

# 13. Approximation

## Einleitendes Beispiel Gütegarantie

Annahmen:

- Sei  $A$  ein Algorithmus, der für jede Instanz  $x$  eines Problems  $X$  eine gültige Lösung mit Lösungswert  $c_A(x) > 0$  liefert.
- Sei  $c_{opt}(x) > 0$  der Wert einer optimalen Lösung.

### ⌚ Definition für Minimierungsprobleme

Falls es ein  $\varepsilon \geq 1$  gibt, sodass für alle Instanzen  $x$  von  $X$  gilt:

$$\frac{c_A(x)}{c_{opt}(x)} \leq \varepsilon$$

dann ist  $A$  ein  $\varepsilon$ -Approximationsalgorithmus und der Wert  $\varepsilon$  heißt Gütegarantie.

### ⌚ Definition für Maximierungsprobleme

Falls es ein  $\varepsilon$  mit  $0 < \varepsilon \leq 1$  gibt, sodass für alle Instanzen  $x$  von  $X$  gilt:

$$\frac{c_A(x)}{c_{opt}(x)} \geq \varepsilon$$

dann ist  $A$  ein  $\varepsilon$ -Approximationsalgorithmus und der Wert  $\varepsilon$  heißt Gütegarantie.

Es folgt:

- für Minimierungsprobleme:  $\varepsilon \geq 1$
- für Maximierungsprobleme:  $0 < \varepsilon \leq 1$
- $\varepsilon = 1 \iff A$  ist ein exakter Algorithmus

# Approximationsalgorithmus für Vertex Cover

## 💡 Definition - Vertex Cover

Ein **Vertex Cover** eines Graphen  $G = (V, E)$  ist eine Menge  $C \subseteq V$ , so dass jede Kante des Graphen zu mindestens einem Knoten aus  $C$  inzident ist.

**Minimales Vertex Cover:** Finde für einen gegebenen Graphen ein Vertex Cover mit minimaler Größe  $|C|$ .

**Aufwand:** Ist NP-schwer, d.h. kann i.A. vermutlich nicht in polynomieller Zeit gelöst werden.

## Minimales Vertex Cover: Approximationsalgorithmus

**2-Approximationsalgorithmus:** Findet ein Vertex Cover in einem Graphen  $G = (V, E)$ , welches höchstens doppelt so groß wie das Optimum ist.

**Approx-Vertex-Cover( $G$ ):**

Approx-Vertex-Cover( $G$ ):

$C \leftarrow \emptyset$

**while**  $E \neq \emptyset$

    Wähle eine beliebige Kante  $(u, v) \in E$

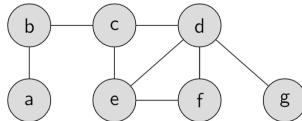
$C \leftarrow C \cup \{u, v\}$

    Entferne aus  $E$  alle Kanten, die inzident zu  $u$  oder  $v$  sind

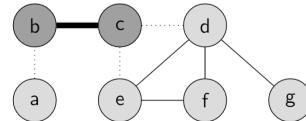
**return**  $C$

☰ Beispiel

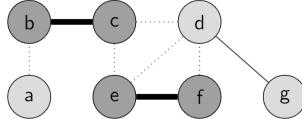
Beispielhafter Ablauf des Algorithmus, Vergleich mit optimaler Lösung.



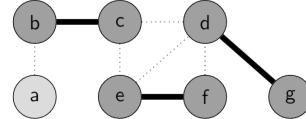
(a) Ausgangssituation



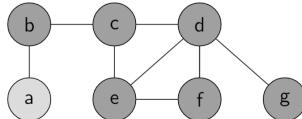
(b) 1. Kante auswählen



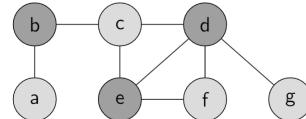
(c) 2. Kante auswählen



(d) 3. Kante auswählen



(e) Ergebnis



(f) Optimale Lösung

Hier können wir jetzt uns die untere Schranke ausrechnen:

$$\frac{6}{3} = 2$$

## Minimales Vertex Cover: Gütegarantie

**Theorem:** Approx-Vertex-Cover ist ein polynomieller Algorithmus mit einer Gütegarantie von 2.

### ✓ Laufzeit

Ist polynomiell.

- In der Schleife werden nacheinander Kanten ausgewählt, zwei Knoten zu  $C$  hinzugefügt und dann alle inzidenten Kanten gelöscht.
- Mit Adjazenzlisten kann dieser Algorithmus mit Laufzeit  $O(n + m)$  implementiert werden.
- Wir schreiben  $n = |V|$  und  $m = |E|$ .

### Gütegarantie:

- Sei  $M$  die Kantenmenge, die vom Algorithmus ausgewählt wird;  $M$  ist ein Matching (Matching... Teilmenge von Kanten im Graphen, sodass keine 2 Kanten in einem Knoten inzident sind, Teilmengen von den Kanten, sodass keine 2 Kanten den selben Endpunkt haben).

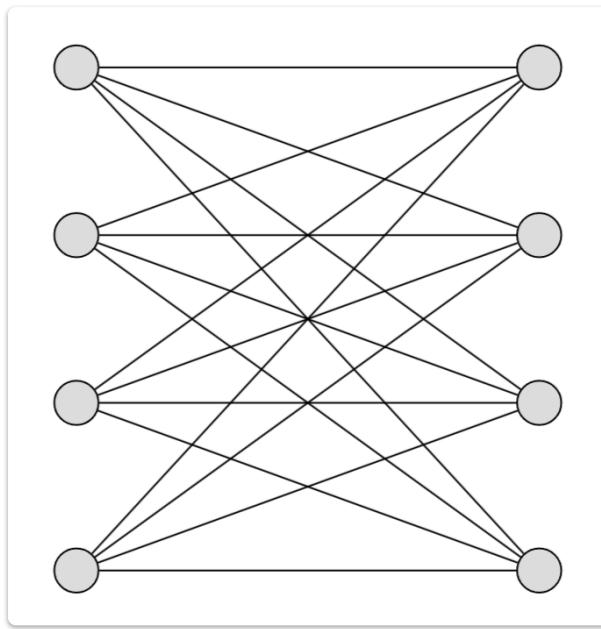
- In einem kleinsten Vertex Cover  $C^*$  muss gelten: Für jede Kante  $e \in M$  existiert ein Knoten  $v \in C^*$ , der inzident zu  $e$  ist.
- Daher muss  $C^*$  zumindest einen der Endpunkte jeder Kante  $e \in M$  enthalten.
- Es folgt, dass  $|C^*| \geq |M|$ .
- Da Approx-Vertex-Cover (kurz AVC) ein Vertex Cover der Größe  $2|M|$  findet, gilt für alle Instanzen  $x$ :

$$c_{AVC}(x) = 2|M| \leq 2 \cdot c_{opt}(x)$$

**Approx-Vertex-Cover:** Für einen bipartiten vollständigen Graphen (kurz  $K_{n,n}$ ) ist die Schranke sogar scharf.

**Hinweis:** Ein einfacher Graph heißt **bipartit** oder paar, wenn sich seine Knoten in zwei disjunkte Teilmengen  $A$  und  $B$  aufteilen lassen, sodass zwischen den Knoten innerhalb einer jeden Teilmenge keine Kanten verlaufen.

**Beispiel:** Approx-Vertex-Cover würde alle Knoten auswählen. Die optimale Lösung besteht aus den Knoten einer Seite.



## Alternativer Algorithmus

**Alternativer Algorithmus:** Wählt immer einen Knoten mit aktuell maximalem Grad.

**Approx-Vertex-Cover2(G):**

Approx-Vertex-Cover2( $G$ ):

$C \leftarrow \emptyset$

**while**  $E \neq \emptyset$

    Wähle einen Knoten  $u$  mit maximalem Grad im aktuellen Graphen

$C \leftarrow C \cup \{u\}$

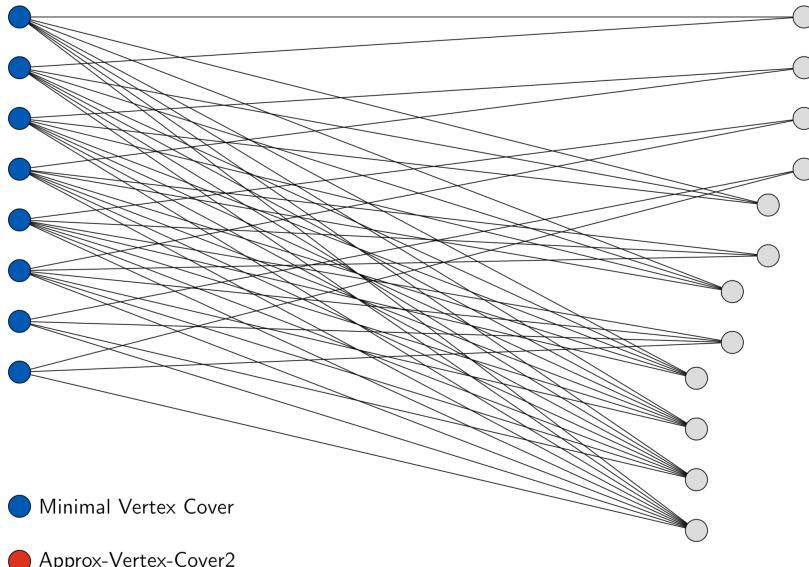
    Entferne aus  $E$  alle Kanten, die inzident zu  $u$  sind

**return**  $C$

**Gütegarantie:** Man kann zeigen, dass dieser Algorithmus eine logarithmische Gütegarantie hat. Die scheinbar intelligenteren Auswahl führt hier nicht zu einer Verbesserung!

