Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Lehrstuhl für
INFORMATIK I
Effiziente Algorithmen und
wissensbasierte Systeme

Institut für Informatik

Algorithmen und Datenstrukturen

Wintersemester 2019/20

23. Vorlesung

Greedy- und Approximationsalgorithmen

Prof. Dr. Alexander Wolff *Lehrstuhl für Informatik I*

Operations Research

Optimierung für Wirtschaftsabläufe:

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

Operations Research

Optimierung für Wirtschaftsabläufe:

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

Werkzeuge:

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

Operations Research

Optimierung für Wirtschaftsabläufe:

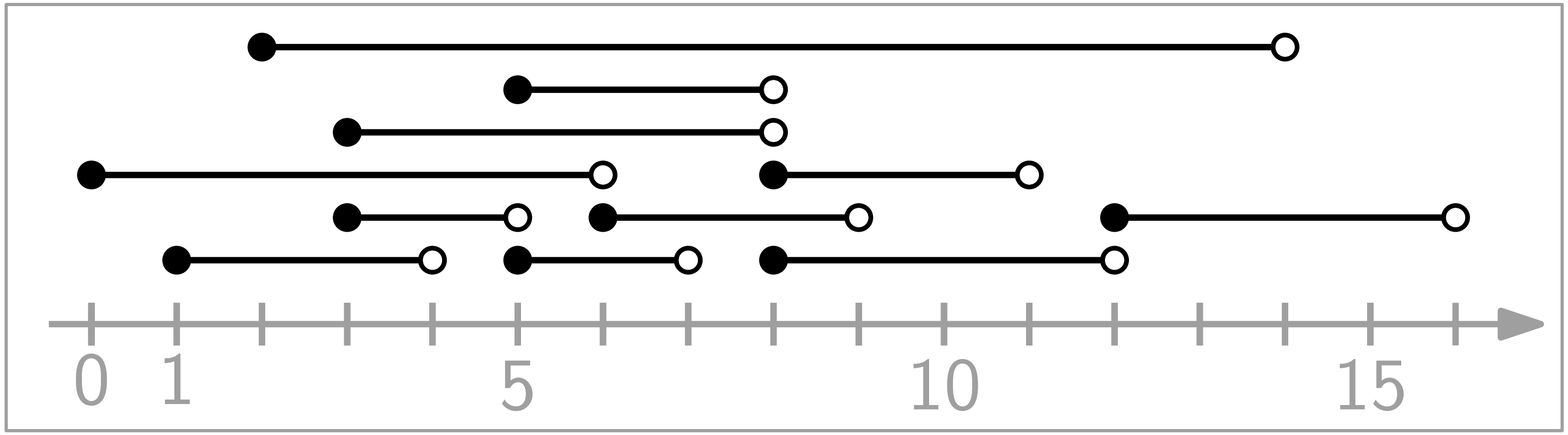
- Standortplanung
- Ablaufplanung
- Flottenmanagement
- Pack- und Zuschnittprobleme
- ...

Werkzeuge:

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

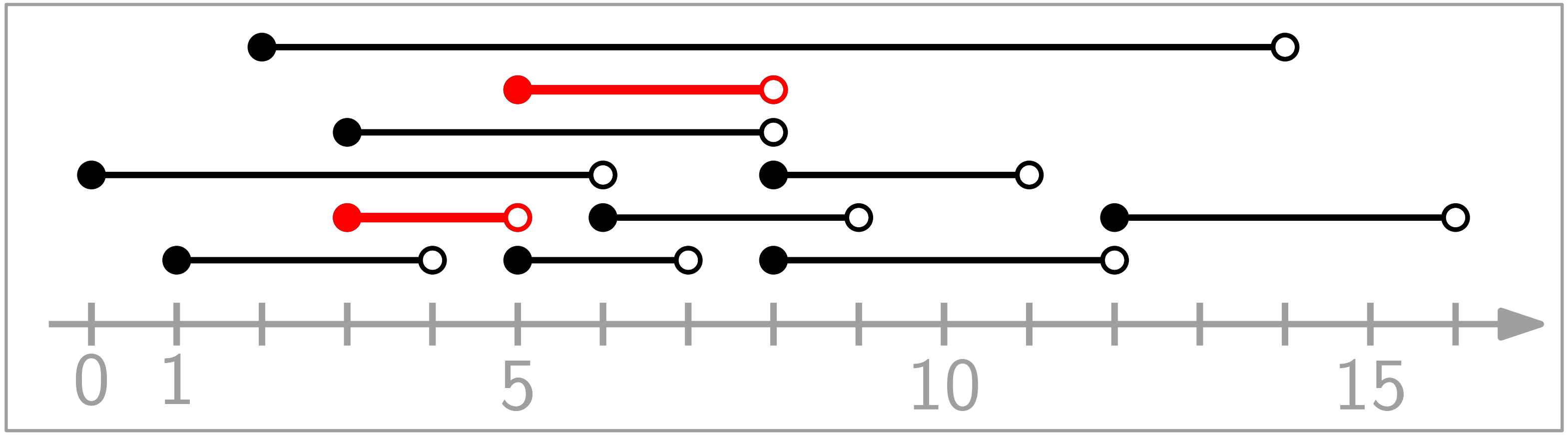
Ein einfaches Problem der Ablaufplanung

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



Ein einfaches Problem der Ablaufplanung

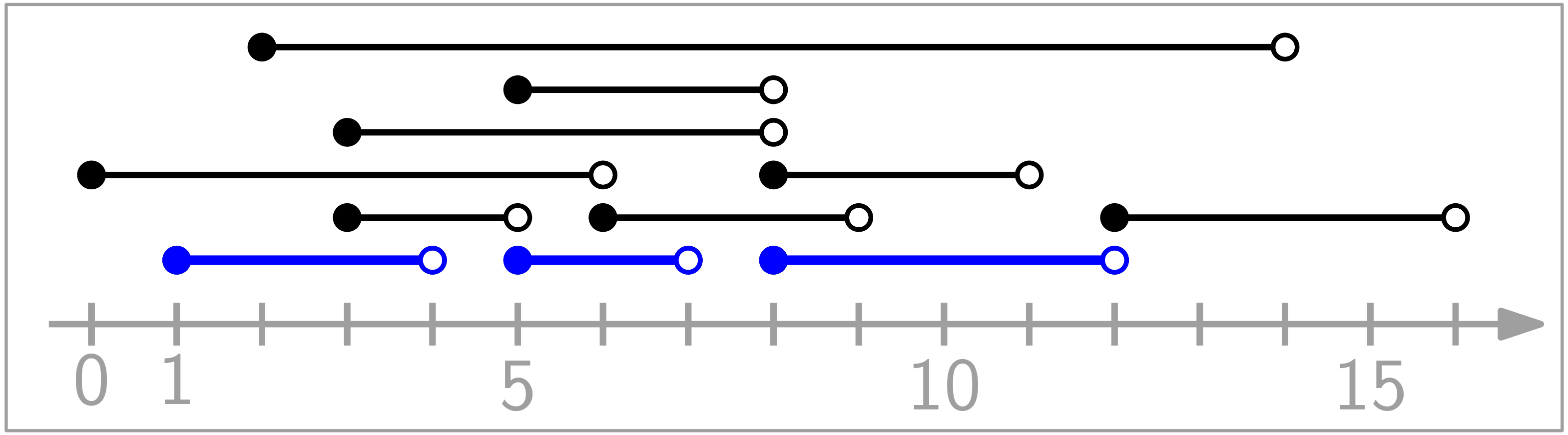
Gegeben: Menge $A = \{a_1, \ldots, a_n\}$ von *Aktivitäten*, wobei für $i = 1, \ldots, n$ gilt $a_i = [s_i, e_i)$.



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

Ein einfaches Problem der Ablaufplanung

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

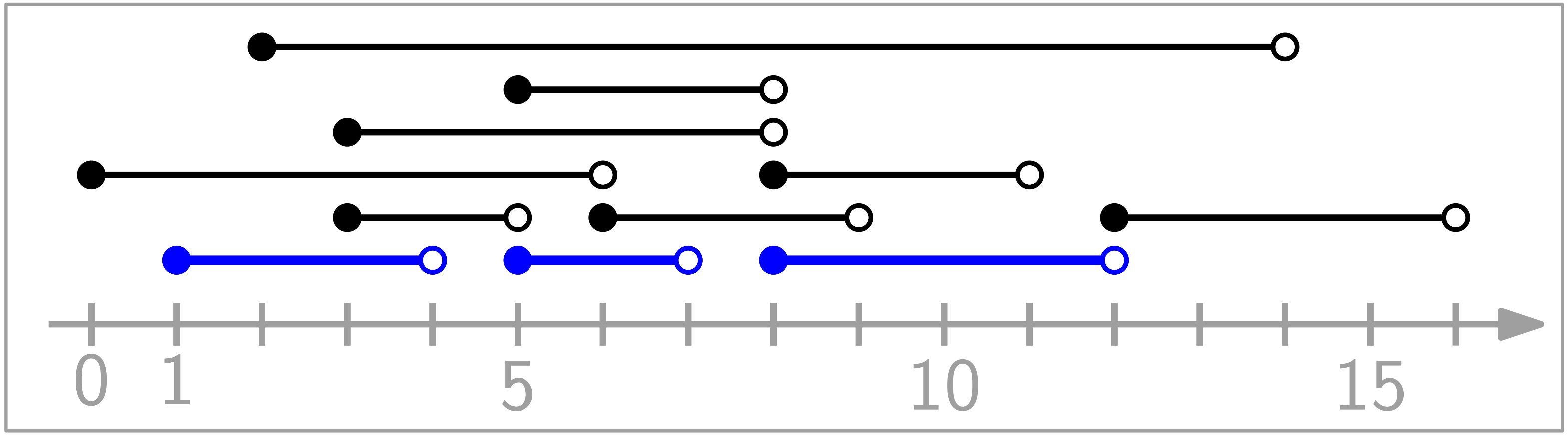


a_i und a_j sind *kompatibel*, wenn $a_i \cap a_j = \emptyset$.

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.

Ein einfaches Problem der Ablaufplanung

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



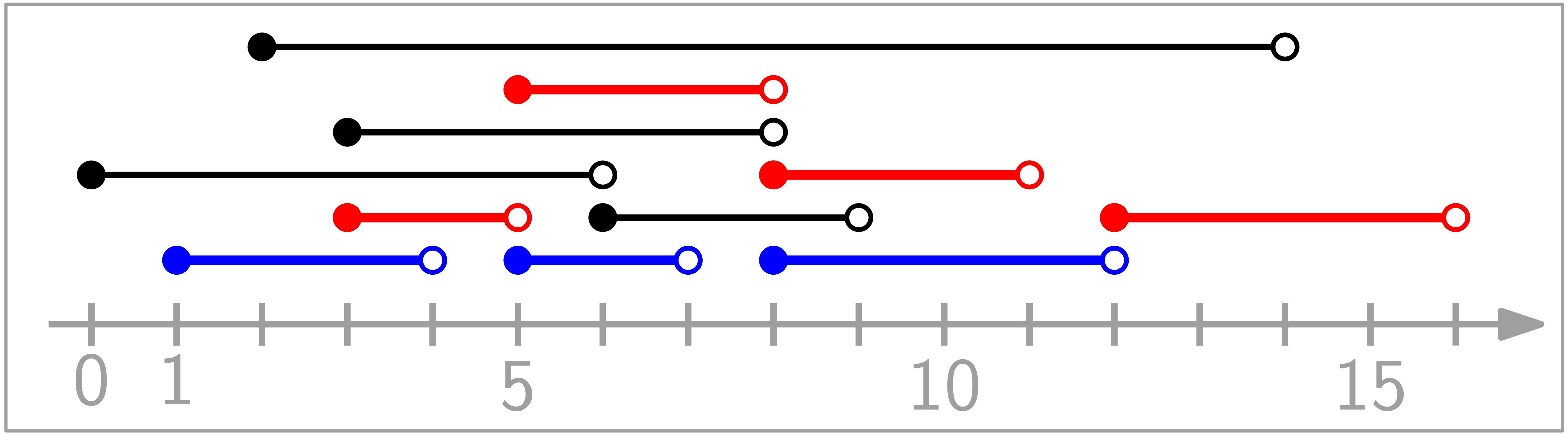
a_i und a_j sind *kompatibel*, wenn $a_i \cap a_j = \emptyset$.

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.

Ein einfaches Problem der Ablaufplanung

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



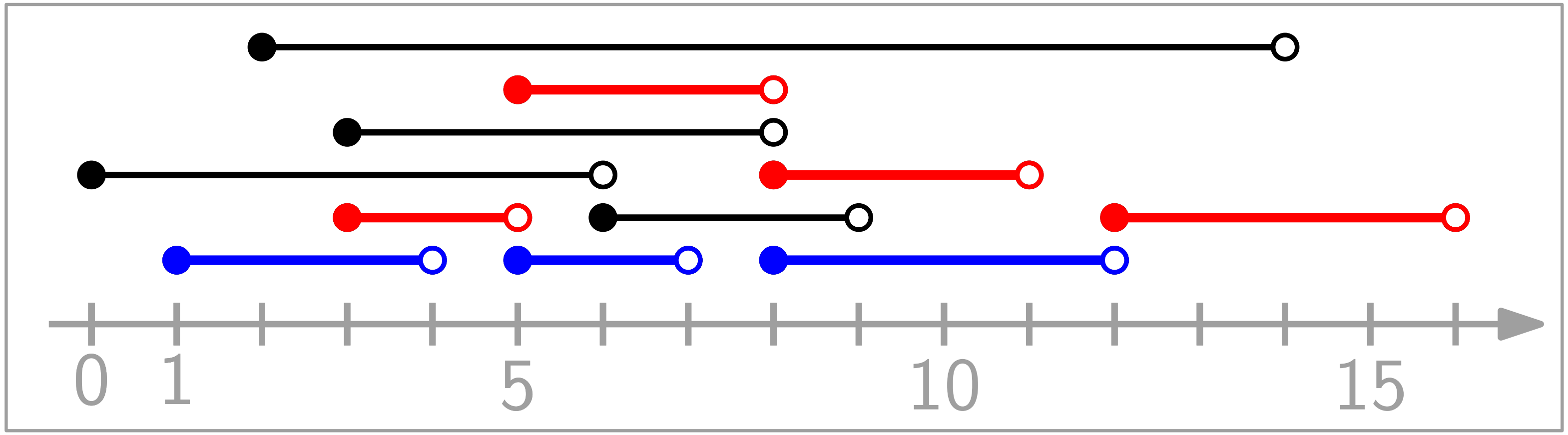
a_i und a_j sind *kompatibel*, wenn $a_i \cap a_j = \emptyset$.

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.

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 $a_i \cap a_j = \emptyset$.

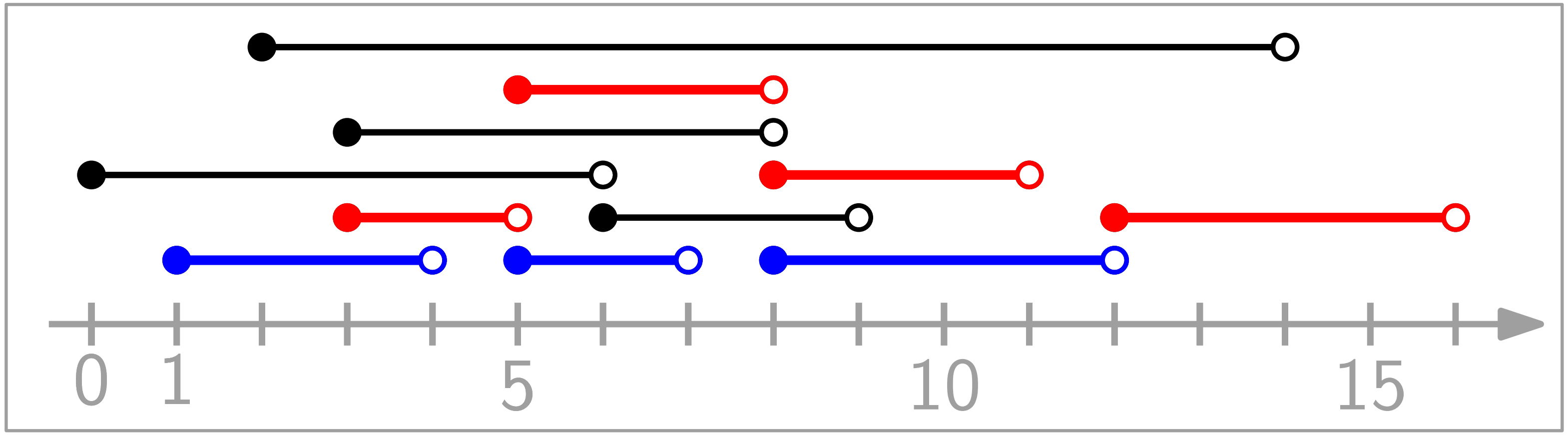
Die Aktivitäten in $A' \subset A$ sind *paarweise kompatibel*, wenn für jedes Paar $a_i, a_j \in A'$ gilt, dass a_i und a_j kompatibel sind.

Gesucht: eine größtmögliche Menge paarweise kompatibler Aktivitäten.

Grund:

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 $a_i \cap a_j = \emptyset$.

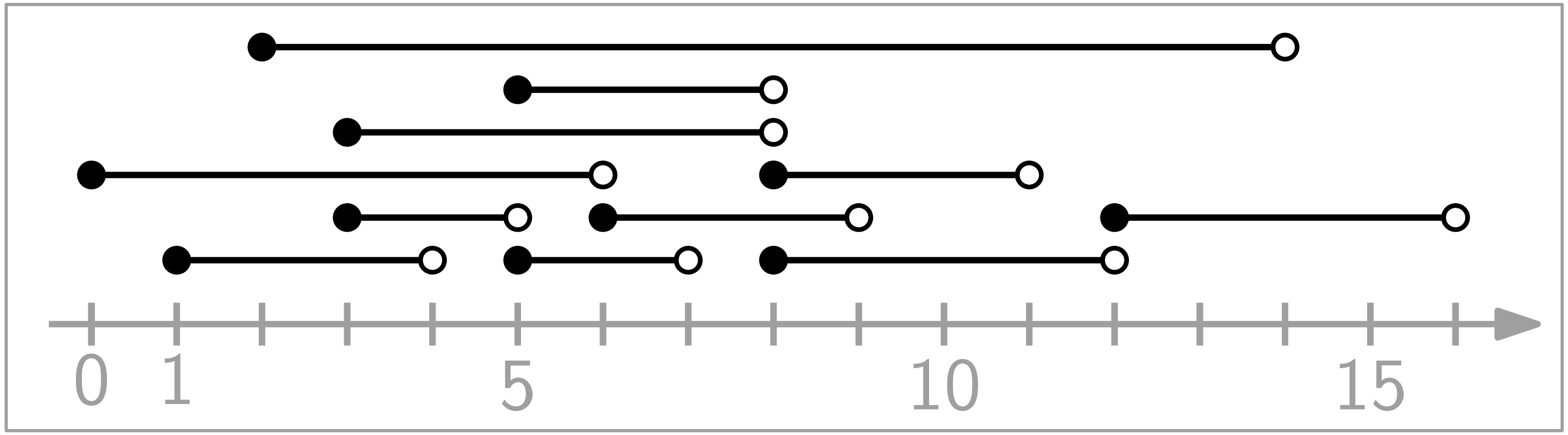
Die Aktivitäten in $A' \subset A$ sind *paarweise kompatibel*, wenn für jedes Paar $a_i, a_j \in A'$ gilt, dass a_i und a_j kompatibel sind.

Gesucht: eine größtmögliche Menge paarweise kompatibler Aktivitäten.

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

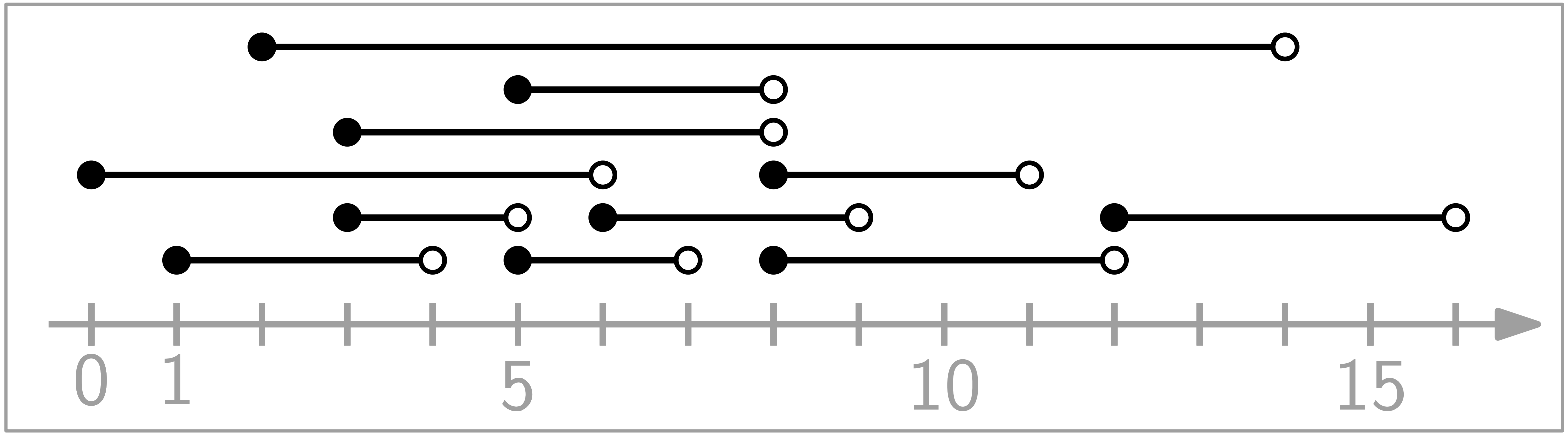
Ein kleiner technischer Trick

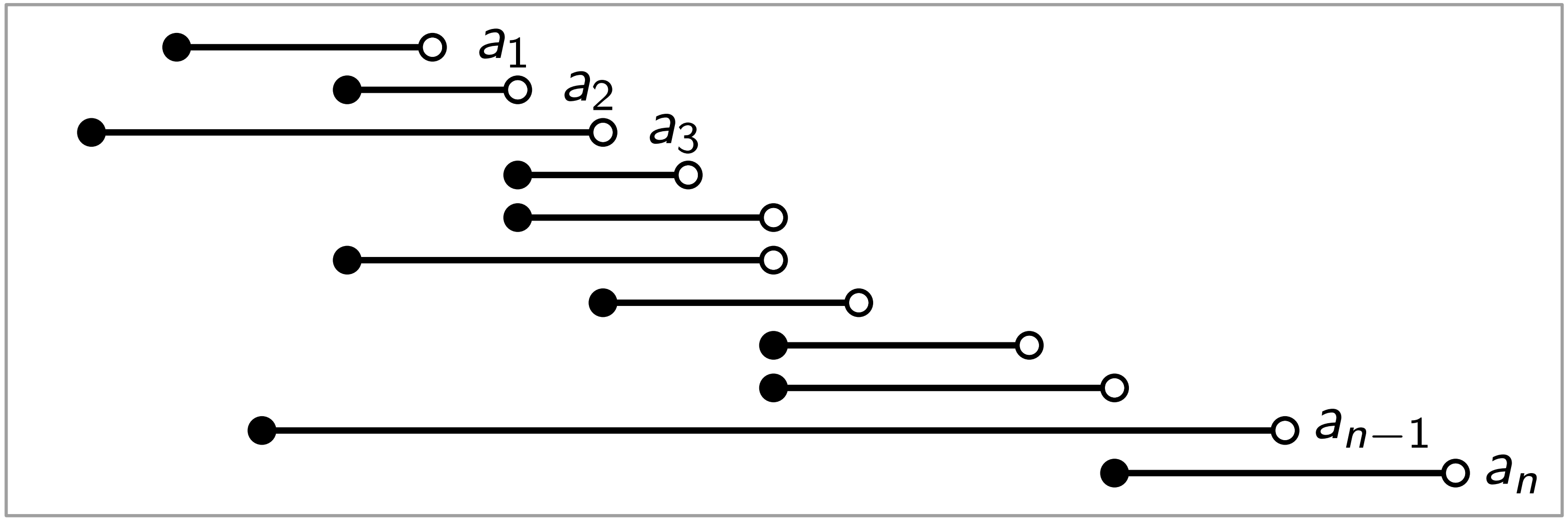
Wir nummerieren (für den Rest der Vorlesung) die Aktivitäten so, dass für die Endtermine gilt $e_1 \leq e_2 \leq \cdots \leq e_n$.



Ein kleiner technischer Trick

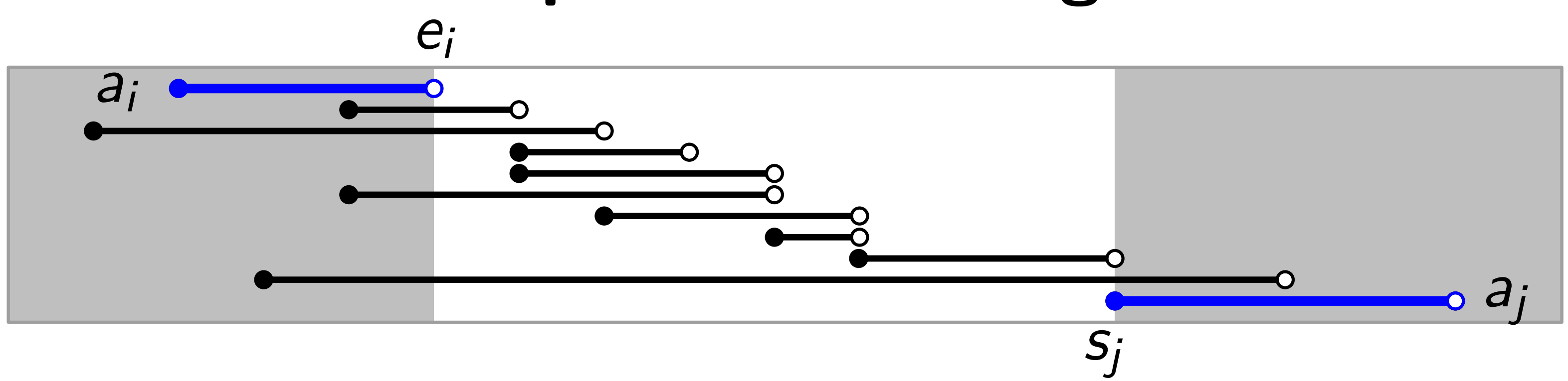
Wir nummerieren (für den Rest der Vorlesung) die Aktivitäten so, dass für die Endtermine gilt $e_1 \leq e_2 \leq \cdots \leq e_n$.





Dynamisches Programmieren?

1. Struktur einer optimalen Lösung charakterisieren

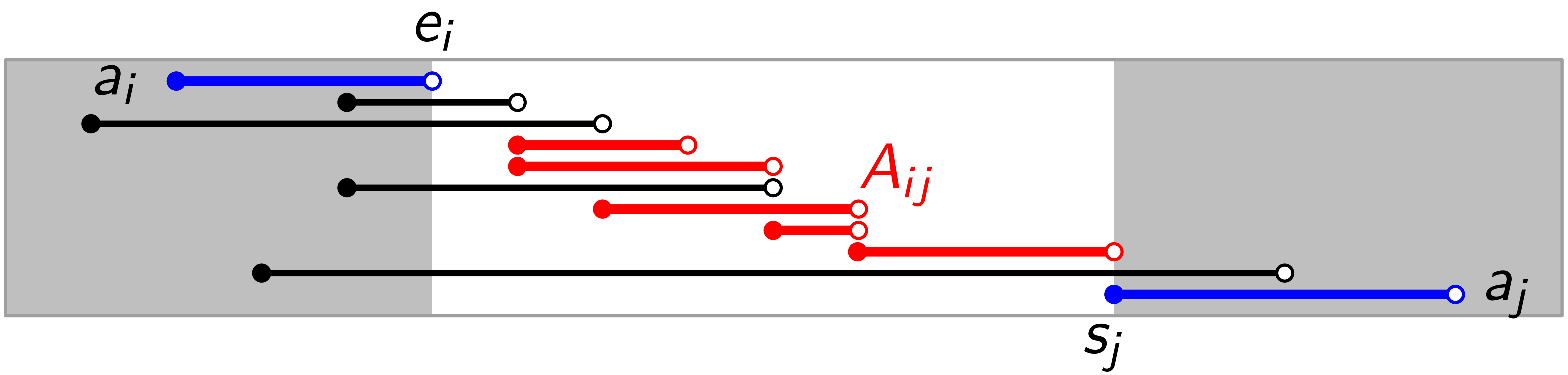


Sei $A_{ij} \subset A$ die Menge aller Aktivitäten in $[e_i, s_j)$, d.h. „zwischen“ a_i und a_j.

