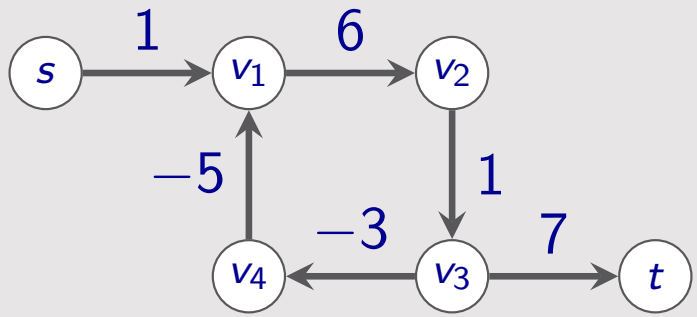
Satz

Sei $G = (V, E, c)$ mit $c : E \to \mathbb{R}_+$ und $|V| = n$ und $|E| = m$. Dann hat Dijkstra's Algorithmus eine Laufzeit von $\mathcal{O}((m + n) \log n)$ und berechnet die kürzeste Distanz $d(s, v)$ für alle $v \in V$.

Negative Kreise

Definitionen

Ein negativer Kreis in einem gewichteten Graphen $G = (V, E, c)$ ist ein Kreis C mit negativer Kantenkostensumme, d.h. $\sum_{e \in C} c_e < 0$.



Lemma

In einem Graphen $G = (V, E, c)$ gilt für $u, v \in V$: $d(u, v) = -\infty$ genau dann, wenn ein Weg W von u nach v existiert mit einem Knoten $w \in W$ wobei w auf einem negativen Kreis liegt.

Annahme für das Kürzeste-Wege Problem: Graph G enthält keine negativen Kreise.

Teilwege kürzester Wege

Lemma

Sei $P = (v_1, \ldots, v_k)$ ein kürzester (v_1, v_k)-Weg. Dann ist für alle i, j mit $1 \leq i \leq j \leq k$: $P' = (v_i, \ldots, v_j)$ ein kürzester (v_i, v_j)-Weg.

Korollar

Sei $s \in V$ ein ausgezeichneter Knoten, dann gilt für alle $(u, v) \in E$:

$$d(s, v) \leq d(s, u) + c(u, v).$$

Satz

Falls G keine negativen Kreise enthält und ein (s, t)-Weg existiert, dann existiert auch ein kürzester (s, t)-Weg mit höchstens $n - 1$ Kanten.

Beweis. Bei mehr Kanten gäbe es einen Kreis, dessen Entfernung die Kosten nicht erhöht.

Algorithmus von Bellman und Ford

(auch Moore-Bellman-Ford Algorithmus)

Bellman-Ford Algorithmus

Input : gewichteter Digraph $G = (V, E, c)$, Startknoten $s \in V$
Output : $d(s, v)$ und kürzesten Weg für alle $v \in V$, oder Information dass G einen negativen Kreis enthält

1 **Initialisiere** $dist(s) \leftarrow 0$, $\forall v \in V \setminus \{s\} : dist(v) \leftarrow \infty$
2 **for** $k := 1, \ldots, n-1$ **do** — $n-1$ Runden
3 **for** $(u, v) \in E$ **do**
4 **if** $dist(u) + c(u, v) < dist(v)$ **then**
5 $dist(v) \leftarrow dist(u) + c(u, v)$ und $pred(v) \leftarrow u$
6 **for** $(u, v) \in E$ **do** — Negative-Kreise-Test
7 **if** $dist(u) + c(u, v) < dist(v)$ **then**
8 Ausgabe, dass negativer Kreis enthalten
9 **return** $dist(v)$ und $pred(v)$ für alle $v \in V$

Analyse Bellman-Ford

Lemma

Am Ende jeder Runde $k \in \{1, \ldots, n-1\}$ gilt für alle $v \in V$

$$dist(v) \leq \min\{c(P) \mid P \text{ ist } (s,v)\text{-Weg mit höchstens } k \text{ Kanten}\}.$$

Beweis. Betrachte ein beliebiges $v \in V$. Wenn kein (s,v)-Weg existiert, dann ist nichts zu zeigen.

