

Algorithmen und Datenstrukturen

Wintersemester 2019/20

23. Vorlesung

Greedy- und Approximationsalgorithmen

Operations Research

Optimierung für Wirtschaftsabläufe:

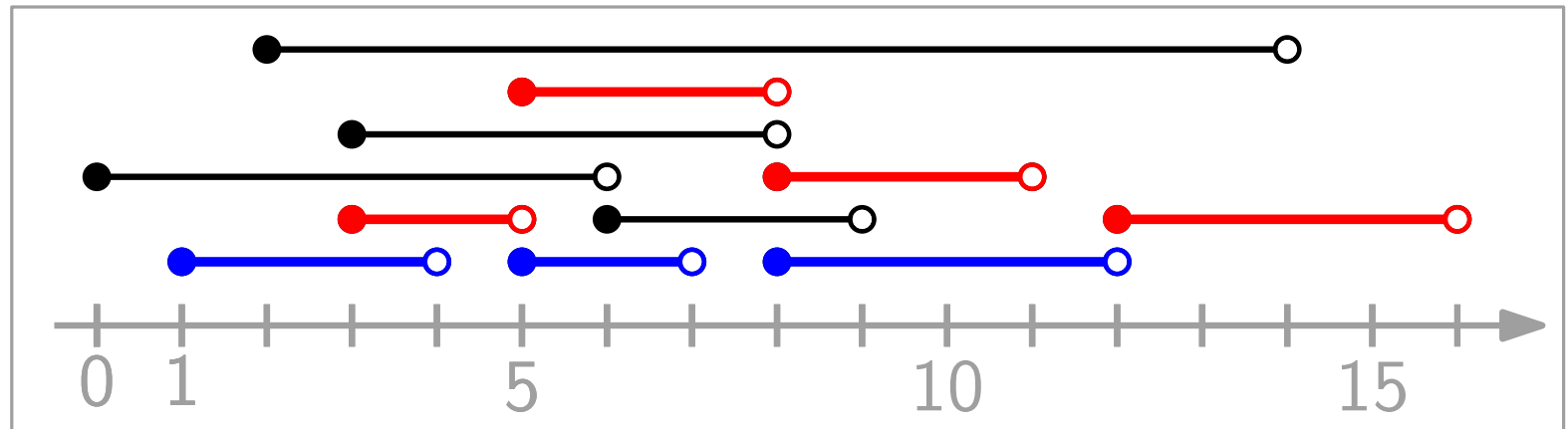
- Standortplanung
- Ablaufplanung
- Flottenmanagement
- Pack- und Zuschnittprobleme
- ...

Werkzeuge:

Statistik, Optimierung, Wahrscheinlichkeitstheorie, Spieltheorie, Graphentheorie, mathematische Programmierung, Simulation...

Ein einfaches Problem der Ablaufplanung

Gegeben: Menge $A = \{a_1, \dots, a_n\}$ von *Aktivitäten*, wobei für $i = 1, \dots, n$ gilt $a_i = [s_i, e_i)$.



a_i und a_j sind *kompatibel*, wenn $a_i \cap a_j = \emptyset$.