2. Wert einer optimalen Lösung rekursiv definieren

Dynamisches Programmieren?

1. Struktur einer optimalen Lösung charakterisieren

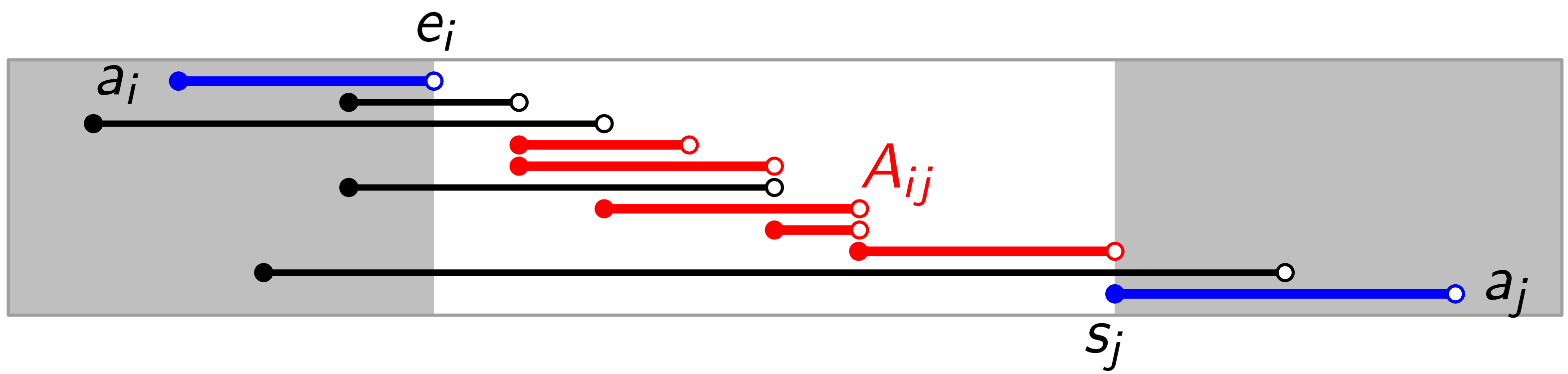


Sei $A_{ij} \subset A$ die Menge aller Aktivitäten in $[e_i, s_j)$, d.h. „zwischen" a_i und a_j.

2. Wert einer optimalen Lösung rekursiv definieren

Dynamisches Programmieren?

1. Struktur einer optimalen Lösung charakterisieren



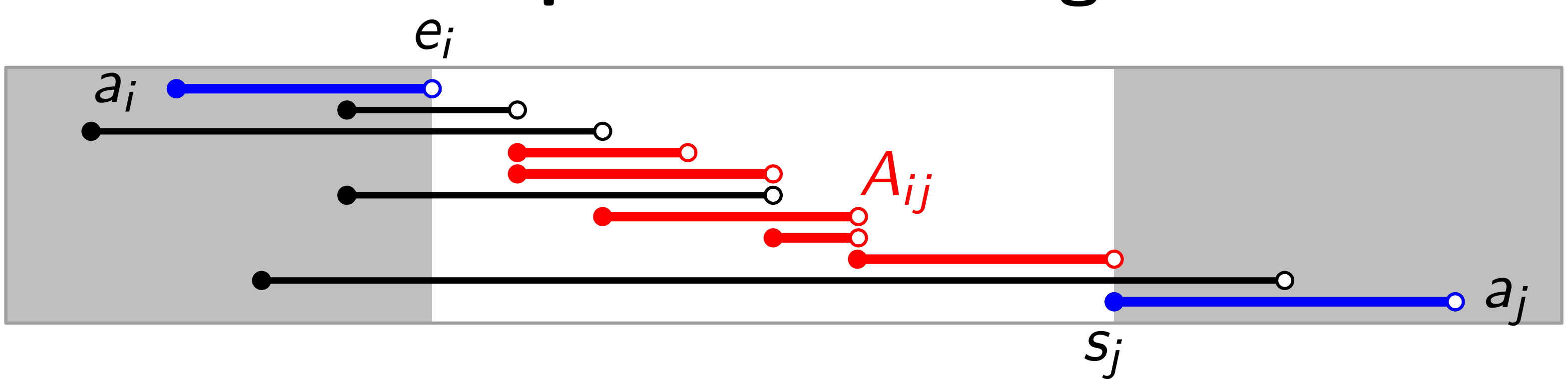
Sei $A_{ij} \subset A$ die Menge aller Aktivitäten in $[e_i, s_j)$, d.h. „zwischen" a_i und a_j.

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}.

2. Wert einer optimalen Lösung rekursiv definieren

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.

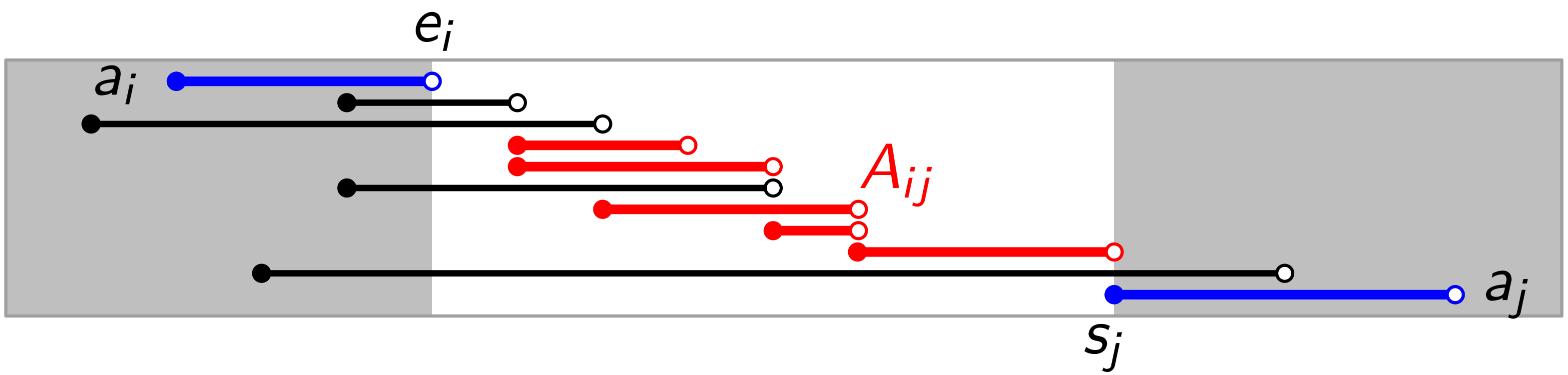
Ang. a_i und a_j sind in einer opt. Lösung $L \subseteq A$ enthalten, dann ist $L \cap A_{ij}$ eine opt. Lösung für A_{ij}.

Beweis?

2. Wert einer optimalen Lösung rekursiv definieren

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.

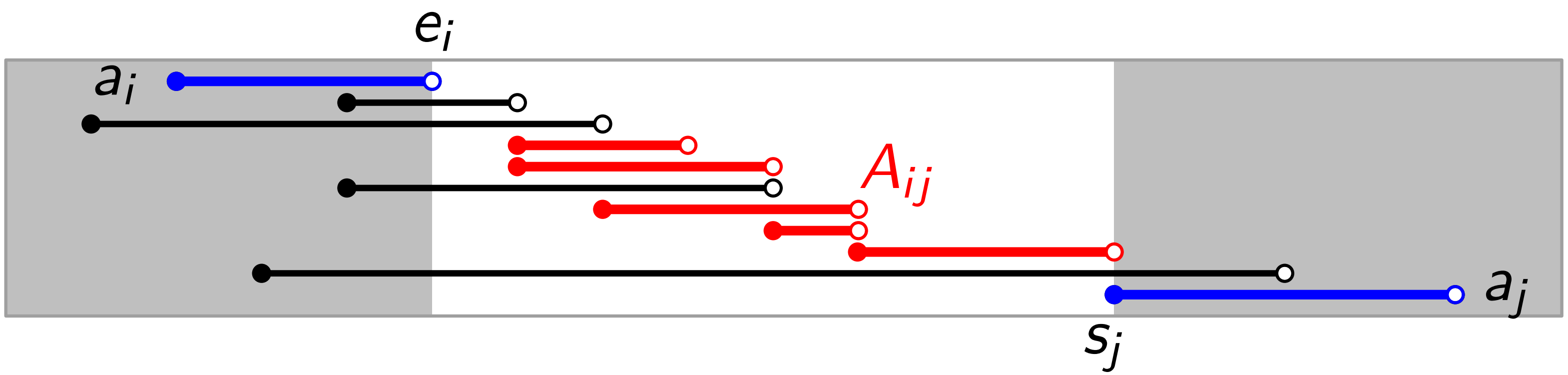
Ang. a_i und a_j sind in einer opt. Lösung $L \subseteq A$ enthalten, dann ist $L \cap A_{ij}$ eine opt. Lösung für A_{ij}.

Beweis? *Austauschargument!*

2. Wert einer optimalen Lösung rekursiv definieren

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}.

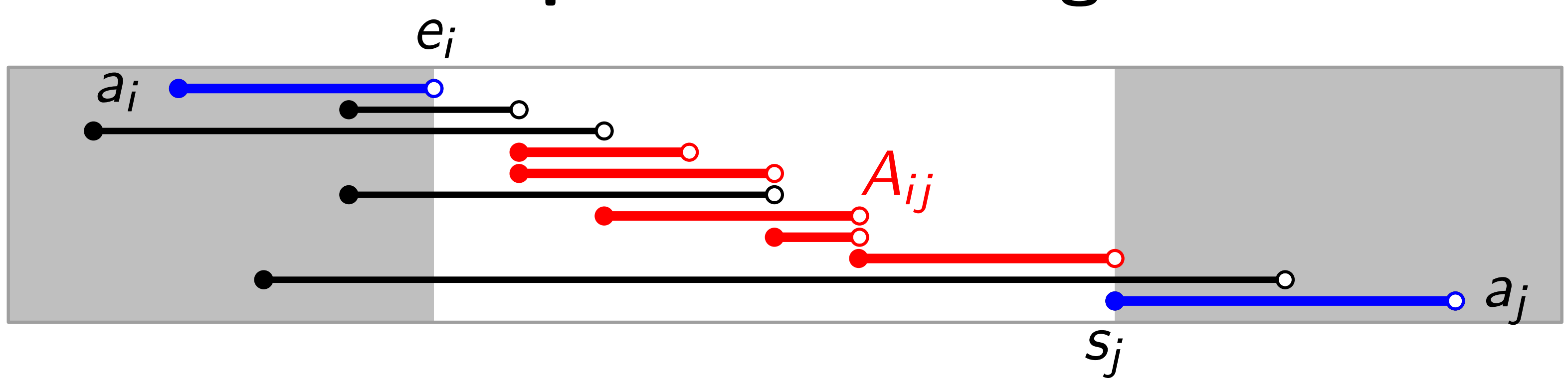
$\Rightarrow$ optimale Substruktur!

Beweis? *Austauschargument!*

2. Wert einer optimalen Lösung rekursiv definieren

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}.

$\Rightarrow$ optimale Substruktur!

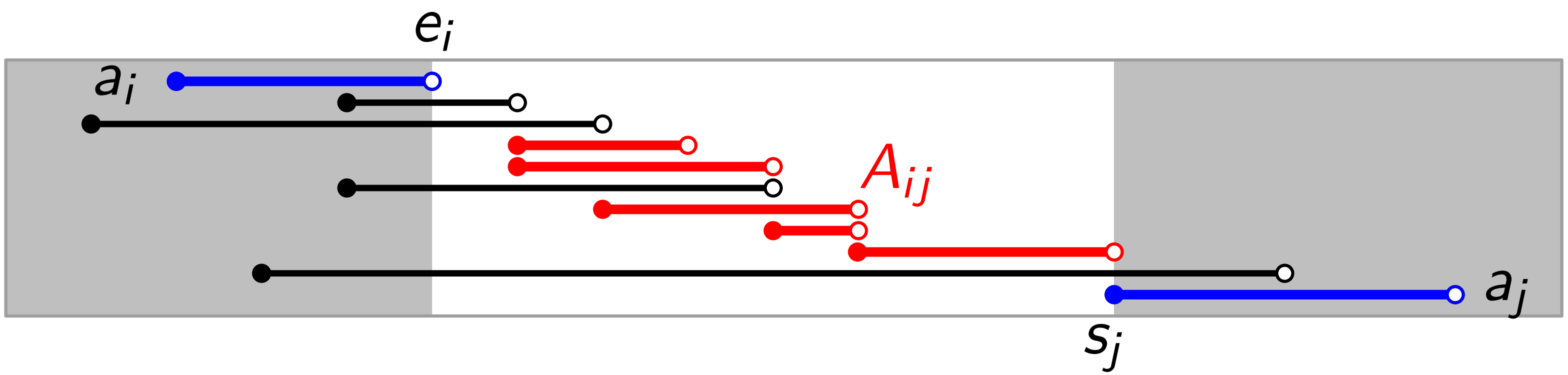
Beweis? *Austauschargument!*

2. Wert einer optimalen Lösung rekursiv definieren

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

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}.

$\Rightarrow$ optimale Substruktur!

Beweis? *Austauschargument!*

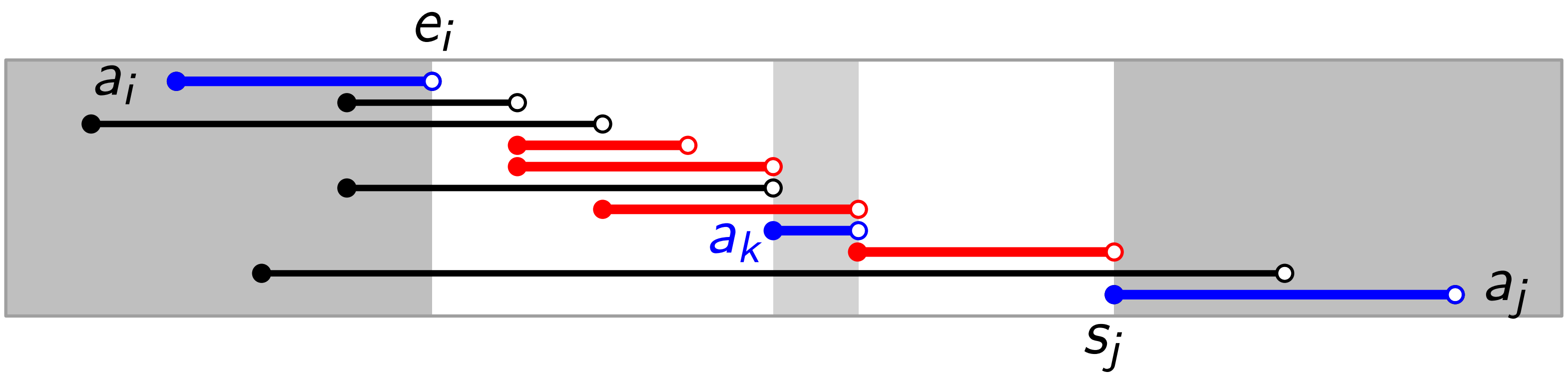
2. Wert einer optimalen Lösung rekursiv definieren

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

Dann gilt: $c_{ij} =$

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}.

$\Rightarrow$ optimale Substruktur! Beweis? *Austauschargument!*

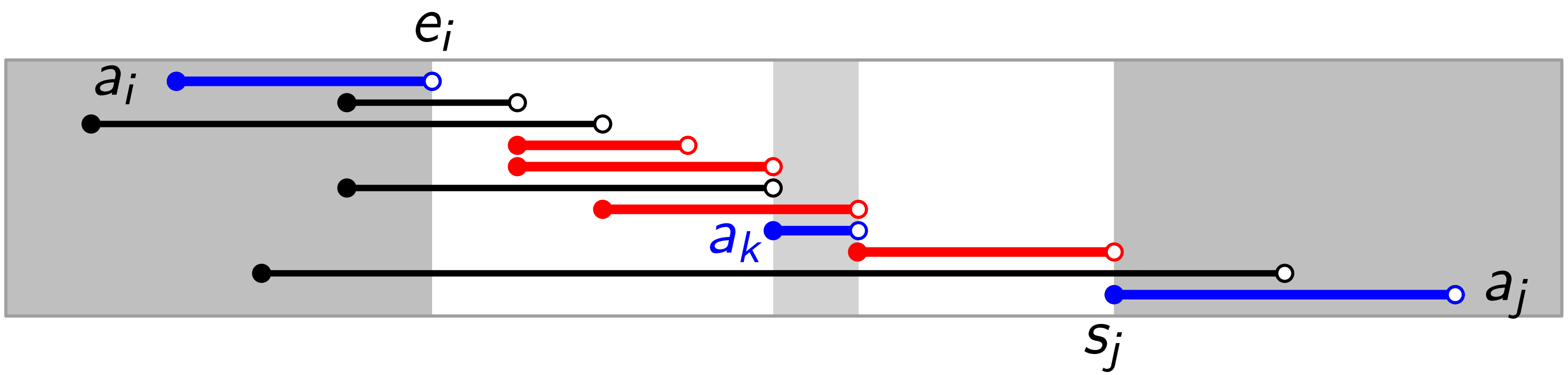
2. Wert einer optimalen Lösung rekursiv definieren

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

Dann gilt: $c_{ij} =$

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}.

$\Rightarrow$ 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

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 $A =$

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 $A = A_{0,n+1}$.

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.

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*

(b) *bottom-up*

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)

$\rightarrow$ liefert c_{ij}

(b) *bottom-up*

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

$\rightarrow$ liefert $c_{0,n+1}$

Siehe Folie
„Zurück zum dynamischen Programmieren“

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)

$\rightarrow$ liefert c_{ij}

(b) *bottom-up*

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

$\rightarrow$ liefert $c_{0,n+1}$

Siehe Folie
„Zurück zum dynamischen Programmieren“

Laufzeit?

2. Wert einer optimalen Lösung rekursiv definieren

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

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

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)

$\rightarrow$ liefert c_{ij}

(b) *bottom-up*

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

$\rightarrow$ 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?

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?

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

Darf's auch etwas einfacher sein?

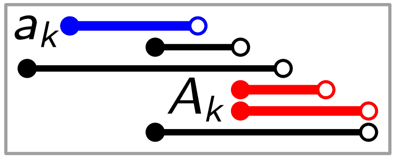
Idee: Sei L opt. Lösung für A. – Welche Aktivität hat gute Chancen die erste („linkeste") in L zu sein?

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

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.

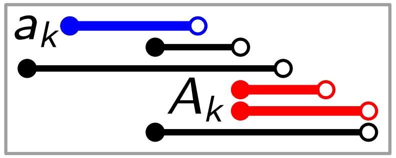


Sei $A_k = \{a_i \in A : s_i \geq e_k\}$ die Menge der Aktivitäten, die nach Ablauf von a_k beginnen.

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.



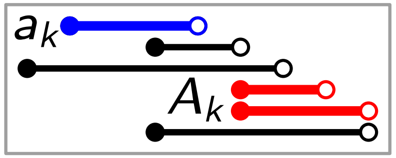
Sei $A_k = \{a_i \in A : s_i \geq e_k\}$ die Menge der Aktivitäten, die nach Ablauf von a_k beginnen.

Sei L_k eine optimale Lösung von A_k.

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.

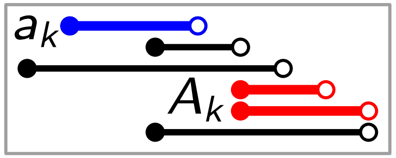
Sei L_k eine optimale Lösung von A_k.

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

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.

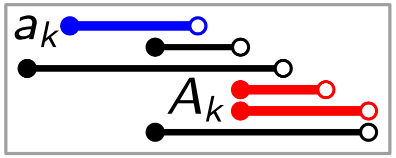
Falls Intuition korrekt, dann ist $\{a_1\} \cup L_1$ optimal.

Satz. Sei $A_k \neq \emptyset$.
Sei a_m Aktivität mit frühester Endzeit in A_k.

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.

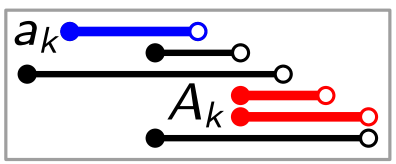
Falls Intuition korrekt, dann ist $\{a_1\} \cup L_1$ optimal.

Satz. Sei $A_k \neq \emptyset$.
Sei a_m Aktivität mit frühester Endzeit in A_k.
$\Rightarrow$ es gibt eine opt. Lösung von A_k, die a_m enthält.

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.

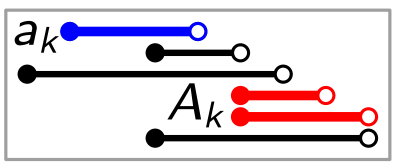
Satz. Sei $A_k \neq \emptyset$.
Sei a_m Aktivität mit frühester Endzeit in A_k.
$\Rightarrow$ es gibt eine opt. Lösung von A_k, die a_m enthält.

Beweis.

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.

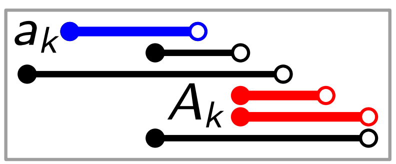
Satz. Sei $A_k \neq \emptyset$.
Sei a_m Aktivität mit frühester Endzeit in A_k.
$\Rightarrow$ es gibt eine opt. Lösung von A_k, die a_m enthält.

Beweis. *Austauschargument!*

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 $A_k \neq \emptyset$.
Sei a_m Aktivität mit frühester Endzeit in A_k.
$\Rightarrow$ es gibt eine opt. Lösung von A_k, die a_m enthält.

optimale Teilstruktur!

Beweis. *Austauschargument!*

Greedy – rekursiv

$\{a_1\} \cup L_1$ optimal.

Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
   e_0 = −∞    // ⇒ A_0 = A
   // Aktivitäten nach Endzeiten sortieren, falls nötig
   return GreedyRecursiveMain(s, e, 0)
```

Greedy – rekursiv

GreedyRecursive(int[] s, int[] e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

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

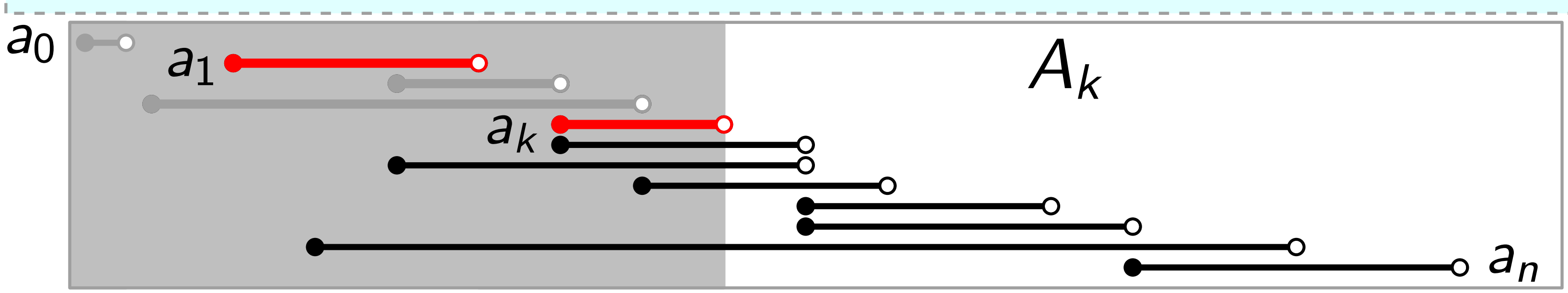
    return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)

Greedy – rekursiv

GreedyRecursive(int[] s, int[] e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

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

  return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)

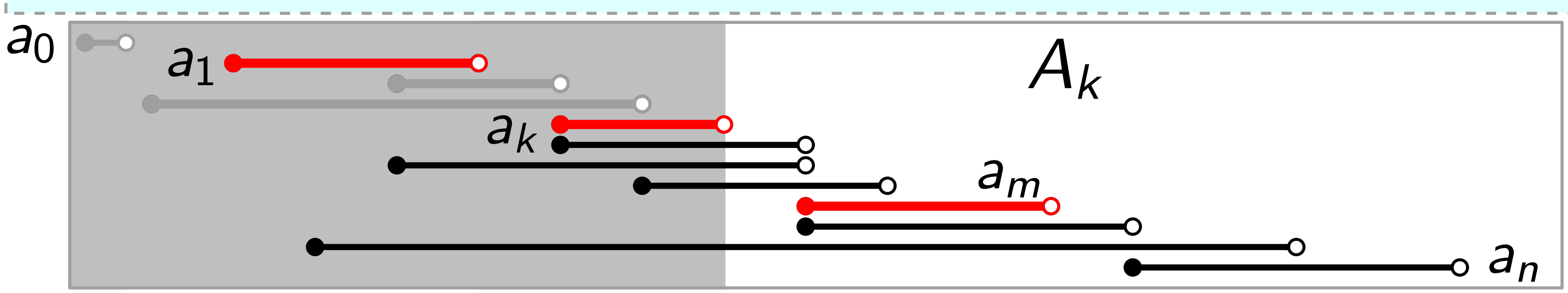


Greedy – rekursiv

GreedyRecursive(int[] s, int[] e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

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

    return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)

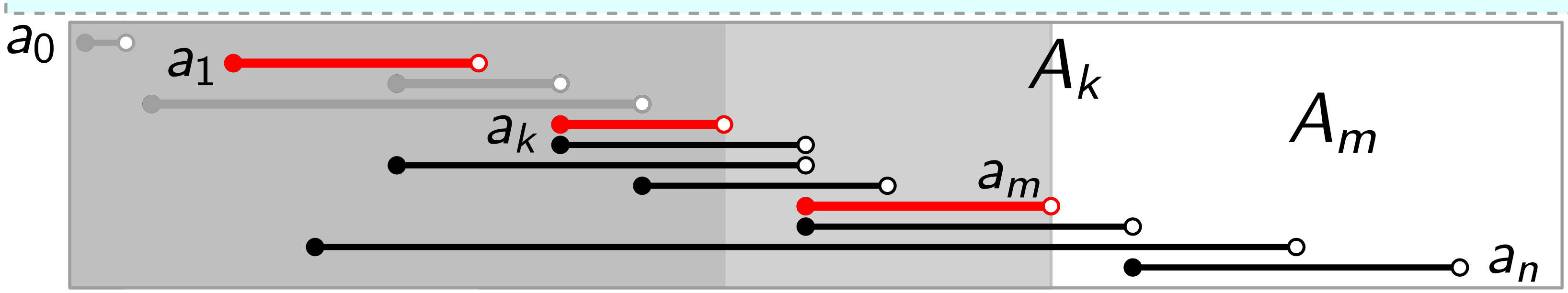


Greedy – rekursiv

GreedyRecursive(int[] s, int[] e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

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

    return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)

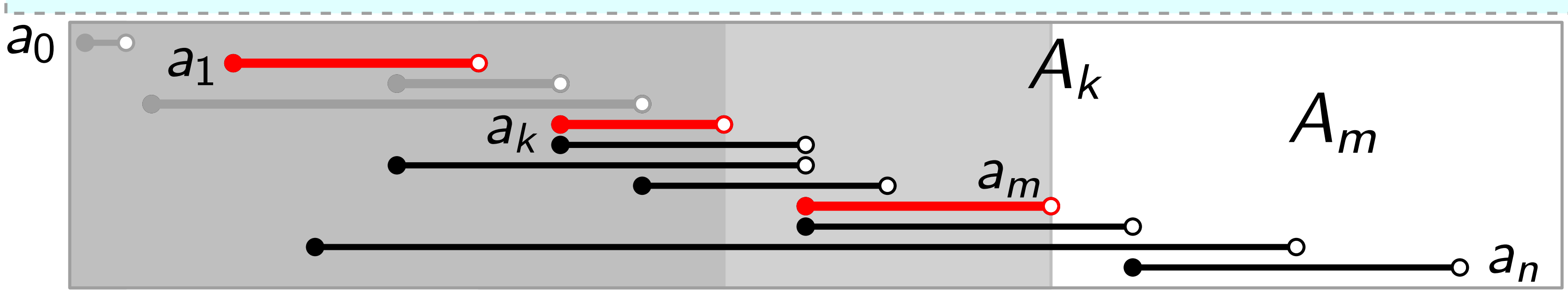


Greedy – rekursiv

GreedyRecursive(int[] s, int[] e)
$e_0 = -\infty$ // $\Rightarrow A_0 = A$
// Aktivitäten nach Endzeiten sortieren, falls nötig
return GreedyRecursiveMain($s, e, 0$)

GreedyRecursiveMain(int[] s, int[] e, int k)// best. Lsg. für A_k
$m = k + 1;\ \ n = s.\mathit{length}$
// Finde Aktivität a_m mit kleinster Endzeit in A_k
while **do**

return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)

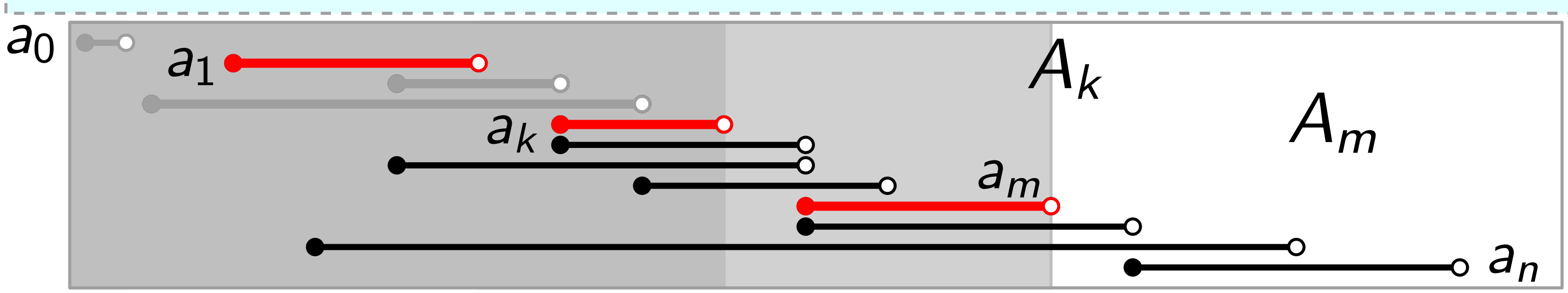


Greedy – rekursiv

GreedyRecursive(int[] s, int[] e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

GreedyRecursiveMain(int[] s, int[] e, int k)// best. Lsg. für A_k
  $m = k + 1$; $n = s.length$
  // Finde Aktivität a_m mit kleinster Endzeit in A_k
  while $m \leq n$ **and** $s[m] < e[k]$ **do**
    $m = m + 1$

    return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

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

$e_0 = -\infty$ // $\Rightarrow A_0 = A$

// Aktivitäten nach Endzeiten sortieren, falls nötig

return GreedyRecursiveMain($s, e, 0$)

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

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

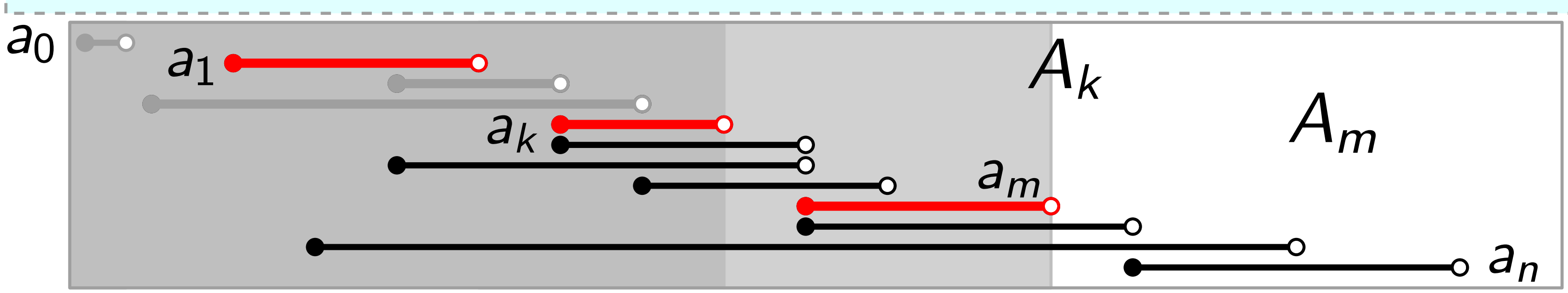
// Finde Aktivität a_m mit kleinster Endzeit in A_k

while $m \leq n$ **and** $s[m] < e[k]$ **do**

$m = m + 1$

if $m > n$ **then return** $\emptyset$

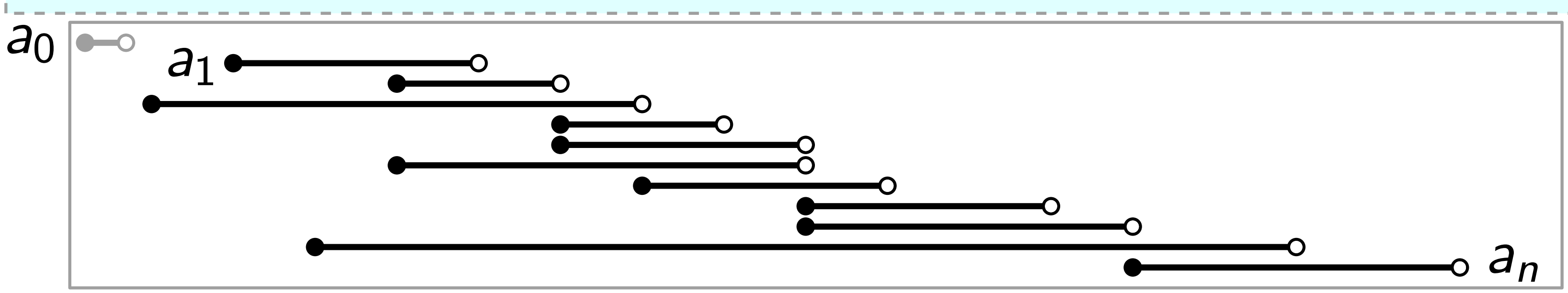
else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

GreedyRecursive(int[] s, int[] e)
$e_0 = -\infty$ // $\Rightarrow A_0 = A$
// Aktivitäten nach Endzeiten sortieren, falls nötig
return GreedyRecursiveMain($s, e, 0$)

GreedyRecursiveMain(int[] s, int[] e, int k)// best. Lsg. für A_k
$m = k + 1$; $n = s.length$
// Finde Aktivität a_m mit kleinster Endzeit in A_k
while $m \leq n$ **and** $s[m] < e[k]$ **do**
  $m = m + 1$
if $m > n$ **then return** $\emptyset$
else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

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

$e_0 = -\infty$ // $\Rightarrow A_0 = A$

// Aktivitäten nach Endzeiten sortieren, falls nötig

return GreedyRecursiveMain($s, e, 0$)

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

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

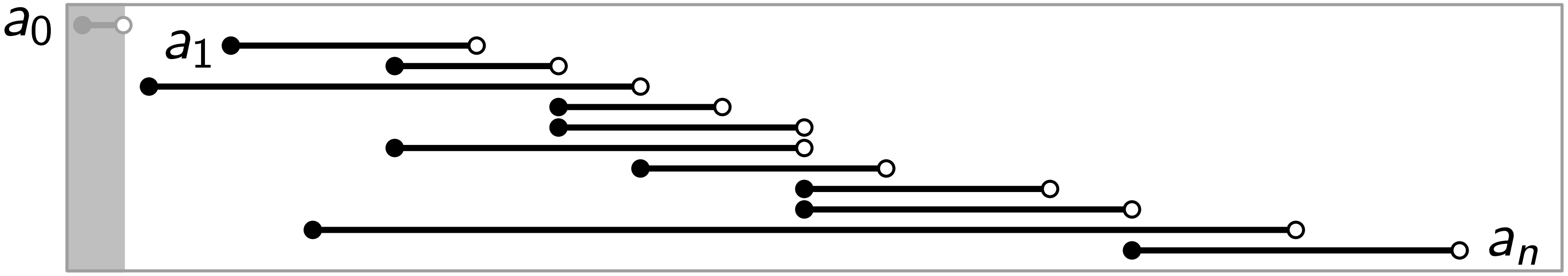
// Finde Aktivität a_m mit kleinster Endzeit in A_k

while $m \leq n$ **and** $s[m] < e[k]$ **do**

$m = m + 1$

if $m > n$ **then return** $\emptyset$

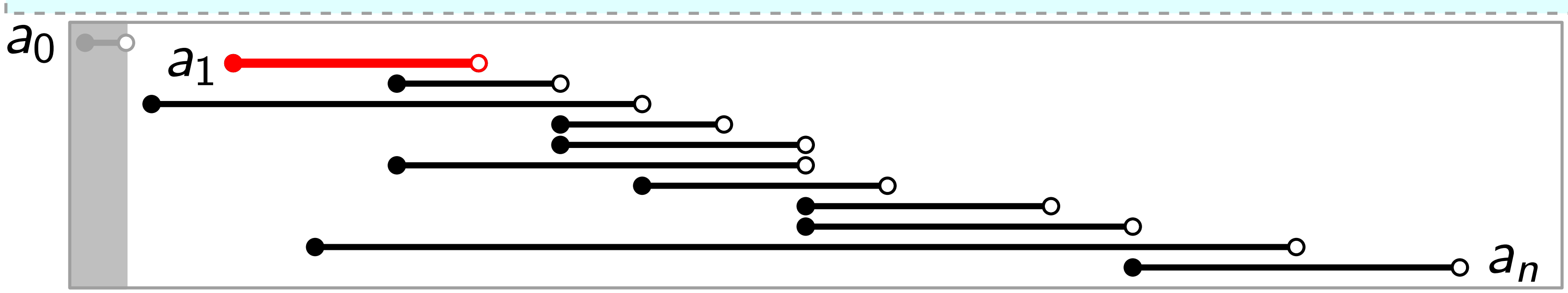
else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

GreedyRecursive(int[] s, int[] e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

GreedyRecursiveMain(int[] s, int[] e, int k) // best. Lsg. für A_k
  $m = k + 1$; $n = s.\mathit{length}$
  // Finde Aktivität a_m mit kleinster Endzeit in A_k
  while $m \leq n$ **and** $s[m] < e[k]$ **do**
    $m = m + 1$
  if $m > n$ **then return** $\emptyset$
  else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

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

  $e_0 = -\infty$ // $\Rightarrow A_0 = A$

  // Aktivitäten nach Endzeiten sortieren, falls nötig

  return GreedyRecursiveMain($s, e, 0$)

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

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

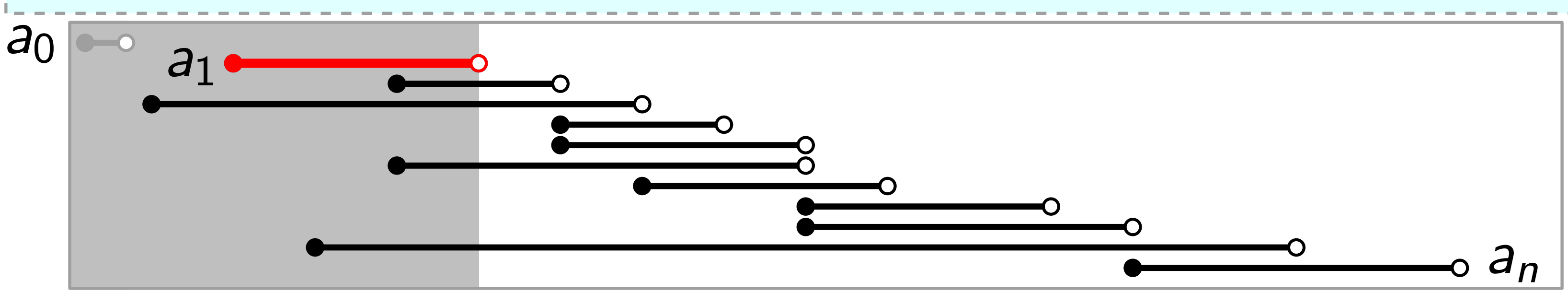
  // Finde Aktivität a_m mit kleinster Endzeit in A_k

  while $m \leq n$ **and** $s[m] < e[k]$ **do**

    $m = m + 1$

  if $m > n$ **then return** $\emptyset$

  else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

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

$e_0 = -\infty$ // $\Rightarrow A_0 = A$

// Aktivitäten nach Endzeiten sortieren, falls nötig

return GreedyRecursiveMain($s, e, 0$)

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

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

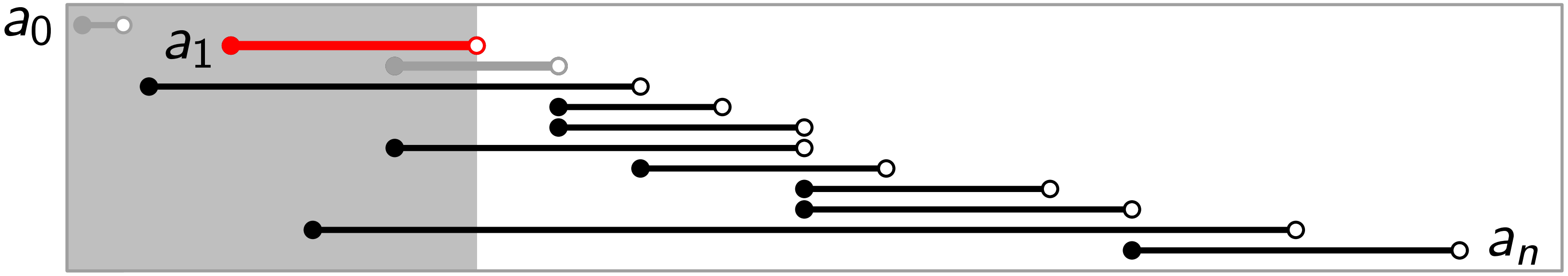
// Finde Aktivität a_m mit kleinster Endzeit in A_k

while $m \leq n$ **and** $s[m] < e[k]$ **do**

 $m = m + 1$

if $m > n$ **then return** $\emptyset$

else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

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

$e_0 = -\infty$ // $\Rightarrow A_0 = A$

// Aktivitäten nach Endzeiten sortieren, falls nötig

return GreedyRecursiveMain($s, e, 0$)

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

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

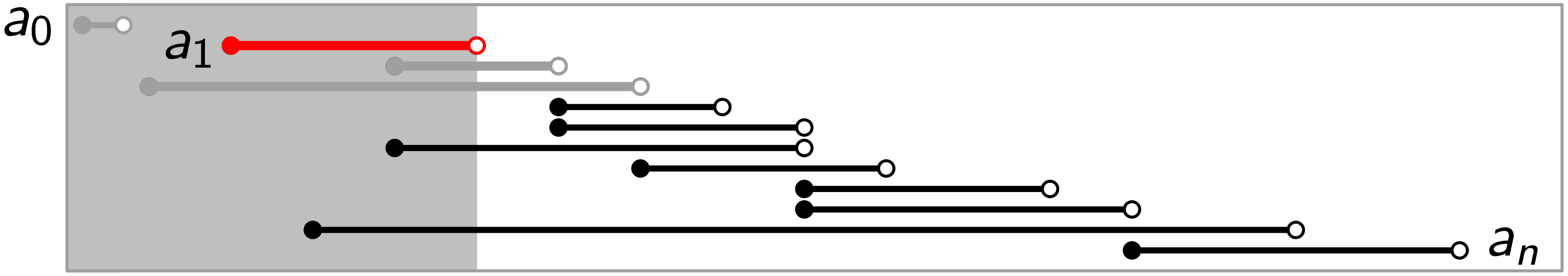
// Finde Aktivität a_m mit kleinster Endzeit in A_k

while $m \leq n$ **and** $s[m] < e[k]$ **do**

$m = m + 1$

if $m > n$ **then return** $\emptyset$

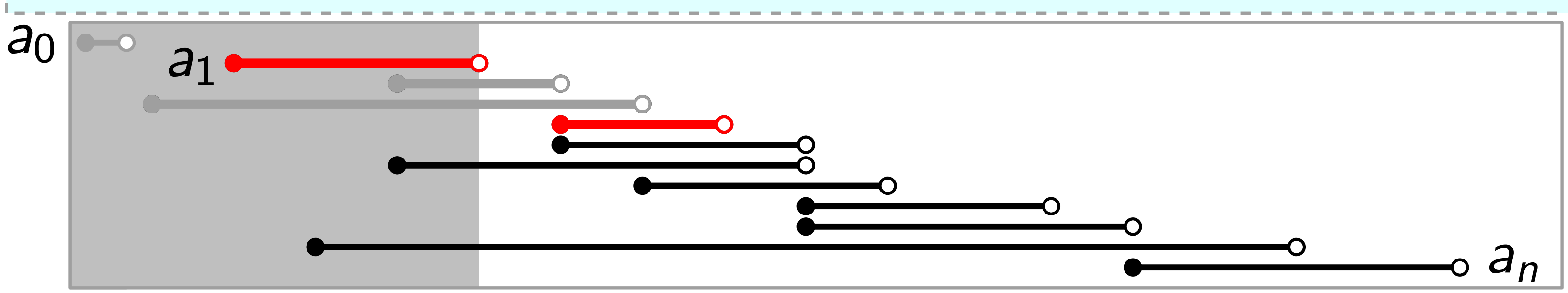
else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

GreedyRecursive(int[] s, int[] e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

GreedyRecursiveMain(int[] s, int[] e, int k)// best. Lsg. für A_k
  $m = k + 1$; $n = s.length$
  // Finde Aktivität a_m mit kleinster Endzeit in A_k
  while $m \leq n$ **and** $s[m] < e[k]$ **do**
    $m = m + 1$
  if $m > n$ **then return** $\emptyset$
  else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

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

  $e_0 = -\infty$ // $\Rightarrow A_0 = A$

  // Aktivitäten nach Endzeiten sortieren, falls nötig

  return GreedyRecursiveMain($s, e, 0$)

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

  $m = k + 1;\ n = s.length$

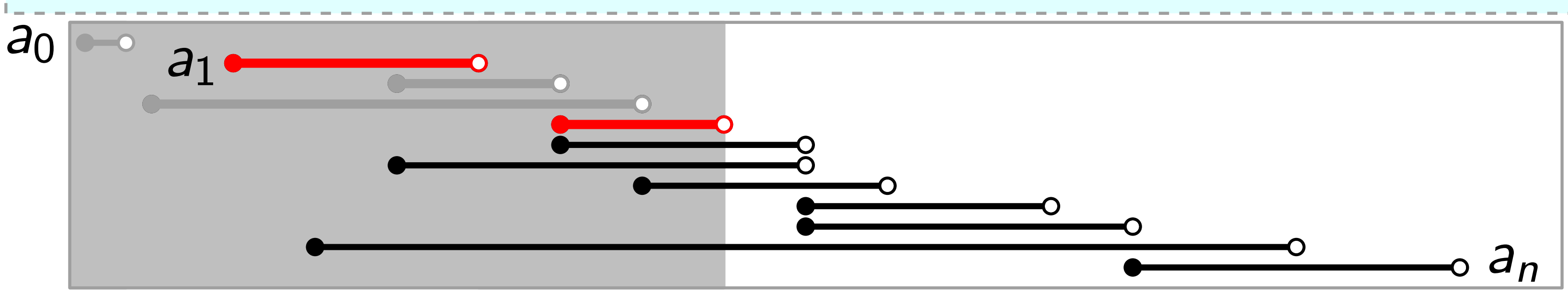
  // Finde Aktivität a_m mit kleinster Endzeit in A_k

  while $m \leq n$ **and** $s[m] < e[k]$ **do**

    $m = m + 1$

  if $m > n$ **then return** $\emptyset$

  else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

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

$e_0 = -\infty$ $\quad // \Rightarrow A_0 = A$

// Aktivitäten nach Endzeiten sortieren, falls nötig

return GreedyRecursiveMain($s, e, 0$)

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

$m = k + 1$; $n = s.\mathit{length}$

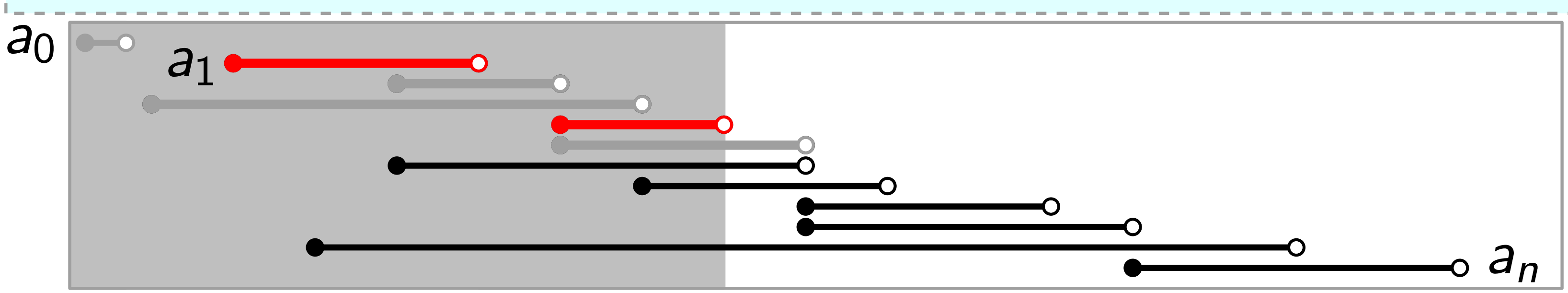
// Finde Aktivität a_m mit kleinster Endzeit in A_k

while $m \leq n$ **and** $s[m] < e[k]$ **do**

$\quad m = m + 1$

if $m > n$ **then return** $\emptyset$

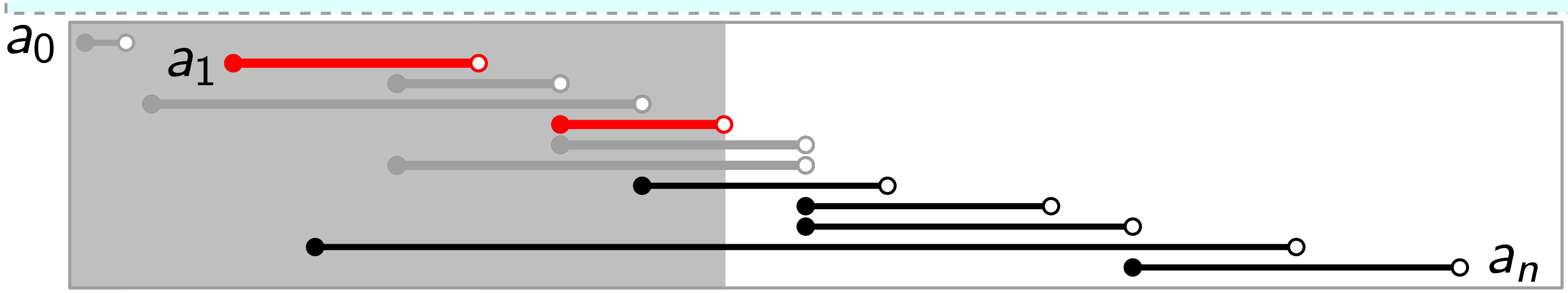
else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

GreedyRecursive(int[] s, int[] e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

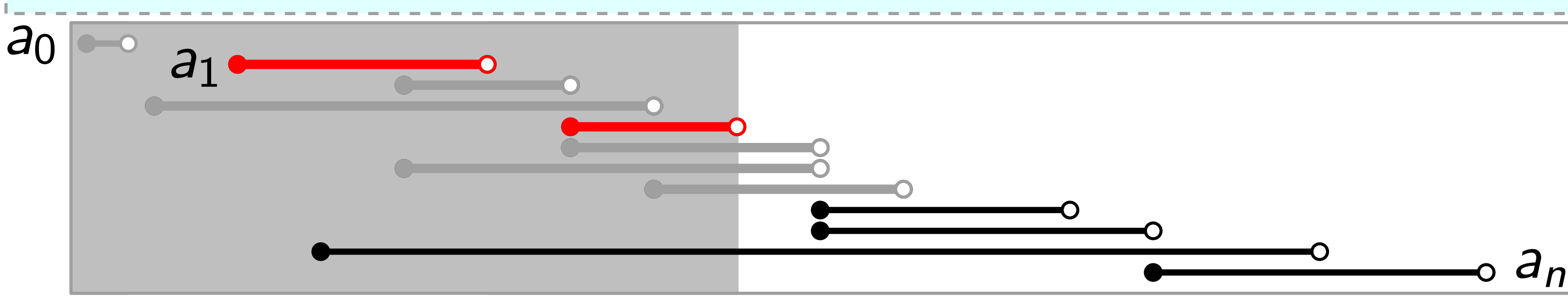
GreedyRecursiveMain(int[] s, int[] e, int k)// best. Lsg. für A_k
  $m = k + 1$; $n = s.length$
  // Finde Aktivität a_m mit kleinster Endzeit in A_k
  while $m \leq n$ **and** $s[m] < e[k]$ **do**
    $m = m + 1$
  if $m > n$ **then return** $\emptyset$
  else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

GreedyRecursive(int[] s, int[] e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

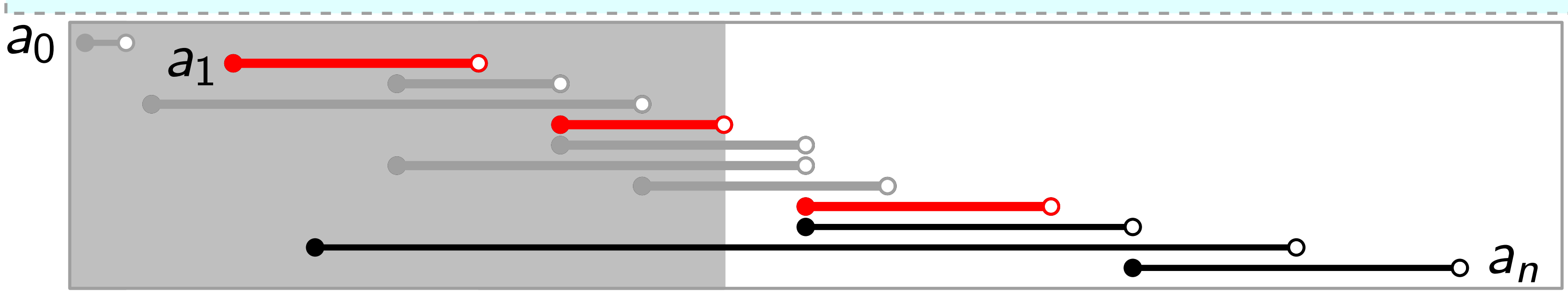
GreedyRecursiveMain(int[] s, int[] e, int k)// best. Lsg. für A_k
  $m = k + 1$; $n = s.length$
  // Finde Aktivität a_m mit kleinster Endzeit in A_k
  while $m \leq n$ **and** $s[m] < e[k]$ **do**
    $m = m + 1$
  if $m > n$ **then return** $\emptyset$
  else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

