GREEDY ALGORITHMEN UND HEURISTIKEN

Minimaler Spannbaum

Minimaler Spannbaum

Eingabe: ein Graph $G(V, E)$ und Kantengewichtung w_e für jede Kante $e \in E$

Minimaler Spannbaum

Eingabe: ein Graph $G(V, E)$ und Kantengewichtung w_e für jede Kante $e \in E$

Ausgabe: Bestimme einen *Spannbaum* mit minimalem Gewicht

(Spannbaum: ein Baum (kreisfreier Graph) $T(V', E')$ mit $V' = V$ und $E' \subseteq E$)

Minimaler Spannbaum

Eingabe: ein Graph $G(V, E)$ und Kantengewichtung w_e für jede Kante $e \in E$

Ausgabe: Bestimme einen *Spannbaum* mit minimalem Gewicht

(Spannbaum: ein Baum (kreisfreier Graph) $T(V', E')$ mit $V' = V$ und $E' \subseteq E$)

Kruskal's Greedy Algorithmus (optimal)

Minimaler Spannbaum

Eingabe: ein Graph $G(V, E)$ und Kantengewichtung w_e für jede Kante $e \in E$

Ausgabe: Bestimme einen *Spannbaum* mit minimalem Gewicht

(Spannbaum: ein Baum (kreisfreier Graph) $T(V', E')$ mit $V' = V$ und $E' \subseteq E$)

Kruskal's Greedy Algorithmus (optimal)

Setze $E' = \emptyset$

WHILE $E \neq \emptyset$ do

- nimm $e \in E$ mit minimalem w_e , setze $E := E \setminus \{e\}$

Minimaler Spannbaum

Eingabe: ein Graph $G(V, E)$ und Kantengewichtung w_e für jede Kante $e \in E$

Ausgabe: Bestimme einen *Spannbaum* mit minimalem Gewicht

(Spannbaum: ein Baum (kreisfreier Graph) $T(V', E')$ mit $V' = V$ und $E' \subseteq E$)

Kruskal's Greedy Algorithmus (optimal)

Setze $E' = \emptyset$

WHILE $E \neq \emptyset$ do

- nimm $e \in E$ mit minimalem w_e , setze $E := E \setminus \{e\}$
- falls $E' \cup \{e\}$ keinen Kreis enthält $E' := E' \cup \{e\}$

Minimaler Spannbaum

Eingabe: ein Graph $G(V, E)$ und Kantengewichtung w_e für jede Kante $e \in E$

Ausgabe: Bestimme einen *Spannbaum* mit minimalem Gewicht

(Spannbaum: ein Baum (kreisfreier Graph) $T(V', E')$ mit $V' = V$ und $E' \subseteq E$)

Kruskal's Greedy Algorithmus (optimal)

Setze $E' = \emptyset$

WHILE $E \neq \emptyset$ do

- nimm $e \in E$ mit minimalem w_e , setze $E := E \setminus \{e\}$
- falls $E' \cup \{e\}$ keinen Kreis enthält $E' := E' \cup \{e\}$
  sonst verwerfe e

(Union-Find Datenstruktur wird verwendet;

Minimaler Spannbaum

Eingabe: ein Graph $G(V, E)$ und Kantengewichtung w_e für jede Kante $e \in E$

Ausgabe: Bestimme einen *Spannbaum* mit minimalem Gewicht

(Spannbaum: ein Baum (kreisfreier Graph) $T(V', E')$ mit $V' = V$ und $E' \subseteq E$)

Kruskal's Greedy Algorithmus (optimal)

Setze $E' = \emptyset$

WHILE $E \neq \emptyset$ do

- nimm $e \in E$ mit minimalem w_e , setze $E := E \setminus \{e\}$
- falls $E' \cup \{e\}$ keinen Kreis enthält $E' := E' \cup \{e\}$
  sonst verwerfe e

(Union-Find Datenstruktur wird verwendet; Laufzeit: $\mathcal{O}(n \log n)$)

Definition: Spannbaum und Steiner-Baum

Definition: Spannbaum und Steiner-Baum

Def: Ein Spannbaum für $G(V, E)$ ist ein Teilgraph $B(V, E')$ von G, der ein Baum ist, und *alle Knoten in* V überdeckt.

Definition: Spannbaum und Steiner-Baum

Def: Ein Spannbaum für $G(V, E)$ ist ein Teilgraph $B(V, E')$ von G, der ein Baum ist, und *alle Knoten in* V überdeckt.

Def: Ein Steiner-Baum für $G(V, E)$ und für eine gegebene Menge von Terminals $T \subseteq V$ ist ein Teilgraph $S(V'E')$ von G, der ein Baum ist, und *alle Knoten in* T überdeckt.

Eine unpräzise Definition

Eine unpräzise Definition

Ein Greedy Algorithmus bestimmt eine Lösung iterativ;

jedes mal wird eine Entscheidung getroffen,

die 'lokal' am vielversprechendsten ist.

Getroffene Entscheidungen werden nicht revidiert.

Definition: Priority Algorithmen

(Konkreter definierbare Greedy Algorithmen)

Definition: Priority Algorithmen

(Konkreter definierbare Greedy Algorithmen)

- Die Eingabe besteht aus *Datenelementen*.

Definition: Priority Algorithmen

(Konkreter definierbare Greedy Algorithmen)

- Die Eingabe besteht aus *Datenelementen*.
- Es gibt eine vollständige Ordnung auf *allen möglichen* Datenelementen.

Definition: Priority Algorithmen

(Konkreter definierbare Greedy Algorithmen)

- Die Eingabe besteht aus *Datenelementen*.
- Es gibt eine vollständige Ordnung auf *allen möglichen* Datenelementen.
- In jedem Schritt erhält der Algorithmus das Datenelement aktuell höchster Priorität laut dieser Ordnung, und

Definition: Priority Algorithmen

(Konkreter definierbare Greedy Algorithmen)

- Die Eingabe besteht aus *Datenelementen*.
- Es gibt eine vollständige Ordnung auf *allen möglichen* Datenelementen.
- In jedem Schritt erhält der Algorithmus das Datenelement aktuell höchster Priorität laut dieser Ordnung, und
- trifft eine nicht-revidierbare Entscheidung für dieses Datenelement.

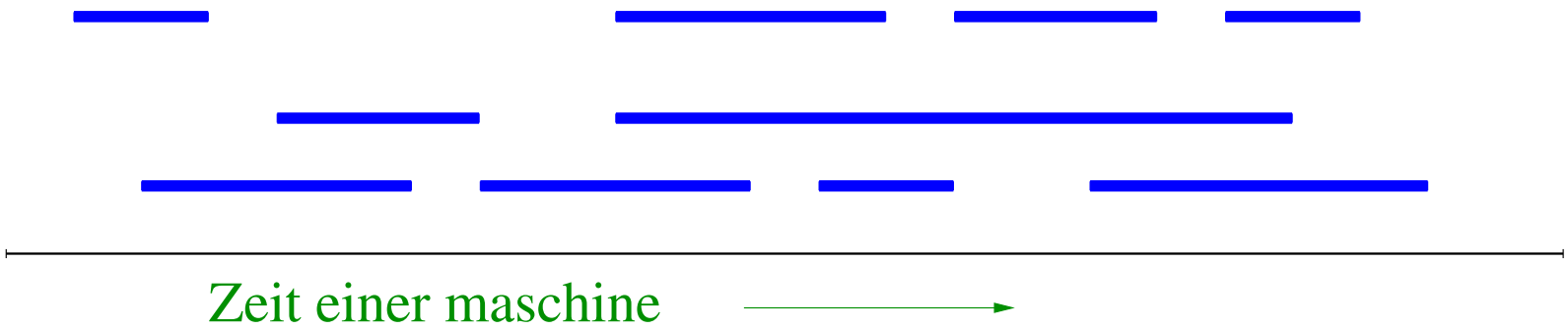
Definition: Priority Algorithmen

(Konkreter definierbare Greedy Algorithmen)

- Die Eingabe besteht aus *Datenelementen*.
- Es gibt eine vollständige Ordnung auf *allen möglichen* Datenelementen.
- In jedem Schritt erhält der Algorithmus das Datenelement aktuell höchster Priorität laut dieser Ordnung, und
- trifft eine nicht-revidierbare Entscheidung für dieses Datenelement.