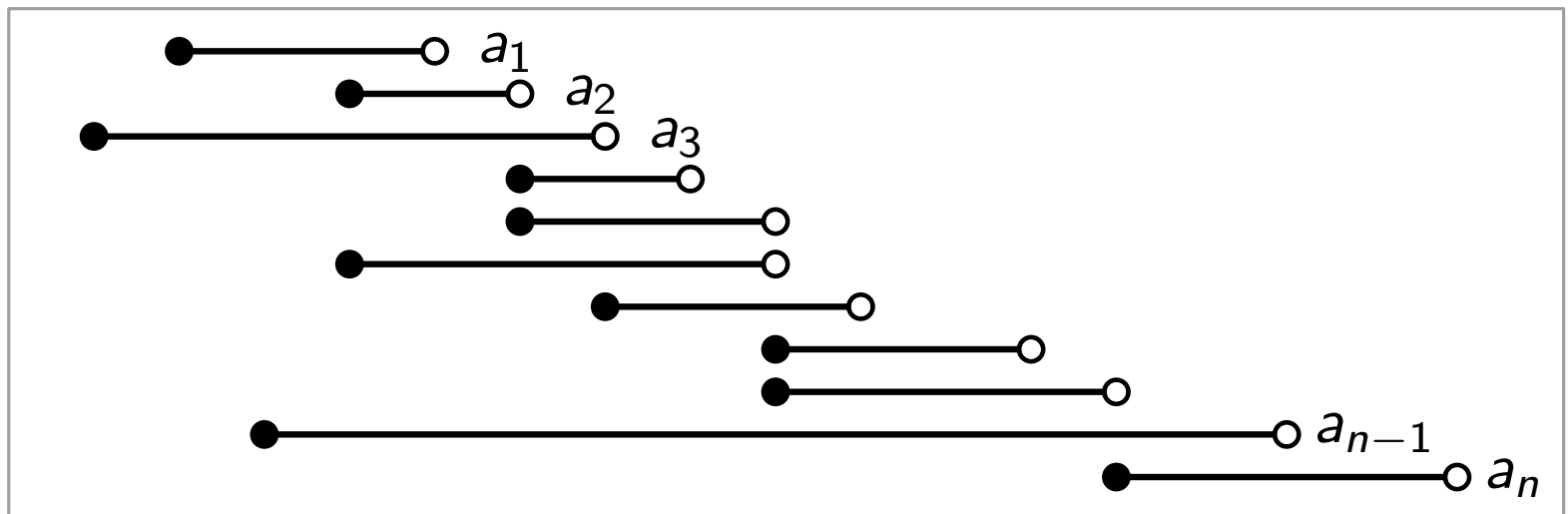
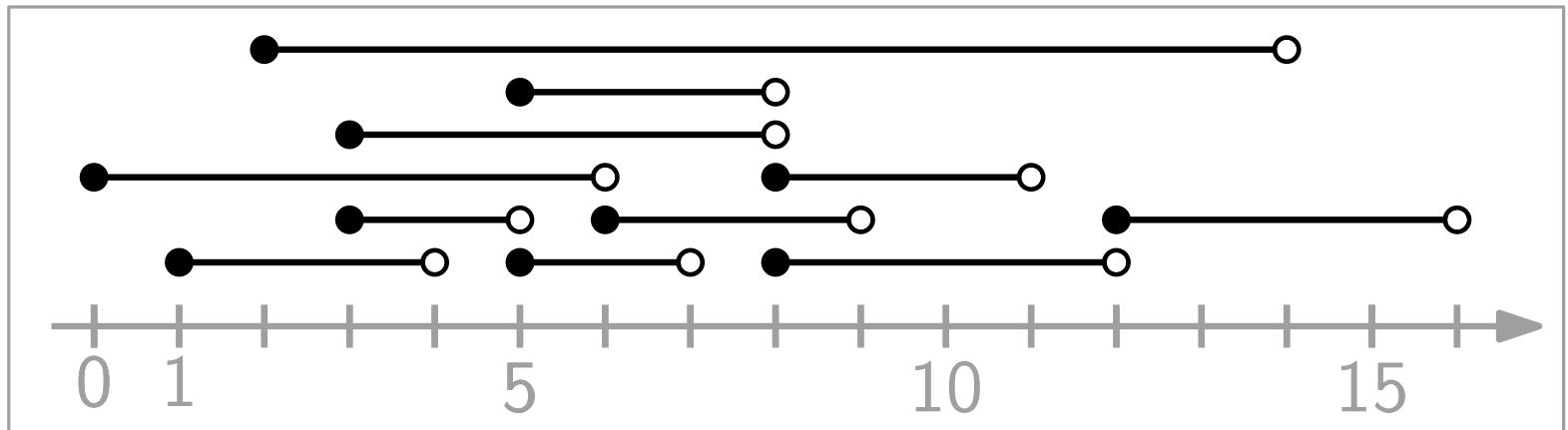
Die Aktivitäten in $A' \subset A$ sind *paarweise kompatibel*, wenn für jedes Paar $a_i, a_j \in A'$ gilt, dass a_i und a_j kompatibel sind.

Gesucht: eine größtmögliche Menge paarweise kompatibler Aktivitäten.