```
GreedyRecursive(int[] s, int[] e)
   e_0 = −∞     // ⇒ A_0 = A
   // Aktivitäten nach Endzeiten sortieren, falls nötig
   return GreedyRecursiveMain(s, e, 0)
```

```
GreedyRecursiveMain(int[] s, int[] e, int k) // best. Lsg. für A_k
   m = k + 1;  n = s.length
   // Finde Aktivität a_m mit kleinster Endzeit in A_k
   while m ≤ n and s[m] < e[k] do
      m = m + 1
   if m > n then return ∅
   else return {a_m} ∪ GreedyRecursiveMain(s, e, m)
```



Greedy – rekursiv

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

$e_0 = -\infty$ // $\Rightarrow A_0 = A$

// Aktivitäten nach Endzeiten sortieren, falls nötig

return GreedyRecursiveMain($s, e, 0$)

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

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

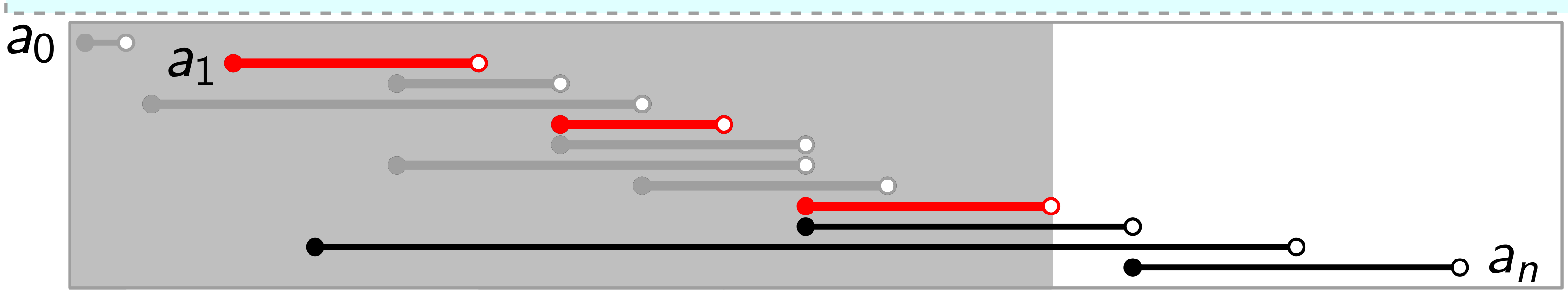
// Finde Aktivität a_m mit kleinster Endzeit in A_k

while $m \leq n$ **and** $s[m] < e[k]$ **do**

  $m = m + 1$

if $m > n$ **then return** $\emptyset$

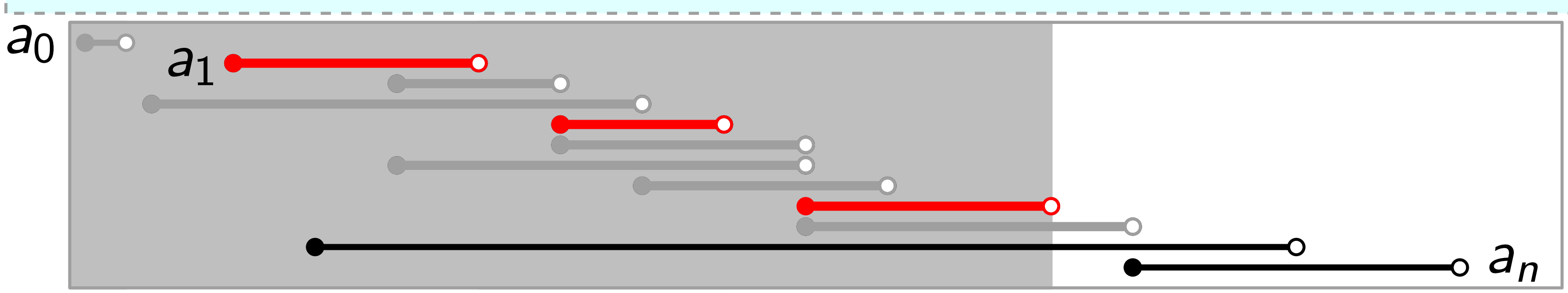
else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

GreedyRecursive(int[] s, int[] e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

GreedyRecursiveMain(int[] s, int[] e, int k) // best. Lsg. für A_k
  $m = k + 1$; $n = s.\mathit{length}$
  // Finde Aktivität a_m mit kleinster Endzeit in A_k
  while $m \leq n$ **and** $s[m] < e[k]$ **do**
    $m = m + 1$
  if $m > n$ **then return** $\emptyset$
  else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

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

  $e_0 = -\infty$ // $\Rightarrow A_0 = A$

  // Aktivitäten nach Endzeiten sortieren, falls nötig

  return GreedyRecursiveMain($s, e, 0$)

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

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

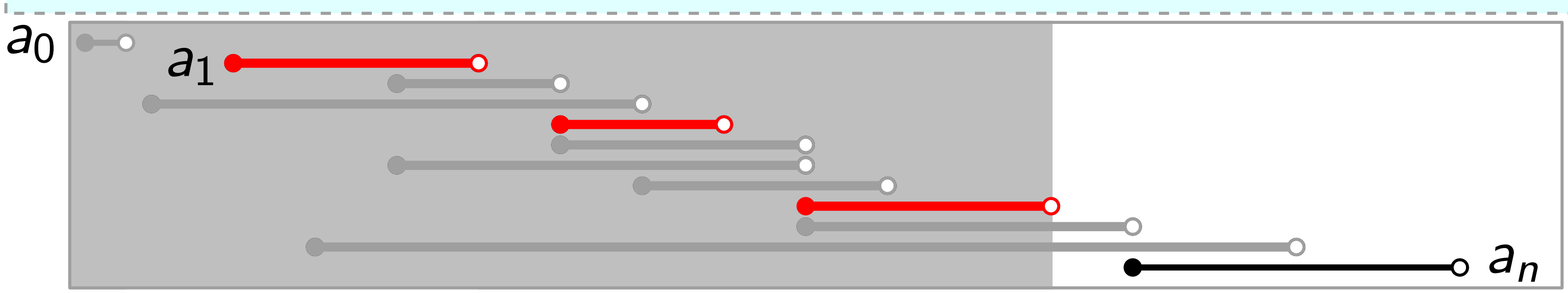
  // Finde Aktivität a_m mit kleinster Endzeit in A_k

  while $m \leq n$ **and** $s[m] < e[k]$ **do**

    $m = m + 1$

  if $m > n$ **then return** $\emptyset$

  else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

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

$e_0 = -\infty$ // $\Rightarrow A_0 = A$

// Aktivitäten nach Endzeiten sortieren, falls nötig

return GreedyRecursiveMain($s, e, 0$)

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

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

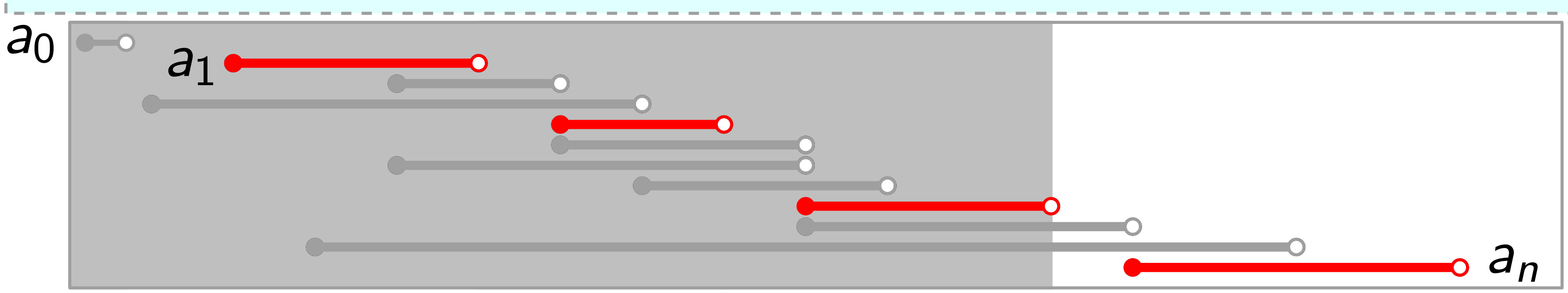
// Finde Aktivität a_m mit kleinster Endzeit in A_k

while $m \leq n$ **and** $s[m] < e[k]$ **do**

  $m = m + 1$

if $m > n$ **then return** $\emptyset$

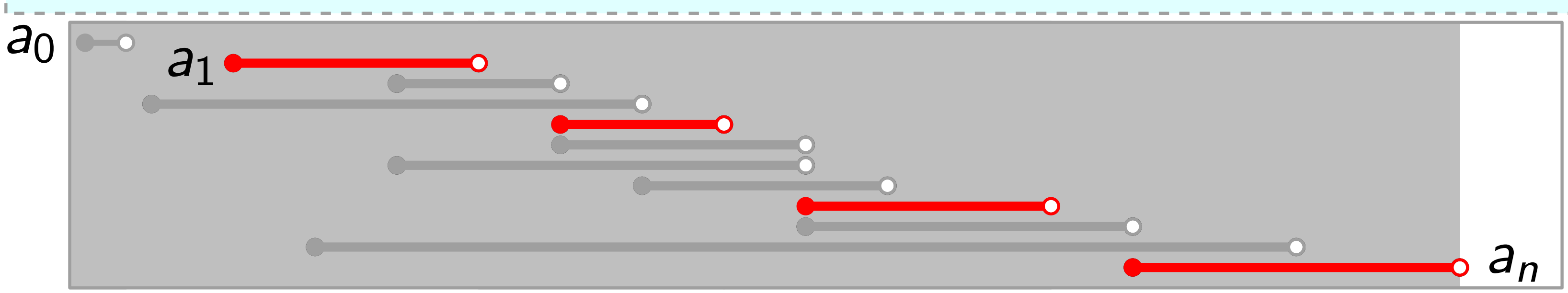
else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

GreedyRecursive(int[] s, int[] e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

GreedyRecursiveMain(int[] s, int[] e, int k)// best. Lsg. für A_k
  $m = k + 1$; $n = s.\mathit{length}$
  // Finde Aktivität a_m mit kleinster Endzeit in A_k
  while $m \leq n$ **and** $s[m] < e[k]$ **do**
    $m = m + 1$
  if $m > n$ **then return** $\emptyset$
  else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)



Greedy – rekursiv

GreedyRecursive(int[] s, int[] e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

GreedyRecursiveMain(int[] s, int[] e, int k)// best. Lsg. für A_k
  $m = k + 1$; $n = s.length$
  // Finde Aktivität a_m mit kleinster Endzeit in A_k
  while $m \leq n$ **and** $s[m] < e[k]$ **do**
    $m = m + 1$
  if $m > n$ **then return** $\emptyset$
  else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)

Laufzeit?

Greedy – rekursiv

GreedyRecursive(int[] s, int[] e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

GreedyRecursiveMain(int[] s, int[] e, int k)// best. Lsg. für A_k
  $m = k + 1$; $n = s.length$
  // Finde Aktivität a_m mit kleinster Endzeit in A_k
  while $m \leq n$ **and** $s[m] < e[k]$ **do**
    $m = m + 1$
  if $m > n$ **then return** $\emptyset$
  else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)

Laufzeit? Wie oft wird m inkrementiert?

Greedy – rekursiv

GreedyRecursive(int[] s, int[] e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

GreedyRecursiveMain(int[] s, int[] e, int k) // best. Lsg. für A_k
  $m = k + 1$; $n = s.length$
  // Finde Aktivität a_m mit kleinster Endzeit in A_k
  while $m \leq n$ **and** $s[m] < e[k]$ **do**
    $m = m + 1$
  if $m > n$ **then return** $\emptyset$
  else return $\{a_m\} \cup$ GreedyRecursiveMain(s, e, m)

Laufzeit? Wie oft wird m inkrementiert?
Insgesamt, über alle rekursiven Aufrufe, n Mal.

Greedy – rekursiv

GreedyRecursive(int[] s, int[] e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

GreedyRecursiveMain(int[] s, int[] e, int k) // best. Lsg. für A_k
  $m = k + 1$; $n = s.\mathit{length}$
  // Finde Aktivität a_m mit kleinster Endzeit in A_k
  while $m \leq n$ **and** $s[m] < e[k]$ **do**
    $m = m + 1$
  if $m > n$ **then return** $\emptyset$
  else return $\{a_m\} \cup$ 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 – ~~rekursiv~~

GreedyRecursive(int[] s, int[] e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

GreedyRecursiveMain(int[] s, int[] e, int k) // best. Lsg. für A_k
  $m = k + 1$; $n = s.\mathit{length}$
  // Finde Aktivität a_m mit kleinster Endzeit in A_k
  while $m \leq n$ **and** $s[m] < e[k]$ **do**
    $m = m + 1$
  if $m > n$ **then return** $\emptyset$
  else return $\{a_m\} \cup$ 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 – ~~rekursiv~~ iterativ!

GreedyRecursive(int[] s, int[] e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

GreedyRecursiveMain(int[] s, int[] e, int k)// best. Lsg. für A_k
  $m = k + 1$; $n = s.length$
  // Finde Aktivität a_m mit kleinster Endzeit in A_k
  while $m \leq n$ **and** $s[m] < e[k]$ **do**
    └ $m = m + 1$
  if $m > n$ **then return** $\emptyset$
  else return $\{a_m\} \cup$ 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 – ~~rekursiv~~ *iterativ!*

Schreiben Sie
GreedyIterative(int[]s, int[]e)!

GreedyRecursive(int[] s, int[] e)
  $e_0 = -\infty$ // $\Rightarrow A_0 = A$
  // Aktivitäten nach Endzeiten sortieren, falls nötig
  return GreedyRecursiveMain($s, e, 0$)

GreedyRecursiveMain(int[] s, int[] e, int k)// best. Lsg. für A_k
  $m = k + 1$; $n = s.length$
  // Finde Aktivität a_m mit kleinster Endzeit in A_k
  while $m \leq n$ **and** $s[m] < e[k]$ **do**
    $m = m + 1$
  if $m > n$ **then return** $\emptyset$
  else return $\{a_m\} \cup$ 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 = {a_1}
    k = 1    // höchster Index in L
    for m = 2 to n do
    |
    |
    |
    └──
    return L
```

Greedy – iterativ

```
GreedyIterative(int[] s, int[] e)
  n = s.length
  if n = 0 then return ∅
  L = {a_1}
  k = 1    // höchster Index in L
  for m = 2 to n do
      if s[m] ≥ e[k] then
          L = L ∪ {a_m}
          k = m
  return L
```

Greedy – iterativ

```
GreedyIterative(int[] s, int[] e)
  n = s.length
  if n = 0 then return ∅
  L = {a_1}
  k = 1    // höchster Index in L
  for m = 2 to n do
      if s[m] ≥ e[k] then
          L = L ∪ {a_m}
          k = m
  return L
```

Laufzeit?

Greedy – iterativ

```
GreedyIterative(int[] s, int[] e)
  n = s.length
  if n = 0 then return ∅
  L = {a_1}
  k = 1      // höchster Index in L
  for m = 2 to n do
    if s[m] ≥ e[k] then
      L = L ∪ {a_m}
      k = m
  return L
```

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

Greedy – iterativ

```
GreedyIterative(int[] s, int[] e)
  n = s.length
  if n = 0 then return ∅
  L = {a_1}
  k = 1    // höchster Index in L
  for m = 2 to n do
     if s[m] ≥ e[k] then
        L = L ∪ {a_m}
        k = m
  return L
```

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

Bemerkung: GreedyIterative berechnet dieselbe optimale Lösung wie GreedyRecursive