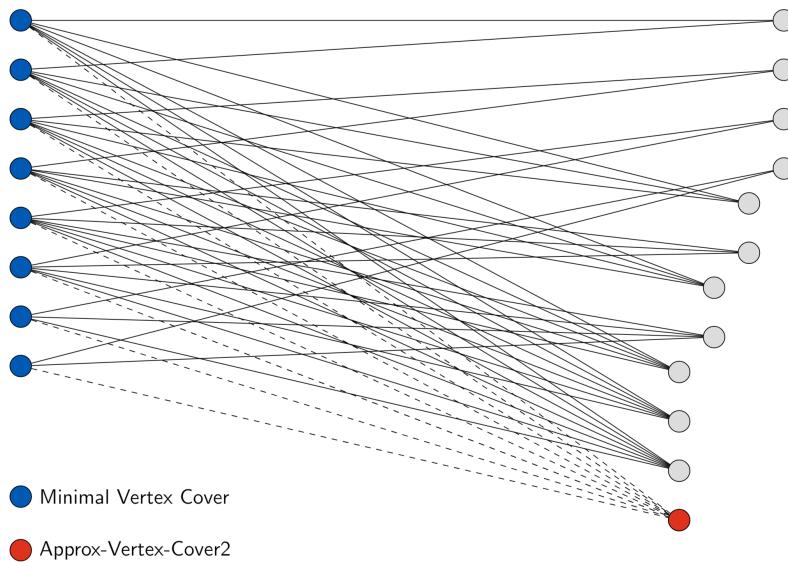
### ☰ Beispiel

Schlechtes Beispiel:

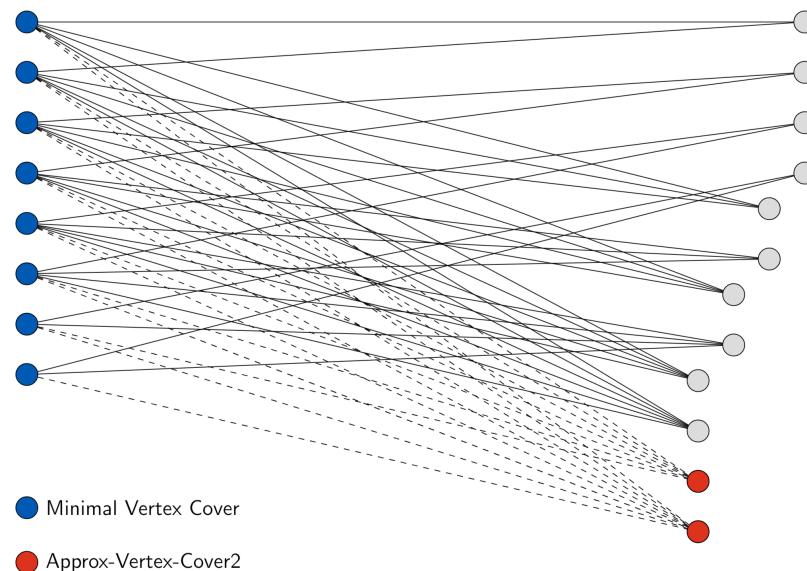


Hier wäre ja das minimale Vertex Cover, wenn man einfach die 8 auf der linken Seite nimmt, da diese alle abdecken würden. Allerdings nimmt unser Greedy Algorithmus ja den mit dem maximalen Grad.

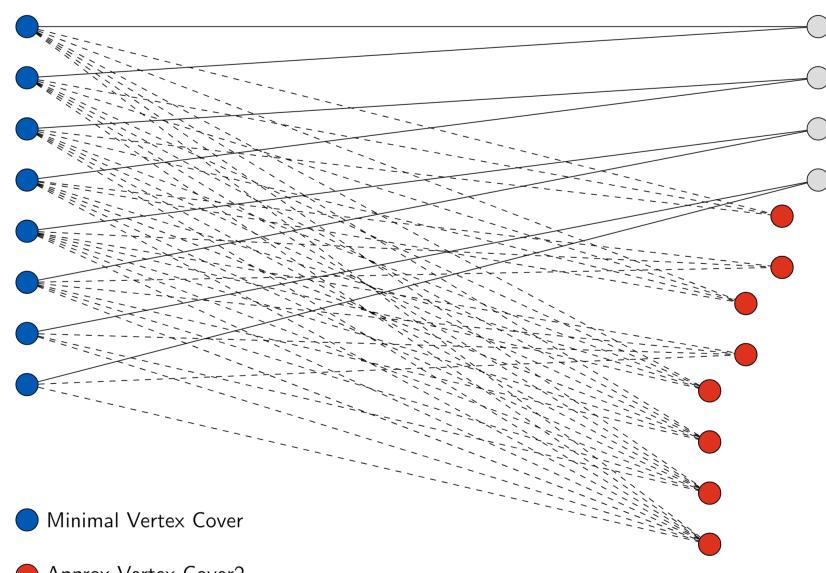
Schlechtes Beispiel:



Schlechtes Beispiel:

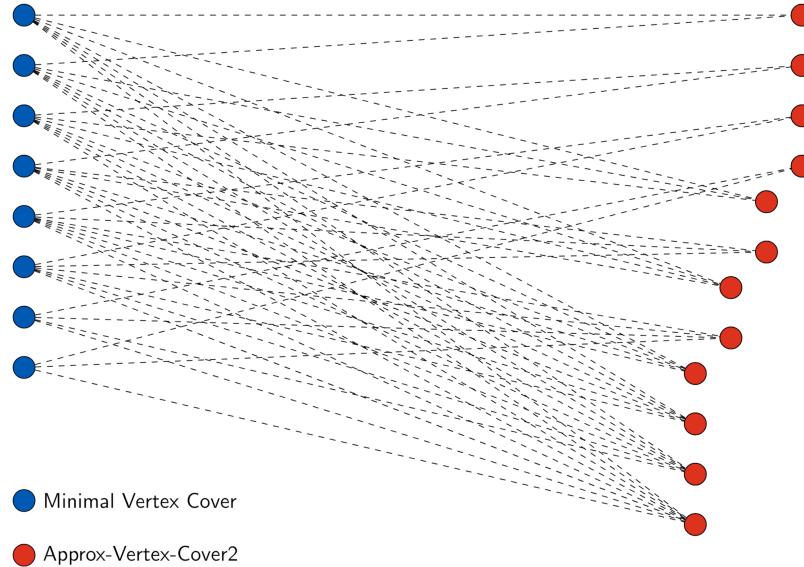


Schlechtes Beispiel:



Hier hat man dann auf der rechten Seite lauter Knoten mit Grad 2, während die linke Seite nur noch Knotengrad 1 hat, daher werden die letzten Knoten auch von rechts gewählt.

### Schlechtes Beispiel:



### Konstruktionsschema für schlechtes Beispiel:

- Wähle  $n$  Knoten auf der linken Seite.
- Für  $i = 2, \dots, n$  geben wir jeweils eine Menge  $R_i$  mit  $\lfloor \frac{n}{i} \rfloor$  Knoten auf der rechten Seite hinzu.
- Jeder Knoten aus der Menge  $R_i$  hat einen Grad  $i$  und ist jeweils mit  $i$  Knoten auf der linken Seite verbunden.

### Beispiel aus vorheriger Folie:

- Links gibt es 8 Knoten.
- Rechts gibt es 12 Knoten:
  - $R_2$  (4 Knoten mit Grad 2)
  - $R_3$  (2 Knoten mit Grad 3)
  - $R_4$  (2 Knoten mit Grad 4)
  - $R_5$  (1 Knoten mit Grad 5)
  - $R_6$  (1 Knoten mit Grad 6)
  - $R_7$  (1 Knoten mit Grad 7)
  - $R_8$  (1 Knoten mit Grad 8)

**Gütegarantie:** Für  $n$  Knoten auf der linken Seite.

- Approx-Vertex-Cover2 wählt alle Knoten auf der rechten Seite, d.h.

$$\sum_{i=2}^n |R_i| = \sum_{i=2}^n \lfloor \frac{n}{i} \rfloor \geq \sum_{i=2}^n (\frac{n}{i} - 1) \geq n \sum_{i=1}^n \frac{1}{i} - 2n = n(H_n - 2)$$

Knoten. Dabei ist  $H_n$  die  $n$ -te harmonische Zahl.

- Es gilt  $H_n = \ln n + \Theta(1)$ .
- Minimales Vertex Cover umfasst  $n$  Knoten. Gütegarantie ist daher:

$$\frac{n(H_n - 2)}{n} = \Omega(\log n)$$

# Spanning-Tree-Heuristik (ST) für das symmetrische TSP

## ⌚ Traveling Salesmen Problem - Wiederholung

- Man sucht eine Reihenfolge (Tour) für den Besuch mehrerer Orte, sodass die gesamte Reisestrecke eines Handlungsreisenden möglichst kurz ist (Minimierungsproblem).
- Jeder Ort wird genau einmal besucht und nach dem letzten Ort wird zum ersten zurückgekehrt.

## ☰ Beispiel

Beispiel: Die 13509 Städte der USA mit mehr als 500 Einwohner.