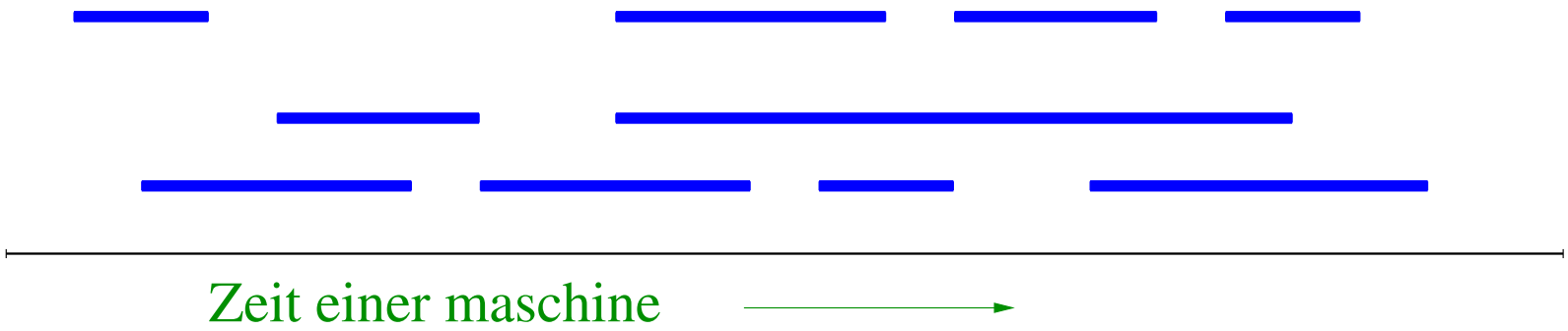
Intervall-Scheduling

Intervall-Scheduling



Eingabe: n Aufgaben $A_1, A_2, \ldots, A_n$
mit Startpunkten $s_1, s_2, \ldots, s_n$, und
Endpunkten $e_1, e_2, \ldots, e_n$

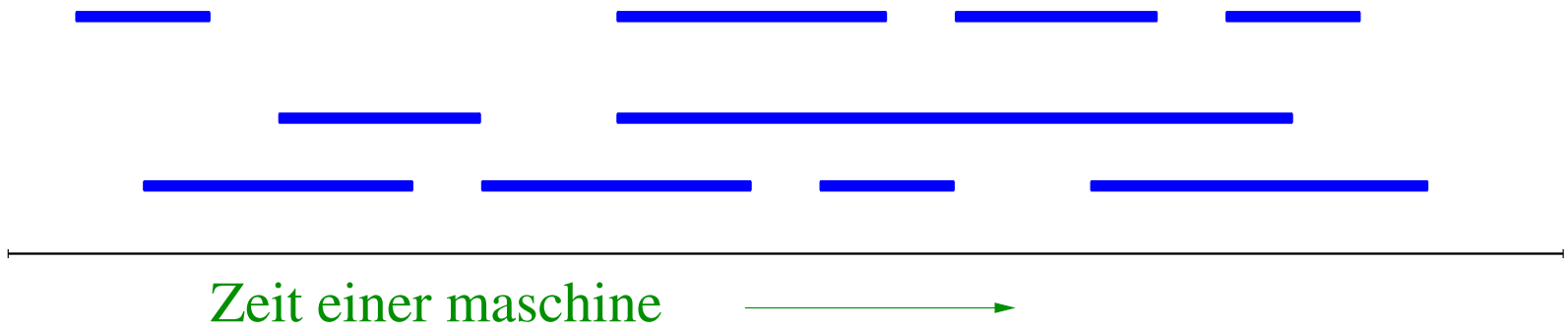
Intervall-Scheduling



Eingabe: n Aufgaben $A_1, A_2, \ldots, A_n$
mit Startpunkten $s_1, s_2, \ldots, s_n$, und
Endpunkten $e_1, e_2, \ldots, e_n$

Ausgabe: Maximiere die Anzahl der Aufgaben die ohne Überlappen ausführbar sind.

Intervall-Scheduling



Eingabe: n Aufgaben $A_1, A_2, \ldots, A_n$
mit Startpunkten $s_1, s_2, \ldots, s_n$, und
Endpunkten $e_1, e_2, \ldots, e_n$

Ausgabe: Maximiere die Anzahl der Aufgaben die ohne Überlappen ausführbar sind.

$$(A_{i_1}, A_{i_2}, \ldots, A_{i_m} \text{ s.d. } [s_{i_k}, e_{i_k}) \cap [s_{i_l} \cap e_{i_l}) = \emptyset \quad (k \neq l))$$

Greedy Intervall-Scheduling

Seien $e^1 \leq e^2 \leq \ldots \leq e^n$ die sortierten Endpunkte,

Greedy Intervall-Scheduling

Seien $e^1 \leq e^2 \leq \ldots \leq e^n$ die sortierten Endpunkte,

entsprehend den Aufgaben $A^1, A^2, \ldots, A^n$

und Endpunkten $s^1, s^2, \ldots, s^n$.

Greedy Intervall-Scheduling

Seien $e^1 \leq e^2 \leq \ldots \leq e^n$ die sortierten Endpunkte,

entsprehend den Aufgaben $A^1, A^2, \ldots, A^n$

und Endpunkten $s^1, s^2, \ldots, s^n$.

Setze $J := \emptyset$ (die Menge der auszuführenden Jobs)
und $E := 0$ (aktueller Endpunkt)

Greedy Intervall-Scheduling

Seien $e^1 \leq e^2 \leq \ldots \leq e^n$ die sortierten Endpunkte,

entsprehend den Aufgaben $A^1, A^2, \ldots, A^n$

und Endpunkten $s^1, s^2, \ldots, s^n$.

Setze $J := \emptyset$ (die Menge der auszuführenden Jobs)
und $E := 0$ (aktueller Endpunkt)

FOR $k = 1$ TO n

if $s^k \geq E$ then
$J := J \cup \{A^k\}$
$E := e^k$

Greedy Intervall-Scheduling

Seien $e^1 \leq e^2 \leq \ldots \leq e^n$ die sortierten Endpunkte,

entsprehend den Aufgaben $A^1, A^2, \ldots, A^n$

und Endpunkten $s^1, s^2, \ldots, s^n$.

Setze $J := \emptyset$ (die Menge der auszuführenden Jobs)
und $E := 0$ (aktueller Endpunkt)

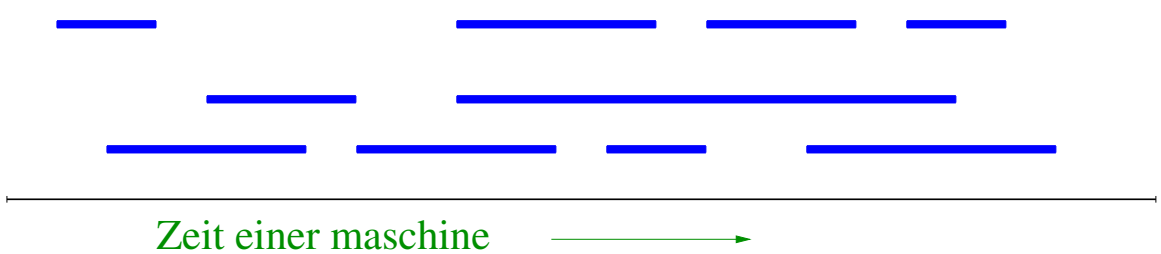
FOR $k = 1$ TO n

if $s^k \geq E$ then

$J := J \cup \{A^k\}$

$E := e^k$

else verwerfe A^k



Greedy Intervall-Scheduling

Seien $e^1 \leq e^2 \leq \ldots \leq e^n$ die sortierten Endpunkte,

entsprehend den Aufgaben $A^1, A^2, \ldots, A^n$

und Endpunkten $s^1, s^2, \ldots, s^n$.

Setze $J := \emptyset$ (die Menge der auszuführenden Jobs)
und $E := 0$ (aktueller Endpunkt)

FOR $k = 1$ TO n

if $s^k \geq E$ then

$J := J \cup \{A^k\}$

$E := e^k$

else verwerfe A^k

Greedy Intervall-Scheduling

Seien $e^1 \leq e^2 \leq \ldots \leq e^n$ die sortierten Endpunkte,

entsprehend den Aufgaben $A^1, A^2, \ldots, A^n$

und Endpunkten $s^1, s^2, \ldots, s^n$.

Setze $J := \emptyset$ (die Menge der auszuführenden Jobs)
und $E := 0$ (aktueller Endpunkt)

FOR $k = 1$ TO n

if $s^k \geq E$ then

$J := J \cup \{A^k\}$

$E := e^k$

else verwerfe A^k

Greedy Intervall-Scheduling

Seien $e^1 \leq e^2 \leq \ldots \leq e^n$ die sortierten Endpunkte,

entsprehend den Aufgaben $A^1, A^2, \ldots, A^n$

und Endpunkten $s^1, s^2, \ldots, s^n$.