Grund: Aktivitäten (à 1€), die gleiche Ressource benutzen

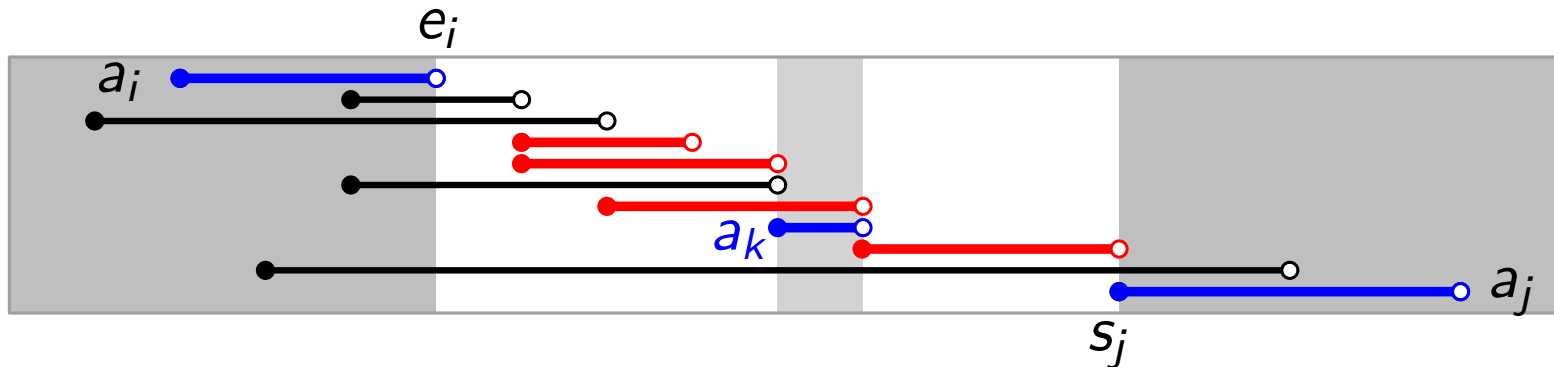
Ein kleiner technischer Trick

Wir nummerieren (für den Rest der Vorlesung) die Aktivitäten so, dass für die Endtermine gilt $e_1 \leq e_2 \leq \dots \leq e_n$.



Dynamisches Programmieren?

1. Struktur einer optimalen Lösung charakterisieren



Sei $A_{ij} \subset A$ die Menge aller Aktivitäten in $[e_i, s_j)$, d.h. „zwischen“ a_i und a_j .

Ang. a_i und a_j sind in einer opt. Lösung $L \subseteq A$ enthalten, dann ist $L \cap A_{ij}$ eine opt. Lösung für A_{ij} .

⇒ optimale Substruktur!

Beweis?

Austauschargument!

2. Wert einer optimalen Lösung rekursiv definieren

Sei c_{ij} die Kardinalität einer opt. Lösung für A_{ij} .

Dann gilt: $c_{ij} = \max_{a_k \in A_{ij}} c_{ik} + 1 + c_{kj}$

Dynamisches Programmieren?

3. Wert einer optimalen Lösung berechnen

Setze $e_0 = -\infty$ und $s_{n+1} = +\infty$. Dann ist $A = A_{0,n+1}$.

Berechne $c_{0,n+1}$, die Kardinalität einer opt. Lösung für A .

(a) *top-down*

TopDownDP(int[]s, int[]e, int i, int j)

→ liefert c_{ij}

(b) *bottom-up*

BottomUpDP(int[]s, int[]e)

→ liefert $c_{0,n+1}$

Siehe Folie

„Zurück zum dynamischen Programmieren“

Laufzeit? $O(n^3)$...

2. Wert einer optimalen Lösung rekursiv definieren

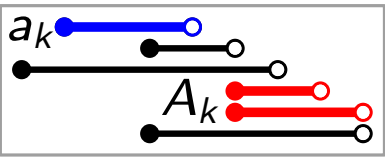
Sei c_{ij} die Kardinalität einer opt. Lösung für A_{ij} .

Dann gilt: $c_{ij} = \max_{a_k \in A_{ij}} c_{ik} + 1 + c_{kj}$

Darf's auch etwas einfacher sein?

Idee: Sei L opt. Lösung für A . – Welche Aktivität hat gute Chancen die erste („linkeste“) in L zu sein?

Intuition: Die Aktivität a_1 mit frühester Endzeit – weil a_1 die gemeinsame Ressource am wenigsten einschränkt.



Sei $A_k = \{a_i \in A : s_i \geq e_k\}$ die Menge der Aktivitäten, die nach Ablauf von a_k beginnen.

Sei L_k eine optimale Lösung von A_k .

Falls Intuition korrekt, dann ist $\{a_1\} \cup L_1$ optimal.

Satz. Sei $A_k \neq \emptyset$.

optimale Teilstruktur! Sei a_m Aktivität mit frühester Endzeit in A_k .

\Rightarrow es gibt eine opt. Lösung von A_k , die a_m enthält.