Greedy – iterativ

```
GreedyIterative(int[] s, int[] e)
  n = s.length
  if n = 0 then return ∅
  L = {a_1}
  k = 1    // höchster Index in L
  for m = 2 to n do
    if s[m] ≥ e[k] then
      L = L ∪ {a_m}
      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

Die Greedy-Strategie

1. Teste, ob das Problem optimale Teilstruktur aufweist.

Die Greedy-Strategie

1. Teste, ob das Problem optimale Teilstruktur aufweist.

2. Entwickle eine rekursive Lösung

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

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!)

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

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?

Food for Thought

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

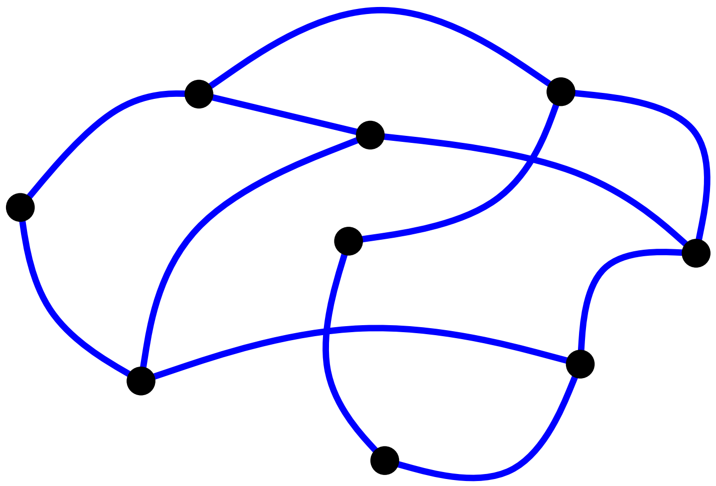
   Zur Erinnerung: Das DP berechnet $c_{ij} = \max_{a_k \in A_{ij}} c_{ik} + 1 + c_{kj}$.

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:

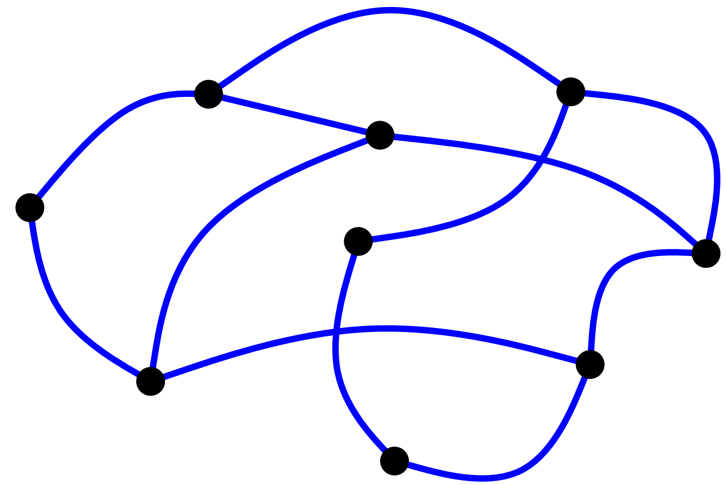


Food for Thought

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:



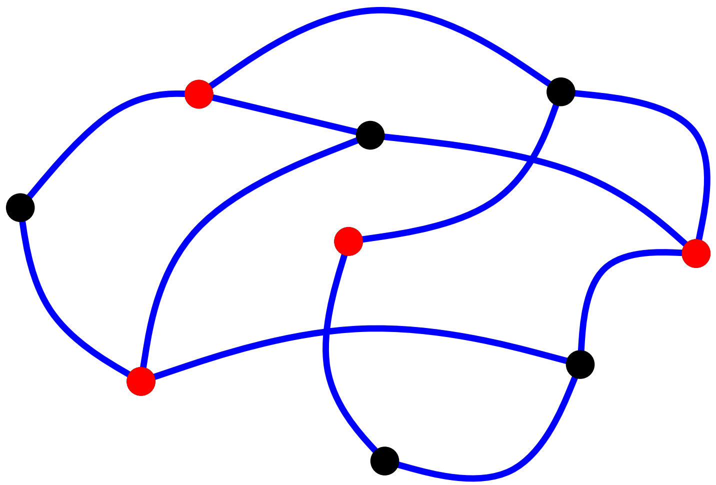
Finde eine größte Teilmenge U der Knoten, so dass keine zwei Knoten in U benachbart sind.

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:



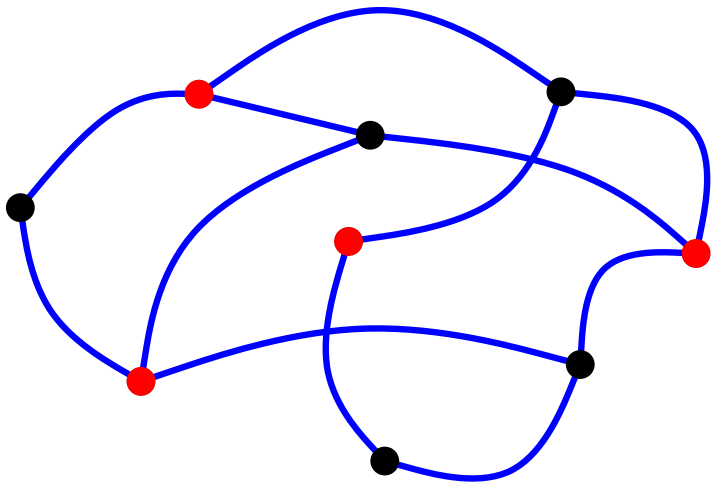
Finde eine größte Teilmenge U der Knoten, so dass keine zwei Knoten in U benachbart sind.

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.

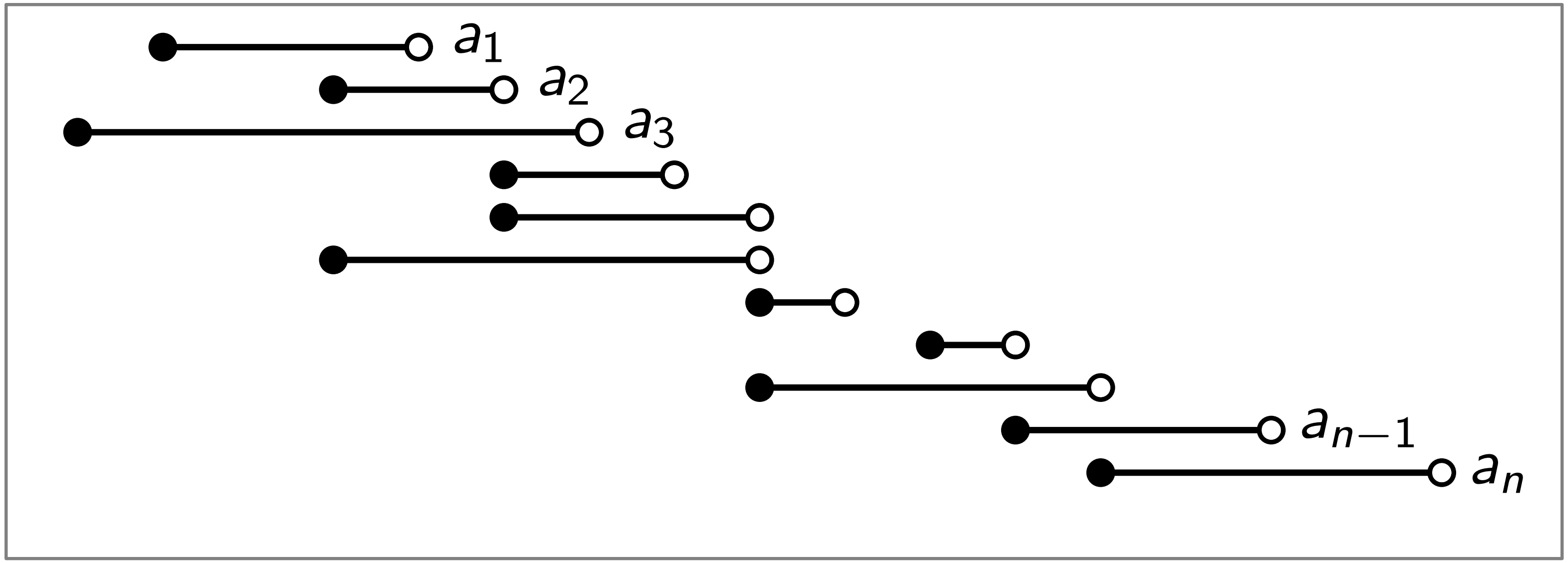
– Was hat guM mit unserem Ablaufplanungsproblem zu tun?

– 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 $i = 1, \dots, n$.

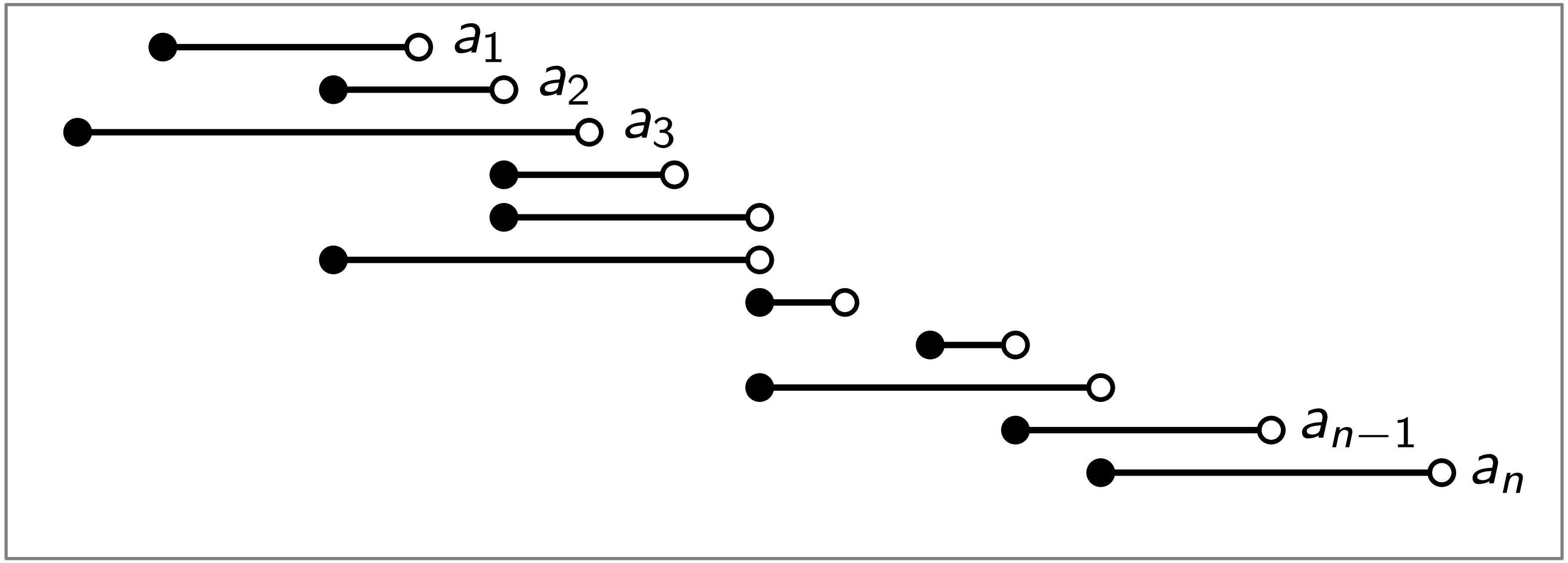
Für die Endpunkte gelte $e_1 \leq e_2 \leq \dots \leq e_n$.



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 $e_1 \leq e_2 \leq \dots \leq e_n$.

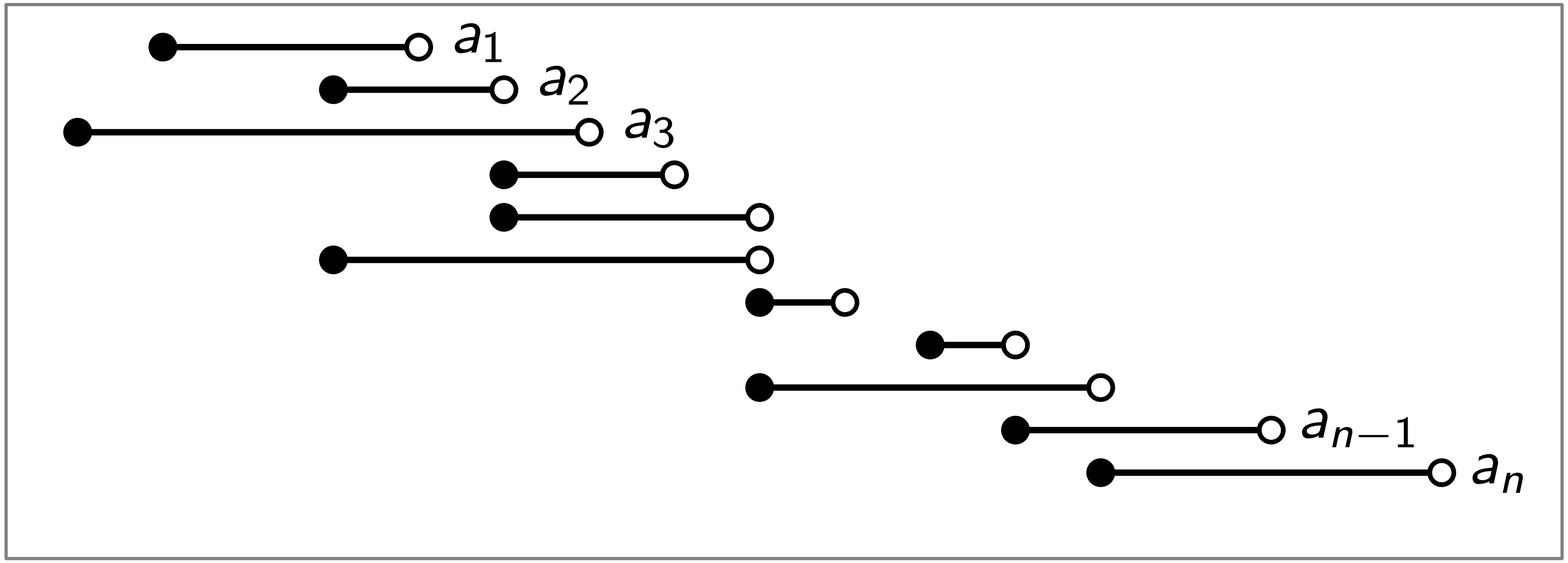


Gesucht: eine Menge $A' \subseteq A$ paarweise disjunkter Intervalle, deren Gesamtlänge $\ell(A')$ maximal ist.

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 $e_1 \leq e_2 \leq \dots \leq e_n$.



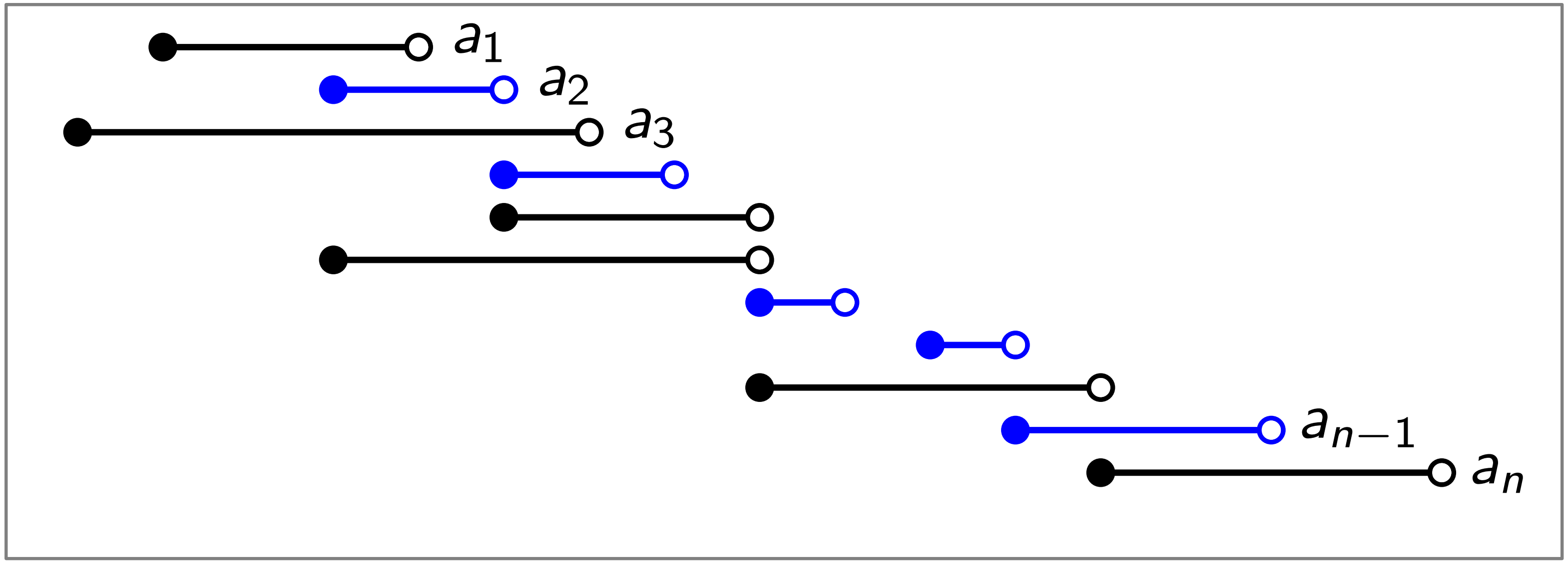
Gesucht: eine Menge $A' \subseteq A$ paarweise disjunkter Intervalle, deren Gesamtlänge $\ell(A')$ maximal ist.

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

Ein ähnliches Problem der Ablaufplanung

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

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



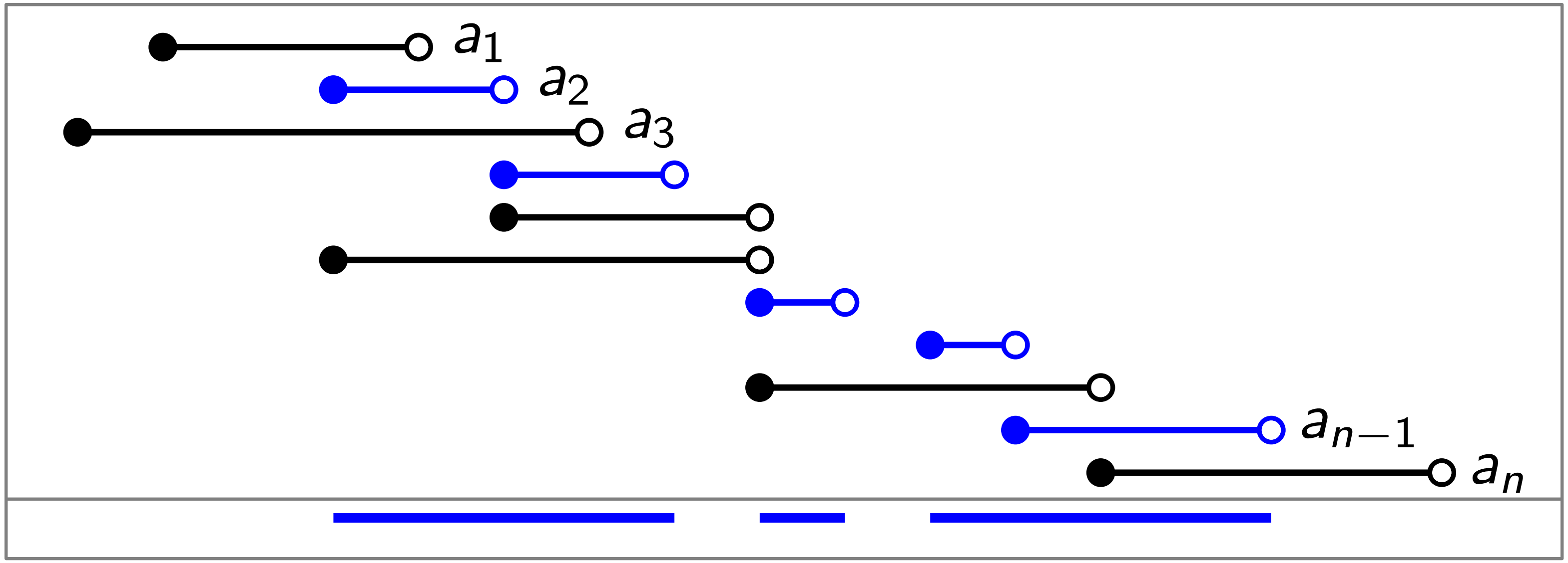
Gesucht: eine Menge $A' \subseteq A$ paarweise disjunkter Intervalle, deren Gesamtlänge $\ell(A')$ maximal ist.

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

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 $e_1 \leq e_2 \leq \dots \leq e_n$.



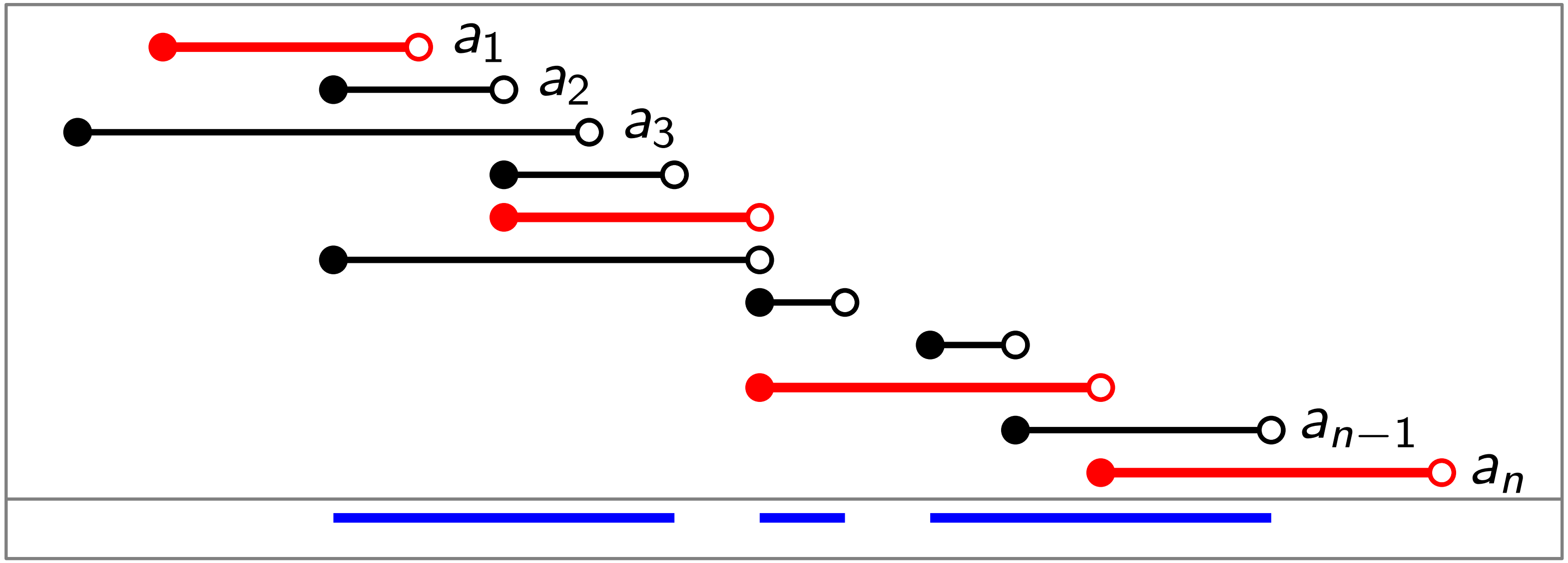
Gesucht: eine Menge $A' \subseteq A$ paarweise disjunkter Intervalle, deren Gesamtlänge $\ell(A')$ maximal ist.

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

Ein ähnliches Problem der Ablaufplanung

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

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



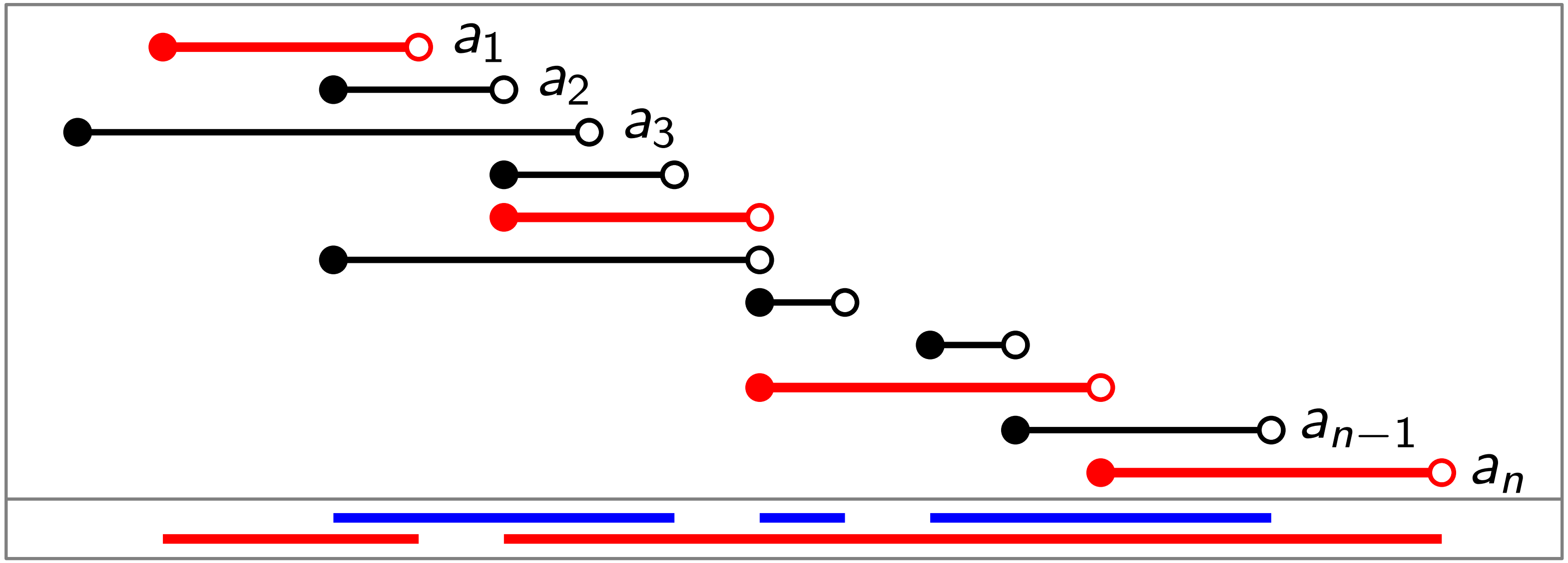
Gesucht: eine Menge $A' \subseteq A$ paarweise disjunkter Intervalle, deren Gesamtlänge $\ell(A')$ maximal ist.

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

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 $e_1 \le e_2 \le \dots \le e_n$.



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

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

Greedy?

Greedy?

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

Greedy?

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

Gegenbsp.:

Greedy?

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

Gegenbsp.:

Greedy?

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

Gegenbsp.:

2. Versuch:

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.*

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.:

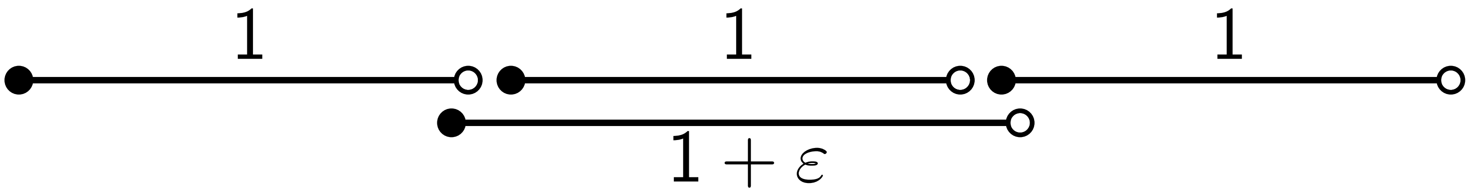
Greedy?

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.:



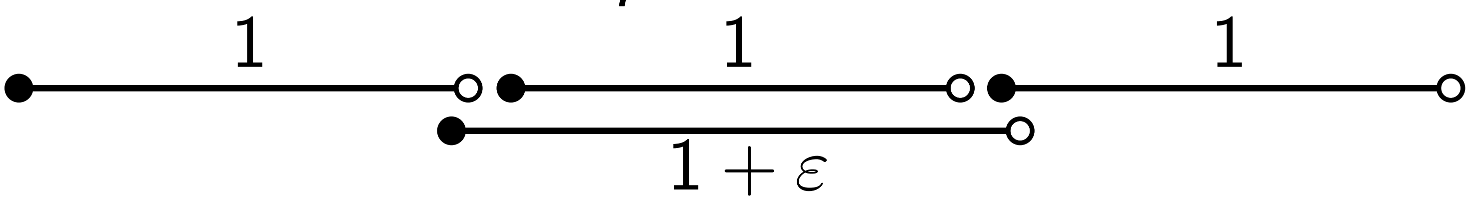
Greedy?

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.*

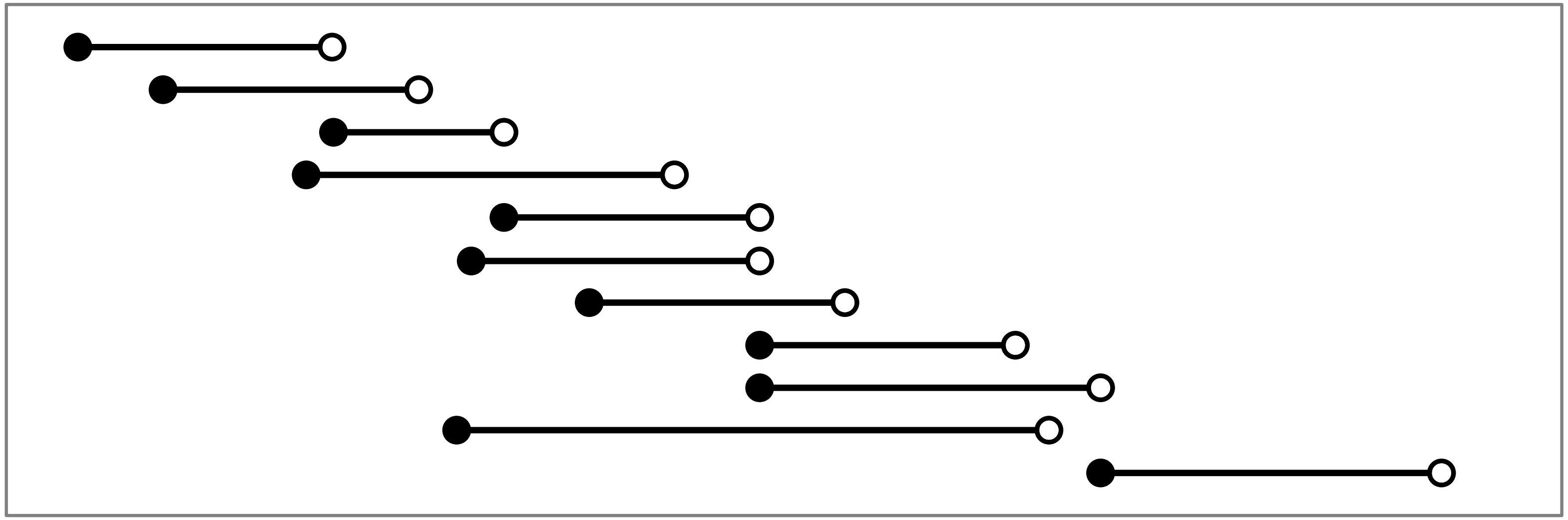
Gegenbsp.:



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

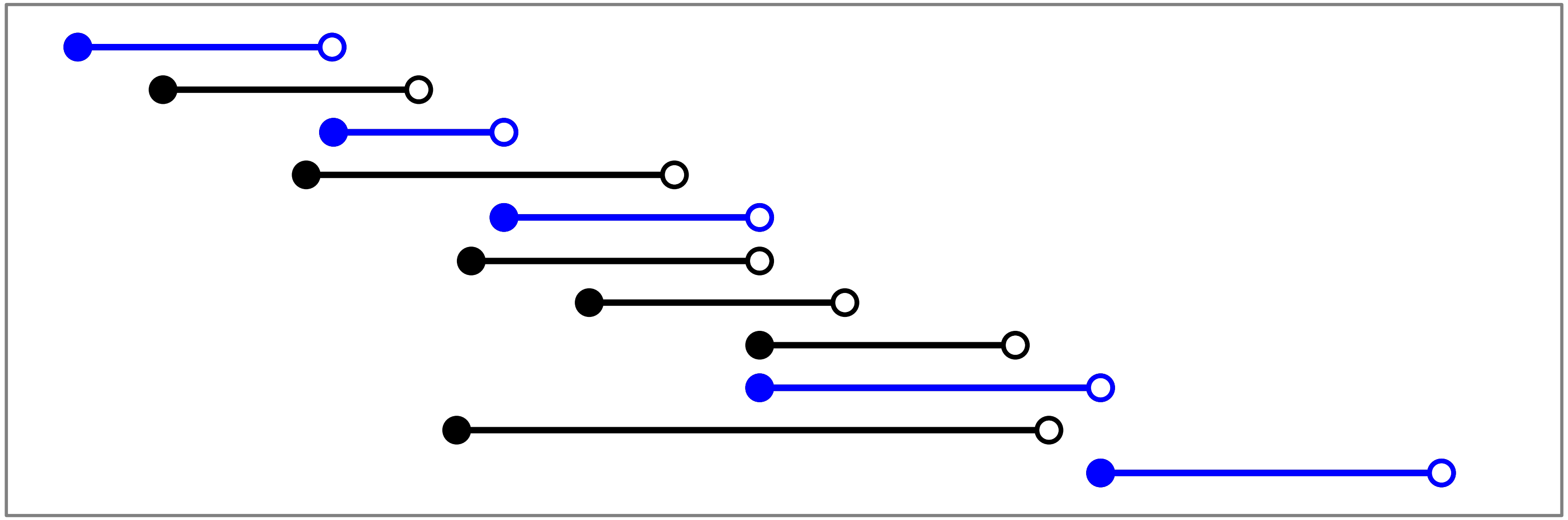
Diskutieren Sie mit Ihrer NachbarIn!

Wie gut/schlecht ist der 2. GA?



Wie gut/schlecht ist der 2. GA?

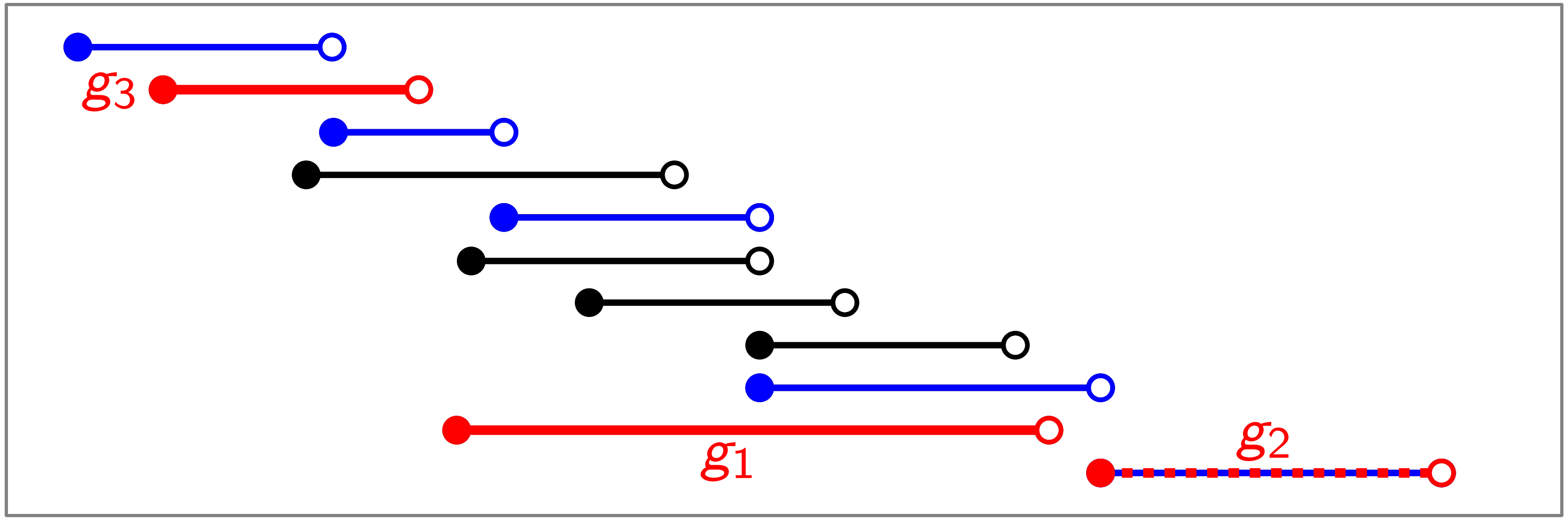
Betrachte eine optimale Lösung $L \subseteq A$.



Wie gut/schlecht ist der 2. GA?

Betrachte eine optimale Lösung $L \subseteq A$.

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

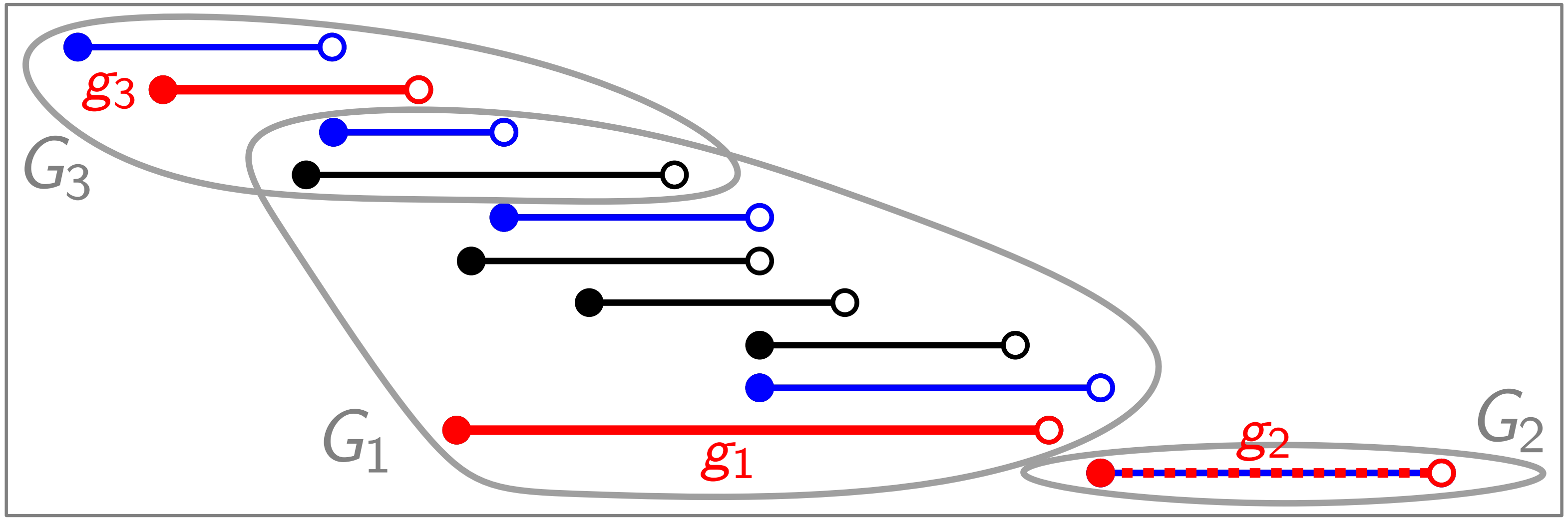


Wie gut/schlecht ist der 2. GA?

Betrachte eine optimale Lösung $L \subseteq A$.

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

Für $i = 1, \ldots, k$ sei $G_i = \{a \in A \mid a \cap g_i \neq \emptyset\}$

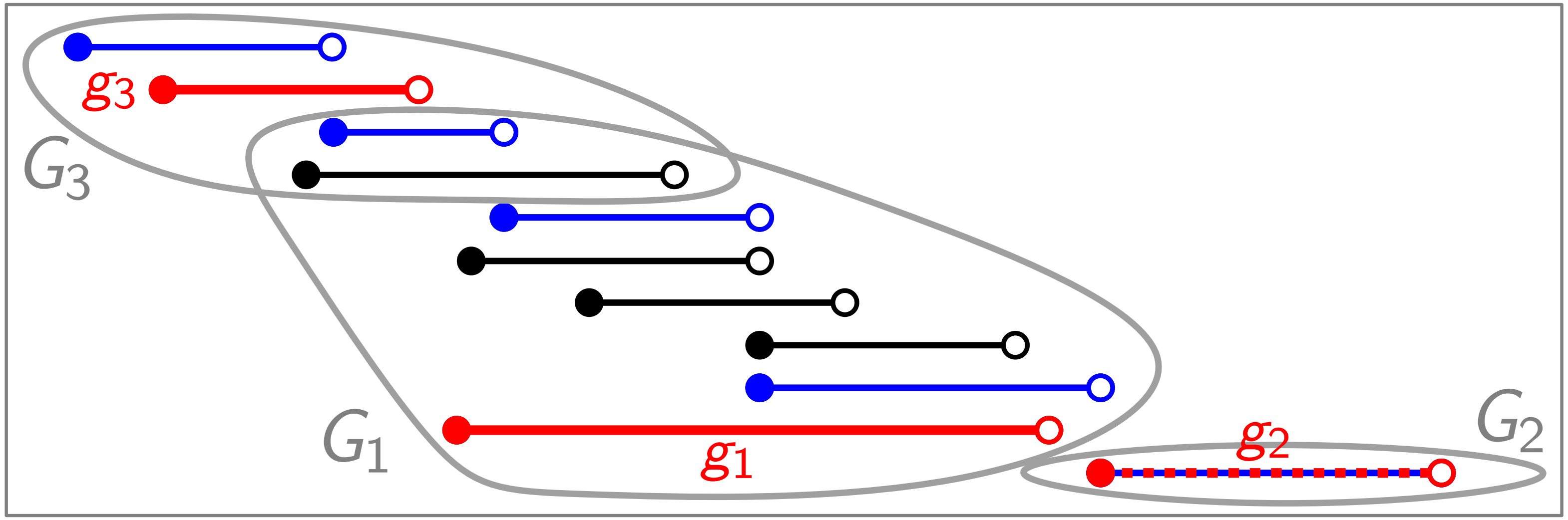


Wie gut/schlecht ist der 2. GA?

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

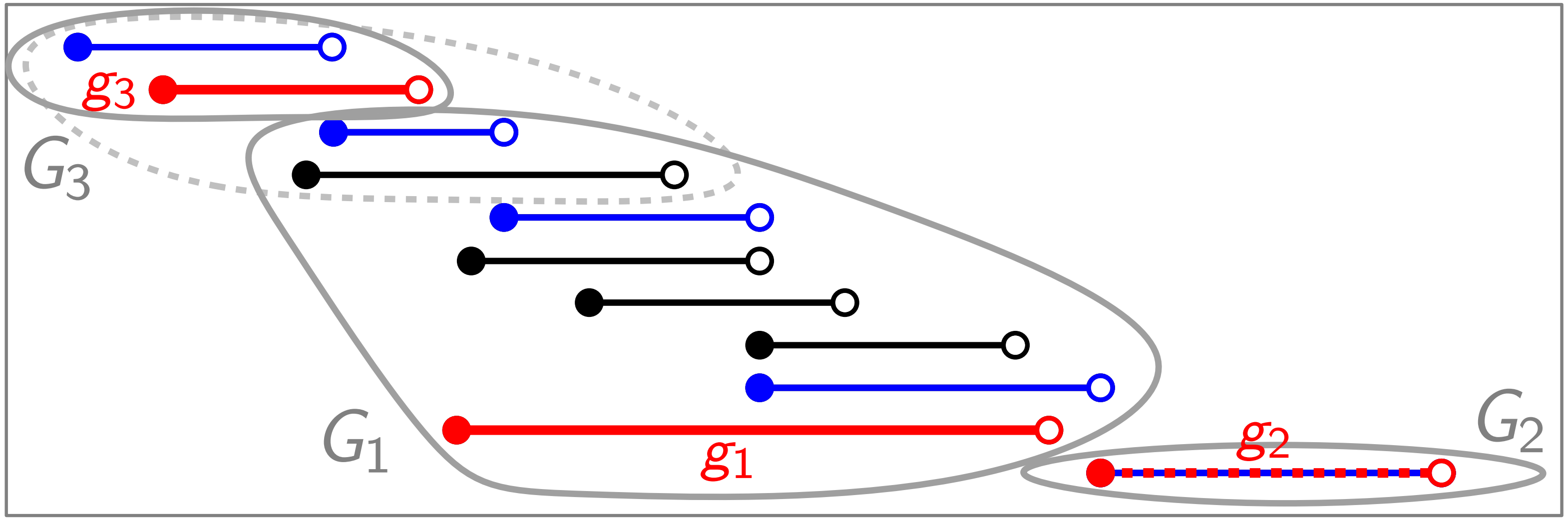


Wie gut/schlecht ist der 2. GA?

Betrachte eine optimale Lösung $L \subseteq A$.

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

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

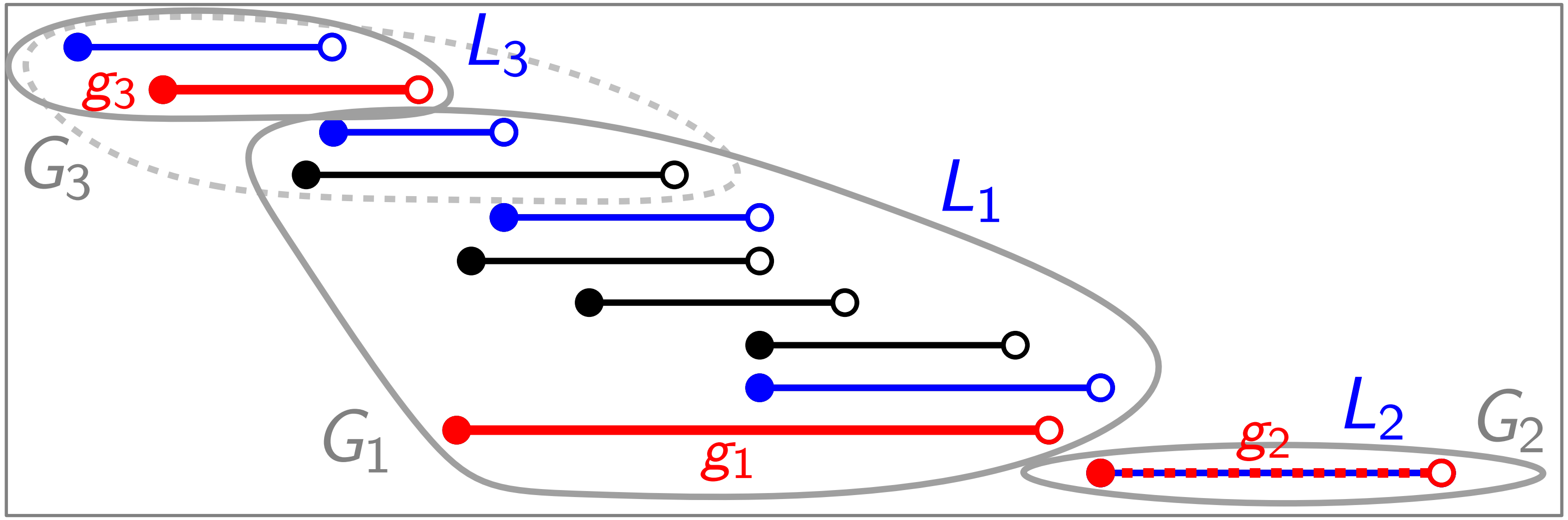


Wie gut/schlecht ist der 2. GA?

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



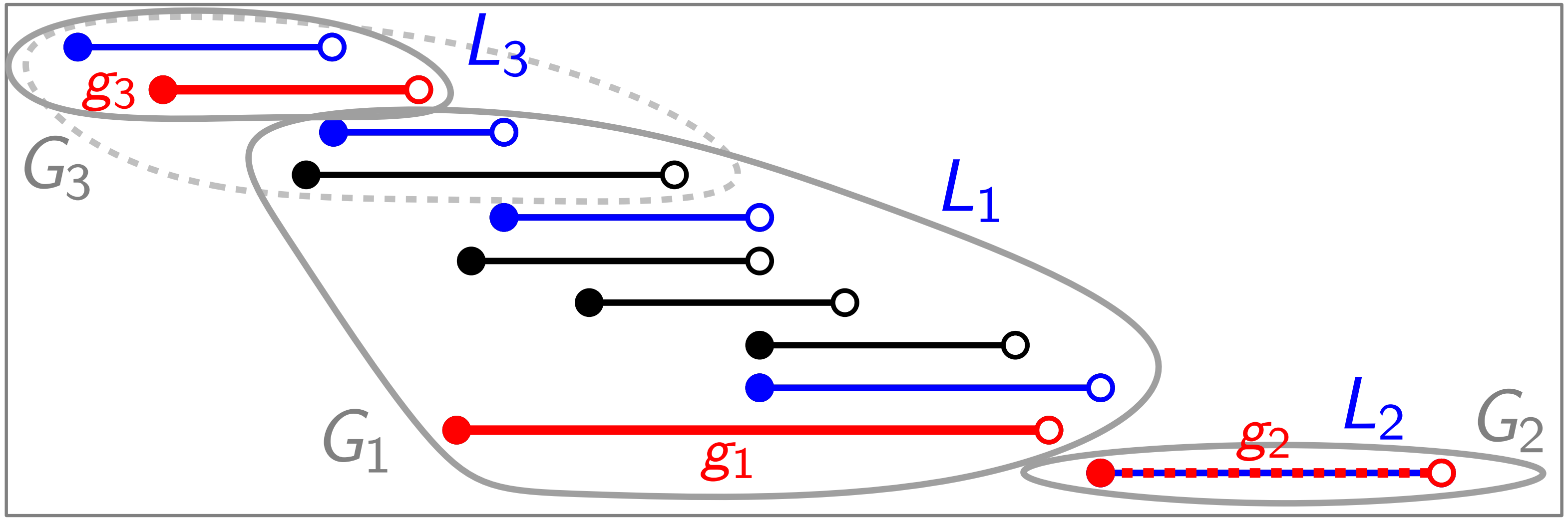
und sei $L_i = L \cap G_i$.

Wie gut/schlecht ist der 2. GA?

Betrachte eine optimale Lösung $L \subseteq A$.

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

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



und sei $L_i = L \cap G_i$.

Dann gilt $A = G_1 \,\dot\cup\, G_2 \,\dot\cup\, \cdots \,\dot\cup\, G_k$

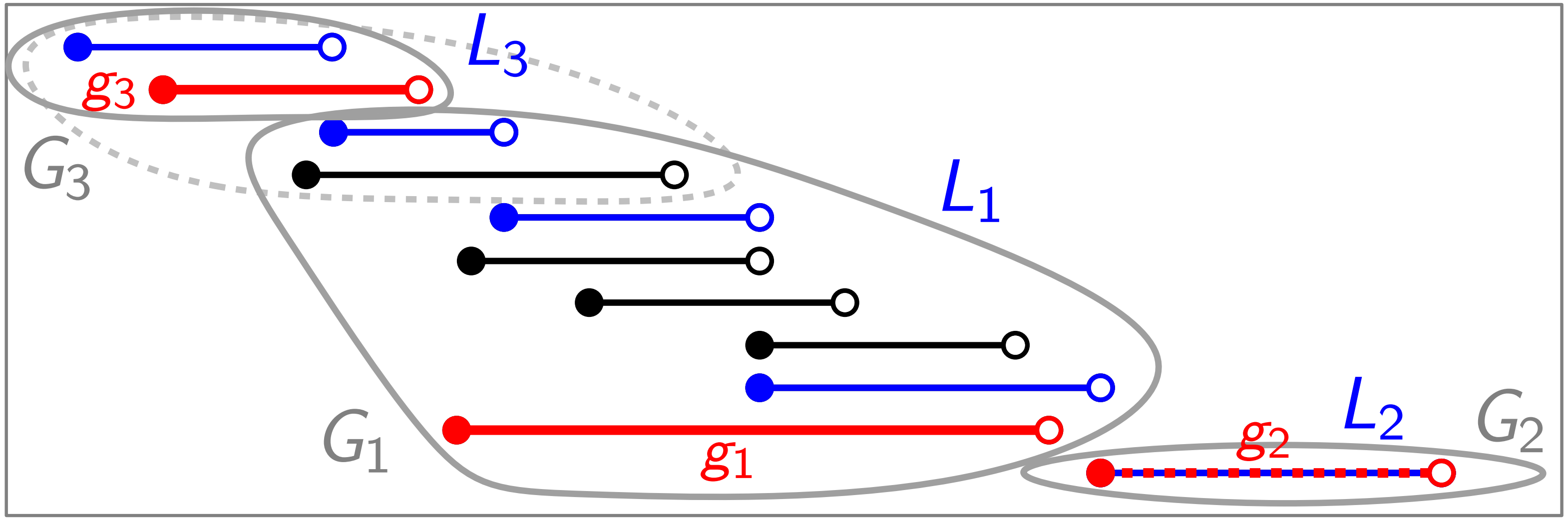
Wie gut/schlecht ist der 2. GA?

Betrachte eine optimale Lösung $L \subseteq A$.

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

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

und sei $L_i = L \cap G_i$.



Dann gilt $A = G_1 \,\dot\cup\, G_2 \,\dot\cup\, \cdots \,\dot\cup\, G_k$

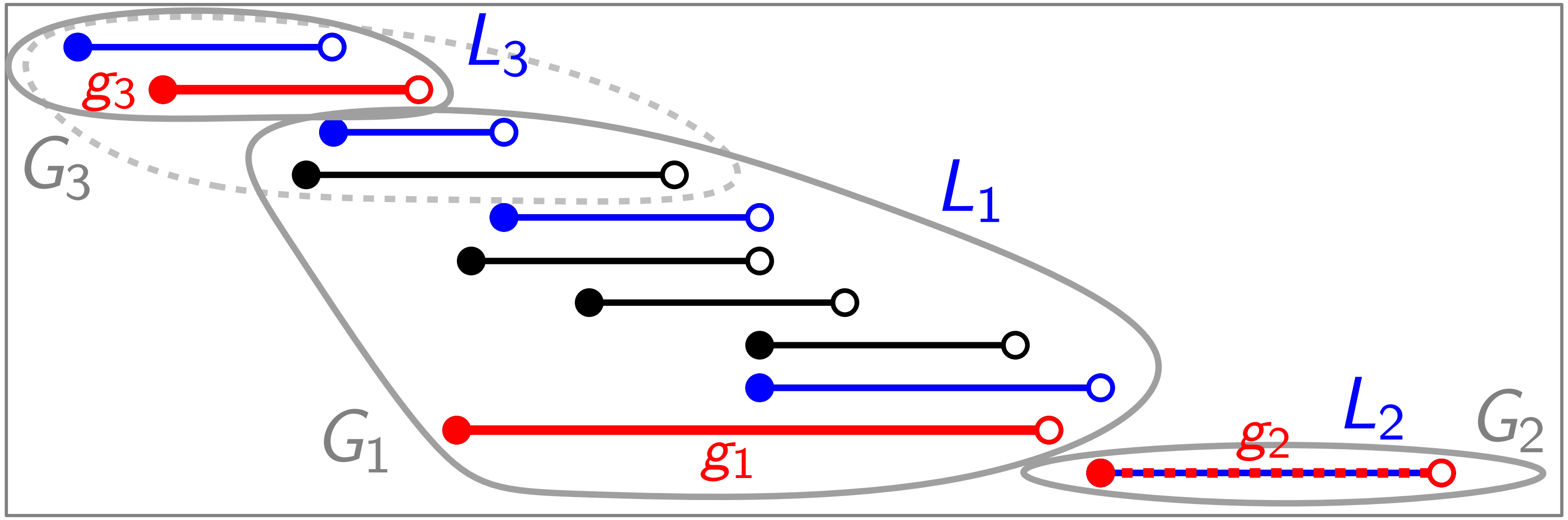
„$\subseteq$“:

Wie gut/schlecht ist der 2. GA?

Betrachte eine optimale Lösung $L \subseteq A$.

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

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



und sei $L_i = L \cap G_i$.

Dann gilt $A = G_1 \,\dot\cup\, G_2 \,\dot\cup\, \cdots \,\dot\cup\, G_k$

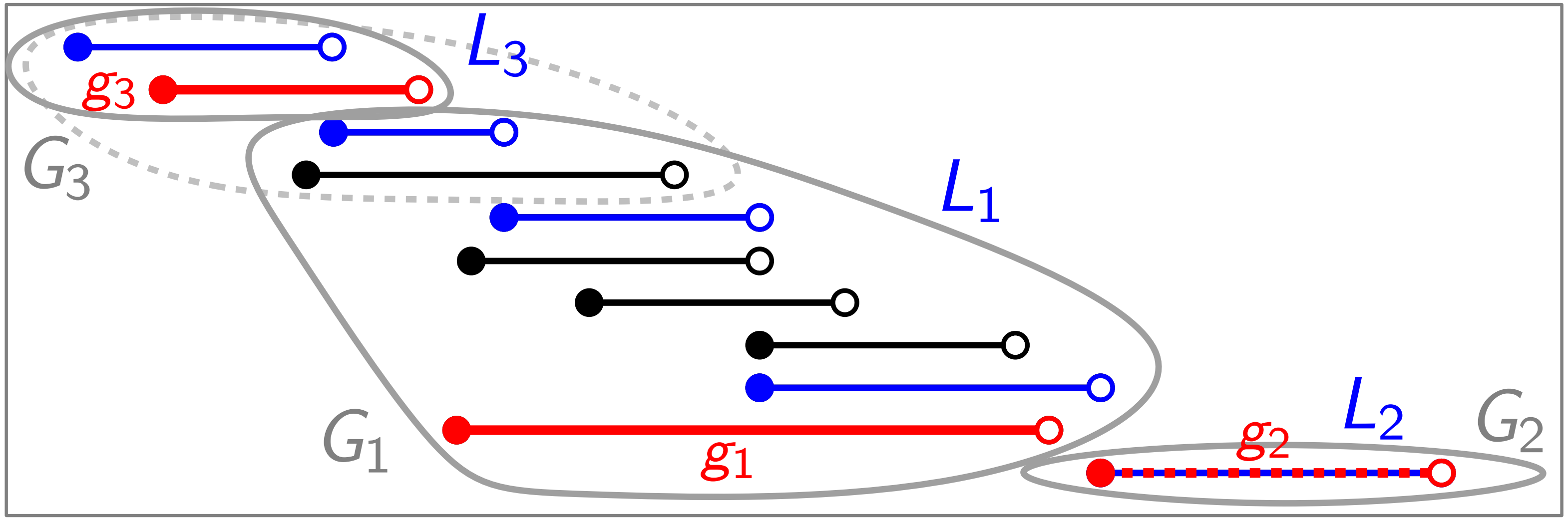
„$\subseteq$“: GA wählt so lange Intervalle aus, bis es keine mehr gibt.

Wie gut/schlecht ist der 2. GA?

Betrachte eine optimale Lösung $L \subseteq A$.

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

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



und sei $L_i = L \cap G_i$.

Dann gilt $A = G_1 \,\dot\cup\, G_2 \,\dot\cup\, \cdots \,\dot\cup\, G_k$

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

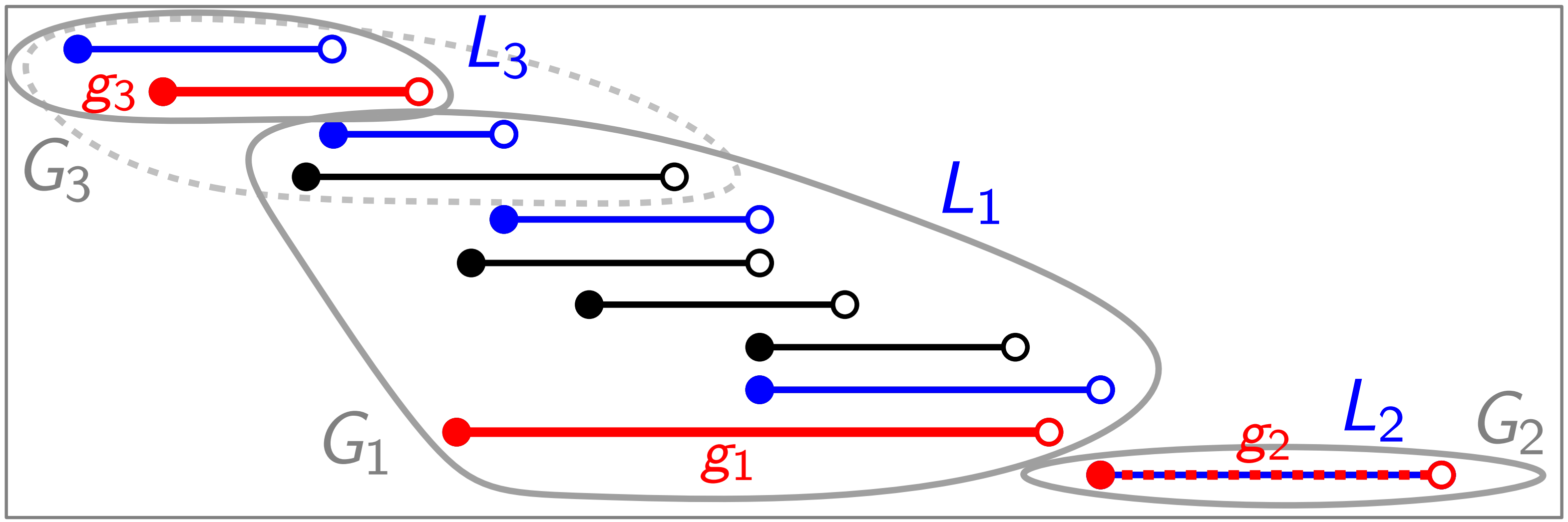
„$\supseteq$“:

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 $A = G_1 \,\dot\cup\, G_2 \,\dot\cup\, \cdots \,\dot\cup\, G_k$

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

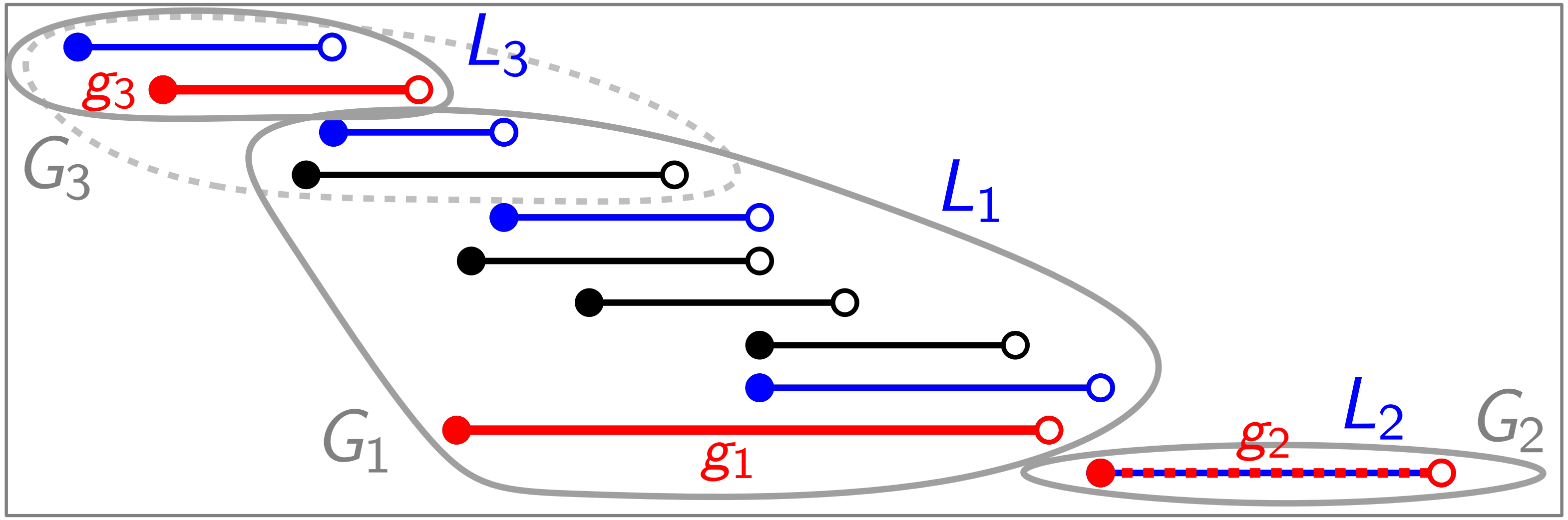
„$\supseteq$“: klar, da $G_1 \subseteq A$, $G_2 \subseteq A$, $\dots$, $G_k \subseteq A$

Wie gut/schlecht ist der 2. GA?

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.

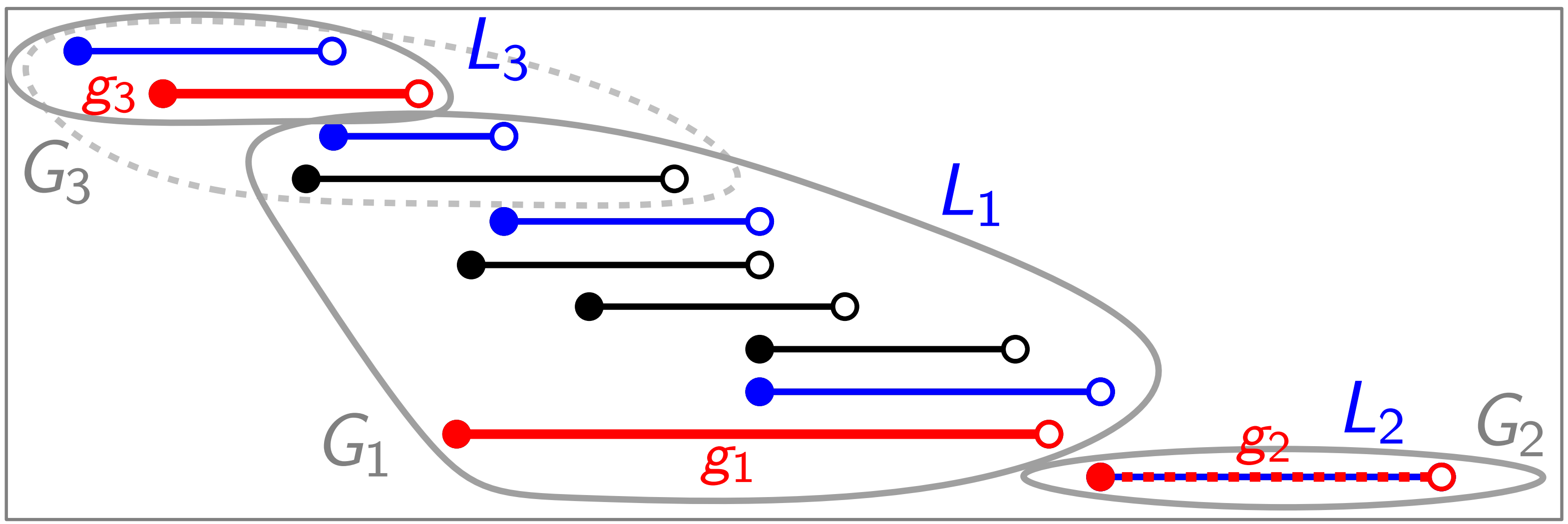
„$\supseteq$": klar, da $G_1 \subseteq A$, $G_2 \subseteq A$, …, $G_k \subseteq A$

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 $L_i = L \cap G_i$.

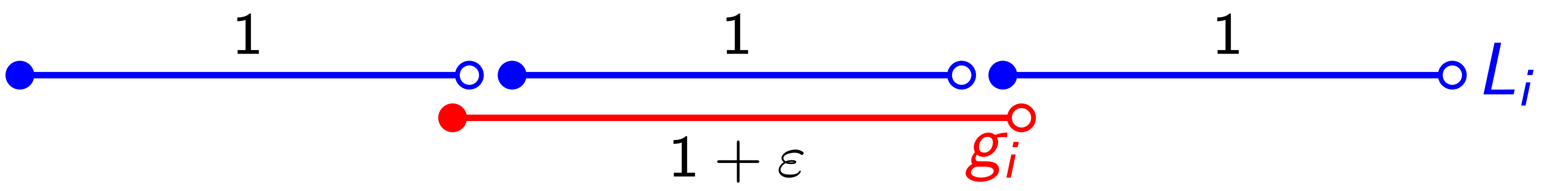
Dann gilt $A = G_1 \,\dot\cup\, G_2 \,\dot\cup\, \cdots \,\dot\cup\, G_k$ und $L = L_1 \,\dot\cup\, L_2 \,\dot\cup\, \dots \,\dot\cup\, L_k$.

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

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

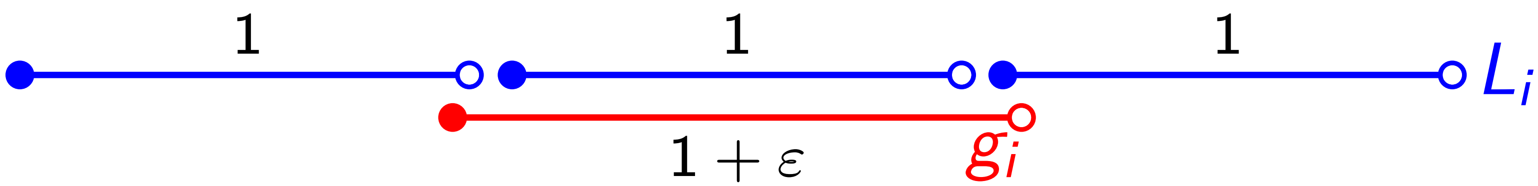
Wie gut/schlecht ist der 2. GA?

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



Wie gut/schlecht ist der 2. GA?

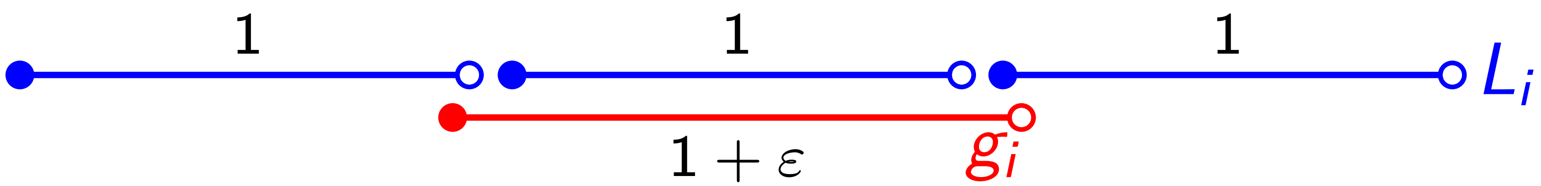
Behauptung: Für $i = 1, \ldots, k$ gilt $\ell(L_i) < 3\ell(g_i)$.



Beweis.

Wie gut/schlecht ist der 2. GA?

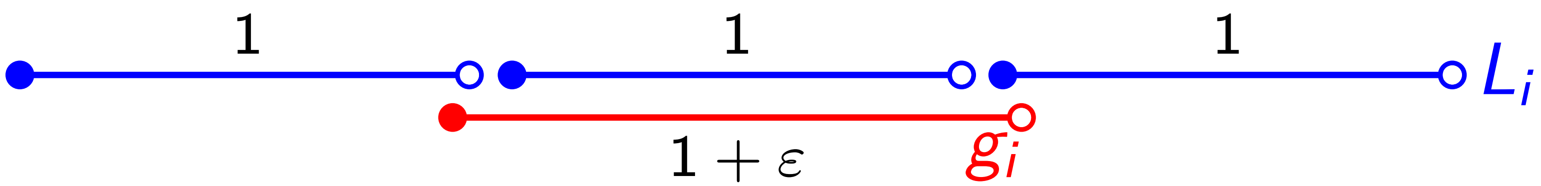
Behauptung: Für $i = 1, \ldots, k$ gilt $\ell(L_i) < 3\ell(g_i)$.



Beweis. (a) g_i ist nach Wahl ein längstes Intervall in G_i

Wie gut/schlecht ist der 2. GA?

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

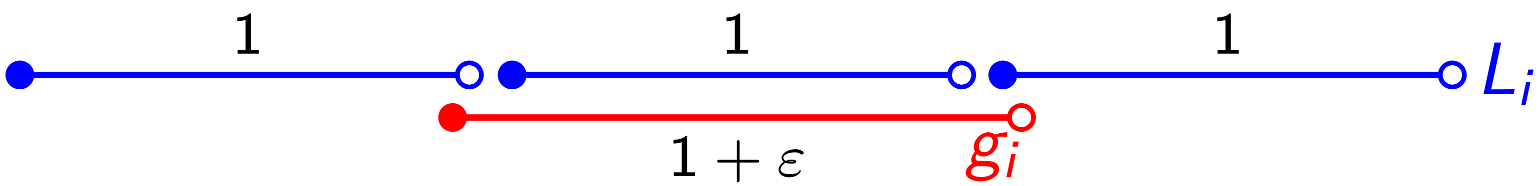


Beweis. (a) g_i ist nach Wahl ein längstes Intervall in G_i

(b) jedes $a \in L_i$ schneidet g_i