Setze $J := \emptyset$ (die Menge der auszuführenden Jobs)
und $E := 0$ (aktueller Endpunkt)

FOR $k = 1$ TO n

if $s^k \geq E$ then

$J := J \cup \{A^k\}$

$E := e^k$

else verwerfe A^k

Greedy Intervall-Scheduling

Seien $e^1 \leq e^2 \leq \ldots \leq e^n$ die sortierten Endpunkte,

entsprehend den Aufgaben $A^1, A^2, \ldots, A^n$

und Endpunkten $s^1, s^2, \ldots, s^n$.

Setze $J := \emptyset$ (die Menge der auszuführenden Jobs)
und $E := 0$ (aktueller Endpunkt)

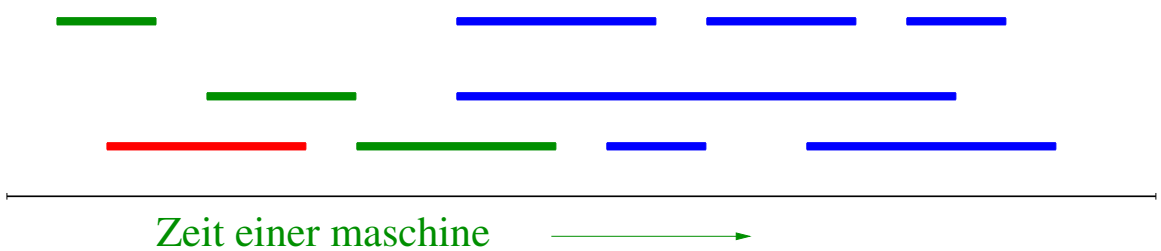
FOR $k = 1$ TO n

if $s^k \geq E$ then

$J := J \cup \{A^k\}$

$E := e^k$

else verwerfe A^k



Greedy Intervall-Scheduling

Seien $e^1 \leq e^2 \leq \ldots \leq e^n$ die sortierten Endpunkte,

entsprehend den Aufgaben $A^1, A^2, \ldots, A^n$

und Endpunkten $s^1, s^2, \ldots, s^n$.

Setze $J := \emptyset$ (die Menge der auszuführenden Jobs)
und $E := 0$ (aktueller Endpunkt)

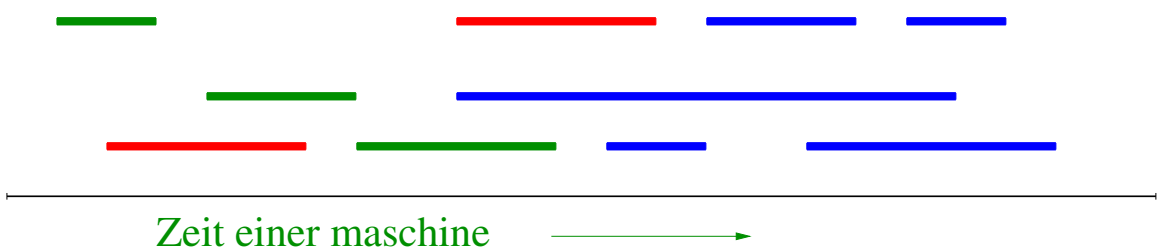
FOR $k = 1$ TO n

if $s^k \geq E$ then

$J := J \cup \{A^k\}$

$E := e^k$

else verwerfe A^k



Greedy Intervall-Scheduling

Seien $e^1 \leq e^2 \leq \ldots \leq e^n$ die sortierten Endpunkte,

entsprehend den Aufgaben $A^1, A^2, \ldots, A^n$

und Endpunkten $s^1, s^2, \ldots, s^n$.

Setze $J := \emptyset$ (die Menge der auszuführenden Jobs)
und $E := 0$ (aktueller Endpunkt)

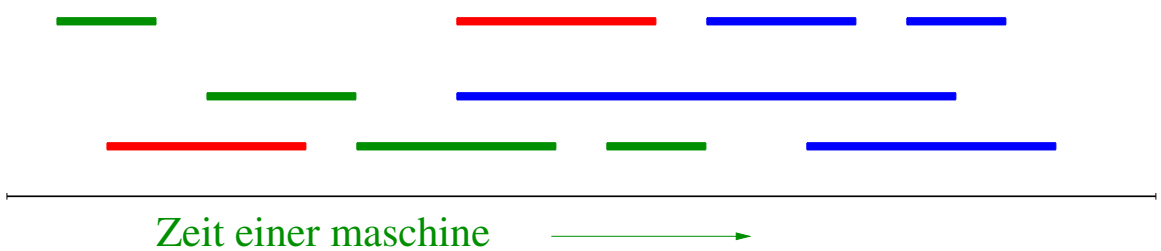
FOR $k = 1$ TO n

if $s^k \geq E$ then

$J := J \cup \{A^k\}$

$E := e^k$

else verwerfe A^k



Greedy Intervall-Scheduling

Seien $e^1 \leq e^2 \leq \ldots \leq e^n$ die sortierten Endpunkte,

entsprehend den Aufgaben $A^1, A^2, \ldots, A^n$

und Endpunkten $s^1, s^2, \ldots, s^n$.

Setze $J := \emptyset$ (die Menge der auszuführenden Jobs)
und $E := 0$ (aktueller Endpunkt)

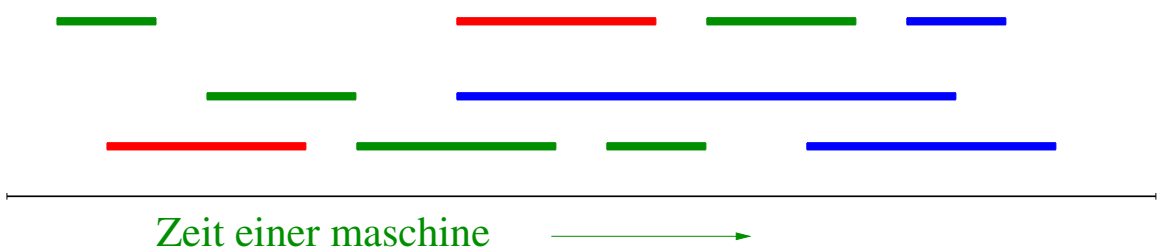
FOR $k = 1$ TO n

if $s^k \geq E$ then

$J := J \cup \{A^k\}$

$E := e^k$

else verwerfe A^k



Greedy Intervall-Scheduling

Seien $e^1 \leq e^2 \leq \ldots \leq e^n$ die sortierten Endpunkte,

entsprehend den Aufgaben $A^1, A^2, \ldots, A^n$

und Endpunkten $s^1, s^2, \ldots, s^n$.

Setze $J := \emptyset$ (die Menge der auszuführenden Jobs)
und $E := 0$ (aktueller Endpunkt)

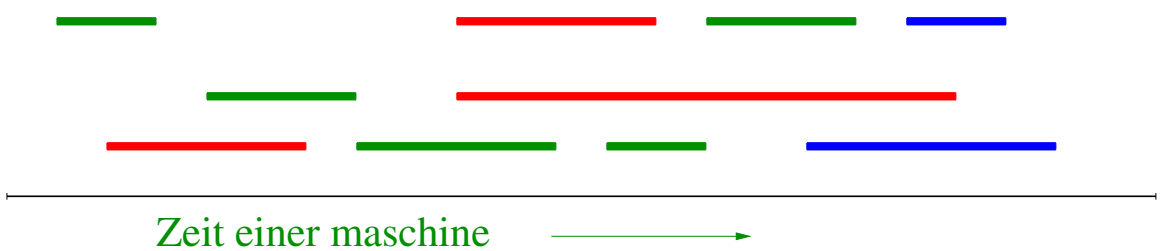
FOR $k = 1$ TO n

if $s^k \geq E$ then

$J := J \cup \{A^k\}$

$E := e^k$

else verwerfe A^k



Greedy Intervall-Scheduling

Seien $e^1 \leq e^2 \leq \ldots \leq e^n$ die sortierten Endpunkte,

entsprehend den Aufgaben $A^1, A^2, \ldots, A^n$

und Endpunkten $s^1, s^2, \ldots, s^n$.