Beispiel: Die 13509 Städte der USA mit mehr als 500 Einwohner: Optimale Tour



## Traveling Salesperson Problem (TSP)

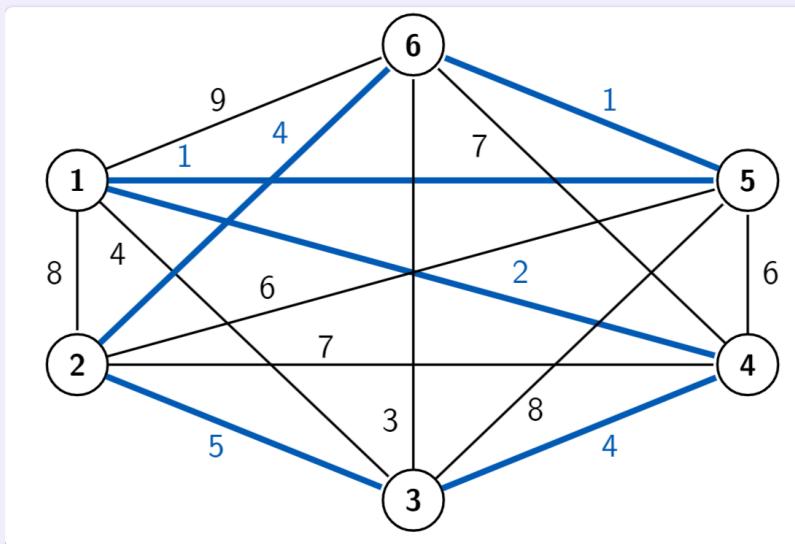
**Darstellung:** Die kürzesten Wege zwischen allen Knotenpaaren können durch einen gewichteten vollständigen Graphen repräsentiert werden.

### Vollständiger Graph:

- Ist ein schlichter Graph, in dem jeder Knoten mit jedem anderen Knoten durch eine Kante verbunden ist.
- Ein vollständiger Graph mit  $n$  Knoten wird als  $K_n$  bezeichnet.

#### ☰ Beispiel

6 Orte, minimale Tour der Länge 17.



## Symmetrisches TSP

## ⌚ Symmetrisches TSP

- Gegeben ist ein ungerichteter vollständiger Graph  $G = (V, E)$  mit Distanzmatrix  $c$  mit  $c_{ii} = +\infty$  und  $c_{ij} \geq 0$ .
- Für alle Knotenpaare  $(i, j)$  sind die Distanzen in beide Richtungen identisch, d.h. es gilt  $c_{ij} = c_{ji}$ .
- Jede Tour hat dieselbe Länge in beide Richtungen.

**Hinweis:** Das TSP ist NP-schwer (Beweis durch Reduktion von HAM-CYCLE).

## Minimum Spanning Tree Heuristik (MST) für das sym. TSP

**Spanning-Tree-Heuristik:** Tour wird aus einem Minimum Spanning Tree (MST) für den gegebenen Graphen  $G$  abgeleitet.

### 📋 Vorgehen

1. Bestimme einen MST  $(V, B_1)$  von  $G$ .
2. Verdopple alle Kanten in  $B_1$ , das ergibt Graph  $(V, B_2)$ . Dieser Graph ist eulersch.
3. Bestimme eine **Euler-Tour**  $T$  im Graphen  $(V, B_2)$ . Gib dieser Tour eine Orientierung, wähle einen Knoten  $p \in V$ , markiere  $p$  als besucht, setze  $T' \leftarrow (p)$ ,  $i \leftarrow 0$ .
4. Sind alle Knoten markiert, setze  $T'$  als Ergebnis-Tour.
5. Laufe von  $p$  entlang der Orientierung von  $T$  bis ein unmarkierter Knoten  $q$  erreicht ist. Setze  $T' \leftarrow T' \cup \{(p, q)\}$ , markiere  $q$ , setze  $p \leftarrow q$  und gehe zu (4).

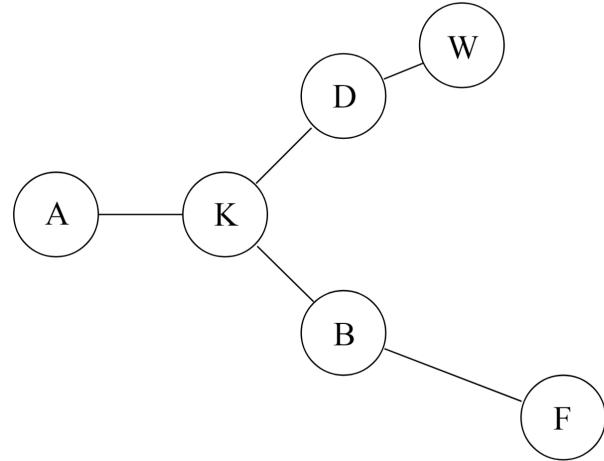
### ⌚ Euler-Tour

Ein Graph hat eine Euler-Tour (ein Rundgang, der jede Kante genau einmal benutzt und zum Startknoten zurückkehrt) genau dann, wenn alle seine Knoten einen geraden Grad haben. Durch das Verdoppeln der Kanten im MST stellen wir sicher, dass jeder Knoten im resultierenden Graphen einen geraden Grad hat.

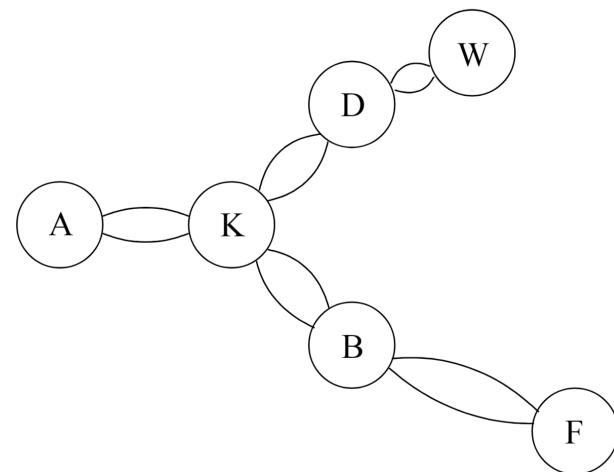
### ☰ Beispiel

Ausgangspunkt: Vollständiger Graph mit 6 Knoten.

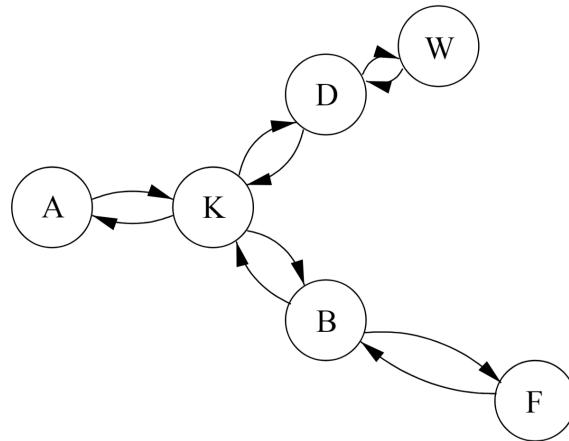
Schritt 1: MST  $(V, B_1)$  sieht beispielsweise folgendermaßen aus:



Schritt 2: Alle Kanten im MST werden verdoppelt  $(V, B_2)$ .



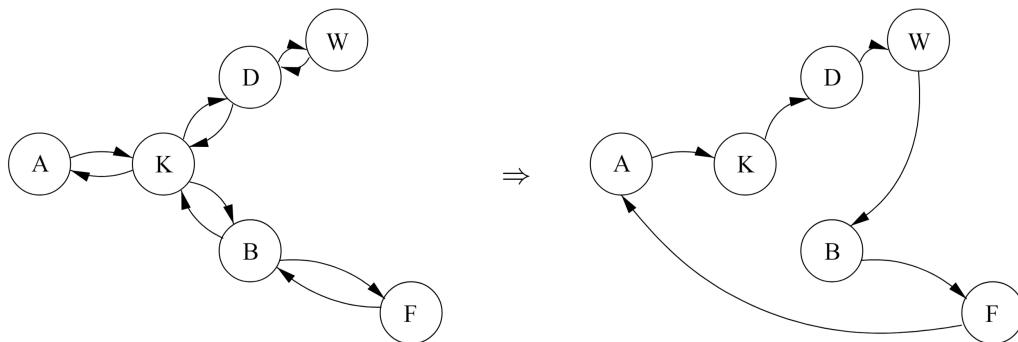
Schritt 3: Bestimme Eulertour  $F$ , die jede Kante genau einmal enthält.



Eulertour:  $F = (A, K, D, W, D, K, B, F, B, K, A)$

Schritt 4 und 5: Eulertour  $\Rightarrow$  TSP-Tour

- $A$  ist der Startknoten.
- In der Tour wird jeder Knoten nur einmal besucht.
- Wird ein Knoten besucht, dann wird er markiert.
- Der nächste TSP-Tour-Knoten ist immer der nächste unmarkierte Knoten auf der Eulertour.



### ⓘ Theorem

Eine Euler-Tour existiert in einem ungerichteten Graphen genau dann, wenn er zusammenhängend ist und jeder Knoten einen geraden Grad hat.

**Hinweis:** Durch die Verdopplung der Kanten im MST hat jeder Knoten geraden Grad. Eine Euler-Tour existiert somit immer.

