Optimierung – Approximation

Algorithmen und Datenstrukturen
VU 186.866, 5.5h, 8 ECTS, 2023S
Letzte Änderung: 15. Juni 2023

Vorlesungsfolien

Optimierung: Roadmap

Branch-and-Bound

Dynamische Programmierung

Approximation(salgorithmen): Erzeuge in polynomieller Zeit eine Näherungslösung, die eine Gütegarantie besitzt. Die Güte eines Algorithmus sagt etwas über die Fähigkeit aus, optimale Lösungen gut oder schlecht anzunähern.

Heuristische Verfahren

Gütegarantie

Annahmen:

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

Für Minimierungsprobleme gilt: Falls es ein $\varepsilon > 0$ gibt, sodass für alle Instanzen x von X

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

gilt, dann ist A ein ε-Approximationsalgorithmus und der Wert ε heißt Gütegarantie.

Gütegarantie

Für Maximierungsprobleme gilt:

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

Es folgt:

- für Minimierungsprobleme: $\varepsilon \geq 1$,
- für Maximierungsprobleme: $0 \leq \varepsilon \leq 1$,
- $\varepsilon = 1 \quad \Leftrightarrow \quad A$ ist ein exakter Algorithmus

Approximationsalgorithmus für Vertex Cover

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 kleinster Größe $|C|$.

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

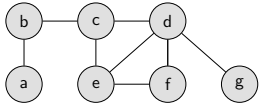
Minimales Vertex Cover: Approximationsalgorithmus

2-Approximationsalgorithmus: Findet ein Vertex Cover in einem Graphen $G = (V, E)$, das höchstens zwei Mal so groß wie ein optimales (minimales) Vertex Cover ist.

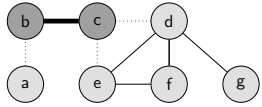
```
Approx-Vertex-Cover(G):
C ← ∅
while E ≠ ∅
    Wähle eine beliebige Kante (u, v) ∈ E
    C ← C ∪ {u, v}
    Entferne aus E alle Kanten, die inzident
          zu u oder v sind
return C
```

Approx-Vertex-Cover: Beispiel

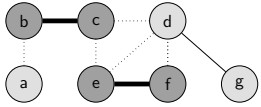
Beispielhafter Ablauf des Algorithmus, Vergleich mit optimaler Lösung.



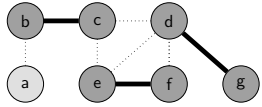
(a) Ausgangssituation



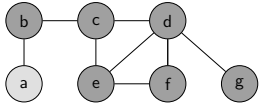
(b) 1. Kante auswählen



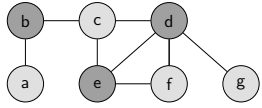
(c) 2. Kante auswählen



(d) 3. Kante auswählen



(e) Ergebnis



(f) Optimale Lösung

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 $m = |E|$.*

Minimales Vertex Cover: Gütegarantie

Gütegarantie:

- Sei M die Kantenmenge, die vom Algorithmus ausgewählt wird; M ist ein Matching.
- 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 $c_{\text{opt}} = |C^*| \geq |M|$.
- Da Approx-Vertex-Cover (kurz AVC) ein Vertex Cover der Größe $2|M|$ findet, gilt für alle Instanzen x:

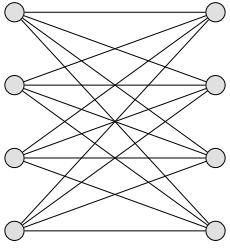
$$c_{\text{AVC}}(x) = 2|M| \leq 2 \cdot c_{\text{opt}}(x)$$

Minimales Vertex Cover: Gütegarantie

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

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.