Setze $J := \emptyset$ (die Menge der auszuführenden Jobs)
und $E := 0$ (aktueller Endpunkt)

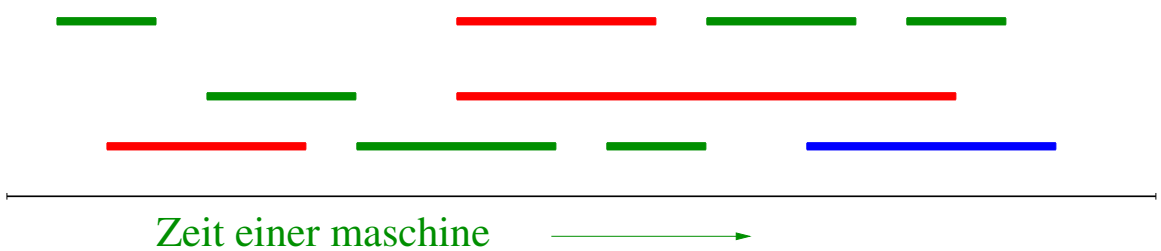
FOR $k = 1$ TO n

if $s^k \geq E$ then

$J := J \cup \{A^k\}$

$E := e^k$

else verwerfe A^k



Greedy Intervall-Scheduling

Seien $e^1 \leq e^2 \leq \ldots \leq e^n$ die sortierten Endpunkte,

entsprehend den Aufgaben $A^1, A^2, \ldots, A^n$

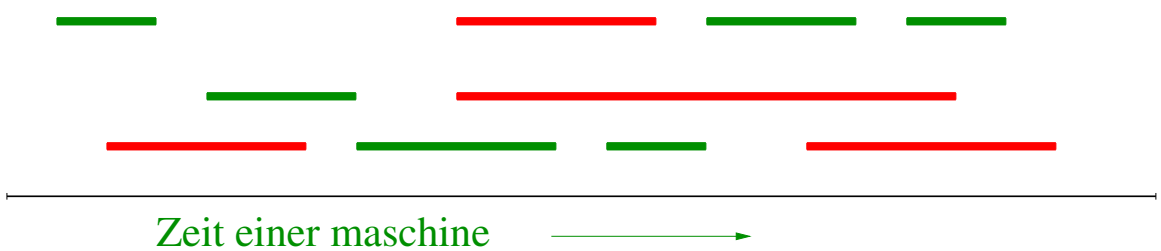
und Endpunkten $s^1, s^2, \ldots, s^n$.

Setze $J := \emptyset$ (die Menge der auszuführenden Jobs)
und $E := 0$ (aktueller Endpunkt)

FOR $k = 1$ TO n

if $s^k \geq E$ then
$J := J \cup \{A^k\}$
$E := e^k$

else verwerfe A^k



Laufzeit: $\mathcal{O}(n \log n)$ für Sortieren und $\mathcal{O}(n)$ sonst

Theorem: Dieser greedy Algorithmus ist optimal.

Huffman-Code

Huffman-Code

Eingabe: Eine Datei mit Buchstaben über dem Alphabet Σ.

Huffman-Code

Eingabe: Eine Datei mit Buchstaben über dem Alphabet Σ.

$H(a)$ ist die Häufigkeit des $a \in \Sigma$ in der Datei

Huffman-Code

Eingabe: Eine Datei mit Buchstaben über dem Alphabet Σ.

$H(a)$ ist die Häufigkeit des $a \in \Sigma$ in der Datei

Ausgabe: Bestimme einen Präfix-Code $\mathrm{code} : \Sigma \to \{0,1\}^*$ so dass die Kodierung der Datei minimiert wird!

Huffman-Code

Eingabe: Eine Datei mit Buchstaben über dem Alphabet Σ.

$H(a)$ ist die Häufigkeit des $a \in \Sigma$ in der Datei

Ausgabe: Bestimme einen Präfix-Code $\mathrm{code} : \Sigma \to \{0,1\}^*$ so dass die Kodierung der Datei minimiert wird!

(Wir kodieren die Buchstaben durch binäre Worte (zB. $\mathrm{code}(a) = 0101$)

so dass kein Codewort Präfix eines anderen ist.)

zur Erinnerung

zur Erinnerung

Jeder Präfix-Code wird von einem binären *Kodierbaum* representiert,

dessen Kanten durch 0 oder 1 markiert sind; jedes Blatt entspricht einem Code-Wort und einem Buchstaben.

zur Erinnerung

Jeder Präfix-Code wird von einem binären *Kodierbaum* representiert,

dessen Kanten durch 0 oder 1 markiert sind; jedes Blatt entspricht einem Code-Wort und einem Buchstaben.

die Länge der kodierten Datei ist

$$||\mathrm{code}|| = \sum_{a \in \Sigma} H(a) \cdot |\mathrm{code}(a)| = \sum_{a \in \Sigma} H(a) \cdot \mathit{Tiefe}(a)$$

zur Erinnerung

Jeder Präfix-Code wird von einem binären *Kodierbaum* representiert,

dessen Kanten durch 0 oder 1 markiert sind; jedes Blatt entspricht einem Code-Wort und einem Buchstaben.

die Länge der kodierten Datei ist

$$||\mathrm{code}|| = \sum_{a \in \Sigma} H(a) \cdot |\mathrm{code}(a)| = \sum_{a \in \Sigma} H(a) \cdot \mathit{Tiefe}(a)$$

Huffman's Algorithmus

Huffman's Algorithmus

der Kodierbaum wird mit einem Greedy Algorithmus aufgebaut

Huffman's Algorithmus

der Kodierbaum wird mit einem Greedy Algorithmus aufgebaut

Anfangs entspricht jedem Buchstaben ein Einzelknoten

Huffman's Algorithmus

der Kodierbaum wird mit einem Greedy Algorithmus aufgebaut

Anfangs entspricht jedem Buchstaben ein Einzelknoten

WHILE es ≥ 2 Buchstaben gibt

- nimm die 2 Buchstaben x und y mit kleinster Häufigkeit

Huffman's Algorithmus

der Kodierbaum wird mit einem Greedy Algorithmus aufgebaut

Anfangs entspricht jedem Buchstaben ein Einzelknoten

WHILE es ≥ 2 Buchstaben gibt

- nimm die 2 Buchstaben x und y mit kleinster Häufigkeit
- erzeuge einen neuen Knoten $\overline{xy}$ mit Häufigkeit $H(\overline{xy}) := H(x) + H(y)$

Huffman's Algorithmus

der Kodierbaum wird mit einem Greedy Algorithmus aufgebaut

Anfangs entspricht jedem Buchstaben ein Einzelknoten

WHILE es ≥ 2 Buchstaben gibt

- nimm die 2 Buchstaben x und y mit kleinster Häufigkeit
- erzeuge einen neuen Knoten $\overline{xy}$ mit Häufigkeit $H(\overline{xy}) := H(x) + H(y)$
- x und y seien die Kinder von $\overline{xy}$ im Binärbaum.

Huffman's Algorithmus

der Kodierbaum wird mit einem Greedy Algorithmus aufgebaut

Anfangs entspricht jedem Buchstaben ein Einzelknoten

WHILE es ≥ 2 Buchstaben gibt

- nimm die 2 Buchstaben x und y mit kleinster Häufigkeit
- erzeuge einen neuen Knoten $\overline{xy}$ mit Häufigkeit $H(\overline{xy}) := H(x) + H(y)$
- x und y seien die Kinder von $\overline{xy}$ im Binärbaum.

(Die $H()$ Werte werden in einem Heap verwaltet, weil neue eingefügt werden.

Laufzeit: $\mathcal{O}(|\Sigma| \log |\Sigma|)$.)

Theorem: Huffman's Algorithmus ist optimal.

GREEDY ALGORITHMEN

GREEDY ALGORITHMEN

Das (metrische) Traveling Salesman Problem (TSP)

Definition: das Traveling Salesman Problem (TSP)

Definition: das Traveling Salesman Problem (TSP)

Eingabe: n Orte $\{1, 2, 3, \ldots, n\}$ und Distanzwerte $d(i, j)$ zwischen je zwei Orten $i \neq j$

Definition: das Traveling Salesman Problem (TSP)

Eingabe: n Orte $\{1, 2, 3, \ldots, n\}$ und Distanzwerte $d(i, j)$ zwischen je zwei Orten $i \neq j$

Ausgabe: Eine Rundreise minimaler Länge

Definition: das Traveling Salesman Problem (TSP)

Eingabe: n Orte $\{1, 2, 3, \ldots, n\}$ und Distanzwerte $d(i, j)$ zwischen je zwei Orten $i \neq j$

Ausgabe: Eine Rundreise minimaler Länge, d.h. eine Permutation π der Orte so dass

$$\sum_{i=1}^{n-1} d(\pi(i), \pi(i+1)) + d(\pi(n), \pi(1))$$

minimiert wird.

Definition: das Traveling Salesman Problem (TSP)

Eingabe: n Orte $\{1, 2, 3, \ldots, n\}$ und Distanzwerte $d(i, j)$ zwischen je zwei Orten $i \neq j$

Ausgabe: Eine Rundreise minimaler Länge, d.h. eine Permutation π der Orte so dass

$$\sum_{i=1}^{n-1} d(\pi(i), \pi(i+1)) + d(\pi(n), \pi(1))$$

minimiert wird.

die Rundreise ist also $(\pi(1), \pi(2), \pi(3), \ldots, \pi(n), \pi(1))$

Definition: das Traveling Salesman Problem (TSP)

Eingabe: n Orte $\{1, 2, 3, \ldots, n\}$ und Distanzwerte $d(i, j)$ zwischen je zwei Orten $i \neq j$

Ausgabe: Eine Rundreise minimaler Länge, d.h. eine Permutation π der Orte so dass

$$\sum_{i=1}^{n-1} d(\pi(i), \pi(i+1)) + d(\pi(n), \pi(1))$$

minimiert wird.

die Rundreise ist also $(\pi(1), \pi(2), \pi(3), \ldots, \pi(n), \pi(1))$

Nichtapproximierbarkeit des *allgemeinen* TSP

Das TSP-Problem lässt effizient keine vernünftige Approximation zu:

Nichtapproximierbarkeit des *allgemeinen* TSP

Das TSP-Problem lässt effizient keine vernünftige Approximation zu:

nicht mal eine 2^n-Approximation!