## Spanning-Tree-Heuristik (ST): Gütegarantie

**Laufzeit:** Der erste Schritt (MST finden) kann beispielsweise mit Prims Algorithmus in Zeit  $O(n^2)$  gelöst werden. Die restlichen Schritte sind nicht aufwendiger und daher läuft die gesamte Heuristik in Zeit  $O(n^2)$ .

## Algorithmus von Hierholzer (1873)

**Hinweis:** Findet eine Euler-Tour (falls vorhanden) in einem Graphen  $G$  in linearer Zeit.

### Grundlegende Idee:

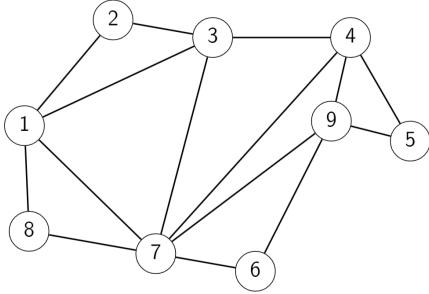
- Beginnend von einem Startknoten wird ein Pfad (nicht notwendigerweise einfach) mit einer noch nicht benutzten Kante fortgesetzt. Wenn alle Knoten geraden Grad haben, kehrt man wieder zum Ausgangsknoten zurück. Dieser geschlossene Pfad bildet einen **Zyklus** mit den Knoten  $v_1, v_2, \dots, v_k, v_1$ .
- Die Kanten  $E(Z) = \{(v_1, v_2), (v_2, v_3), \dots, (v_{k-1}, v_k), (v_k, v_1)\}$  des gefundenen Zyklus  $Z$  werden aus dem Graphen entfernt. Im Restgraphen sind, wenn eine Euler-Tour vorhanden ist, wiederum alle Knotengrade gerade.
- Gibt es in einem Zyklus  $Z$  einen Knoten mit einem Grad größer 0, dann kann man von dort aus wiederum einen Zyklus bilden.

### Vorgehen

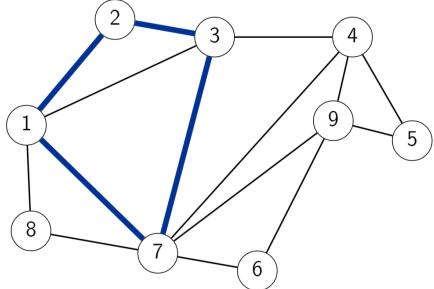
1. Wähle einen beliebigen Knoten  $v_0$  des Graphen und konstruiere von  $v_0$  ausgehend einen geschlossenen Pfad  $Z$  in  $G$ , der keine Kante in  $G$  zweimal durchläuft. Wir nennen  $Z$  einen Zyklus.
2. Wenn  $Z$  eine Euler-Tour ist (also alle Knoten beinhaltet), terminiere. Andernfalls:
3. Lösche nun alle Kanten aus  $G$ .
4. An einem Knoten  $v$  des Zyklus  $Z$ , der dessen Grad im aktuellen (Rest-)Graphen größer 0 ist, startet man nun einen neuen Zyklus  $Z'$ .
5. Füge  $Z'$  in den Zyklus  $Z$  ein, indem der Startpunkt von  $Z'$  beim ersten Auftreten in  $Z$  durch alle Knoten von  $Z'$  in der durchlaufenden Reihenfolge ersetzt wird.
6. Fahre bei Schritt 2 fort.

### Beispiel

Ausgangsgraph:

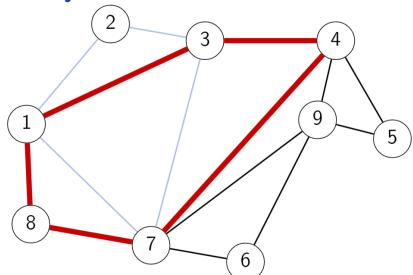


1. Zyklus:



$$Z = (1, 2, 3, 7, 1)$$

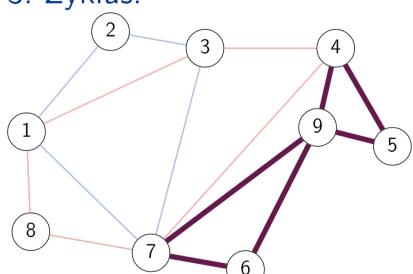
2. Zyklus:



$$Z' = (1, 3, 4, 7, 8, 1)$$

$$Z = (1, 3, 4, 7, 8, 1, 2, 3, 7, 1)$$

3. Zyklus:

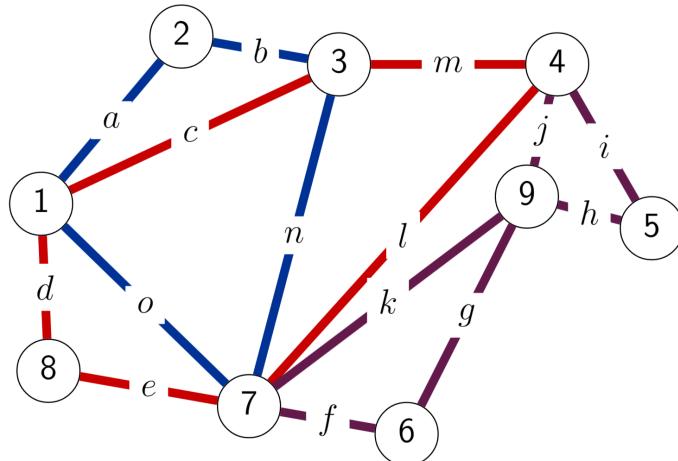


$$Z' = (7, 6, 9, 5, 4, 9, 7)$$

$$Z = (1, 3, 4, 7, 6, 9, 5, 4, 9, 7, 8, 1, 2, 3, 7, 1)$$

## Mögliche Eulertour: Weiteres Beispiel

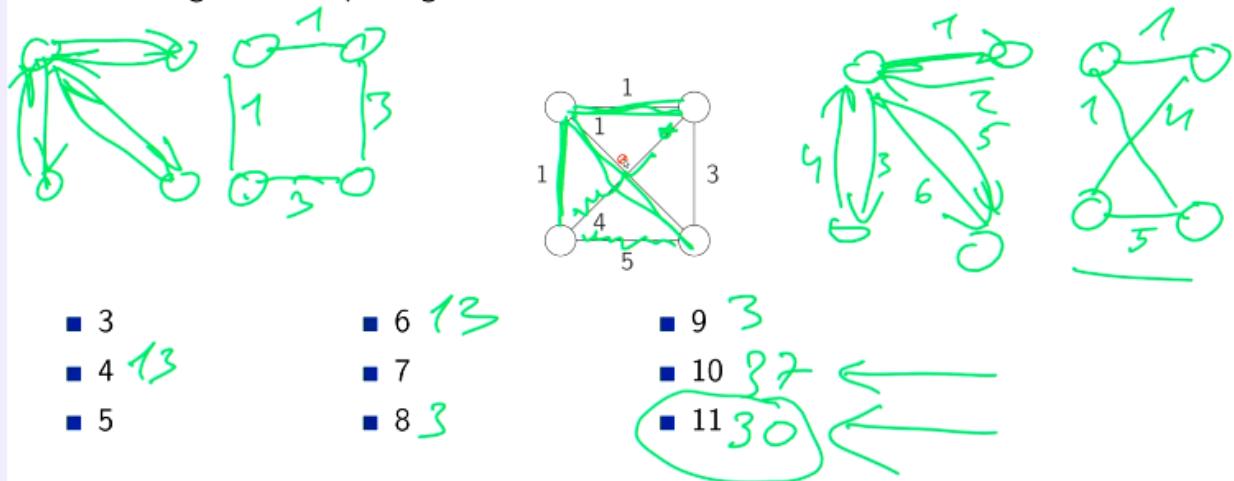
- Kantenfolge: a, b, c, d, e, f, g, h, i, j, k, l, m, n, o
- Knotenfolge: 1, 2, 3, 1, 8, 7, 6, 9, 5, 4, 9, 7, 4, 3, 7, 1



Laufzeit: Liegt in  $O(n + m)$ .

## ☰ Beispiel

Frage 3: Welche Länge hat die schlechteste Tour, die von der Spanning-Tree-Heuristik auf dem folgenden Graphen gefunden wird?



## Ist das symmetrische TSP 2-approximierbar?

- **Theorem:** Angenommen  $P \neq NP$ . Dann gibt es keinen polynomiellen 2-Approximationsalgorithmus für das symmetrische TSP (Travelling Salesperson Problem).

### ✓ Beweis (durch Widerspruch)

1. **Annahme:** Es existiert ein polynomieller Approximationsalgorithmus  $A$  mit einer Gütegarantie 2 für das symmetrische TSP.

2. Sei  $G = (V, E)$  ein Graph, für den wir bestimmen wollen, ob er einen Hamiltonkreis enthält (dies ist ein NP-vollständiges Problem).
3. Wir möchten nun Algorithmus  $A$  benutzen, um den Hamiltonkreis effizient zu finden.
4. Dazu wird Graph  $G$  in eine TSP-Instanz  $G' = (V, E', c)$  transformiert.

- Transformation von  $G$  in eine TSP-Instanz  $G' = (V, E', c)$ :

- Sei  $(V, E')$  der vollständige Graph, d.h.  $E' = \{(u, v) : u, v \in V, u \neq v\}$ .

$$c_{u,v} = \begin{cases} 1 & \text{wenn } (u, v) \in E \\ 2 \cdot n + 1 & \text{sonst} \end{cases}$$

Die Kantenkosten  $c_{u,v}$  sind definiert als:

1, wenn die Kante  $(u, v)$  in  $E$  (dem ursprünglichen Graphen) enthalten ist.