Beweis. *Austauschargument!*

Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
```

```
  e0 = -∞ // ⇒ A0 = A
```

```
  // Aktivitäten nach Endzeiten sortieren, falls nötig
```

```
  return GreedyRecursiveMain(s, e, 0)
```

```
GreedyRecursiveMain(int[] s, int[] e, int k) // best. Lsg. für Ak
```

```
  m = k + 1; n = s.length
```

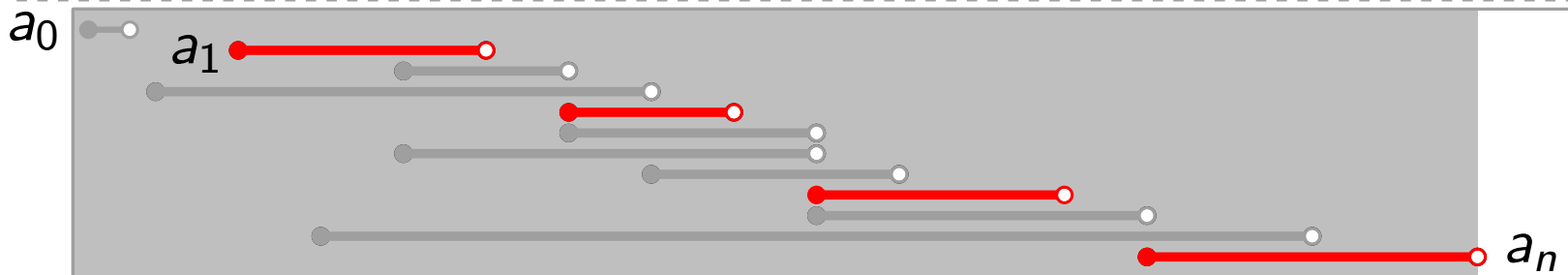
```
  // Finde Aktivität am mit kleinster Endzeit in Ak
```

```
  while m ≤ n and s[m] < e[k] do
```

```
    ⊥ m = m + 1
```

```
  if m > n then return ∅
```

```
  else return {am} ∪ GreedyRecursiveMain(s, e, m)
```



Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
```

```
  e0 = -∞ // ⇒ A0 = A
```

```
  // Aktivitäten nach Endzeiten sortieren, falls nötig
```

```
  return GreedyRecursiveMain(s, e, 0)
```

```
GreedyRecursiveMain(int[] s, int[] e, int k) // best. Lsg. für Ak
```

```
  m = k + 1; n = s.length
```

```
  // Finde Aktivität am mit kleinster Endzeit in Ak
```

```
  while m ≤ n and s[m] < e[k] do
```

```
    ⊥ m = m + 1
```

```
  if m > n then return ∅
```

```
  else return {am} ∪ GreedyRecursiveMain(s, e, m)
```

Laufzeit?

Wie oft wird m **inkrementiert**?

Insgesamt, über alle rekursiven Aufrufe, n Mal.

D.h. GreedyRecursive läuft (ohne Sortieren) in $\Theta(n)$ Zeit.

Greedy – iterativ

```

GreedyIterative(int[] s, int[] e)
  n = s.length
  if n = 0 then return ∅
  L = {a1}
  k = 1 // höchster Index in L
  for m = 2 to n do
    if s[m] ≥ e[k] then
      L = L ∪ {am}
      k = m
  return L

```

Laufzeit? GreedyIterative läuft ebenfalls in $\Theta(n)$ Zeit.

Bemerkung: GreedyIterative berechnet dieselbe optimale Lösung wie GreedyRecursive – die „linkeste“.

Die Greedy-Strategie

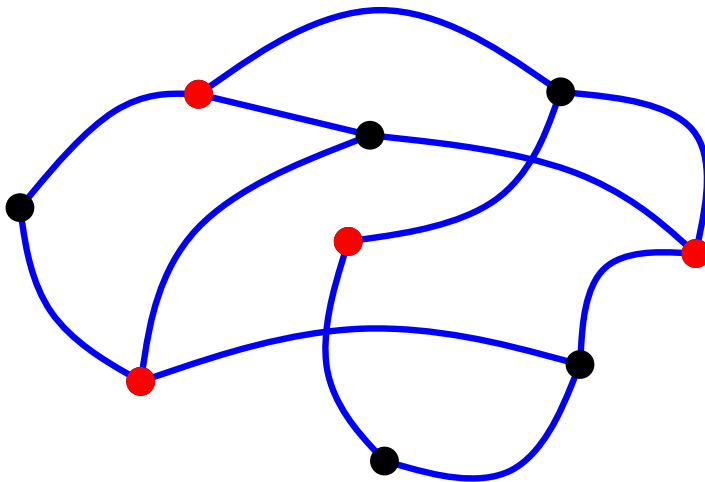
1. Teste, ob das Problem optimale Teilstruktur aufweist.
2. Entwickle eine rekursive Lösung
3. Zeige, dass bei einer Greedy-Entscheidung nur *ein* Teilproblem bleibt
4. Beweise, dass die Greedy-Wahl „sicher“ ist (vgl. Kruskal!)
5. Entwickle einen rekursiven Greedy-Algorithmus
6. Konvertiere den rekursiven in einen iterativen Algorithmus