Nichtapproximierbarkeit des *allgemeinen* TSP

Das TSP-Problem lässt effizient keine vernünftige Approximation zu:

nicht mal eine 2^n-Approximation!

Theorem: Es existiert kein effizienter $\alpha(n)$-approximativer Algorithmus für TSP für $\alpha(n) = \mathcal{O}(2^n)$

Nichtapproximierbarkeit des *allgemeinen* TSP

Das TSP-Problem lässt effizient keine vernünftige Approximation zu:

nicht mal eine 2^n-Approximation!

Theorem: Es existiert kein effizienter $\alpha(n)$-approximativer Algorithmus für TSP für $\alpha(n) = \mathcal{O}(2^n)$

Definition: das metrische Traveling Salesman Problem

Definition: das metrische Traveling Salesman Problem

In dieser Einschränkung des TSP ist gefordert, dass die Distanzwerte $d(i, j)$ die *Dreiecksungleichung* erfüllen.

Definition: das metrische Traveling Salesman Problem

In dieser Einschränkung des TSP ist gefordert, dass die Distanzwerte $d(i,j)$ die *Dreiecksungleichung* erfüllen.

[Eine Distanzfunktion $d(i,j)$ über allen Punktpaaren einer Menge X heißt eine Metrik, wenn

1. $d(i,j) = 0 \Leftrightarrow i = j$;

Definition: das metrische Traveling Salesman Problem

In dieser Einschränkung des TSP ist gefordert, dass die Distanzwerte $d(i,j)$ die *Dreiecksungleichung* erfüllen.

[Eine Distanzfunktion $d(i,j)$ über allen Punktpaaren einer Menge X heißt eine Metrik, wenn

1. $d(i,j) = 0 \Leftrightarrow i = j$;

2. $d(i,j) = d(j,i) \quad \forall i,j$

Definition: das metrische Traveling Salesman Problem

In dieser Einschränkung des TSP ist gefordert, dass die Distanzwerte $d(i,j)$ die *Dreiecksungleichung* erfüllen.

[Eine Distanzfunktion $d(i,j)$ über allen Punktpaaren einer Menge X heißt eine Metrik, wenn

1. $d(i,j) = 0 \Leftrightarrow i = j$;
2. $d(i,j) = d(j,i) \quad \forall i,j$
3. $d(i,k) \leq d(i,j) + d(j,k) \quad \forall i,j,k$

Definition: das metrische Traveling Salesman Problem

In dieser Einschränkung des TSP ist gefordert, dass die Distanzwerte $d(i,j)$ die *Dreiecksungleichung* erfüllen.

[Eine Distanzfunktion $d(i,j)$ über allen Punktpaaren einer Menge X heißt eine Metrik, wenn

1. $d(i,j) = 0 \Leftrightarrow i = j$;
2. $d(i,j) = d(j,i) \quad \forall i,j$
3. $d(i,k) \leq d(i,j) + d(j,k) \quad \forall i,j,k$

die Bedingung 3. nennt man Dreiecksungleichung]

Beobachtungen

Beobachtungen

Denken wir an die Orte als an Knoten eines vollständigen Graphen, mit Kantengewichten $d(i, j)$.

Beobachtungen

Denken wir an die Orte als an Knoten eines vollständigen Graphen, mit Kantengewichten $d(i, j)$.

$Rundreise_{MIN}$: Länge einer minimalen Rundreise

$Spannbaum_{MIN}$: die Länge eines minimalen Spannbaums.

Beobachtungen

Denken wir an die Orte als an Knoten eines vollständigen Graphen, mit Kantengewichten $d(i, j)$.

$Rundreise_{MIN}$: Länge einer minimalen Rundreise

$Spannbaum_{MIN}$: die Länge eines minimalen Spannbaums.

Behauptung 1. $Spannbaum_{MIN} \leq Rundreise_{MIN}$

Beobachtungen

Denken wir an die Orte als an Knoten eines vollständigen Graphen, mit Kantengewichten $d(i,j)$.

$Rundreise_{MIN}$: Länge einer minimalen Rundreise

$Spannbaum_{MIN}$: die Länge eines minimalen Spannbaums.

Behauptung 1. $Spannbaum_{MIN} \leq Rundreise_{MIN}$

Behauptung 2. Es gibt eine Rundreise mit Länge $\leq 2 \cdot Spannbaum_{MIN}$.

Beobachtungen

Denken wir an die Orte als an Knoten eines vollständigen Graphen, mit Kantengewichten $d(i,j)$.

$Rundreise_{MIN}$: Länge einer minimalen Rundreise

$Spannbaum_{MIN}$: die Länge eines minimalen Spannbaums.

Behauptung 1. $Spannbaum_{MIN} \leq Rundreise_{MIN}$

Behauptung 2. Es gibt eine Rundreise mit Länge $\leq 2 \cdot Spannbaum_{MIN}$.

1. die Spannbaum-Heuristik

1. die Spannbaum-Heuristik

1. Bestimmen wir einen minimalen Spannbaum (für den *vollständigen* Graphen auf $\{1, 2, 3, \ldots, n\}$ mit Kantengewichten $d(i, j)$).

1. die Spannbaum-Heuristik

1. Bestimmen wir einen minimalen Spannbaum (für den *vollständigen* Graphen auf $\{1, 2, 3, \ldots, n\}$ mit Kantengewichten $d(i, j)$).

2. Besuche die Orte in der Reihenfolge in der der Algorithmus Präorder (Tiefensuche) die Knoten besucht.

1. die Spannbaum-Heuristik

1. Bestimmen wir einen minimalen Spannbaum (für den *vollständigen* Graphen auf $\{1, 2, 3, \ldots, n\}$ mit Kantengewichten $d(i, j)$).

2. Besuche die Orte in der Reihenfolge in der der Algorithmus Präorder (Tiefensuche) die Knoten besucht.

Dank der Dreiecksungleichung, enthält diese Rundreise tatsächlich Abkürzungen im Vergleich zu einer Tiefensuche mit hin-und-zurück Durchlaufen jeder Kante.

1. die Spannbaum-Heuristik

1. Bestimmen wir einen minimalen Spannbaum (für den *vollständigen* Graphen auf $\{1, 2, 3, \ldots, n\}$ mit Kantengewichten $d(i, j)$).

2. Besuche die Orte in der Reihenfolge in der der Algorithmus Präorder (Tiefensuche) die Knoten besucht.

Dank der Dreiecksungleichung, enthält diese Rundreise tatsächlich Abkürzungen im Vergleich zu einer Tiefensuche mit hin-und-zurück Durchlaufen jeder Kante.

Theorem: Die Spannbaum-Heuristik berechnet eine 2-approximative Rundreise für das metrische TSP.

2. Der Algorithmus von Christofides (Vorbereitung)

2. Der Algorithmus von Christofides (Vorbereitung)

Zur Erinnerung:

Sei $G(V, E)$ ein zusammenhängender Graph.

Es gibt eine Euler-Tour in G
(d.h. die jede Kante in E genau einmal durchläuft)

2. Der Algorithmus von Christofides (Vorbereitung)

Zur Erinnerung:

Sei $G(V, E)$ ein zusammenhängender Graph.