$2 \cdot n + 1$ , wenn die Kante  $(u, v)$  nicht\* in  $E$  enthalten ist. Hierbei ist  $n$  die Anzahl der Knoten in  $V$ .

- Graph  $G$  kann in polynomieller Zeit in  $G'$  transformiert werden.

- Fall 1:  $G$  enthält einen Hamiltonkreis  $H$

- Wenn  $G$  einen Hamiltonkreis  $H$  enthält, dann werden jeder Kante von  $H$  die Kosten 1 zugewiesen.
- $G'$  enthält somit eine Tour mit den Kosten  $n$  (da ein Hamiltonkreis  $n$  Kanten hat und jede Kante Kosten 1 hat).

- Fall 2:  $G$  enthält keinen Hamiltonkreis

- Wenn  $G$  keinen Hamiltonkreis enthält, dann besitzt jede Tour in  $G'$  mindestens eine Kante, die sich *nicht* in  $E$  befindet (da jede Tour, die nur Kanten aus  $E$  enthält, per Definition ein Hamiltonkreis in  $G$  wäre).
- Solch eine Tour hat aber Kosten von mindestens:
  - $(2n + 1)$  für die eine Kante, die nicht in  $E$  ist.
  - Plus  $(n - 1)$  für die restlichen  $n - 1$  Kanten (die mindestens Kosten 1 haben).
  - Gesamtkosten:  $(2n + 1) + (n - 1) = 2n + n = 3n > 2n$ .

- Daher sind die Kosten einer Tour in  $G'$ , die *kein* Hamiltonkreis in  $G$  ist, zumindest um den Faktor 3 größer als die einer Tour, die *einen* Hamiltonkreis in  $G$  ist.
  - (Begründung: Kosten einer Hamiltonkreis-Tour sind  $n$ . Kosten einer Nicht-Hamiltonkreis-Tour sind  $3n$ . Das Verhältnis ist  $3n/n = 3$ ).
- Nun wenden wir Algorithmus  $A$  auf die TSP-Instanz  $G'$  an.
- Algorithmus  $A$  liefert garantiert eine Tour zurück, deren Kosten höchstens um den Faktor 2 über denen einer optimalen Tour liegen (dies ist die definierte Gütegarantie 2 von Algorithmus  $A$ ).

- **Fall 1: Wenn  $G$  einen Hamiltonkreis enthält,** dann muss Algorithmus  $A$  ihn zurückliefern.
  - Begründung: Die optimale Tour in  $G'$  hätte Kosten  $n$  (den Hamiltonkreis). Da  $A$  eine Tour liefert, deren Kosten höchstens  $2 \cdot n$  betragen, und die einzige andere Option (eine Tour ohne Hamiltonkreis in  $G$ ) Kosten von  $3n$  oder mehr hat, muss  $A$  den Hamiltonkreis finden.
- **Fall 2: Wenn  $G$  keinen Hamiltonkreis enthält,** dann liefert Algorithmus  $A$  eine Tour mit Kosten größer als  $2 \cdot n$ .
  - Begründung: Wenn  $G$  keinen Hamiltonkreis enthält, ist die optimale Tour in  $G'$  eine, die mindestens eine Kante mit hohen Kosten ( $2n + 1$ ) enthält. Die Kosten einer solchen Tour sind mindestens  $3n$ . Da  $A$  eine Tour mit Kosten von *höchstens*  $2 \times$  (Kosten der optimalen Tour) liefert, kann  $A$  keine Tour mit Kosten  $\leq 2n$  finden, wenn die optimale Tour Kosten  $\geq 3n$  hat.
- Daraus folgt: Algorithmus  $A$  kann benutzt werden, um einen Hamiltonkreis in polynomieller Zeit zu finden.
  - Dies ist aber ein Widerspruch dazu, dass HAM-CYCLE NP-schwer ist.
  - Dieser Widerspruch tritt nur auf, wenn  $P = NP$  gilt.
  - Da wir von  $P \neq NP$  ausgegangen sind, ist die ursprüngliche Annahme (Existenz eines 2-Approximationsalgorithmus) falsch.
  - Daher kann es keinen polynomiellen 2-Approximationsalgoritmus für das symmetrische TSP geben, *wenn*  $P \neq NP$ .

## Spanning-Tree-Heuristik (ST): Gütegarantie

- 
- **Theorem:** Ähnlich wie für den Fall der 2-Approximation kann für ein **allgemeines**  $\varepsilon > 1$  gezeigt werden, dass es keinen polynomiellen  $\varepsilon$ -Approximationsalgoritmus für das symmetrische TSP gibt.
  - Das symmetrische TSP ist somit "**nicht approximierbar**" (im allgemeinen Fall, wenn  $P \neq NP$  gilt).

### ✓ Beweis

- Der Beweis folgt dem gleichen Prinzip wie auf den vorherigen Folien (für die 2-Approximation).
- Der einzige Unterschied liegt in der Definition der Kantenkosten:
  - Nun gilt  $c(u, v) = \varepsilon \cdot n + 1$ , wenn  $(u, v) \notin E$  (wobei  $\varepsilon$  ein beliebiger Wert größer 1 ist).

## ② Frage

- Wozu dann der ganze Aufwand, wenn eine beliebige Approximation (d.h. mit einem beliebigen  $\varepsilon$ ) nicht effizient möglich ist?

Antwort:

- Unter einer **einfachen Voraussetzung** wird das Ergebnis besser (d.h. es gibt Fälle, in denen eine Approximation doch möglich oder effizienter ist).

## Metrisches TSP

### ⌚ Metrisches TSP

- Ein TSP (Travelling Salesperson Problem) heißt **metrisch**, wenn für die Distanzmatrix  $C$  die **Dreiecksungleichung** gilt.
- Das bedeutet, für alle Knoten  $i, j, k$  muss die Bedingung erfüllt sein:
  - $c_{ik} \leq c_{ij} + c_{jk}$
  - (Intuitiv: Der direkte Weg von  $i$  nach  $k$  ist nie länger als der Weg von  $i$  über  $j$  nach  $k$ . Man kann keinen "Short-Cut" über einen dritten Punkt finden.)
- **Hinweis:** Insbesondere ist auch das **Euklidische TSP** metrisch. Hierbei entsprechen die Knoten Punkten in der euklidischen Ebene, und die Distanzen sind die euklidischen Distanzen.

## ⓘ Theorem

- Das metrische TSP besitzt einen polynomiellen Approximationsalgorithmus mit einer **Gütegarantie von 2**.
- Das bedeutet, für eine gegebene TSP-Instanz  $x$ :
  - $\frac{c_{ST}(x)}{c_{opt}(x)} \leq 2$
  - Hierbei ist  $c_{ST}(x)$  die Kosten der Tour, die von der Spanning-Tree-Heuristik gefunden wird, und  $c_{opt}(x)$  sind die Kosten der optimalen Tour.
  - (Die von der ST-Heuristik gefundene Tour ist also maximal doppelt so lang wie die optimale Tour.)

## Laufzeit:

- Die Spanning-Tree-Heuristik läuft in **polynomieller Zeit**.

- Der **aufwändigste Schritt** ist die Ermittlung des **MST** (Minimum Spanning Tree) im ersten Schritt.

### ✓ Beweis für Gütegarantie

Beweis für Gütegarantie: Es gilt

$$c_{\text{ST}}(x) \leq c_{B_2}(x) = 2c_B(x) \leq 2c_{\text{opt}}(x)$$

- Gilt wegen der Dreiecksungleichung.
- Gilt, da  $B_2$  durch Verdopplung der Kanten aus  $B$  entsteht. ■ Gilt, da ein MST die kürzeste Möglichkeit ist, in einem Graphen alle Knoten zu verbinden.

- Begründung der Ungleichungen:**

- $c_{\text{ST}}(x) \leq c_{B_2}(x)$ : Gilt wegen der Dreiecksungleichung.
  - (Dies bezieht sich auf die Umwandlung eines "doppelt durchlaufenen" MST in eine Tour, wobei Kanten unter Umständen abgekürzt werden können, da die Dreiecksungleichung gilt.)
- $c_{B_2}(x) = 2c_B(x)$ : Gilt, da  $B_2$  durch Verdopplung der Kanten aus  $B$  entsteht.
  - (Hier ist  $B$  der minimale Spannbaum und  $B_2$  ein Graph, in dem jede Kante des MST zweimal vorkommt, was einen Eulerkreis ermöglicht, der dann in eine Tour umgewandelt wird.)
- $c_B(x) \leq c_{\text{opt}}(x)$ : Gilt, da ein MST die kürzeste Möglichkeit ist, in einem Graphen alle Knoten zu verbinden (ohne Zyklen).
  - (Jede TSP-Tour enthält einen Spannbaum. Der MST ist der "billigste" Spannbaum, daher sind seine Kosten kleiner oder gleich den Kosten jedes anderen Spannbaums, einschließlich des Spannbaums, der Teil einer optimalen TSP-Tour ist.)

Anschaulich siehe: [Beispiel dazu](#)

# Lastverteilung (Load Balancing)

- **Eingabe:**

- $m$  identische Maschinen
- $n$  Jobs (Aufgaben)
- Jeder Job  $j$  hat eine Bearbeitungszeit  $t_j$ .
- Jeder Job  $j$  muss ununterbrochen auf einer Maschine ausgeführt werden.
- Eine Maschine kann nur einen Job auf einmal ausführen.