Wie gut/schlecht ist der 2. GA?

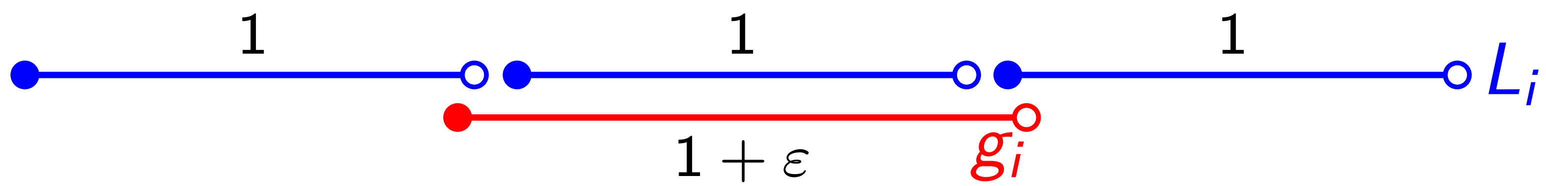
Behauptung: Für $i = 1, \dots, k$ gilt $\ell(L_i) < 3\ell(g_i)$.



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

Wie gut/schlecht ist der 2. GA?

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



Beweis. (a) g_i ist nach Wahl ein längstes Intervall in G_i

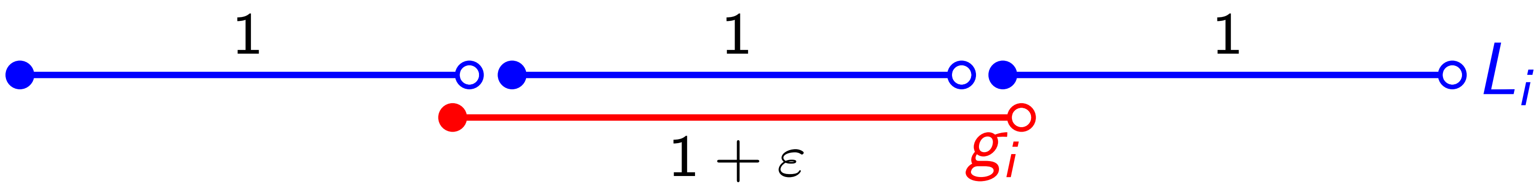
(b) jedes $a \in L_i$ schneidet g_i

(c) Intervalle in L_i sind paarweise disjunkt

$\Rightarrow$

Wie gut/schlecht ist der 2. GA?

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



Beweis. (a) g_i ist nach Wahl ein längstes Intervall in G_i

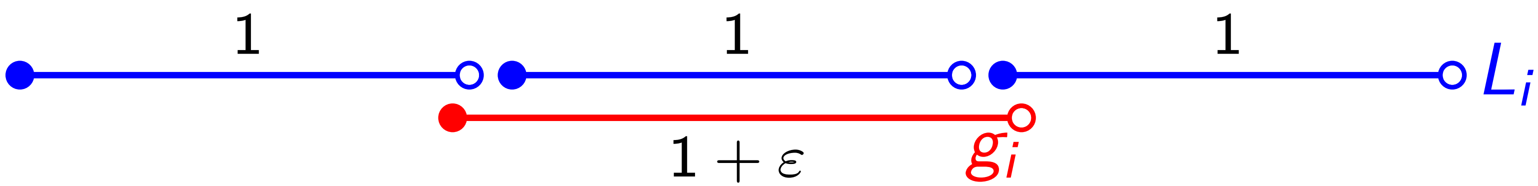
(b) jedes $a \in L_i$ schneidet g_i

(c) Intervalle in L_i sind paarweise disjunkt

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

Wie gut/schlecht ist der 2. GA?

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



Beweis. (a) g_i ist nach Wahl ein längstes Intervall in G_i

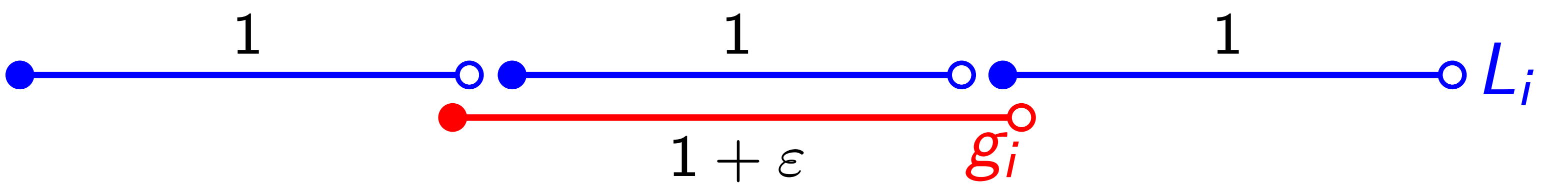
(b) jedes $a \in L_i$ schneidet g_i

(c) Intervalle in L_i sind paarweise disjunkt

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

Wie gut/schlecht ist der 2. GA?

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



Beweis. (a) g_i ist nach Wahl ein längstes Intervall in G_i

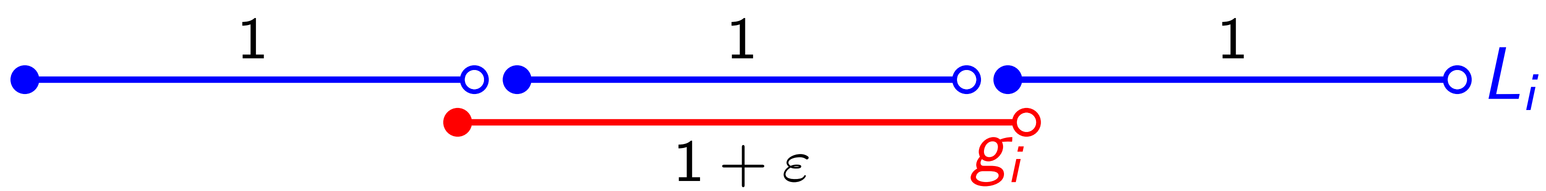
(b) jedes $a \in L_i$ schneidet g_i

(c) Intervalle in L_i sind paarweise disjunkt

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

Wie gut/schlecht ist der 2. GA?

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



Beweis. (a) g_i ist nach Wahl ein längstes Intervall in G_i

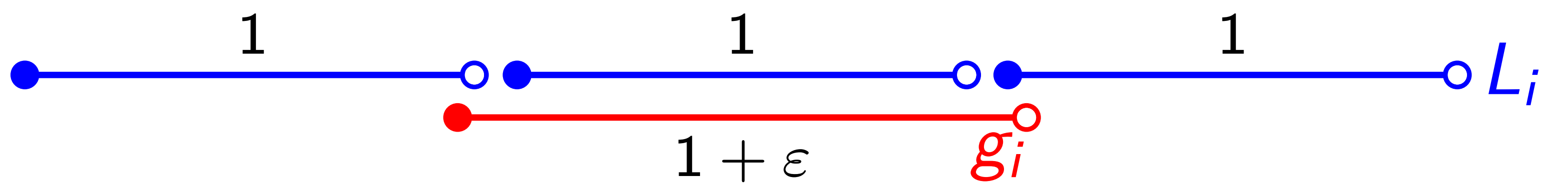
(b) jedes $a \in L_i$ schneidet g_i

(c) Intervalle in L_i sind paarweise disjunkt

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

Wie gut/schlecht ist der 2. GA?

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



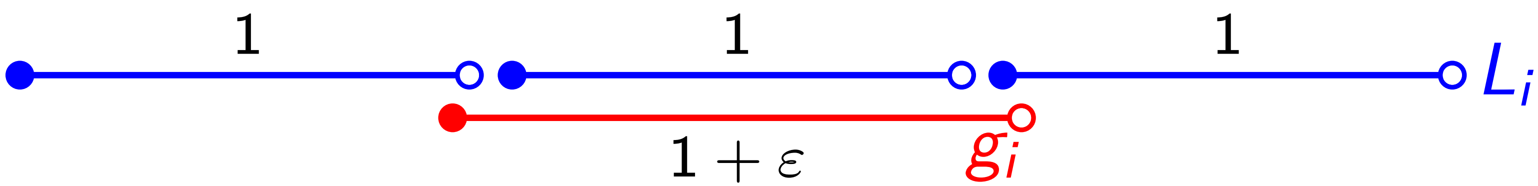
Beweis.
(a) g_i ist nach Wahl ein längstes Intervall in G_i
(b) jedes $a \in L_i$ schneidet g_i
(c) Intervalle in L_i sind paarweise disjunkt

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

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

Wie gut/schlecht ist der 2. GA?

Behauptung: Für $i = 1, \ldots, k$ 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

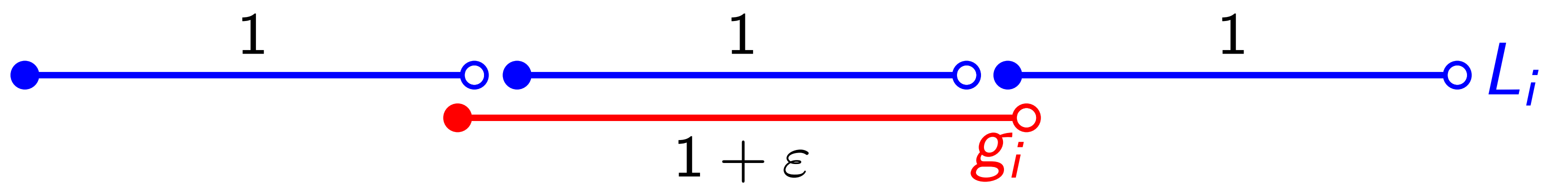
$\Rightarrow$ OPT $= \ell(L) = \sum_{i=1}^{k} \ell(L_i) < 3 \sum_{i=1}^{k} \ell(g_i) = 3\ell(G)$

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

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

Wie gut/schlecht ist der 2. GA?

Behauptung: Für $i = 1, \ldots, k$ 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

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

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

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

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

Approxi. . . *hä?*

Approxi. . . *hä?*

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

Bertrand Russell (1872–1970)

Approxi. . . *hä?*

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

Bertrand Russell (1872–1970)

Sei Π ein *Maximierungsproblem*.

Approxi. . . *hä?*

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

Bertrand Russell (1872–1970)

Sei Π ein *Maximierungsproblem*.

z.B. Ablaufplanung

Approxi. . . *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 $\mapsto \mathbb{Q}_{\geq 0}$.

Approxi. . . *hä?*

„*All exact science is dominated by the idea of approximation.*“
Bertrand Russell (1872–1970)

Sei Π ein *Maximierungsproblem*.

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

z.B. Ablaufplanung

$\zeta = \ell$

Approxi... *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. $\gamma = 1/3$

Approxi. . . *hä?*

„All exact science is dominated by the idea of approximation.“
Bertrand Russell (1872–1970)

Sei Π ein *Maximierungsproblem*.

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

Sei γ eine Zahl ≤ 1.

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

z.B. Ablaufplanung

$\zeta = \ell$

$\gamma = 1/3$

Approxi. . . *hä?*

„*All exact science is dominated by the idea of approximation.*“
Bertrand Russell (1872–1970)

Sei Π ein *Maximierungsproblem*.

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

Sei γ eine Zahl ≤ 1.

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

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

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

z.B. Ablaufplanung

$\zeta = \ell$

$\gamma = 1/3$

Approxi. . . *hä?*

„*All exact science is dominated by the idea of approximation.*“
Bertrand Russell (1872–1970)

Sei Π ein *Maximierungsproblem*.

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

Sei γ eine Zahl ≤ 1.

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

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

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

z.B. Ablaufplanung

$\zeta = \ell$

$\gamma = 1/3$

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

Approxi. . . *hä?*

„*All exact science is dominated by the idea of approximation.*“
Bertrand Russell (1872–1970)

Sei Π ein *Maximierungsproblem*.

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

Sei γ eine Zahl ≤ 1.

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

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

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

z.B. Ablaufplanung

$\zeta = \ell$

$\gamma = 1/3$

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

Approxi. . . *hä?*

„All exact science is dominated by the idea of approximation.“
Bertrand Russell (1872–1970)

Sei Π ein *Maximierungsproblem*.

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

Sei γ eine Zahl ≤ 1.

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

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

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

ζ(optimale Lösung)

z.B. Ablaufplanung

$\zeta = \ell$

$\gamma = 1/3$

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

Approxi. . . *hä?*

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

Bertrand Russell (1872–1970)

Sei Π ein *Maximierungsproblem*.

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

Sei γ eine Zahl ≤ 1.

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

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

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

ζ(optimale Lösung)

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

z.B. Ablaufplanung

$\zeta = \ell$

$\gamma = 1/3$

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

Approxi. . . *hä?*

„All exact science is dominated by the idea of approximation.“
Bertrand Russell (1872–1970)

Sei Π ein *Maximierungsproblem*.

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

Sei γ eine Zahl ≤ 1.

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

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

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

ζ(optimale Lösung)

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

z.B. Ablaufplanung

$\zeta = \ell$

$\gamma = 1/3$

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

Approxi. . . *hä?*

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

Bertrand Russell (1872–1970)

Sei Π ein *Maximierungsproblem*.

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

Sei γ eine Zahl ≤ 1.

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

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

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

ζ(optimale Lösung)

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

Größe der Instanz I

z.B. Ablaufplanung

$\zeta = \ell$

$\gamma = 1/3$

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

Approxi. . . *hä?*

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

Bertrand Russell (1872–1970)

Sei Π ein *Maximierungsproblem*.

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

Sei γ eine Zahl ≤ 1.

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

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

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

ζ(optimale Lösung)

Größe der Instanz I

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

z.B. Ablaufplanung

$\zeta = \ell$

$\gamma = 1/3$

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

$O(n \log n)$

Zurück zum dynamischen Programmieren

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

2. Wert einer optimalen Lösung rekursiv definieren

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

Zurück zum dynamischen Programmieren