Es gibt eine Euler-Tour in G
(d.h. die jede Kante in E genau einmal durchläuft)

$$\Updownarrow$$

für jeden v Knoten $grad(v)$ eine gerade Zahl ist

Der Algorithmus von Christofides:

Der Algorithmus von Christofides:

- Berechne einen minimalen Spannbaum T_{min}

Der Algorithmus von Christofides:

- Berechne einen minimalen Spannbaum T_{min}
- sei $U \subseteq \{1, 2, 3, \ldots, n\}$ die Menge der Knoten mit *ungeradem Grad im Baum* T_{min} ;

Der Algorithmus von Christofides:

- Berechne einen minimalen Spannbaum T_{min}
- sei $U \subseteq \{1, 2, 3, \ldots, n\}$ die Menge der Knoten mit *ungeradem Grad im Baum* T_{min} ;
- berechne ein minimum-Gewicht perfekt Matching M_{min} auf den Knoten von U ;

Der Algorithmus von Christofides:

- Berechne einen minimalen Spannbaum T_{min}
- sei $U \subseteq \{1, 2, 3, \ldots, n\}$ die Menge der Knoten mit *ungeradem Grad im Baum* T_{min} ;
- berechne ein minimum-Gewicht perfekt Matching M_{min} auf den Knoten von U ;
- berechne auf $T_{\text{min}} \uplus M_{\text{min}}$ eine Euler-Tour;

Der Algorithmus von Christofides:

- Berechne einen minimalen Spannbaum T_{min}
- sei $U \subseteq \{1, 2, 3, \ldots, n\}$ die Menge der Knoten mit *ungeradem Grad im Baum* T_{min} ;
- berechne ein minimum-Gewicht perfekt Matching M_{min} auf den Knoten von U ;
- berechne auf $T_{\text{min}} \uplus M_{\text{min}}$ eine Euler-Tour;
- besuche die Knoten in der Reihenfolge wie die Euler-Tour, aber mit weiteren Abkürzungen.

Der Algorithmus von Christofides:

- Berechne einen minimalen Spannbaum $T_{\min}$
- sei $U \subseteq \{1, 2, 3, \ldots, n\}$ die Menge der Knoten mit *ungeradem Grad im Baum* $T_{\min}$;
- berechne ein minimum-Gewicht perfekt Matching $M_{\min}$ auf den Knoten von U ;
- berechne auf $T_{\min} \uplus M_{\min}$ eine Euler-Tour;
- besuche die Knoten in der Reihenfolge wie die Euler-Tour, aber mit weiteren Abkürzungen.

Theorem: Der Algorithmus von Christofides ist ein 3/2-Approximationsalgorithmus. Seine Laufzeit für n Punkte ist $\mathcal{O}(n^3)$.

Nearest-Insertion Heuristik:

Nearest-Insertion Heuristik:

nimm die *kürzeste* Kante $\{i, j\}$, setze $V := V \setminus \{i, j\}$, und die Partielle Rundreise sei (i, j, i)

Nearest-Insertion Heuristik:

nimm die *kürzeste* Kante $\{i, j\}$, setze $V := V \setminus \{i, j\}$, und die Partielle Rundreise sei (i, j, i)

REPEAT

- nimm den Knoten k mit *kürzestem* Abstand zum nächstliegenden Knoten der bisherigen Partiellen Rundreise; setze $V = V \setminus \{k\}$;

Nearest-Insertion Heuristik:

nimm die *kürzeste* Kante $\{i,j\}$, setze $V := V \setminus \{i,j\}$, und die Partielle Rundreise sei (i,j,i)

REPEAT

- nimm den Knoten k mit *kürzestem* Abstand zum nächstliegenden Knoten der bisherigen Partiellen Rundreise; setze $V = V \setminus \{k\}$;
- füge k zwischen zwei benachbarten Knoten der Partiellen Rundreise, mit kleinstem Anstieg der Länge der Rundreise.

Nearest-Insertion Heuristik:

nimm die *kürzeste* Kante $\{i,j\}$, setze $V := V \setminus \{i,j\}$, und die Partielle Rundreise sei (i,j,i)

REPEAT

- nimm den Knoten k mit *kürzestem* Abstand zum nächstliegenden Knoten der bisherigen Partiellen Rundreise; setze $V = V \setminus \{k\}$;
- füge k zwischen zwei benachbarten Knoten der Partiellen Rundreise, mit kleinstem Anstieg der Länge der Rundreise.

UNTIL $V = \emptyset$

Farthest-Insertion Heuristik:

Farthest-Insertion Heuristik:

nimm die *längste* Kante $\{i, j\}$, setze $V := V \setminus \{i, j\}$, und die Partielle Rundreise sei (i, j, i)

Farthest-Insertion Heuristik:

nimm die *längste* Kante $\{i, j\}$, setze $V := V \setminus \{i, j\}$, und die Partielle Rundreise sei (i, j, i)

REPEAT

- nimm den Knoten k mit *weitestem* Abstand zum nächstliegenden Knoten der bisherigen Partiellen Rundreise; setze $V = V \setminus \{k\}$;

Farthest-Insertion Heuristik:

nimm die *längste* Kante $\{i, j\}$, setze $V := V \setminus \{i, j\}$, und die Partielle Rundreise sei (i, j, i)

REPEAT

- nimm den Knoten k mit *weitestem* Abstand zum nächstliegenden Knoten der bisherigen Partiellen Rundreise; setze $V = V \setminus \{k\}$;
- füge k zwischen zwei benachbarten Knoten der Partiellen Rundreise, mit kleinstem Anstieg der Länge der Rundreise.

Farthest-Insertion Heuristik:

nimm die *längste* Kante $\{i, j\}$, setze $V := V \setminus \{i, j\}$, und die Partielle Rundreise sei (i, j, i)

REPEAT

- nimm den Knoten k mit *weitestem* Abstand zum nächstliegenden Knoten der bisherigen Partiellen Rundreise; setze $V = V \setminus \{k\}$;
- füge k zwischen zwei benachbarten Knoten der Partiellen Rundreise, mit kleinstem Anstieg der Länge der Rundreise.

UNTIL $V = \emptyset$

Durchschnittlicher Approximationsfaktor

Experimentell berechnet auf 100 000 zufällig gewählten Punkten in $[0, 1] \times [0, 1]$.

Durchschnittlicher Approximationsfaktor

Experimentell berechnet auf 100 000 zufällig gewählten Punkten in $[0, 1] \times [0, 1]$.

1. Christofides' Algorithmus $\alpha \approx 1.1$

Durchschnittlicher Approximationsfaktor

Experimentell berechnet auf 100 000 zufällig gewählten Punkten in $[0, 1] \times [0, 1]$.

1. Christofides' Algorithmus $\alpha \approx 1.1$
2. Farthest-Insertion $\alpha \approx 1.1$

Durchschnittlicher Approximationsfaktor

Experimentell berechnet auf 100 000 zufällig gewählten Punkten in $[0, 1] \times [0, 1]$.

1. Christofides' Algorithmus $\alpha \approx 1.1$
2. Farthest-Insertion $\alpha \approx 1.1$
3. Nearest-Insertion $\alpha \approx 1.25$

Durchschnittlicher Approximationsfaktor

Experimentell berechnet auf 100 000 zufällig gewählten Punkten in $[0, 1] \times [0, 1]$.

1. Christofides' Algorithmus $\alpha \approx 1.1$
2. Farthest-Insertion $\alpha \approx 1.1$
3. Nearest-Insertion $\alpha \approx 1.25$
4. Spannbaum Heuristik $\alpha \approx 1.4$

Das Euklidsche TSP

Eingabe: n Punkte $V \subseteq \mathbb{R}^d$ mit den euklidschen Distanzen:

Das Euklidsche TSP

Eingabe: n Punkte $V \subseteq \mathbb{R}^d$ mit den euklidschen Distanzen:

für $a = (a_1, a_2, \ldots, a_d)$ und $b = (b_1, b_2, \ldots, b_d)$

$$d(a, b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \ldots + (a_d - b_d)^2}$$

Das Euklidsche TSP

Eingabe: n Punkte $V \subseteq \mathbb{R}^d$ mit den euklidschen Distanzen:

für $a = (a_1, a_2, \ldots, a_d)$ und $b = (b_1, b_2, \ldots, b_d)$

$$d(a, b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \ldots + (a_d - b_d)^2}$$

Ausgabe: Eine kürzeste Rundreise die alle n Punkte genau einmal besucht.