Food for Thought

1. Welches allgemeinere Ablaufproblem könnte das DP lösen – aber nicht der GA?

Zur Erinnerung: Das DP berechnet $c_{ij} = \max_{a_k \in A_{ij}} c_{ik} + 1 + c_{kj}$.

2. Problem *größte unabhängige Menge (guM)* in Graphen:



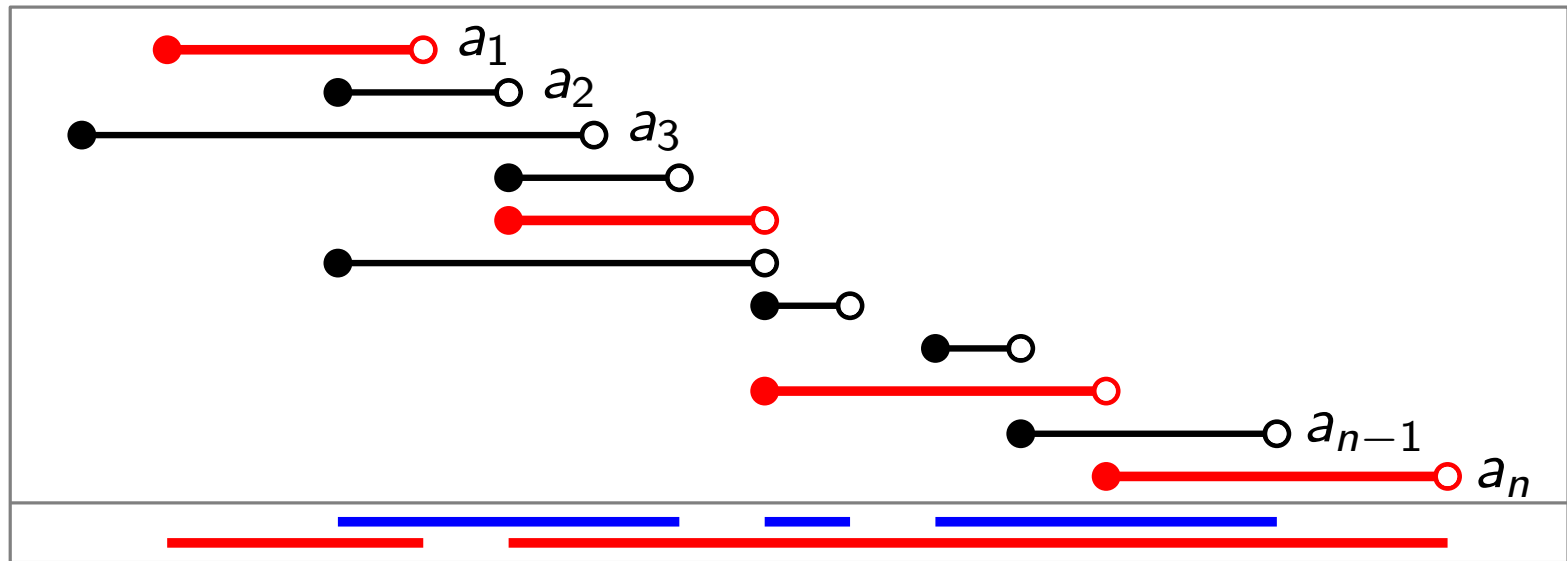
Finde eine größte Teilmenge U der Knoten, so dass keine zwei Knoten in U benachbart sind.

- Was hat guM mit unserem Ablaufplanungsproblem zu tun?
- Kann man guM mit DP oder GA lösen?

Ein ähnliches Problem der Ablaufplanung

Gegeben: Menge $A = \{a_1, \dots, a_n\}$ von halboffenen Intervallen, mit $a_i = [s_i, e_i)$ für $i = 1, \dots, n$.

Für die Endpunkte gelte $e_1 \leq e_2 \leq \dots \leq e_n$.



Gesucht: eine Menge $A' \subseteq A$ paarweise disjunkter Intervalle, deren **Gesamtlänge $\ell(A')$ maximal** ist.