Minimales Vertex Cover: Alternativer Algorithmus

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

```
Approx-Vertex-Cover2(G):
C ← ∅
while E ≠ ∅
    Wähle einen Knoten u mit maximalem Grad im aktuellen Graphen
    C ← C ∪ {u}
    Entferne aus E alle Kanten, die inzident zu u sind
return C
```

Minimales Vertex Cover: Alternativer Algorithmus

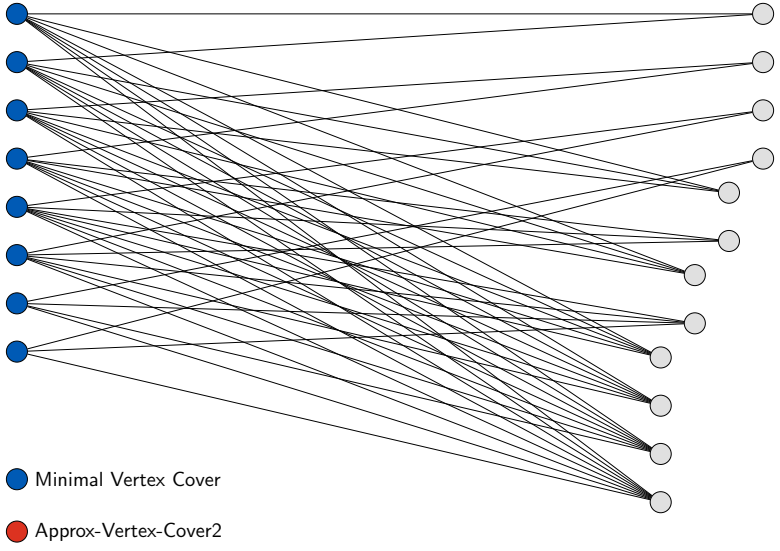
Alternativer Algorithmus: Wählt immer einen Knoten mit aktuell maximalem Grad.

```
Approx-Vertex-Cover2(G):
C ← ∅
while E ≠ ∅
    Wähle einen Knoten u mit maximalem Grad im aktuellen Graphen
    C ← C ∪ {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 intelligentere Auswahl führt hier nicht zu einer Verbesserung!

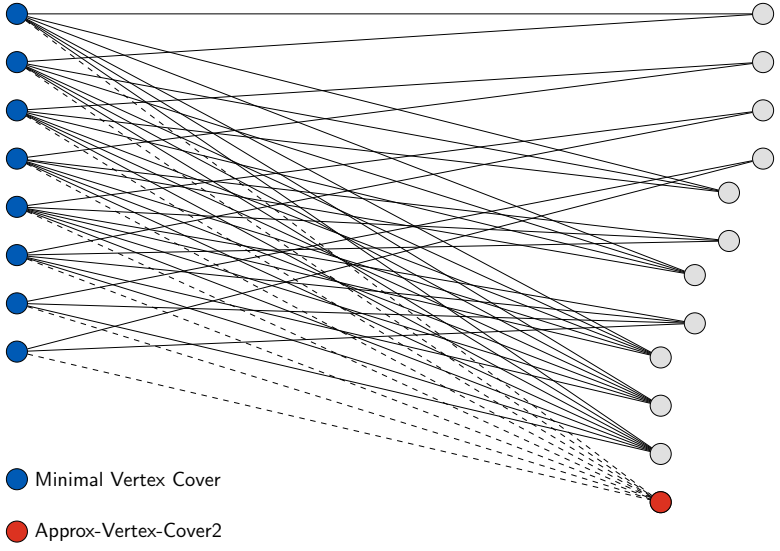
Minimales Vertex Cover: Alternativer Algorithmus

Schlechtes Beispiel:



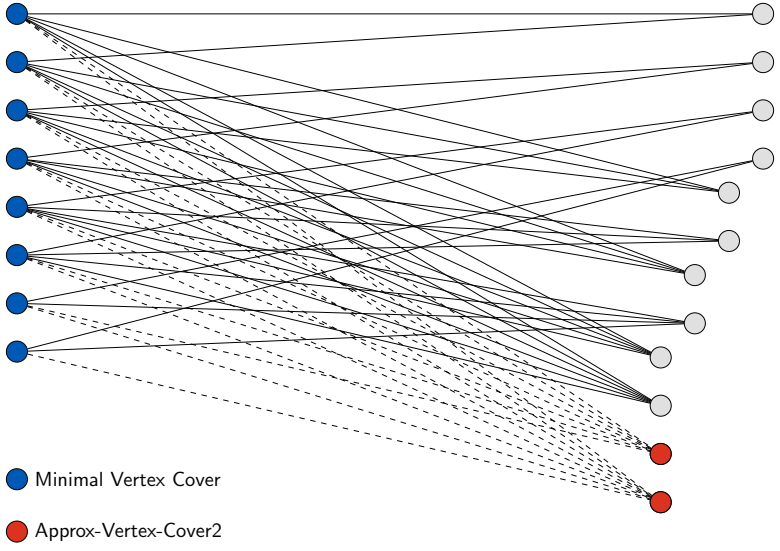
Minimales Vertex Cover: Alternativer Algorithmus

Schlechtes Beispiel:



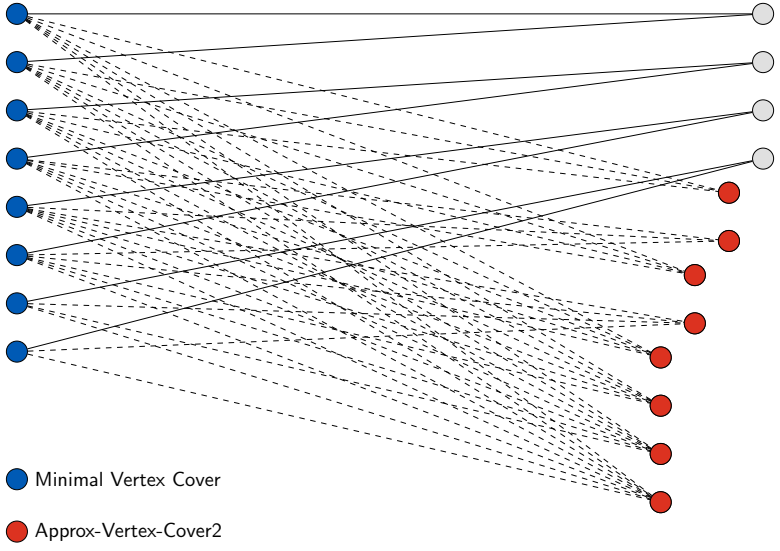
Minimales Vertex Cover: Alternativer Algorithmus

Schlechtes Beispiel:



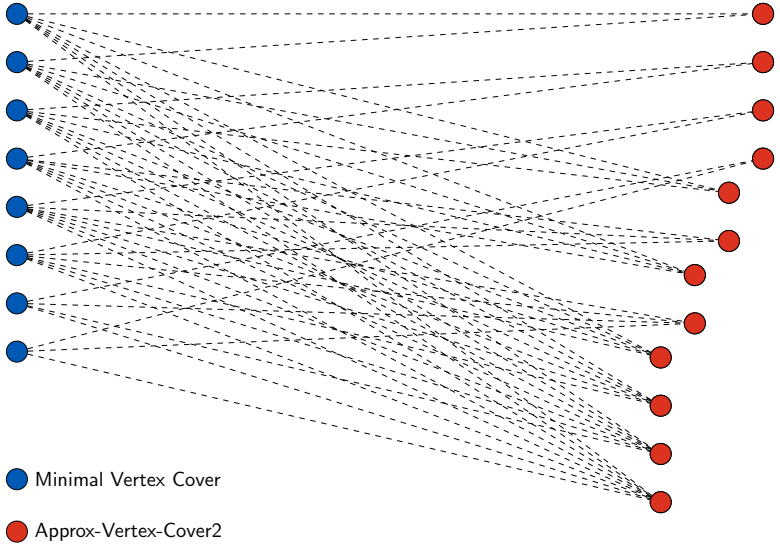
Minimales Vertex Cover: Alternativer Algorithmus

Schlechtes Beispiel:



Minimales Vertex Cover: Alternativer Algorithmus

Schlechtes Beispiel:



Minimales Vertex Cover: Alternativer Algorithmus

Konstruktionsschema für schlechtes Beispiel:

- Wähle n Knoten auf der linken Seite.
- Für $i = 2, \ldots n$ 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).

Minimales Vertex Cover: Alternativer Algorithmus

Gütegarantie: Für n Knoten auf der linken Seite.