Zusammenfassung: Approximierbarkeit von TSP

Zusammenfassung: Approximierbarkeit von TSP

das euklidsche TSP: $(1+\varepsilon)$-approximierbar (Arora's PTAS)

Zusammenfassung: Approximierbarkeit von TSP

das euklidsche TSP: $(1+\varepsilon)$-approximierbar (Arora's PTAS)

das metrische TSP: $\frac{3}{2}$-approximierbar (Christofides)

Zusammenfassung: Approximierbarkeit von TSP

das euklidsche TSP: $(1+\varepsilon)$-approximierbar (Arora's PTAS)

das metrische TSP: $\frac{3}{2}$-approximierbar (Christofides)

keine polynomielle Approximation $< \frac{220}{219}$

Zusammenfassung: Approximierbarkeit von TSP

das euklidsche TSP: $(1+\varepsilon)$-approximierbar (Arora's PTAS)

das metrische TSP: $\frac{3}{2}$-approximierbar (Christofides)

keine polynomielle Approximation $< \frac{220}{219}$

das allgemeine TSP

Zusammenfassung: Approximierbarkeit von TSP

das euklidsche TSP: $(1+\varepsilon)$-approximierbar (Arora's PTAS)

das metrische TSP: $\frac{3}{2}$-approximierbar (Christofides)

keine polynomielle Approximation $< \frac{220}{219}$

das allgemeine TSP 'gar nicht' approximierbar

Zusammenfassung: Approximierbarkeit von TSP

das euklidsche TSP: $(1+\varepsilon)$-approximierbar (Arora's PTAS)

das metrische TSP: $\frac{3}{2}$-approximierbar (Christofides)

keine polynomielle Approximation $< \frac{220}{219}$

das allgemeine TSP 'gar nicht' approximierbar

Das BIN PACKING Problem

Das BIN PACKING Problem

Eingabe: n Objekte mit Gewichten $g_1, g_2, \ldots, g_n$ $\qquad (0 \leq g_i \leq 1)$

Das BIN PACKING Problem

Eingabe: n Objekte mit Gewichten $g_1, g_2, \ldots, g_n$ $\quad (0 \leq g_i \leq 1)$

Ausgabe: Verteile die Objekte in eine *minimale* Anzahl von Behälter mit Kapazität 1

Greedy Algorithmen für BIN PACKING (On-line)

Greedy Algorithmen für BIN PACKING (On-line)

1. Next Fit

Greedy Algorithmen für BIN PACKING (On-line)

1. Next Fit

   sei $B = 1$; (Anzahl geöffneter Behälter)

Greedy Algorithmen für BIN PACKING (On-line)

1. Next Fit

   sei $B = 1$; (Anzahl geöffneter Behälter)

   FOR $i = 1$ TO n DO

   - falls möglich, füge g_i in Behälter B ein;

Greedy Algorithmen für BIN PACKING (On-line)

1. Next Fit

   sei $B = 1$; (Anzahl geöffneter Behälter)

   FOR $i = 1$ TO n DO

   - falls möglich, füge g_i in Behälter B ein;
   - sonst $B := B + 1$ und füge g_i in den neuen Behälter ein;

Greedy Algorithmen für BIN PACKING (On-line)

1. Next Fit

   sei $B = 1$; (Anzahl geöffneter Behälter)

   FOR $i = 1$ TO n DO

   - falls möglich, füge g_i in Behälter B ein;
   - sonst $B := B + 1$ und füge g_i in den neuen Behälter ein;

Beobachtung: Next Fit benutzt $\leq 2 \cdot OPT - 1$ Behälter.

Greedy Algorithmen für BIN PACKING (On-line)

1. Next Fit

   sei $B = 1$; (Anzahl geöffneter Behälter)

   FOR $i = 1$ TO n DO

   - falls möglich, füge g_i in Behälter B ein;
   - sonst $B := B + 1$ und füge g_i in den neuen Behälter ein;

Beobachtung: Next Fit benutzt $\leq 2 \cdot OPT - 1$ Behälter.

2. First Fit

Greedy Algorithmen für BIN PACKING (On-line)

1. Next Fit

   sei $B = 1$; (Anzahl geöffneter Behälter)

   FOR $i = 1$ TO n DO

   - falls möglich, füge g_i in Behälter B ein;
   - sonst $B := B + 1$ und füge g_i in den neuen Behälter ein;

Beobachtung: Next Fit benutzt $\leq 2 \cdot OPT - 1$ Behälter.

2. First Fit

   sei $B = 1$; (Anzahl geöffneter Behälter)

Greedy Algorithmen für BIN PACKING (On-line)

1. Next Fit

   sei $B = 1$; (Anzahl geöffneter Behälter)

   FOR $i = 1$ TO n DO

   - falls möglich, füge g_i in Behälter B ein;
   - sonst $B := B + 1$ und füge g_i in den neuen Behälter ein;

Beobachtung: Next Fit benutzt $\leq 2 \cdot OPT - 1$ Behälter.

2. First Fit

   sei $B = 1$; (Anzahl geöffneter Behälter)

   FOR $i = 1$ TO n DO

   - füge g_i in den *ersten* Behälter B ein wo es reinpasst;

Greedy Algorithmen für BIN PACKING (On-line)

1. Next Fit

   sei $B = 1$; (Anzahl geöffneter Behälter)

   FOR $i = 1$ TO n DO

   - falls möglich, füge g_i in Behälter B ein;
   - sonst $B := B + 1$ und füge g_i in den neuen Behälter ein;

Beobachtung: Next Fit benutzt $\leq 2 \cdot OPT - 1$ Behälter.

2. First Fit

   sei $B = 1$; (Anzahl geöffneter Behälter)

   FOR $i = 1$ TO n DO

   - füge g_i in den *ersten* Behälter B ein wo es reinpasst;
   - sonst $B = B + 1$ und füge g_i in den neuen Behälter ein;

Greedy Algorithmen für BIN PACKING (On-line)

1. Next Fit

   sei $B = 1$; (Anzahl geöffneter Behälter)

   FOR $i = 1$ TO n DO

   - falls möglich, füge g_i in Behälter B ein;
   - sonst $B := B + 1$ und füge g_i in den neuen Behälter ein;

Beobachtung: Next Fit benutzt $\leq 2 \cdot OPT - 1$ Behälter.

2. First Fit

   sei $B = 1$; (Anzahl geöffneter Behälter)

   FOR $i = 1$ TO n DO

   - füge g_i in den *ersten* Behälter B ein wo es reinpasst;
   - sonst $B = B + 1$ und füge g_i in den neuen Behälter ein;
   - höchstens 1.7-approximativ (oB.)

Greedy Algorithmen für BIN PACKING (Off-line)

Greedy Algorithmen für BIN PACKING (Off-line)

3. First-Fit-Decreasing (FFD)

Greedy Algorithmen für BIN PACKING (Off-line)

3. First-Fit-Decreasing (FFD)
   - sortiere die Gewichte absteigend: $g_1 \geq g_2 \geq \ldots \geq g_n$

Greedy Algorithmen für BIN PACKING (Off-line)

3. First-Fit-Decreasing (FFD)
   - sortiere die Gewichte absteigend: $g_1 \geq g_2 \geq \ldots \geq g_n$
   - wende First-Fit an

Greedy Algorithmen für BIN PACKING (Off-line)

3. First-Fit-Decreasing (FFD)
   - sortiere die Gewichte absteigend: $g_1 \geq g_2 \geq \ldots \geq g_n$
   - wende First-Fit an

Theorem: FFD benutzt $\leq \frac{3}{2} \cdot OPT + 1$ Behälter.

Greedy Algorithmen für BIN PACKING (Off-line)

3. First-Fit-Decreasing (FFD)
   - sortiere die Gewichte absteigend: $g_1 \geq g_2 \geq \ldots \geq g_n$
   - wende First-Fit an

Theorem: FFD benutzt $\leq \frac{3}{2} \cdot OPT + 1$ Behälter.

4. Best-Fit-Decreasing

Greedy Algorithmen für BIN PACKING (Off-line)

3. First-Fit-Decreasing (FFD)
   - sortiere die Gewichte absteigend: $g_1 \geq g_2 \geq \ldots \geq g_n$
   - wende First-Fit an

Theorem: FFD benutzt $\leq \frac{3}{2} \cdot OPT + 1$ Behälter.

4. Best-Fit-Decreasing

   sortiere die Gewichte absteigend: $g_1 \geq g_2 \geq \ldots \geq g_n$

Greedy Algorithmen für BIN PACKING (Off-line)

3. First-Fit-Decreasing (FFD)
   - sortiere die Gewichte absteigend: $g_1 \geq g_2 \geq \ldots \geq g_n$
   - wende First-Fit an

Theorem: FFD benutzt $\leq \frac{3}{2} \cdot OPT + 1$ Behälter.

4. Best-Fit-Decreasing

   sortiere die Gewichte absteigend: $g_1 \geq g_2 \geq \ldots \geq g_n$

   FOR $i = 1$ TO n DO
   - füge Objekt i in den Behälter ein wo es am knappsten ist

Greedy Algorithmen für BIN PACKING (Off-line)

3. First-Fit-Decreasing (FFD)
   - sortiere die Gewichte absteigend: $g_1 \geq g_2 \geq \ldots \geq g_n$
   - wende First-Fit an

Theorem: FFD benutzt $\leq \frac{3}{2} \cdot OPT + 1$ Behälter.

4. Best-Fit-Decreasing

   sortiere die Gewichte absteigend: $g_1 \geq g_2 \geq \ldots \geq g_n$

   FOR $i = 1$ TO n DO
   - füge Objekt i in den Behälter ein wo es am knappsten ist (dessen verbleibende Restkapazität minimal ist)

'Nicht-Approximierbarkeit' des BIN PACKING

'Nicht-Approximierbarkeit' des BIN PACKING

Theorem: BIN PACKING besitzt *keinen* effizienten α-approximativen Algorithmus für $\alpha < 3/2$ falls $\mathcal{P} \neq \mathcal{NP}$.