 **Definition: Last einer Maschine**

- Sei  $J_i$  die Teilmenge von Jobs, die Maschine  $i$  zugewiesen wurde.
- Die Last der Maschine  $i$  ist

$$L_i = \sum_{j \in J_i} t_j$$

. (Summe der Bearbeitungszeiten aller Jobs auf Maschine  $i$ ).

 **Definition: Bearbeitungsdauer (Makespan)**

- Die Bearbeitungsdauer (oder Makespan) ist die maximale Last auf irgendeiner Maschine.
- $L = \max_{i=1, \dots, m} L_i$ .

- **Ziel der Lastverteilung:**

- Teile jeden Job einer Maschine so zu, dass die Bearbeitungsdauer (Makespan) minimiert wird.

# Lastverteilung: List-Scheduling

## List-Scheduling-Algorithmus:

- Berücksichtigt  $n$  Jobs mit einer fixen Ordnung (d.h. die Jobs werden in einer bestimmten, vorher festgelegten Reihenfolge verarbeitet).
- **Greedy-Algorithmus:** Teile Job  $j$  einer Maschine mit der aktuell kleinsten Last zu (lokale optimale Entscheidung).
  - $L_i$ : Aktuelle Last auf Maschine  $i$ .
  - $J_i$ : Menge der Jobs, die Maschine  $i$  zugewiesen wurden.

## List-Scheduling( $m, n, t_1, t_2, \dots, t_n$ ):

```

List-Scheduling( $m, n, t_1, t_2, \dots, t_n$ ):
  for  $i \leftarrow 1$  bis  $m$ 
     $J_i \leftarrow \emptyset$ 
     $L_i \leftarrow 0$ 
  for  $j \leftarrow 1$  bis  $n$ 
     $i = \operatorname{argmin}_{k=1, \dots, m} L_k$ 
     $J_i \leftarrow J_i \cup \{j\}$ 
     $L_i \leftarrow L_i + t_j$ 
  return  $J_1, \dots, J_m$ 
```

■ Maschine  $i$  hat geringste Last

**Laufzeit:**  $O(n \log m)$  mit priority Queue

### ☰ Beispiel

Frage 4: Gegeben sind drei Maschinen und die folgenden Jobs:

Job	$j_1$	$j_2$	$j_3$	$j_4$	$j_5$	$j_6$
Bearbeitungszeit	2	2	5	5	6	10



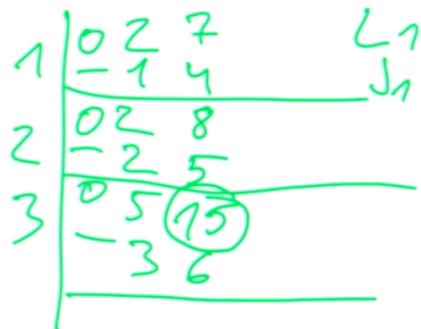
Für welche Reihenfolge an Jobs liefert der List-Scheduling Algorithmus das **schlechteste** Ergebnis?

- ✗ (A)  $j_1, j_2, j_3, j_4, j_5, j_6$
- ✗ (B)  $j_6, j_5, j_4, j_3, j_2, j_1$
- ✓ (C)  $j_1, j_2, j_5, j_4, j_3, j_6$
- ✗ (D)  $j_1, j_4, j_3, j_2, j_5, j_6$

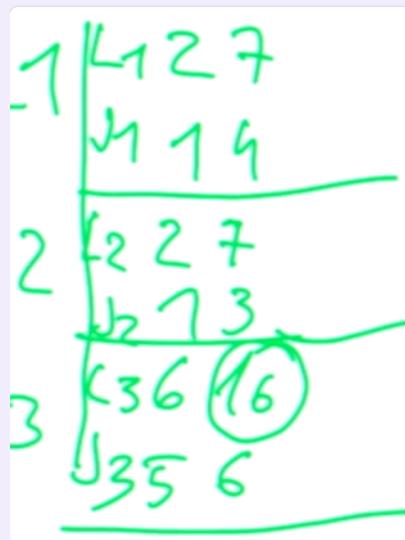
(A) Liefert uns das Ergebnis 15

Für welche Reihenfolge an Jobs liefert der List-Scheduling Algorithmus das schlechteste Ergebnis?

- (A)  $j_1, j_2, j_3, j_4, j_5, j_6$  ←
- (B)  $j_6, j_5, j_4, j_3, j_2, j_1$
- (C)  $j_1, j_2, j_5, j_4, j_3, j_6$
- (D)  $j_1, j_4, j_3, j_2, j_5, j_6$



(C) liefert uns das Ergebnis 16



Die anderen muss man sich genau so anschauen

## Theorem:

- [Graham, 1966] List-Scheduling ist ein **2-Approximationsalgorithmus**.
- Dies ist die erste Worst-Case-Analyse eines Approximationsalgorithmus.
- Dazu muss die resultierende Lösung (vom List-Scheduling) mit der **optimalen Bearbeitungsdauer  $L^*$**  verglichen werden.

### ① Lemma 1

Für die optimale Dauer  $L^*$  gilt in jedem Fall:  $L^* \geq \max_j t_j$ .

#### ✓ Beweis

Eine Maschine muss den aufwendigsten Job ( $j$ ) mit der Bearbeitungszeit  $t_j$  verarbeiten. Da die optimale Dauer  $L^*$  die **maximale** Last auf einer Maschine ist,

muss sie mindestens so groß sein wie die Bearbeitungszeit des längsten einzelnen Jobs.

### ⓘ Lemma 2

Für die optimale Dauer  $L^*$  gilt in jedem Fall:

$$L^* \geq \frac{1}{m} \sum_j t_j$$

#### ✓ Beweis

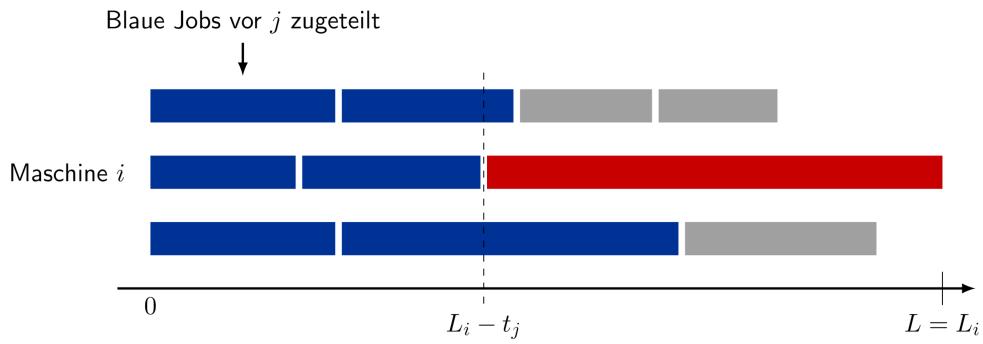
- Die gesamte Verarbeitungszeit aller Jobs ist  $\sum_j t_j$ .
- Eine der  $m$  Maschinen muss zumindest den  $1/m$ -ten Teil der gesamten Arbeit machen (im Durchschnitt). Da  $L^*$  die maximale Last ist, muss sie mindestens so groß sein wie der Durchschnitt der Lasten über alle Maschinen.

### Theorem:

List-Scheduling ist ein 2-Approximationsalgorithmus.

#### ✓ Beweis

- Wir betrachten die Last  $L_i$  einer Maschine  $i$ , die einen Engpass darstellt (d.h. die Maschine mit der maximalen Last, also  $L_i = L$ ).
- Sei  $j$  der letzte zugeteilte Job auf Maschine  $i$ .
- **Wenn Job  $j$  Maschine  $i$  zugewiesen wird, hat  $i$  die geringste Last.**
  - Die Last auf Maschine  $i$  vor der Zuteilung von Job  $j$  war  $L_i - t_j$ .
  - Da Maschine  $i$  zu diesem Zeitpunkt die geringste Last hatte, muss gelten:  $L_i - t_j \leq L_k$  für alle  $1 \leq k \leq m$ .
  - Somit ist  $L_i - t_j \leq \min_{k=1,\dots,m} L_k$ .