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

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

  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

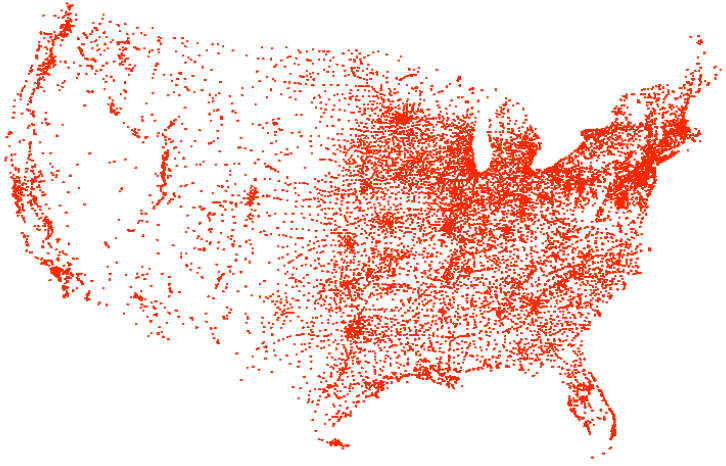
Traveling Salesman Problem (TSP): Wiederholung

Traveling Salesman Problem (TSP):

- 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.

Traveling Salesman Problem (TSP): Beispiel

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



Traveling Salesman Problem (TSP): Beispiel

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

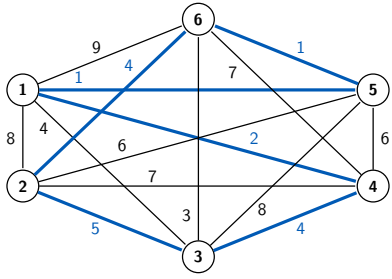
Traveling Salesman Problem (TSP)

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).

Spanning-Tree-Heuristik (ST) für das sym. TSP

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

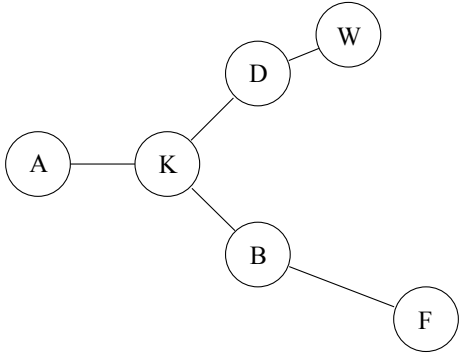
Vorgehen:

1. Bestimme einen MST (V, B_1) von G.
2. Verdopple alle Kanten in B_1, das ergibt Graph (V, B_2).
3. Bestimme eine Eulertour F im Graphen (V, B_2). Gib dieser Tour eine Orientierung, wähle einen Knoten $i \in V$, markiere i, setze $p \leftarrow i, T \leftarrow \emptyset$.
4. Sind alle Knoten markiert, setze $T \leftarrow T \cup \{(p, i)\}$ und retourniere T als Ergebnis-Tour.
5. Laufe von p entlang der Orientierung von F bis ein unmarkierter Knoten q erreicht ist. Setze $T \leftarrow T \cup \{(p, q)\}$, markiere q, setze $p \leftarrow q$ und gehe zu (4).

Spanning-Tree-Heuristik (ST): Beispiel

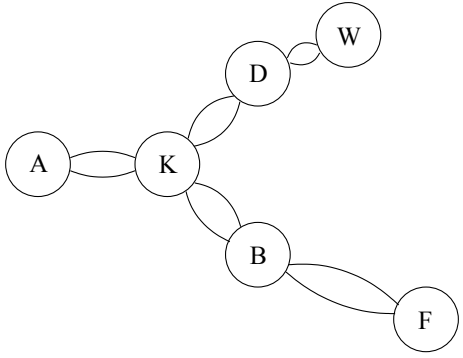
Ausgangspunkt: Vollständiger Graph mit 6 Knoten.

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



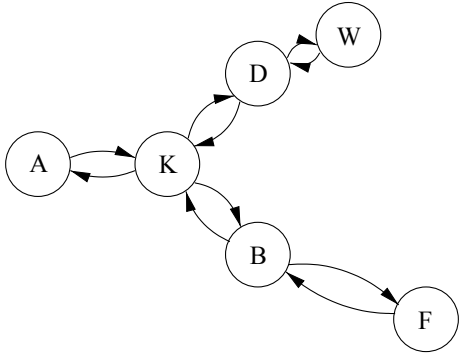