Angenommen v ist von s erreichbar. Induktion über k.

- IA $k = 1$: klar.

  $\min \ldots < \infty$ nur für $P = (s,v)$; es gilt $dist(v) = c(s,v)$.
- IS: Betrachte (s,v)-Weg $P = (s = v_0, v_1, \ldots, v_{k-1}, v_k = v)$ mit genau k Kanten. Für den Teilweg $P' = (s, \ldots, v_{k-1})$ gilt nach IV $dist(v_{k-1}) \leq c(P')$. In Runde k wird auch Kante (v_{k-1}, v_k) betrachtet und es gilt

$$dist(v_k) \leq dist(v_{k-1}) + c(v_{k-1}, v_k) \leq c(P') + c(v_{k-1}, v_k) = c(P).$$

□

Analyse Bellman-Ford

Satz

Wenn G keine von s erreichbaren negativen Kreise enthält, so berechnet der Algorithmus $dist(v) = d(s, v)$ für jeden Knoten v. Er erkennt korrekt, ob G einen von s erreichbaren negativen Kreis enthält und hat Laufzeit $\mathcal{O}(nm)$.

Beweisskizze.

- Keine von s erreichbaren negativen Kreise: Betrachte $v \in V$ und einen kürzesten (s, v)-Weg P (falls ein solcher existiert, sonst gilt trivialerweise $dist(v) = d(s, v) = \infty$). Dieser hat $k \leq n - 1$ Kanten. Mit vorigem Lemma gilt nach Phase k, dass $dist(v) \leq c(P) = d(s, v)$. Labels werden nie erhöht und nur gesenkt, wenn ein Weg existiert.

Analyse Bellman-Ford

- Negative Kreise:
  (1) Wenn kein von s erreichbarer negativer Kreis ex., dann gilt $dist(v) = d(s,v) \Rightarrow dist(v) \leq dist(u) + c(u,v)$, $\forall (u,v) \in E$ und der Alg. meldet korrekt, dass kein von s erreichbarer neg. Kreis ex.

  (2) Wenn von s erreichbarer neg. Kreis $C = v_1, \ldots, v_{k+1}$ (mit $v_1 = v_{k+1}$) mit $\sum_{i=1}^{k} c(v_i, v_{i+1}) < 0$ ex., dann gibt der Alg. dies zurück: Angenommen nicht. Der Alg. hat geprüft: $dist(u) + c(u,v) \geq dist(v)$ für alle $(u,v) \in E$. Da $v_1 = v_{k+1}$, gilt $\sum_{i=1}^{k} dist(v_i) = \sum_{i=1}^{k} dist(v_{i+1})$. Also

$$\sum_{i=1}^{k} dist(v_{i+1}) > \sum_{i=1}^{k} dist(v_{i+1}) + \sum_{i=1}^{k} c(v_i, v_{i+1})$$

$$= \sum_{i=1}^{k} \big(dist(v_i) + c(v_i, v_{i+1})\big) \geq \sum_{i=1}^{k} dist(v_{i+1}),$$

  ein Widerspruch.
- Laufzeit: $n-1$ Runden, pro Runde m Kantenupdates □

Zusammenfassung

Kürzeste Wege

- Keine negativen Kantenkosten: Dijkstra Algorithmus
- Keine negativen Kreise: Bellman-Ford Algorithmus

Designprinzip Dynamische Programmierung

Dynamische Programmierung (DP)

Designprinzip (Paradigma) für den Entwurf von effizienten Algorithmen

- **Andere Paradigmen:** Greedy, Divide-and-Conquer, Brute-Force

Designprinzip: Dynamische Programmierung

- Teile Probleminstanz in kleinere Teilprobleme auf.
- Löse kleine Teilprobleme durch erschöpfende Suche.
- Löse größere (Teil)probleme mit Hilfe von Lösungen für kleinere Teilprobleme.
- Speichere Teillösungen um mehrfache Berechung zu vermeiden.