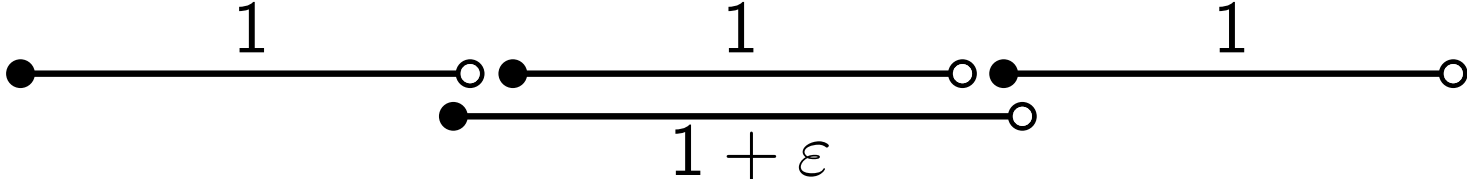
Grund: Intervalle $\hat{=}$ Prozesse, die die gleiche Ressource nutzen; der Gesamtertrag ist proportional zur Auslastung.

Greedy?

1. Versuch: *Nimm Aktivität mit frühestem Endtermin, streiche dazu inkompatible Aktivitäten und iteriere.*

Gegenbsp.: 

2. Versuch: *Nimm längste Aktivität, streiche dazu inkompatible Aktivitäten und iteriere.*

Gegenbsp.: 

Aufgabe: Können Sie den 2. GA in $O(n \log n)$ Zeit implementieren?

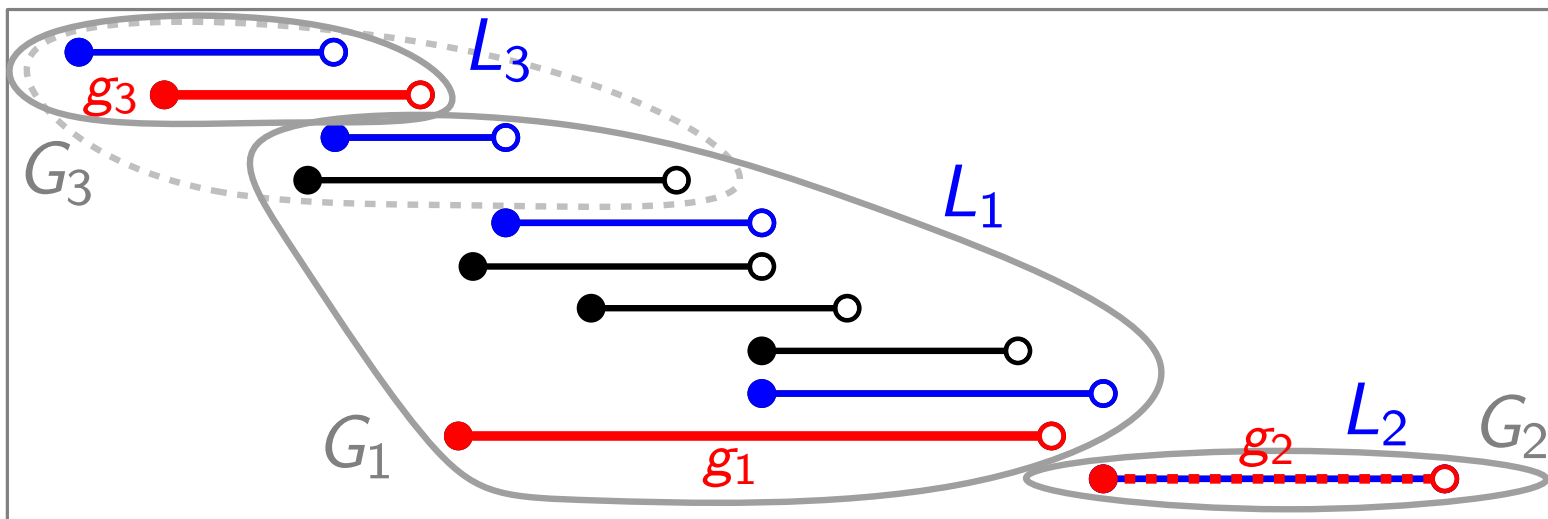
Diskutieren Sie mit Ihrer NachbarIn!

Wie gut/schlecht ist der 2. GA?

Betrachte eine optimale Lösung $L \subseteq A$.