Spanning-Tree-Heuristik (ST): Beispiel

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



Spanning-Tree-Heuristik (ST): Beispiel

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

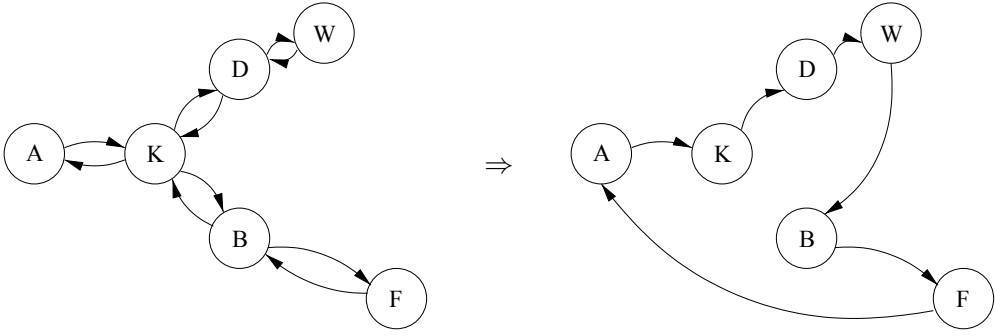


Eulertour: $F =$ (A, K, D, W, D, K, B, F, B, K, A)

Spanning-Tree-Heuristik (ST): Beispiel

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.



Spanning-Tree-Heuristik (ST) für das sym. TSP

Eulertour: Eine Rundtour, die jede Kante des Graphen genau einmal beinhaltet.

Theorem: Eine Eulertour existiert in einem ungerichteten Graphen genau dann wenn er zusammenhängend ist und jeder Knoten einen geraden Grad hat.

Hinweis: Durch die Verdopplung der Kanten in ST hat jeder Knoten geraden Grad.
Eine Eulertour 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)

Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren.

Von Carl Hierholzer.

Mitgetheilt von Chr. Wiener*).

In einem beliebig verschlungenen Linienzuge mögen *Zweige* eines Punktes diejenigen verschiedenen Theile des Zuges heissen, auf welchen man den fraglichen Punkt verlassen kann. Ein Punkt mit mehreren Zweigen heisse ein *Knotenpunkt*, der so vielfach genannt werde, als

Hinweis: Findet eine Eulertour (falls vorhanden) in einem Graphen G in linearer Zeit.

Eulertour: Algorithmus von Hierholzer

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 $v_1, v_2, ..., v_k, v_1$.

- Die Kanten $E(Z) = \{(v_1, v_2), (v_2, v_3), ..., (v_{k-1}, v_k), (v_k, v_1)\}$ des gefundenen Zyklus Z werden aus dem Graphen entfernt. Im Restgraphen sind, wenn eine Eulertour 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.