'Nicht-Approximierbarkeit' des BIN PACKING

Theorem: BIN PACKING besitzt *keinen* effizienten α-approximativen Algorithmus für $\alpha < 3/2$ falls $\mathcal{P} \neq \mathcal{NP}$.

So ein Algorithmus könnte effizient entscheiden ob für n beliebige positive Zahlen $z_1, z_2, \ldots, z_n$ zwei Behälter mit Kapazität $\frac{\sum z_i}{2}$ ausreichen, und so das PARTITION Problem effizient lösen!

Ob es einen Algorithmus gibt der $OPT + 1$ Behälter benutzt, ist nicht bekannt...

Ob es einen Algorithmus gibt der $OPT + 1$ Behälter benutzt, ist nicht bekannt...

Aber:

Für jedes $\varepsilon \in (0, \frac{1}{2})$ gibt es einen Algorithmus A_ε
der $(1 + \varepsilon) \cdot OPT(I) + 1$ Behälter benutzt für jede Eingabe I.

Ob es einen Algorithmus gibt der $OPT + 1$ Behälter benutzt, ist nicht bekannt...

Aber:

Für jedes $\varepsilon \in (0, \frac{1}{2})$ gibt es einen Algorithmus A_ε
der $(1 + \varepsilon) \cdot OPT(I) + 1$ Behälter benutzt für jede Eingabe I.

Vorbereitung

BIN PACKING ist 'effizient' optimierbar falls es nur G verschiedene mögliche Gewichte gibt, und alle sind grösser als eine Konstante δ.

Vorbereitung

BIN PACKING ist 'effizient' optimierbar falls es
nur G verschiedene mögliche Gewichte gibt,
und alle sind grösser als eine Konstante δ.

1. Wir merken *alle* möglichen Bepackungen *eines* Behälters:

Vorbereitung

BIN PACKING ist 'effizient' optimierbar falls es
nur G verschiedene mögliche Gewichte gibt,
und alle sind grösser als eine Konstante δ.

1. Wir merken *alle* möglichen Bepackungen *eines* Behälters:
   die Anzahl der Möglichkeiten sei T. Es gilt: $T < (1/\delta)^G$.

Vorbereitung

BIN PACKING ist 'effizient' optimierbar falls es
nur G verschiedene mögliche Gewichte gibt,
und alle sind grösser als eine Konstante δ.

1. Wir merken *alle* möglichen Bepackungen *eines* Behälters:
   die Anzahl der Möglichkeiten sei T. Es gilt: $T < (1/\delta)^G$.

2. Höchstens n Behälter mit T Bepackungstypen ergibt
   höchstens n^T mögliche Verteilungen der Objekte

Vorbereitung

BIN PACKING ist 'effizient' optimierbar falls es
nur G verschiedene mögliche Gewichte gibt,
und alle sind grösser als eine Konstante δ.

1. Wir merken *alle* möglichen Bepackungen *eines* Behälters:
   die Anzahl der Möglichkeiten sei T. Es gilt: $T < (1/\delta)^G$.

2. Höchstens n Behälter mit T Bepackungstypen ergibt
   höchstens n^T mögliche Verteilungen der Objekte

   (genauere Berechnung ergibt $\binom{n+T}{T}$ wobei $T \leq \binom{1/\delta+G}{G}$)

Vorbereitung

BIN PACKING ist 'effizient' optimierbar falls es
nur G verschiedene mögliche Gewichte gibt,
und alle sind grösser als eine Konstante δ.

1. Wir merken *alle* möglichen Bepackungen *eines* Behälters:
   die Anzahl der Möglichkeiten sei T. Es gilt: $T < (1/\delta)^G$.

2. Höchstens n Behälter mit T Bepackungstypen ergibt
   höchstens n^T mögliche Verteilungen der Objekte

   (genauere Berechnung ergibt $\binom{n+T}{T}$ wobei $T \leq \binom{1/\delta+G}{G}$)

   Wir haben in Poly(n) aber astronomisch hohe Laufzeit
   erhalten!

Asymptotisches PTAS

Asymptotisches PTAS

Definition: Ein *asymptotisches polynomielles Approximationsschema* (für Minimierungsprobleme) ist eine Familie von polynomiellen Algorithmen (A_ε) zusammen mit einer Konstante c, so dass jeder Algorithmus A_ε eine Lösung mit Wert höchstens $(1+\varepsilon) \cdot OPT + c$ berechnet.

Asymptotisches PTAS

Definition: Ein *asymptotisches polynomielles Approximationsschema* (für Minimierungsprobleme) ist eine Familie von polynomiellen Algorithmen (A_ε) zusammen mit einer Konstante c, so dass jeder Algorithmus A_ε eine Lösung mit Wert höchstens $(1+\varepsilon) \cdot OPT + c$ berechnet.

(APTAS: Asymptotic Polynomial-Time Approximation Scheme)

Asymptotisches PTAS

Definition: Ein *asymptotisches polynomielles Approximationsschema* (für Minimierungsprobleme) ist eine Familie von polynomiellen Algorithmen (A_ε) zusammen mit einer Konstante c, so dass jeder Algorithmus A_ε eine Lösung mit Wert höchstens $(1+\varepsilon) \cdot OPT + c$ berechnet.

(APTAS: Asymptotic Polynomial-Time Approximation Scheme)

(der Approximationsfaktor geht gegen $(1+\varepsilon)$ als $OPT \to \infty$)

Um den APTAS für *beliebige* Gewichte zu definieren, verwenden wir die Idee des obigen optimalen Algorithmus.

Um den APTAS für *beliebige* Gewichte zu definieren, verwenden wir die Idee des obigen optimalen Algorithmus.

Wir brauchen dass...

Um den APTAS für *beliebige* Gewichte zu definieren, verwenden wir die Idee des obigen optimalen Algorithmus.

Wir brauchen dass...

1. alle Gewichte eine Mindestgröße haben ($\geq \varepsilon$)

Um den APTAS für *beliebige* Gewichte zu definieren, verwenden wir die Idee des obigen optimalen Algorithmus.

Wir brauchen dass...

1. alle Gewichte eine Mindestgröße haben ($\geq \varepsilon$)

2. es nur konstant viele verschiedene Gewichte gibt

APTAS für BIN PACKING

– lege *kleine* Objekte mit $g_i \leq \varepsilon$ an die Seite;

APTAS für BIN PACKING

- lege *kleine* Objekte mit $g_i \leq \varepsilon$ an die Seite;

  (sei n die Anzahl der *großen* Objekte)

- zerlege $[0, 1]$ in G Gewichtsklassen mit jeweils $E = n/G$ Gewichten;

APTAS für BIN PACKING

- lege *kleine* Objekte mit $g_i \leq \varepsilon$ an die Seite;

  (sei n die Anzahl der *großen* Objekte)

- zerlege $[0, 1]$ in G Gewichtsklassen mit jeweils $E = n/G$ Gewichten;

- runde jedes Gewicht auf das größte Gewicht seiner Klasse;

APTAS für BIN PACKING

- lege *kleine* Objekte mit $g_i \leq \varepsilon$ an die Seite;
  (sei n die Anzahl der *großen* Objekte)
- zerlege $[0, 1]$ in G Gewichtsklassen mit jeweils $E = n/G$ Gewichten;
- runde jedes Gewicht auf das größte Gewicht seiner Klasse;
- berechne *optimales* Bin Packing für die gerundeten Gewichte

APTAS für BIN PACKING

- lege *kleine* Objekte mit $g_i \leq \varepsilon$ an die Seite;
  (sei n die Anzahl der *großen* Objekte)
- zerlege $[0, 1]$ in G Gewichtsklassen mit jeweils $E = n/G$ Gewichten;
- runde jedes Gewicht auf das größte Gewicht seiner Klasse;
- berechne *optimales* Bin Packing für die gerundeten Gewichte
- füge die *kleinen* Objekte mit First Fit ein.

APTAS für BIN PACKING

- lege *kleine* Objekte mit $g_i \leq \varepsilon$ an die Seite;
  (sei n die Anzahl der *großen* Objekte)
- zerlege $[0, 1]$ in G Gewichtsklassen mit jeweils $E = n/G$ Gewichten;
- runde jedes Gewicht auf das größte Gewicht seiner Klasse;
- berechne *optimales* Bin Packing für die gerundeten Gewichte
- füge die *kleinen* Objekte mit First Fit ein.

Theorem: Dieses APTAS benötigt $\leq (1 + 2\varepsilon) \cdot OPT + 1$ Bins
bei $E = \lfloor n \cdot \varepsilon^2 \rfloor$ und $G = \lceil 1/\varepsilon^2 \rceil$.