  $\rightarrow$ zentraler Unterschied zu Divide-and-Conquer

Fibonacci-Folge via Divide-and-Conquer

Fibonacci-Folge:

Die Fibonacci-Folge $f_1, f_2, f_3, \ldots$ ist die unendliche Folge von natürlichen Zahlen, die durch das rekursive Bildungsgesetz

- $f_n = f_{n-1} + f_{n-2}$ für $n > 2$

mit den Anfangswerten

- $f_1 = 1$ und $f_2 = 1$

definiert ist.

Fibonacci mit Divide-and-Conquer

```
1 Function Fib(n):
2     if n ≤ 2 then
3         return 1
4     else
5         return Fib(n − 1) + Fib(n − 2)
```

Fibonacci-Folge via Divide-and-Conquer

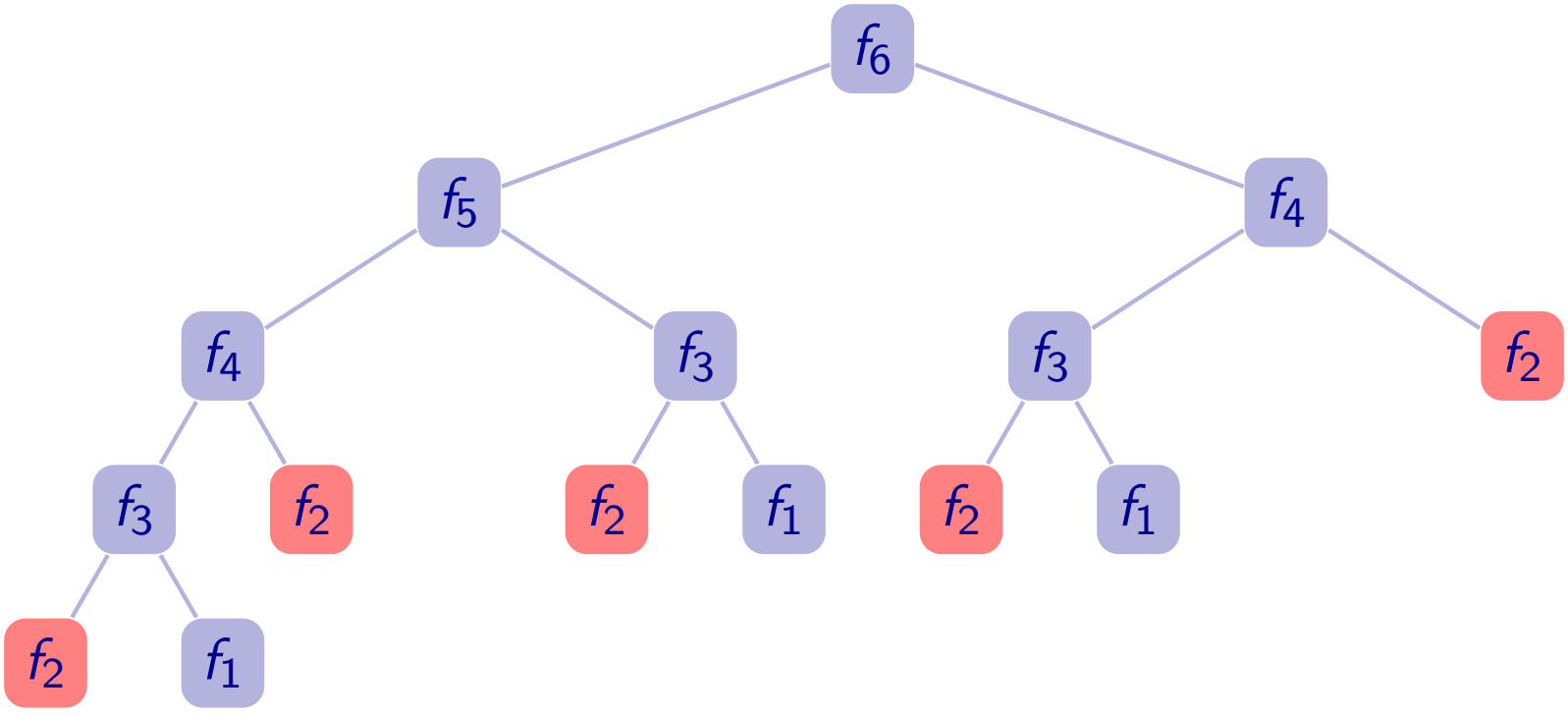
Fibonacci mit Divide-and-Conquer

```
1 Function Fib(n):
2     if n ≤ 2 then
3         return 1
4     else
5         return Fib(n − 1) + Fib(n − 2)
```