Sei $G = \{g_1, g_2, \dots, g_k\} \subseteq A$ die Greedy-Lösung (in dieser Rf.).

Für $i = 1, \dots, k$ sei $G_i = \{a \in A \mid a \cap g_i \neq \emptyset\} \setminus (G_1 \cup \dots \cup G_{i-1})$



und sei
 $L_i = L \cap G_i$.

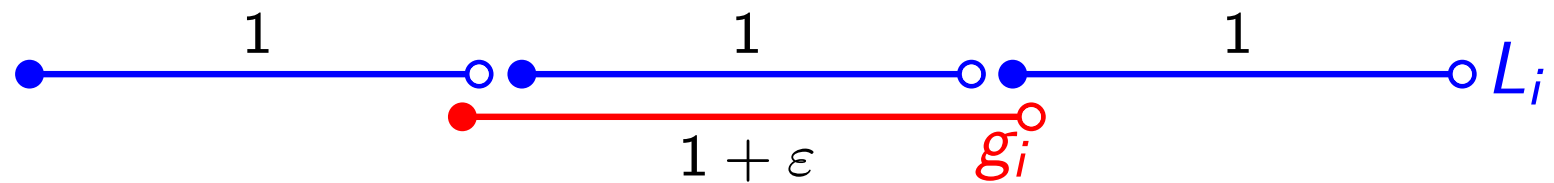
Dann gilt $A = G_1 \dot{\cup} G_2 \dot{\cup} \dots \dot{\cup} G_k$ und $L = L_1 \dot{\cup} L_2 \dot{\cup} \dots \dot{\cup} L_k$.

„ \subseteq “: GA wählt so lange Intervalle aus, bis es keine mehr gibt.

„ \supseteq “: klar, da $G_1 \subseteq A$, $G_2 \subseteq A$, \dots , $G_k \subseteq A$

Wie gut/schlecht ist der 2. GA?

Behauptung: Für $i = 1, \dots, k$ gilt $\ell(L_i) < 3\ell(g_i)$.



Beweis.

- (a) g_i ist nach Wahl ein längstes Intervall in G_i
- (b) jedes $a \in L_i$ schneidet g_i
- (c) Intervalle in L_i sind paarweise disjunkt

$$\Rightarrow \text{OPT} = \ell(L) = \sum_{i=1}^k \ell(L_i) < 3 \sum_{i=1}^k \ell(g_i) = 3\ell(G)$$

$$\Rightarrow \ell(G) > \text{OPT}/3$$

\Rightarrow 2. GA liefert *immer* mind. $1/3$ der maximalen Gesamtlänge.

Also ist der 2. GA ein **Faktor-(1/3)-Approximationsalgorithmus**.

Approxim. . . hä?

„All exact science is dominated by the idea of approximation.“

Bertrand Russell (1872–1970)

Sei Π ein *Maximierungsproblem*.

z.B. Ablaufplanung

Sei ζ die *Zielfunktion* von Π : Lösung $\mapsto \mathbb{Q}_{\geq 0}$.

$$\zeta = \ell$$

Sei γ eine Zahl ≤ 1 .

$$\gamma = 1/3$$

Ein Algorithmus \mathcal{A} heißt γ -*Approximation*, wenn

- \mathcal{A} für jede Instanz I von Π eine Lösung $\mathcal{A}(I)$ berechnet, so dass

1/3-Approx.
liefert Menge von
Aktivitäten, deren
Gesamtlänge
mindestens 1/3
der maximal mög-
lichen Länge ist.

$$\frac{\zeta(\mathcal{A}(I))}{\text{OPT}(I)} \geq \gamma$$

$\zeta(\text{optimale Lösung})$ \rightarrow $\text{OPT}(I)$

\rightarrow Größe der Instanz I

- die Laufzeit von \mathcal{A} polynomiell in $|I|$ ist.

$O(n \log n)$

Zurück zum dynamischen Programmieren

```
BottomUpDPWeighted(int[] s, int[] e)
```

```
n = s.length
```

```
c = new int[0..n][1..n + 1] // cij = Wert einer opt. Lsg. für Aij
```

```
for d = 1 to n - 1 do // d = „Distanz“ zwischen j und i
```

```
  for i = 0 to (n + 1) - d do
```

```
    j = i + d
```

```
    if ai und aj kompatibel then // falls ai ∩ aj = ∅
```

```
      c = 0
```

```
      for k = i + 1 to j - 1 do
```

```
        c' = c[i][k] + l(ak) + c[k][j]
```

```
        if c' > c then c = c'
```

```
      c[i][j] = c
```

```
    else c[i][j] = 0
```

```
  return c[0, n + 1]
```

NEU! Im ungewichteten Fall stand hier eine Eins.