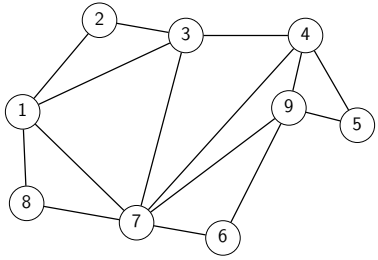
Eulertour: Algorithmus von Hierholzer

Vorgehen:

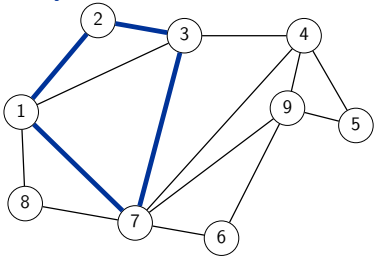
1. Wähle einen beliebigen Knoten v_0 des Graphen und konstruiere von v_0 ausgehend einen geschlossenen Pfad Z in G, der keine Kante in G zweimal durchläuft. Wir nennen Z einen Zyklus.
2. Wenn Z eine Eulertour ist (also alle Knoten beinhaltet), terminiere. Andernfalls:
3. Lösche nun alle Kanten in Z aus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

Eulertour: Algorithmus von Hierholzer: Beispiel

Ausgangsgraph:



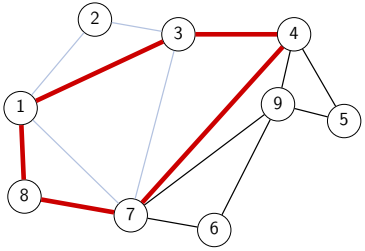
1. Zyklus:



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

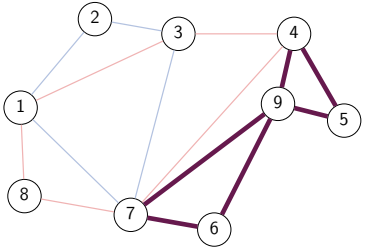
Eulertour: Algorithmus von Hierholzer: Beispiel

2. Zyklus:



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

3. Zyklus:

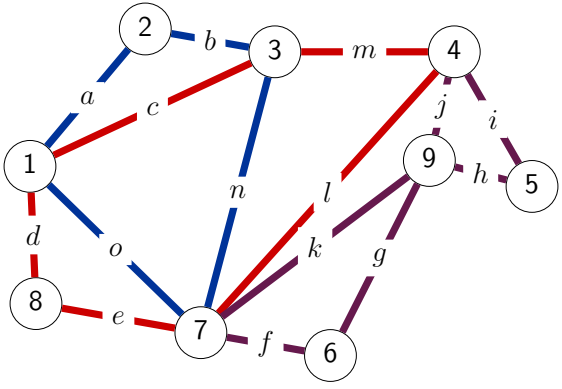


$Z' = (7,6,9,5,4,9,7)$
$Z = (1,3,4,7,6,9,5,4,9,7,8,1,2,3,7,1)$

Eulertour: Algorithmus von Hierholzer: Beispiel

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 $O(n+m)$.

Ist das symmetrische TSP 2-approximierbar?

Theorem: Angenommen P$\neq$NP. Dann gibt es keinen polynomiellen 2-Approximationsalgorithmus für das symmetrische TSP.

Beweis: (durch Widerspruch)

- Angenommen, es existiert ein polynomieller Approximationsalgorithmus A mit einer Gütegarantie 2.
- Sei $G = (V, E)$ ein Graph, für den wir bestimmen wollen, ob der Graph einen Hamiltonkreis enthält ($\mathrm{HAM\text{-}CYCLE}$-Problem ist NP-vollständig).