Laufzeit (Anzahl Funktionsaufrufe):

- $T(1) = T(2) = 1$.
- $T(n) = 1 + T(n-1) + T(n-2)$.
- $T(n) = \frac{1}{\sqrt{5}} \left[\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right] \approx 1.45 \cdot 1.62^n$

Fibonacci-Folge via Divide-and-Conquer



Der rekursive Algorithmus berechnet viele Werte mehrmals und verschwendet somit Rechenzeit.

Bsp.: Bei Berechnung von `Fib(70)` werden $\approx 1.3 \cdot 10^6$ Aufrufe auf `Fib(40)` gemacht, jeder Aufruf braucht etwa 10 Sekunden.

Dynamische Programmierung (DP)

Vermeide mehrfache Berechnung gleicher Funktionswerte durch Speichern bereits berechneter Werte:

- Verwende DP-Tabelle M um Zwischenergebnisse zu speichern.

  Fibonacci:

  $M[i] :=$ Zwischenergebnis f_i

- Berechne und speichere zunächst die Basisfälle.

  Fibonacci:

  ```
  M[]=new array;      M[1]=1; M[2]=1;
  ```

- Berechne den Rest der Tabelle entweder rekursiv (Memoization) oder iterativ (Tabularization).
- Größter Eintrag der DP-Tabelle enthält am Ende das Gesamtergebnis (**Fibonacci**: $M[n]$).

Memoization

Entwurfsprinzip Memoization:

- Rekursives Programm
- Speichere die Lösungen der Teilprobleme
- Teste bei Funktionsaufruf zunächst, ob der Wert der Funktion bereits berechnet wurde und gebe im positiven Fall den gespeicherten Wert zurück. Sonst setze die Rekursion zur Berechnung fort.

Fibonacci mit Memoization

```
1 Initialisiere Array M[·] und setze M[1] := 1 und M[2] := 1
2 Function Fib(n):
3     if M[n] undefiniert then
4         M[n] := Fib(n − 1) + Fib(n − 2)
5     return M[n]
```

Tabularization

Entwurfsprinzip Tabularization:

- Iteratives Programm
- Berechne bottom-up die Werte der Teillösungen und speichere sie in einer Tabelle. Wenn ein Teilproblem gelöst werden soll sind bereits die benötigten Teillösungen berechnet.

Fibonacci mit Tabularization

1 Initialisiere Array $M[\cdot]$ und setze $M[1] := 1$ und $M[2] := 1$

2 **Function** `Fib(`n`)`**:**

3 **for** $i = 3, \ldots, n$ **do**

4 $M[i] := M[i-1] + M[i-2]$

5 **return** $M[n]$

Fibonacci-Folge via Dynamischer Programmierung

Fibonacci mit Memoization

```
1 Initialisiere Array M[·] und setze M[1] := 1 und M[2] := 1
2 Function Fib(n):
3     if M[n] undefiniert then
4         M[n] := Fib(n − 1) + Fib(n − 2)
5     return M[n]
```

Fibonacci mit Tabularization

```
1 Initialisiere Array M[·] und setze M[1] := 1 und M[2] := 1
2 Function Fib(n):
3     for i = 3, . . . , n do
4         M[i] := M[i − 1] + M[i − 2]
5     return M[n]
```

- Laufzeit in beiden Fällen: $\mathcal{O}(n)$

Memoization vs. Tabularization

- Im rekursiven Ansatz werden genau die Werte berechnet, die zur Lösung benötigt werden.
- In der Implementierung können hashmaps/maps benutzt werden um Speicherverbrauch zu optimieren.
- Der iterative Ansatz vermeidet Overhead für Funktionsaufrufe und erreicht in der Praxis oft bessere Laufzeiten, wenn alle Werte berechnet werden müssen.