```
BottomUpDPWeighted(int[] s, int[] e)
  n = s.length
  c = new int[0..n][1..n+1]
  for d = 1 to n − 1 do
    for i = 0 to (n+1) − d do
      j = i + d
      if a_i und a_j kompatibel then
        c = 0
        for k = i + 1 to j − 1 do
          c' = c[i][k] + ℓ(a_k) + c[k][j]
          if c' > c then c = c'
        c[i][j] = c
      else c[i][j] = 0
  return c[0, n+1]
```

2. Wert einer optimalen Lösung rekursiv definieren

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

Zurück zum dynamischen Programmieren

```
BottomUpDPWeighted(int[] s, int[] e)
  n = s.length
  c = new int[0..n][1..n+1]
  for d = 1 to n − 1 do
      for i = 0 to (n+1) − d do
          j = i + d
          if a_i und a_j kompatibel then
              c = 0
              for k = i+1 to j−1 do
                  c' = c[i][k] + ℓ(a_k) + 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.

2. Wert einer optimalen Lösung rekursiv definieren

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

Zurück zum dynamischen Programmieren

```
BottomUpDPWeighted(int[] s, int[] e)
  n = s.length
  c = new int[0..n][1..n+1]          // c_ij = Wert einer opt. Lsg. für A_ij
  for d = 1 to n − 1 do
    for i = 0 to (n + 1) − d do
      j = i + d
      if a_i und a_j kompatibel then
        c = 0
        for k = i + 1 to j − 1 do
          c′ = c[i][k] + ℓ(a_k) + 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.

2. Wert einer optimalen Lösung rekursiv definieren

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

Zurück zum dynamischen Programmieren

```
BottomUpDPWeighted(int[] s, int[] e)
  n = s.length
  c = new int[0..n][1..n+1]           // c_ij = Wert einer opt. Lsg. für A_ij
  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 a_i und a_j kompatibel then
        c = 0
        for k = i+1 to j − 1 do
          c' = c[i][k] + ℓ(a_k) + 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.

2. Wert einer optimalen Lösung rekursiv definieren

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

Zurück zum dynamischen Programmieren

```
BottomUpDPWeighted(int[] s, int[] e)
  n = s.length
  c = new int[0..n][1..n + 1]              // c_ij = Wert einer opt. Lsg. für A_ij
  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 a_i und a_j kompatibel then   // falls a_i ∩ a_j = ∅
              c = 0
              for k = i + 1 to j − 1 do
                  c' = c[i][k] + ℓ(a_k) + 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.

2. Wert einer optimalen Lösung rekursiv definieren

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

Zurück zum dynamischen Programmieren

```
BottomUpDPWeighted(int[] s, int[] e)
  n = s.length
  c = new int[0..n][1..n+1]          // c_ij = Wert einer opt. Lsg. für A_ij
  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 a_i und a_j kompatibel then   // falls a_i ∩ a_j = ∅
        c = 0
        for k = i+1 to j−1 do
          c' = c[i][k] + ℓ(a_k) + 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?

2. Wert einer optimalen Lösung rekursiv definieren

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

Zurück zum dynamischen Programmieren

BottomUpDPWeighted(int[] s, int[] e)
- $n = s.length$
- $c =$ new int[0..n][1..$n+1$] // c_{ij} = Wert einer opt. Lsg. für A_{ij}
- **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** a_i und a_j kompatibel **then** // falls $a_i \cap a_j = \emptyset$
      - $c = 0$
      - **for** $k = i+1$ **to** $j-1$ **do**
        - $c' = c[i][k] + \ell(a_k) + c[k][j]$
        - **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? $O(n^3)$