- Die Abbildung zeigt, dass die Summe der "blauen" Jobs (also die Jobs auf allen Maschinen vor der Zuteilung des letzten Jobs  $j$  auf Maschine  $i$ ) mindestens so groß ist wie  $m \cdot (L_i - t_j)$ , da jede Maschine zu diesem Zeitpunkt eine Last von mindestens  $L_i - t_j$  hatte.
- Wir betrachten die Last  $L_i$  einer Maschine  $i$ , die einen Engpass darstellt (d.h.,  $L_i$  ist die maximale Last  $L$ ).
- Sei  $j$  der letzte zugeteilte Job auf Maschine  $i$ .
- Wenn Job  $j$  Maschine  $i$  zugewiesen wird, hatte  $i$  die geringste Last.
  - Die Last auf Maschine  $i$  vor der Zuteilung von Job  $j$  war  $L_i - t_j$ .
  - Es gilt:  $L_i - t_j \leq L_k$  für alle Maschinen  $k$  (da  $i$  die geringste Last hatte).
- Wir summieren alle Ungleichungen über alle Maschinen  $k$  und dividieren durch  $m$ :

$$\begin{aligned} L_i - t_j &\leq \frac{1}{m} \sum_k L_k \\ &= \frac{1}{m} \sum_j t_j \\ &\leq L^* \end{aligned}$$

Nun ist  $L_i = \underbrace{(L_i - t_j)}_{\leq L^*} + \underbrace{t_j}_{\leq L^*} \leq 2L^*$ .  $\square$

□ Lemma 2 □ Lemma 1

# Center Selection

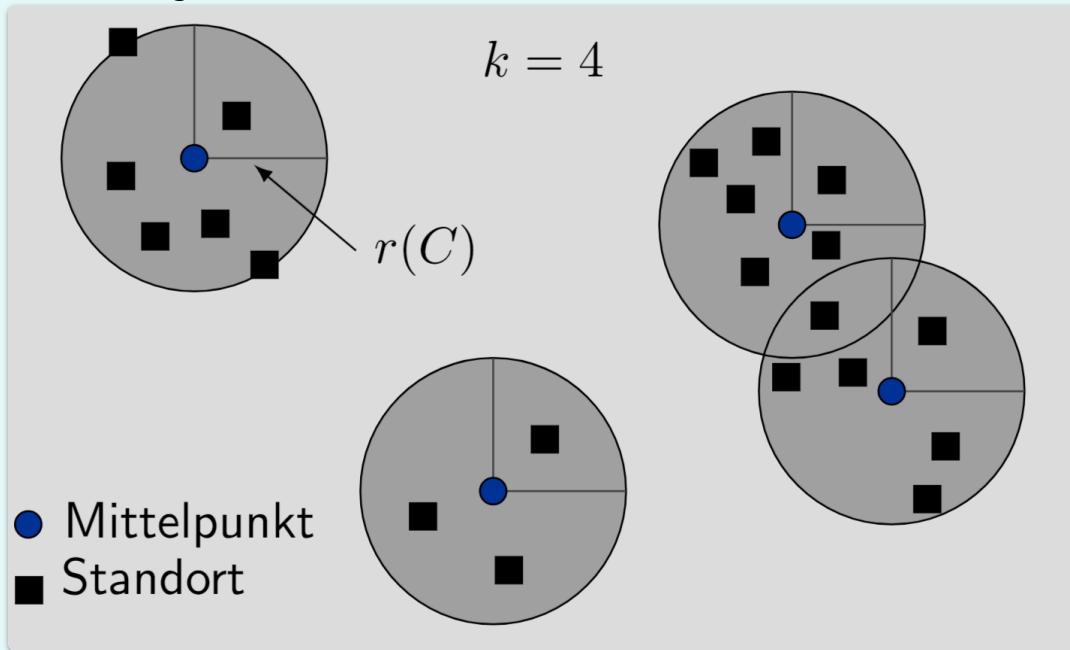
## Definition

Eingabe:

- Eine Menge von  $n$  Standorten  $s_1, \dots, s_n$ .
- Eine ganze Zahl  $k > 0$ .

Problem

- Wähle  $k$  Mittelpunkte  $C$ , sodass die **maximale Distanz** von einem Standort zu einem nächsten Mittelpunkt minimiert wird.
- (Ziel ist es,  $k$  "Zentren" so zu platzieren, dass der am weitesten entfernte Standort so nah wie möglich an einem Zentrum ist.)



Notation:

- $\text{dist}(x, y)$ : Distanz zwischen  $x$  und  $y$ .
- $\text{dist}(s_i, C)$ : Distanz von Standort  $s_i$  zu seinem nächsten Mittelpunkt.
  - Formal:  $\text{dist}(s_i, C) = \min_{c \in C} \text{dist}(s_i, c)$ .
- $r(C)$ : Der kleinste überdeckende Radius für eine gegebene Menge von Mittelpunkten  $C$ .
  - Formal:  $r(C) = \max_{s_i \in S} \text{dist}(s_i, C)$ .
  - (Dies ist die Metrik, die minimiert werden soll – der Radius des kleinsten Kreises, der alle Standorte überdeckt, wenn die Kreise um die Mittelpunkte liegen.)

Ziel:

- Finde eine Menge von Mittelpunkten  $C$ , die  $r(C)$  minimiert, unter Berücksichtigung, dass die Kardinalität von  $C$  gleich  $k$  sein muss ( $|C| = k$ ).

## Eigenschaften der Distanzfunktion:

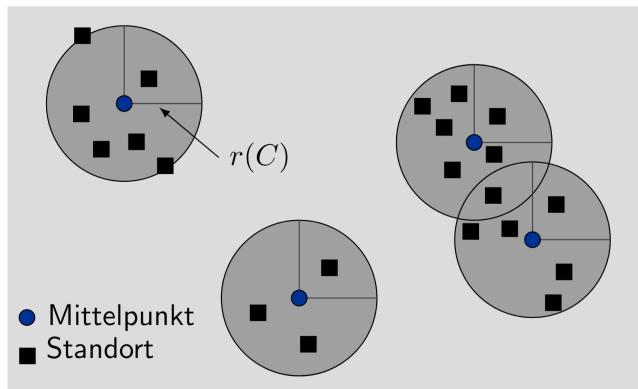
Eine gültige Distanzfunktion (Metrik) erfüllt folgende Eigenschaften:

- $\text{dist}(x, x) = 0$  (Identität: Die Distanz von einem Punkt zu sich selbst ist Null).
- $\text{dist}(x, y) = \text{dist}(y, x)$  (Symmetrie: Die Distanz von  $x$  nach  $y$  ist gleich der Distanz von  $y$  nach  $x$ ).
- $\text{dist}(x, y) \leq \text{dist}(x, z) + \text{dist}(z, y)$  (Dreiecksungleichung: Der direkte Weg ist nie länger als ein Umweg über einen dritten Punkt).

### :≡ Beispiel

**Beispiel:** Jeder Standort ist ein Punkt in der Ebene, ein Mittelpunkt kann jeder Punkt in der Ebene sein,  $\text{dist}(x, y) =$  Euklidische Distanz.

**Anmerkung:** Es gibt unendlich viele potentielle Lösungen!



## Greedy-Algorithmus

### ⚡ Falsche Ansatz

- **Greedy-Algorithmus (falscher Ansatz):**
  1. Wähle für den ersten Mittelpunkt einen "besten" Platz für einen Mittelpunkt.
  2. Danach füge iterativ Mittelpunkte hinzu, sodass der Abdeckradius immer am meisten reduziert wird.
- **Anmerkung:** Dieser Ansatz kann beliebig schlecht sein!



Mittelpunkt 1



Schlechte Wahl für  
 $k = 2$  Mittelpunkte

- Mittelpunkt
- Standort

- Das Bild zeigt ein Beispiel, wo ein initialer Mittelpunkt "intuitiv" in die Mitte einer Gruppe gesetzt wird. Wenn dann ein zweiter Mittelpunkt hinzugefügt werden soll, kann es sein, dass dieser nicht optimal platziert wird, weil die erste Wahl schon eine schlechte Ausgangsbasis geschaffen hat.

## Richtiger Ansatz

### Greedy-Algorithmus (korrekter Ansatz für Center Selection)

1. Wähle den ersten Mittelpunkt **beliebig** aus der Menge aller Standorte.
2. Wähle wiederholt als nächsten Mittelpunkt einen Standort, der am **weitesten von jedem existierenden Mittelpunkt entfernt ist**.
  - (Dies ist eine Max-Min-Strategie: Maximiere den minimalen Abstand zum nächsten Zentrum, um eine gute Verteilung zu erzielen.)

### Greedy-Center-Selection( $k, n, s_1, s_2, \dots, s_n$ ):

```

Greedy-Center-Selection( $k, n, s_1, s_2, \dots, s_n$ )
 $C = \{s_1\}$ 
wiederhole  $k - 1$  mal
  Wähle einen Standort  $s_i$  mit maximaler  $\text{dist}(s_i, C)$ 
  Füge  $s_i$  zu  $C$  hinzu
return  $C$ 
```

- **Standort am weitesten von jedem Mittelpunkt:** Das bedeutet, der Standort  $s_t$  wird so gewählt, dass  $\min_{c \in C} \text{dist}(s_t, c)$  maximiert wird.