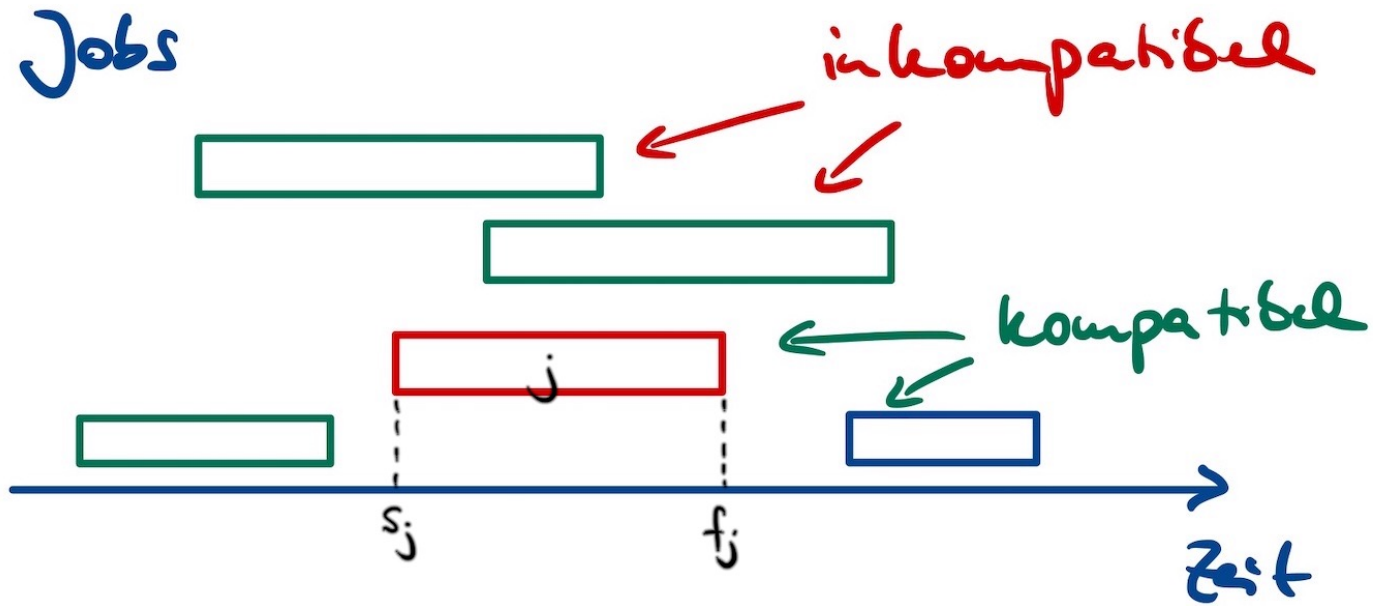
Dynamische Programmierung

- Wir betrachten eine Reihe von Beispiel-Problemen
- Dabei benutzen wir Tabularization, d.h., wir erstellen eine DP-Tabelle und berechnen alle Einträge
- Beachte: Häufig brauchen wir Tabellen mit Dimension 2 oder 3
- Schwierigkeit: Welche Teilergebnisse sollen in der Tabelle gespeichert werden?
- In der Tabelle berechnen wir zunächst Basisfälle. Die anderen Einträge werden sukzessive aus den "vorherigen" Einträgen berechnet

Gewichtetes Interval Scheduling

Wiederholung: Ungewichtetes Interval Scheduling

- Prozesse $1, \ldots, n$ haben Startzeiten s_i und Endzeiten (deadlines) f_i für $1 \leq i \leq n$.
- Prozess i und j sind kompatibel, wenn $[s_i, f_i) \cap [s_j, f_j) = \emptyset$.



- **Ziel**: Finde eine Teilmenge maximaler Größe (Kardinalität) von kompatiblen Prozessen.

Wiederholung: Ungewichtetes Interval Scheduling

Satz

Der Greedy Algorithmus mit Auswahlstrategie „früheste Beendigungszeit“ liefert einen optimalen Plan für Interval Scheduling.