2. Wert einer optimalen Lösung rekursiv definieren

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

Zurück zum dynamischen Programmieren

```
BottomUpDPWeighted(int[] s, int[] e)
  n = s.length
  c = new int[0..n][1..n+1]          // c_ij = Wert einer opt. Lsg. für A_ij
  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 a_i und a_j kompatibel then // falls a_i ∩ a_j = ∅
        c = 0
        for k = i+1 to j−1 do
          c' = c[i][k] + ℓ(a_k) + 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? $O(n^3)$

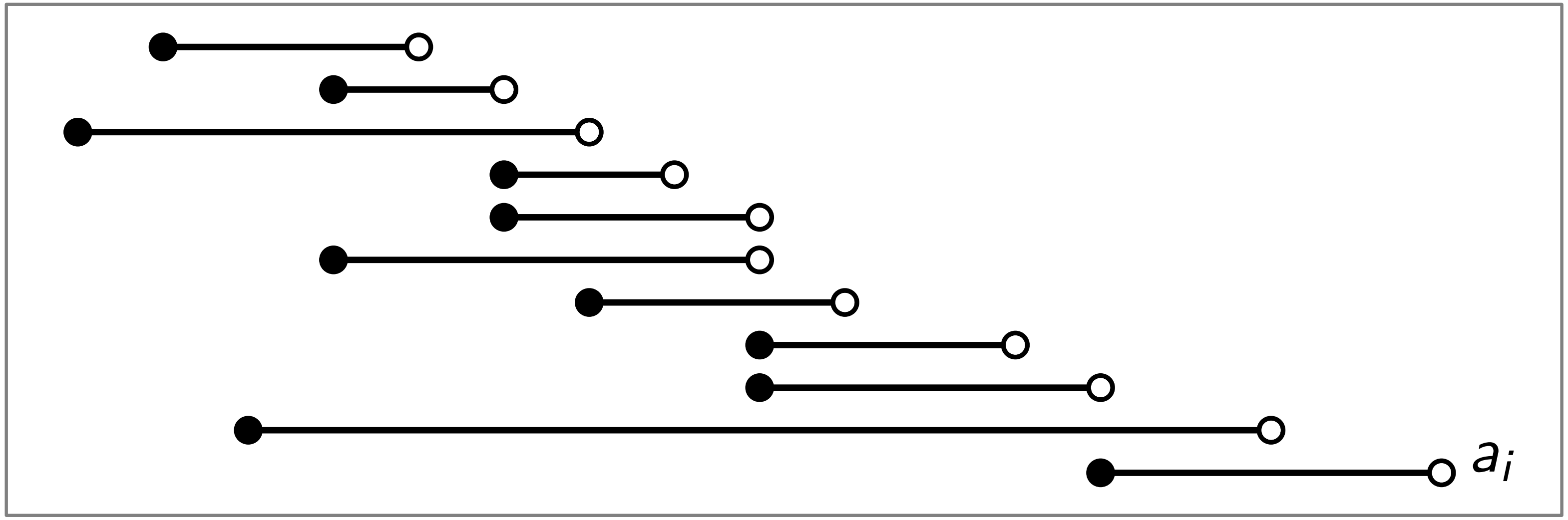
Aber warum verzweigen wir hier zweimal?

2. Wert einer optimalen Lösung rekursiv definieren

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

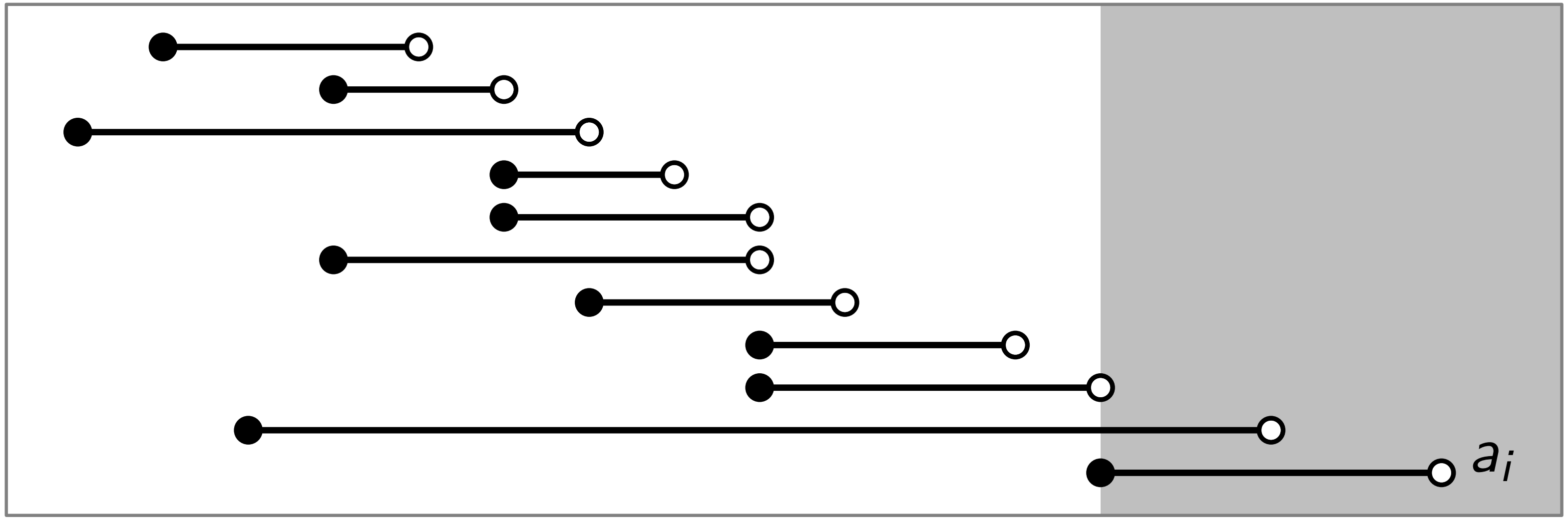
Dynamisches Programmieren, aber einfacher

Für $i = 1, \ldots, n$ 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 $A_{n+1} = A$.)



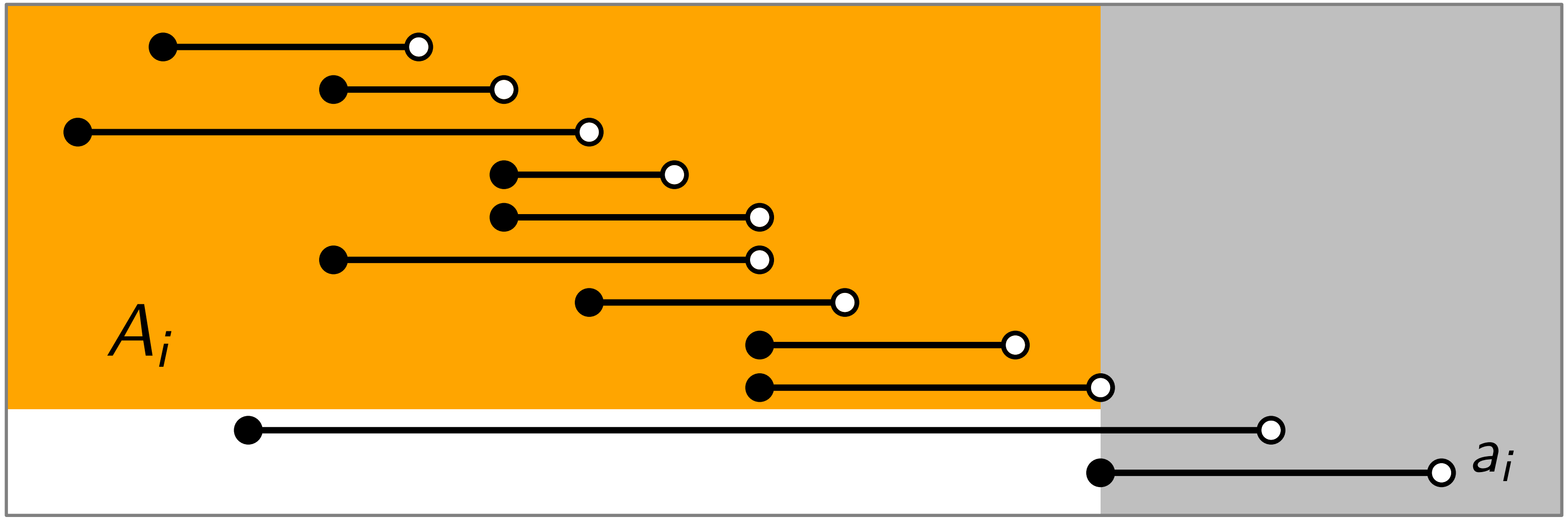
Dynamisches Programmieren, aber einfacher

Für $i = 1, \ldots, n$ 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 $A_{n+1} = A$.)



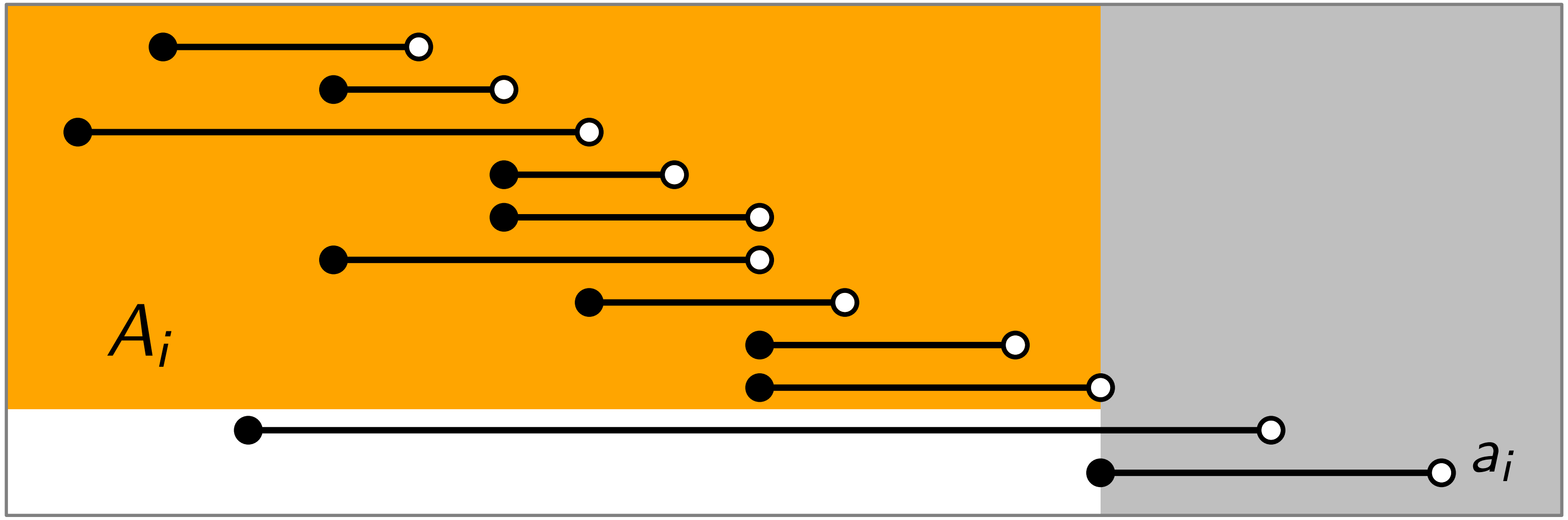
Dynamisches Programmieren, aber einfacher

Für $i = 1, \ldots, n$ 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 $A_{n+1} = A$.)



Dynamisches Programmieren, aber einfacher

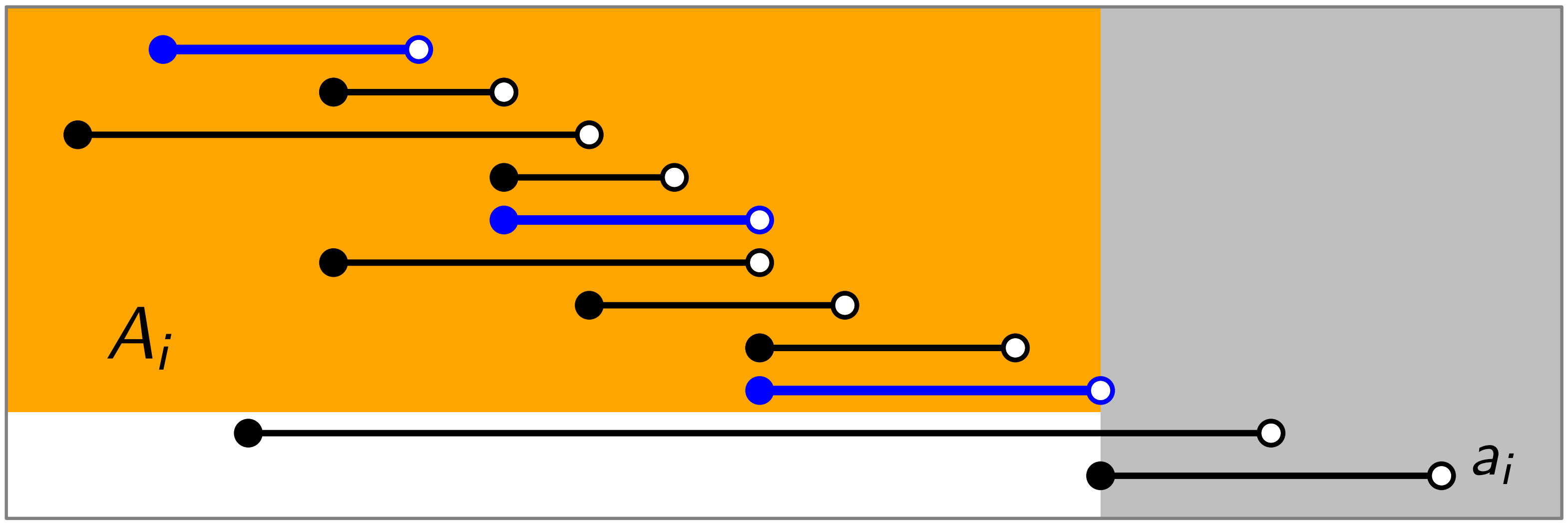
Für $i = 1, \ldots, n$ sei $A_i = \{a_j \in A \mid e_j \leq s_i\}$ die Menge aller Intervalle in A, die enden, bevor a_i beginnt. (Setze $A_{n+1} = A$.)



Eine optimale Lösung für A_i besteht aus:
– *einem* letzten Intervall a_k und
– einer optimalen Lösung für A_k.

Dynamisches Programmieren, aber einfacher

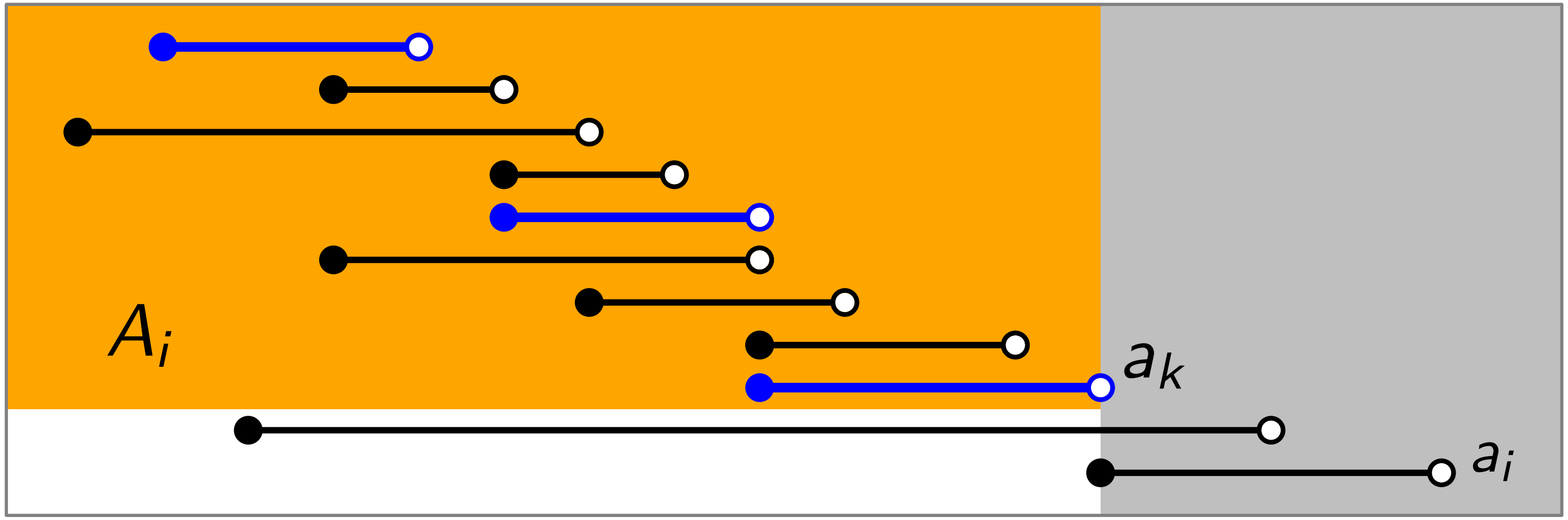
Für $i = 1, \ldots, n$ sei $A_i = \{a_j \in A \mid e_j \leq s_i\}$ die Menge aller Intervalle in A, die enden, bevor a_i beginnt. (Setze $A_{n+1} = A$.)



Eine optimale Lösung für A_i besteht aus:
– *einem* letzten Intervall a_k und
– einer optimalen Lösung für A_k.

Dynamisches Programmieren, aber einfacher

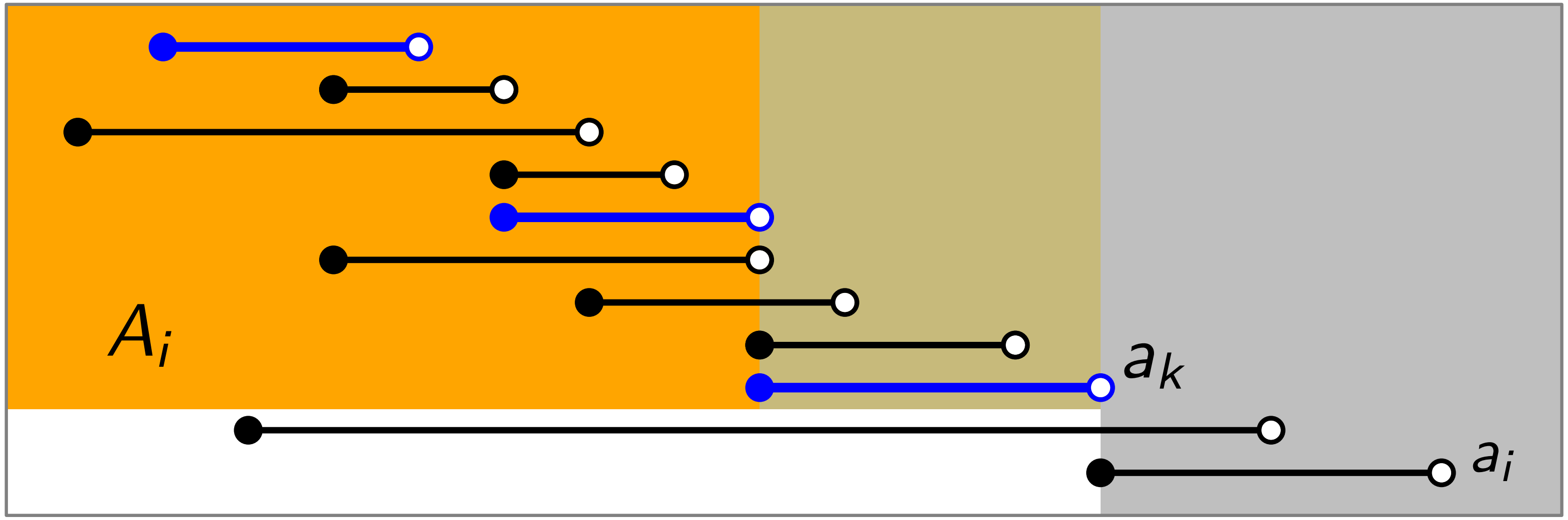
Für $i = 1, \ldots, n$ sei $A_i = \{a_j \in A \mid e_j \leq s_i\}$ die Menge aller Intervalle in A, die enden, bevor a_i beginnt. (Setze $A_{n+1} = A$.)



Eine optimale Lösung für A_i besteht aus:
– *einem* letzten Intervall a_k und
– einer optimalen Lösung für A_k.

Dynamisches Programmieren, aber einfacher

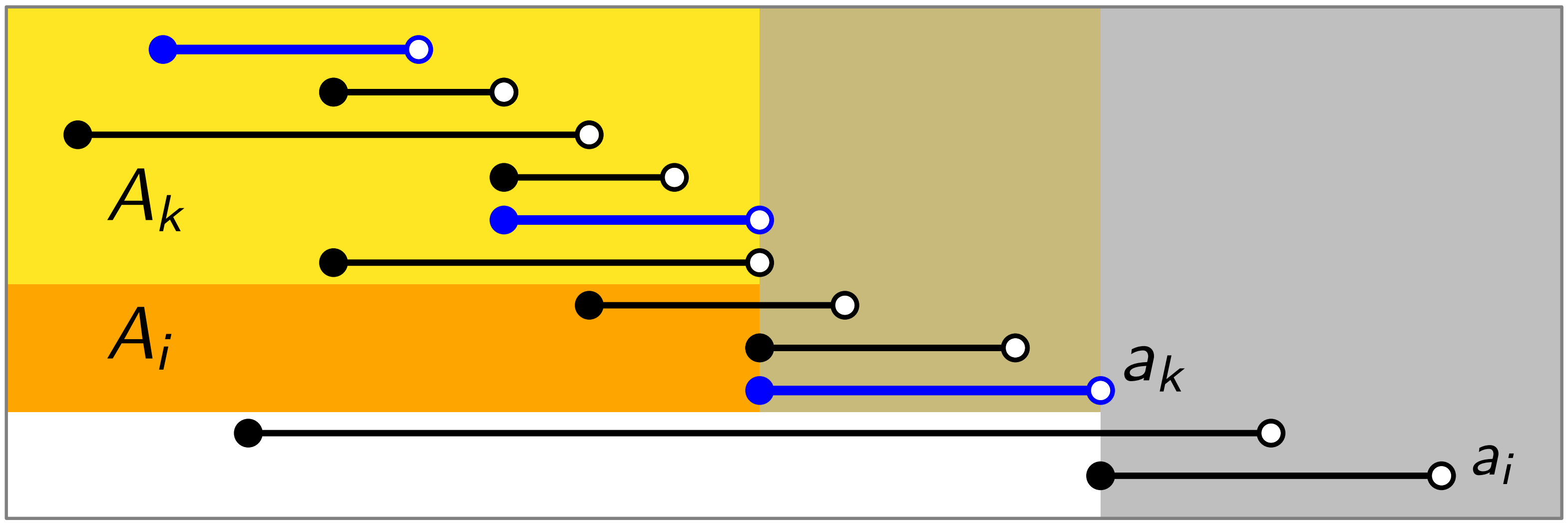
Für $i = 1, \ldots, n$ sei $A_i = \{a_j \in A \mid e_j \leq s_i\}$ die Menge aller Intervalle in A, die enden, bevor a_i beginnt. (Setze $A_{n+1} = A$.)



Eine optimale Lösung für A_i besteht aus:
– *einem* letzten Intervall a_k und
– einer optimalen Lösung für A_k.

Dynamisches Programmieren, aber einfacher

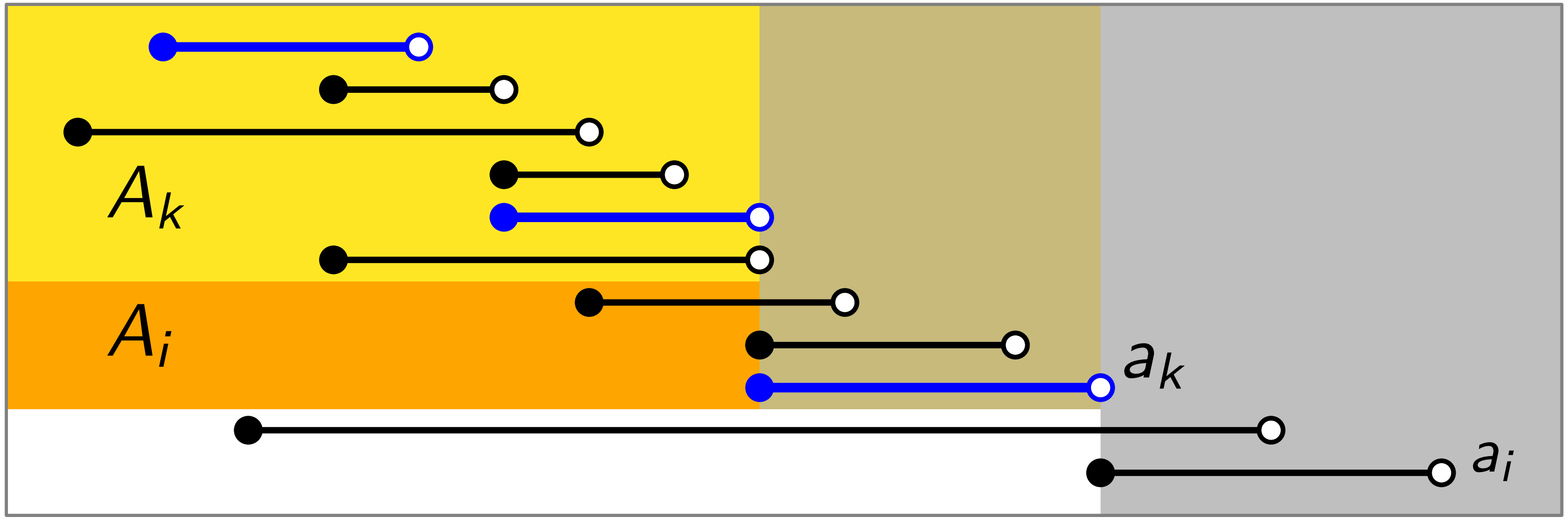
Für $i = 1, \ldots, n$ sei $A_i = \{a_j \in A \mid e_j \leq s_i\}$ die Menge aller Intervalle in A, die enden, bevor a_i beginnt. (Setze $A_{n+1} = A$.)



Eine optimale Lösung für A_i besteht aus:
– *einem* letzten Intervall a_k und
– einer optimalen Lösung für A_k.

Dynamisches Programmieren, aber einfacher

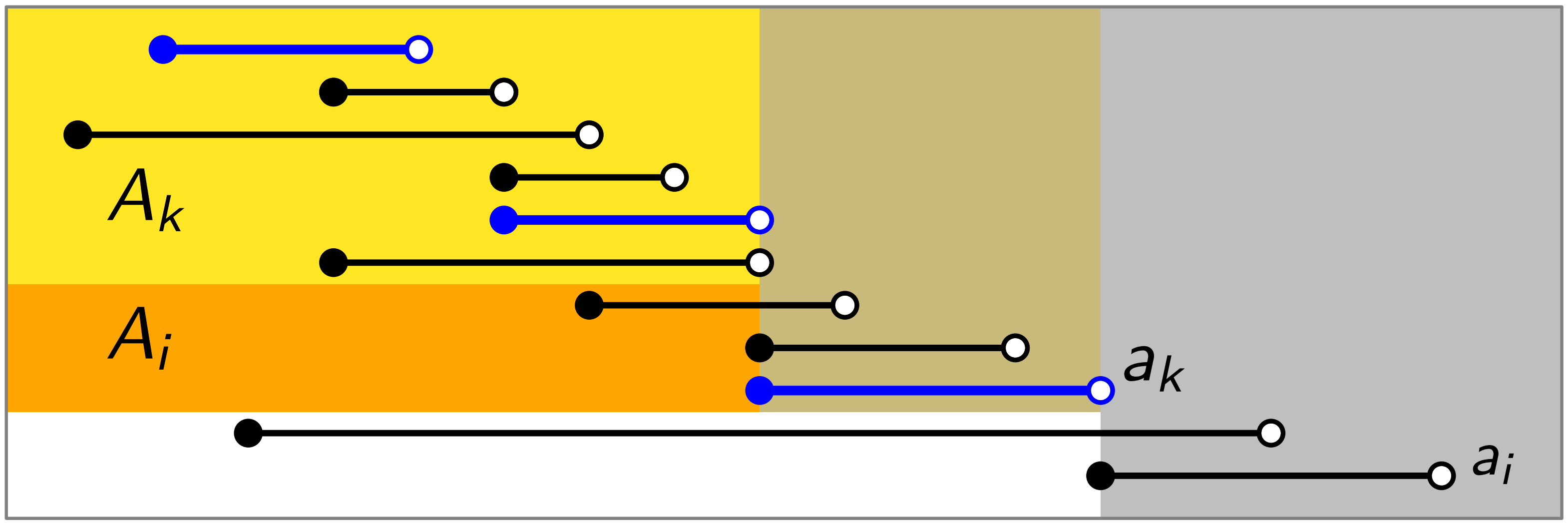
Für $i = 1, \ldots, n$ sei $A_i = \{a_j \in A \mid e_j \leq s_i\}$ die Menge aller Intervalle in A, die enden, bevor a_i beginnt. (Setze $A_{n+1} = A$.)



$$\left.\begin{array}{l}\text{Eine optimale Lösung für } A_i \text{ besteht aus:}\\ \text{– \textit{einem} letzten Intervall } a_k \text{ und}\\ \text{– einer optimalen Lösung für } A_k.\end{array}\right\}\ \begin{array}{l}\text{optimale}\\ \text{Teilstruktur!}\end{array}$$

Dynamisches Programmieren, aber einfacher

Für $i = 1, \ldots, n$ 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 $A_{n+1} = A$.)



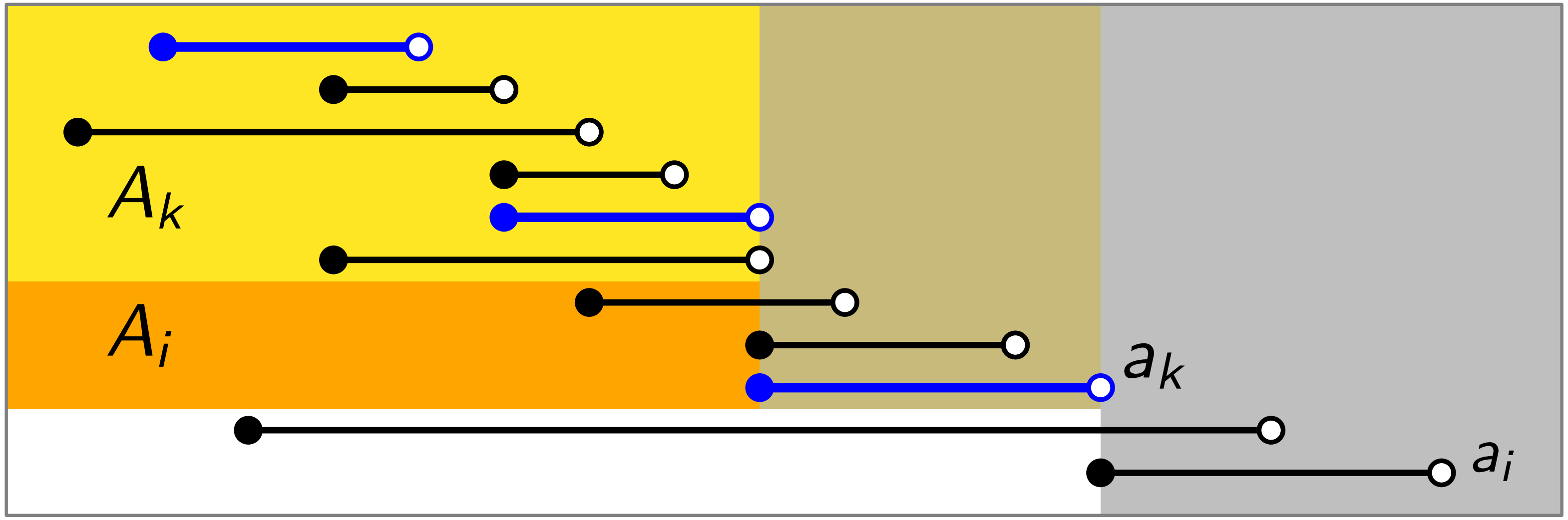
Eine optimale Lösung für A_i besteht aus:
– *einem* letzten Intervall a_k und
– einer optimalen Lösung für A_k.

$\Big\}$ optimale Teilstruktur!

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

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:

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

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 $A_{n+1} = A$.

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 $c_1, \ldots, c_{n+1}$ zu berechnen.

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 $c_1, \ldots, c_{n+1}$ zu berechnen.

Laufzeit?

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 $c_1, \ldots, c_{n+1}$ zu berechnen.

Laufzeit? $O(n^2)$

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 $c_1, \ldots, c_{n+1}$ zu berechnen.

Laufzeit? $O(n^2)$ **Work out the details!**

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 $c_1, \ldots, c_{n+1}$ zu berechnen.

Laufzeit? $O(n^2)$ **Work out the details!**

Resultate:

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.

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 $c_1, \ldots, c_{n+1}$ 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.

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 $c_1, \ldots, c_{n+1}$ 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!