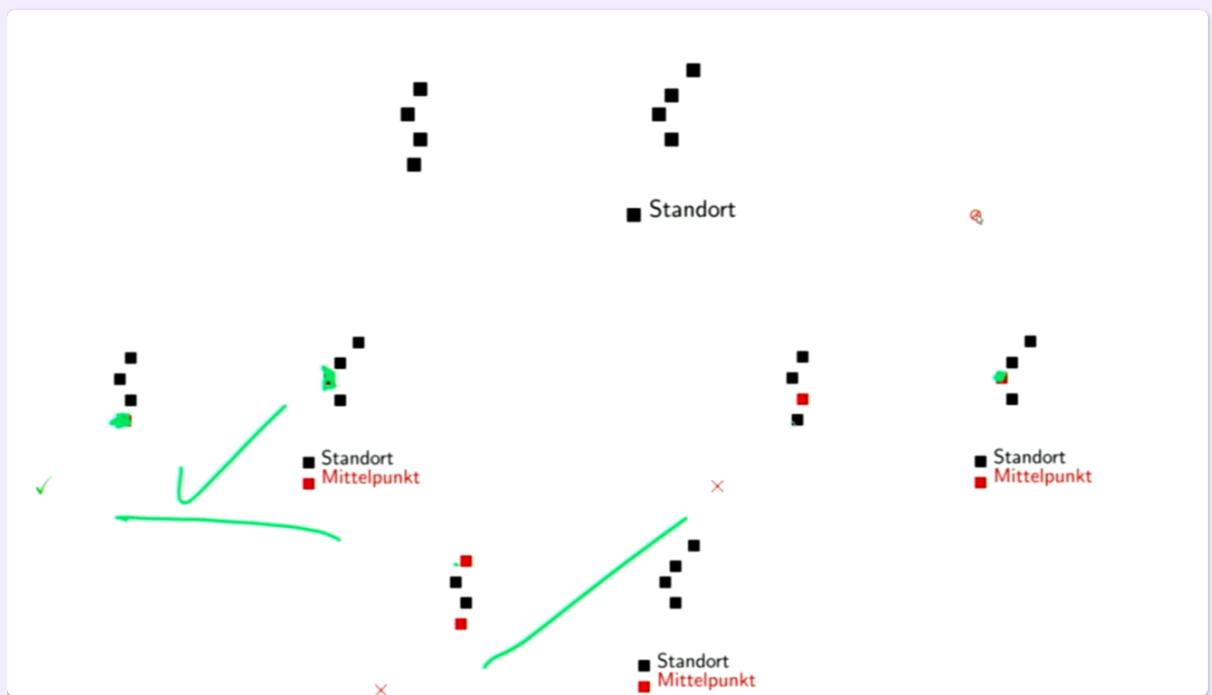
### Beobachtung:

- Nach der Terminierung sind alle Mittelpunkte in  $C$  paarweise zumindest  $r(C)$  voneinander entfernt.
  - (Dies ist eine wichtige Eigenschaft des Algorithmus: Die Mittelpunkte sind gut verteilt und nicht zu nah beieinander.)

### ✓ Beweis

- Durch Konstruktion des Algorithmus (indem immer der am weitesten entfernte Punkt gewählt wird, stellt man sicher, dass die neu hinzugefügten Mittelpunkte einen gewissen Mindestabstand zu den bestehenden haben).

### ☰ Beispiel



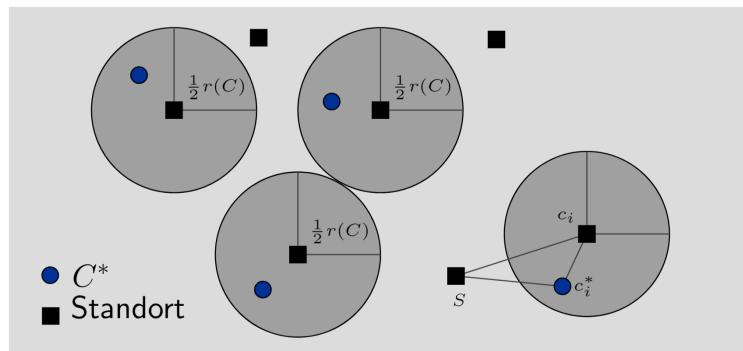
Nur das linke stimmt

# Greedy-Algorithmus: Analyse (für das Center Selection Problem)

**Theorem:** Sei  $C^*$  eine optimale Menge von Mittelpunkten. Dann ist  $r(C) \leq 2r(C^*)$ .

**Beweis:** (durch Widerspruch) Angenommen  $r(C^*) < \frac{1}{2}r(C)$ .

- Für jeden Standort  $c_i$  in  $C$  betrachte den Radius  $\frac{1}{2}r(C)$  um ihn herum.
- Nur ein  $c_i^* \in C^*$  in jedem Radius; sei  $c_i$  der Standort mit  $c_i^*$ .
- Betrachte einen beliebigen Standort  $s$  und seinen nächsten Mittelpunkt  $c_i^*$  in  $C^*$ .
- $\text{dist}(s, C) \leq \text{dist}(s, c_i) \leq \text{dist}(s, c_i^*) + \text{dist}(c_i^*, c_i) \leq 2r(C^*)$ .
- Daher  $r(C) \leq 2r(C^*)$ .  $\square$
- $\Delta$ -Ungleichung  $\leq r(C^*)$  da  $c_i^*$  der nächste Mittelpunkt ist.



## Theorem: Optimalität der Menge von Mittelpunkten

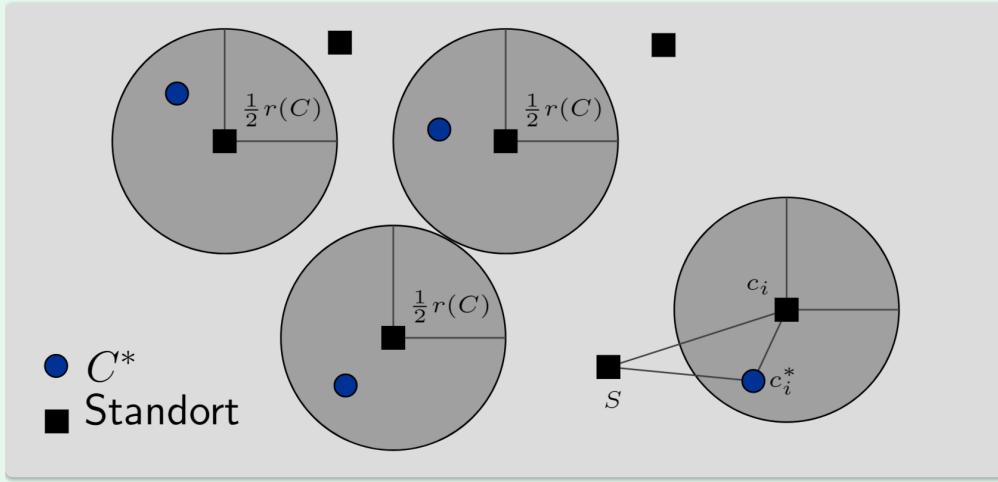
Sei  $C^*$  eine optimale Menge von Mittelpunkten. Dann gilt:  $r(C) \leq 2r(C^*)$ .

### ✓ Beweis (durch Widerspruch)

Angenommen, das Gegenteil ist der Fall, d.h.,  $r(C^*) < \frac{1}{2}r(C)$ .

- Für jeden Standort  $c_i$  in  $C$  (der Menge der Standorte) wird der Radius  $\frac{1}{2}r(C)$  um ihn herum betrachtet.
- Nur ein Mittelpunkt  $c_j^* \in C^*$  befindet sich in jedem dieser Radien; sei  $c_i^*$  der Mittelpunkt, der zu  $c_i$  gehört (also der nächste Mittelpunkt zu  $c_i$  ist).
- Betrachte einen beliebigen Standort  $s$  und seinen nächsten Mittelpunkt  $c_i^*$  in  $C^*$ .
- Es gilt die Dreiecksungleichung:  $\text{dist}(s, c_i) \leq \text{dist}(s, c_i^*) + \text{dist}(c_i^*, c_i) \leq 2r(C^*)$ .
  - Erläuterung zur Ungleichung:
    - $\text{dist}(s, c_i)$  ist der Abstand von  $s$  zum nächsten Standort  $c_i$  in  $C$ .
    - $\text{dist}(s, c_i^*)$  ist der Abstand von  $s$  zu seinem nächsten Mittelpunkt  $c_i^*$  in  $C^*$ .
    - $\text{dist}(c_i^*, c_i)$  ist der Abstand zwischen dem Mittelpunkt  $c_i^*$  und dem Standort  $c_i$ .
  - Die Dreiecksungleichung besagt, dass der direkte Weg zwischen zwei Punkten ( $\text{dist}(s, c_i)$ ) nie länger ist als der Umweg über einen dritten Punkt ( $\text{dist}(s, c_i^*) + \text{dist}(c_i^*, c_i)$ ).

- Der Ausdruck  $\leq 2r(C^*)$  kommt daher, dass sowohl  $dist(s, c_i^*)$  als auch  $dist(c_j^*, c_i)$  maximal  $r(C^*)$  betragen können, da  $r(C^*)$  der *maximale Radius* ist, der nötig ist, um alle Standorte von einem Mittelpunkt in  $C^*$  aus abzudecken.
- Daraus folgt, dass  $r(C) \leq 2r(C^*)$ . (Dies widerspricht der Annahme  $r(C^*) < \frac{1}{2}r(C)$ .)



# Theorem: Gütegarantie des Greedy-Algorithmus

---

## Theorem: Optimalität und Greedy-Algorithmus

- Sei  $C^*$  eine optimale Menge von Mittelpunkten. Dann gilt:  $r(C) \leq 2r(C^*)$ .
- Der Greedy-Algorithmus ist ein 2-Approximationsalgorithmus für das Center-Selection-Problem.

## Anmerkung: Greedy-Algorithmus und Gütegarantie

- Der Greedy-Algorithmus platziert Mittelpunkte immer auf Standorten.
- Trotzdem erreicht er eine Gütegarantie von 2 gegenüber einer optimalen Lösung, selbst wenn die optimalen Mittelpunkte überall platziert werden dürfen (z.B. Punkte in einer Ebene).

## Frage & Theorem: Bessere Approximationsalgorithmen

- Gibt es Hoffnung auf einen 3/2- oder 4/3-Approximationsalgorithmus? Eher nein!
- Ein  $\epsilon$ -Approximationsalgorithmus für das Center-Selection-Problem mit beliebigem  $\epsilon < 2$  existiert nur, wenn P = NP gilt.
  - **Bedeutung:** Eine bessere Approximationsgarantie als 2 ist so schwer wie die Frage, ob P=NP, was das Problem als NP-schwer einstuft.