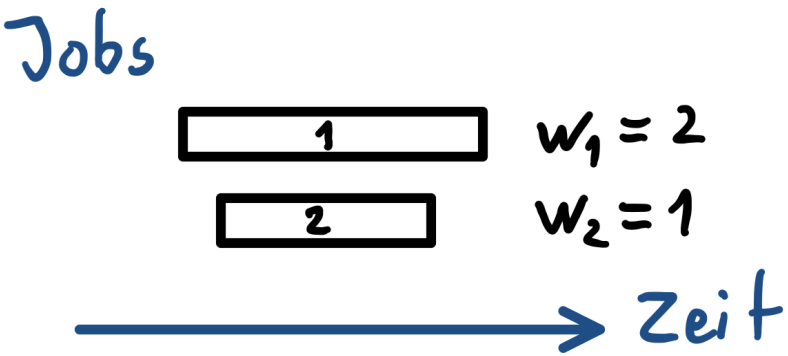
```
1 Sortiere Jobs, so dass f_1 ≤ f_2 ≤ ... ≤ f_n
2 A := ∅
3 t := 0
4 for i := 1 to n do
5     if t ≤ s_i then
6         A := A ∪ {i}
7         t := f_i
8 return A
```

Gewichtetes Interval Scheduling

Gewichtetes Interval Scheduling:

- Prozesse $1, \ldots, n$ haben Startzeiten s_i, Endzeiten f_i, und Gewichte (Prioritäten) w_i für $1 \leq i \leq n$.
- Prozess i und j sind kompatibel, wenn $[s_i, f_i) \cap [s_j, f_j) = \emptyset$.
- **Ziel**: Finde eine Teilmenge S von kompatiblen Prozessen, maximiere Zielfunktion $w(S) = \sum_{i \in S} w_i$.

„Früheste Beendigungszeit“ ist nicht mehr optimal:



Dynamisches Programm

Definition DP-Tabelle:

- Die Prozesse $1, \ldots, n$ seien nach aufsteigenden Beendigungszeiten sortiert: $f_1 \leq f_2 \leq \ldots \leq f_n$.
- DP-Tabelle M (Teilprobleme):
  $M[i] :=$ Opt. Zielfunktionswert nur mit Prozessen aus $\{1, \ldots, i\}$
- Basisfälle: $M[0] = 0$ und $M[1] = w_1$,
- Wie berechnen wir $M[i]$, $i > 1$, aus den Einträgen $M[j]$, $j < i$?

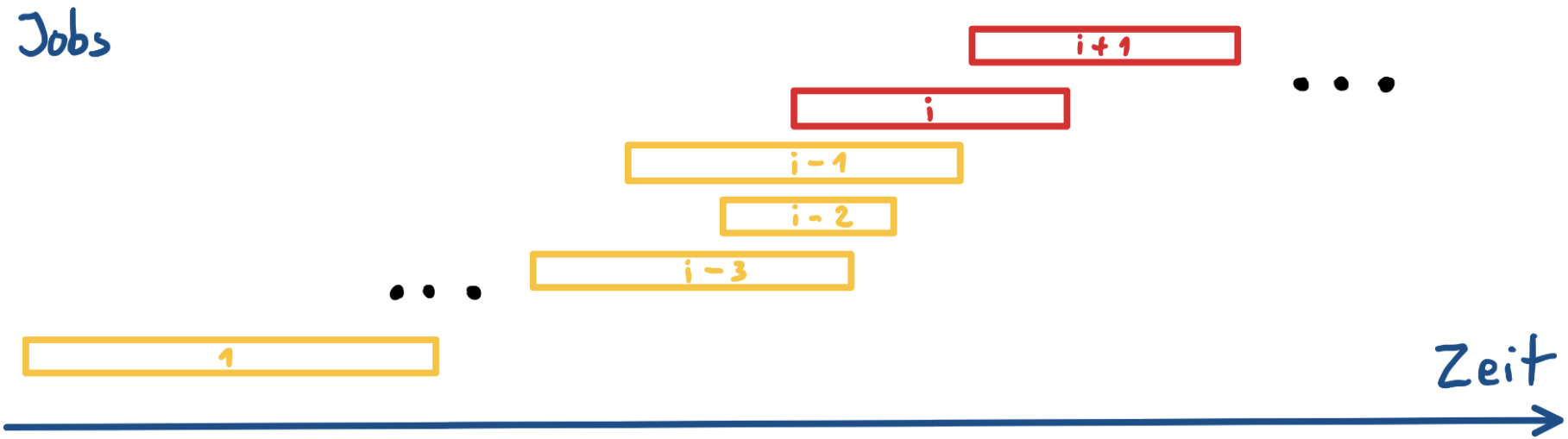
Dynamisches Programm

Wie berechnen wir $M[i]$, $i > 1$, aus den Einträgen $M[j]$, $j < i$?