Laufzeit? $O(n^3)$

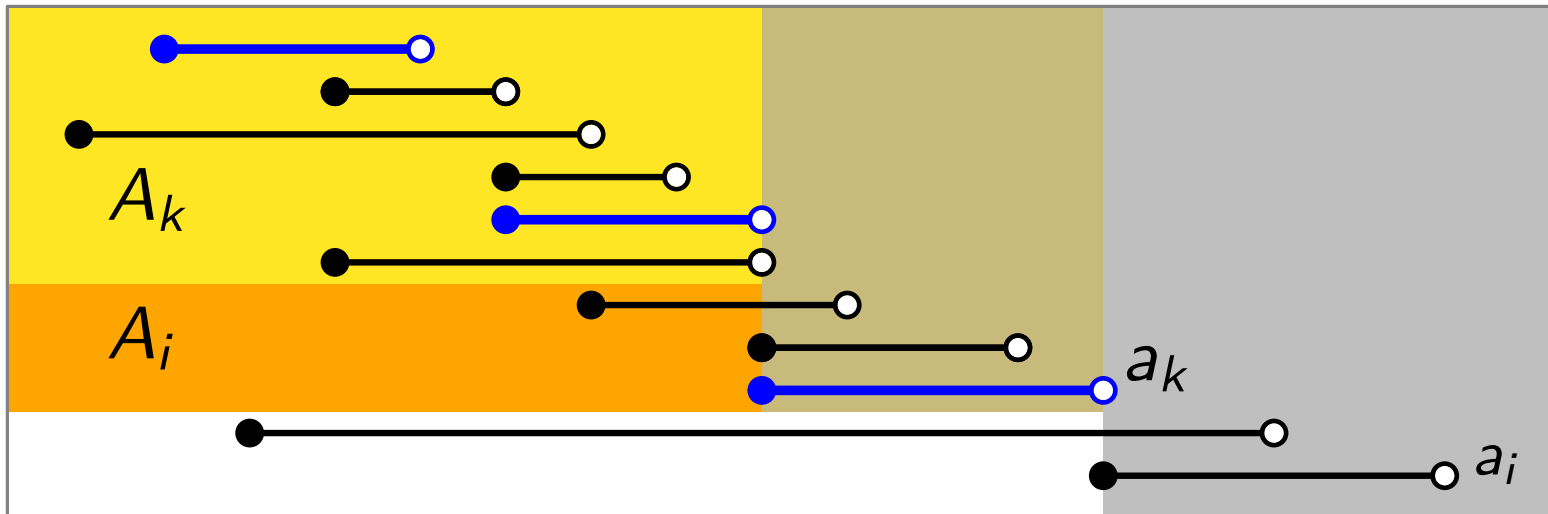
Aber warum verzweigen wir hier zweimal?

2. Wert einer optimalen Lösung rekursiv definieren

$$c_{ij} = \max_{a_k \in A_{ij}} c_{ik} + 1 + c_{kj}$$

Dynamisches Programmieren, aber einfacher

Für $i = 1, \dots, n$ sei $A_i = \{a_j \in A \mid e_j \leq s_i\}$ die Menge aller Intervalle in A , die enden, bevor a_i beginnt. (Setze $A_{n+1} = A$.)



Eine optimale Lösung für A_i besteht aus:

- *einem* letzten Intervall a_k und
- einer optimalen Lösung für A_k .

} optimale Teilstruktur!

Also gilt für den Wert c_i einer optimalen Lösung für A_i :

$$c_i = \max_{a_k \in A_i} c_k + \ell(a_k)$$

Dynamisches Programmieren, aber einfacher

Also gilt für den Wert c_i einer optimalen Lösung für A_i :

$$c_i = \max_{a_k \in A_i} c_k + \ell(a_k)$$

Erinnern wir uns...

c_{n+1} ist der Wert der optimalen Lösung für $A_{n+1} = A$.

Also genügt es c_1, \dots, c_{n+1} zu berechnen.

Laufzeit? $O(n^2)$

Work out the details!

Resultate:

- Der 2. Greedy-Alg. findet in $O(n \log n)$ Zeit eine Lösung, die *mindestens 1/3 des maximalen Ertrags* garantiert.
- Unser neues DP findet in $O(n^2)$ Zeit eine Lösung mit *maximalem Ertrag*.

Trade-Off zwischen Zeit und Qualität!