Entweder die opt. Lösung für $\{1, \ldots, i\}$ enthält i oder nicht:

- 1. Fall Die Lösung enthält Prozess i nicht:

$$M[i] = M[i-1]$$



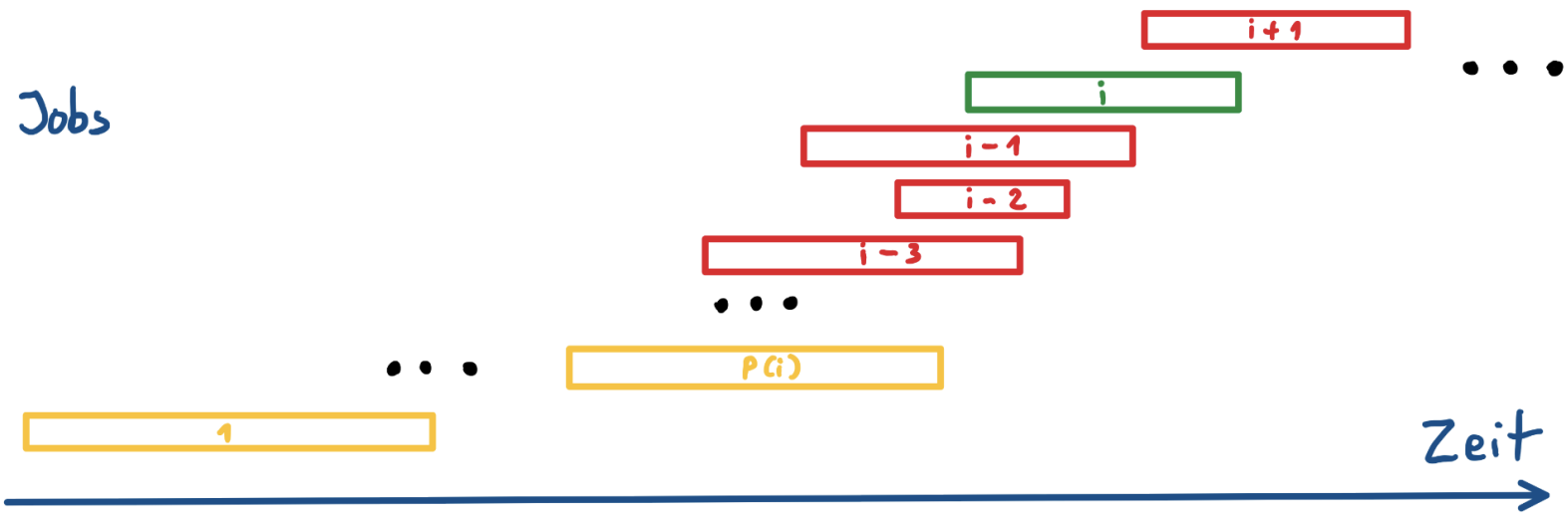
Dynamisches Programm

Wie berechnen wir $M[i]$, $i > 1$, aus den Einträgen $M[j]$, $j < i$?

Entweder die opt. Lösung für $\{1, \ldots, i\}$ enthält i oder nicht:

- 2. Fall Die Lösung enthält Prozess i:
  Sei $p(i) \in \{1, \ldots, i-1\}$ maximal mit $[s_i, f_i) \cap [s_{p(i)}, f_{p(i)}) = \emptyset$.

$$M[i] = w_i + M[p(i)]$$



Dynamisches Programm

Wie berechnen wir $M[i]$, $i > 1$, aus den Einträgen $M[j]$, $j < i$?

Entweder die opt. Lösung für $\{1, \ldots, i\}$ enthält i oder nicht:

- 1. Fall Die Lösung enthält Prozess i nicht:
  $$M[i] = M[i-1]$$
- 2. Fall Die Lösung enthält Prozess i:
  Sei $p(i) \in \{1, \ldots, i-1\}$ maximal mit $[s_i, f_i) \cap [s_{p(i)}, f_{p(i)}) = \emptyset$.
  Dann: $M[i] = w_i + M[p(i)]$
- Welcher Fall? Max Wert!
  $$M[i] = \max\{M[i-1], w_i + M[p(i)]\}$$

Dynamisches Programm

Definition DP-Tabelle:

- Die Prozesse $1, \ldots, n$ seien nach aufsteigenden Beendigungszeiten sortiert: $f_1 \leq f_2, \leq \ldots \leq f_n$.
- DP-Tabelle M (Teilprobleme):
  $M[i] :=$ Opt. Zielfunktionswert nur mit Prozessen aus $\{1, \ldots, i\}$
- Basisfälle: $M[0] = 0$ und $M[1] = w_1$
- Wie berechnen wir $M[i]$, $i > 1$, aus den Einträgen $M[j]$, $j < i$?
  $$M[i] = \max\{M[i-1], w_i + M[p(i)]\}$$
- $M[n]$ enthält am Ende den optimalen Zielfunktionswert.

Algorithmus

Algorithmus für Gewichtetes Interval Scheduling

Input : n Prozesse mit Startzeit s_i, Beendigungszeit f_i und Gewicht w_i

Output: Gewicht einer optimalen Lösung

1 Sortiere Prozesse aufsteigend nach f_*
2 Initialisiere Array $M[\cdot]$ der Länge $n+1$
3 Setze $M[0] := 0$ und $M[1] := w_1$
4 **for** $i = 2, \ldots, n$ **do**
5 $\quad M[i] := \max\left(M[i-1], w_i + M[p(i)]\right)$
6 **return** $M[n]$

Laufzeit: $\mathcal{O}(n \log(n))$

- $p(i)$ kann mit binärer Suche in Zeit $\mathcal{O}(\log n)$ gefunden werden

Berechnung der eigentlichen Lösung

- Nach Ausführung des DP enthält $M[n]$ den optimalen Zielfunktionswert.
- Wie kann aus der DP-Tabelle M die zugehörige Lösung $S \subseteq \{1, \ldots, n\}$ mit $w(S) = \sum_{i \in S} w_i = M[n]$ berechnet werden?

Rekonstruktion der Lösung aus M via Backtracking

```
1 i := n
2 S := ∅
3 while i ≥ 1 do
4     if M[i] ≠ M[i − 1] then
5         S := S ∪ {i}
6         i := p(i)
7     else
8         i := i − 1
```

Gesamter Algorithmus

Algorithmus für Gewichtetes Interval Scheduling

Input : n Prozesse mit Startzeit s_i, Beendigungszeit f_i und Gewicht w_i
Output : Optimalen Lösung

Sortiere Prozesse aufsteigend nach f_*
Initialisiere Array $M[\cdot]$ der Länge $n+1$ und setze $M[0] := 0$ und $M[1] := w_1$
for $i = 2, \ldots, n$ **do**
    $M[i] := \max\left(M[i-1], w_i + M[p(i)]\right)$
$i := n$
$S := \emptyset$
while $i \geq 1$ **do**
    if $M[i] \neq M[i-1]$ **then**
        $S := S \cup \{i\}$ und $i := p(i)$
    else
        $i := i - 1$
return S

Zusammenfassung

Designprinzip Dynamische Programmierung (DP)

- Lösung durch (rekursive) Aufteilung in kleinere Teilprobleme.
- Speichere Teillösungen um mehrfache Berechung zu vermeiden.
- Herzstück eines DP: DP-Tabelle.
- Berechnung der DP-Tabelle via Tabularization oder Memoization.
- Lösungen via Backtracking aus DP-Tabelle rekonstruieren.

Dynamische Programme für:

- Fibonacci-Folge,
- gewichtetes Interval Scheduling