- Wir möchten jetzt A benutzen, um den Hamiltonkreis effizient zu finden.
- Dazu wird G in eine TSP-Instanz $G' = (V, E', c)$ transformiert.

Ist das symmetrische TSP 2-approximierbar?

Beweis (Fortsetzung):

- Transformation von G in eine TSP-Instanz $G' = (V, E', c)$:
  Sei (V, E') der vollständige Graph:
  - $E' = \{(u, v) : u, v \in V, \quad u \neq v\}$.
  - mit Kantenkosten:

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

- G kann in polynomieller Zeit in G' transformiert werden.
- Wenn G einen Hamiltonkreis H enthält, dann werden jeder Kante von H die Kosten 1 zugewiesen und G' enthält eine Tour mit den Kosten n.
- Wenn G keinen Hamiltonkreis enthält, dann benutzt jede Tour in G' mindestens eine Kante, die sich nicht in E befindet. Solch eine Tour hat aber die Kosten von mindestens

$$(2 \cdot n + 1) + (n - 1) = 2 \cdot n + n = 3 \cdot n > 2 \cdot n$$

.

Ist das symmetrische TSP 2-approximierbar?

Beweis (Fortsetzung):

- 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 ein Hamiltonkreis in G ist.
- 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.
- Wenn G einen Hamiltonkreis enthält, dann muss A ihn zurückliefern.
- Wenn G keinen Hamiltonkreis enthält, dann liefert A eine Tour mit Kosten größer als $2 \cdot n$.
- Daher kann A benutzt werden, einen Hamiltonkreis in polynomieller Zeit zu finden. Das ist aber ein Widerspruch dazu, dass $\mathrm{HAM\text{-}CYCLE}$ NP-schwer ist. Das könnte nur der Fall sein, wenn P=NP gilt. $\square$

Spanning-Tree-Heuristik (ST): Gütegarantie

Theorem: Ähnlich kann für ein allgemeines $\varepsilon > 1$ gezeigt werden, dass es es keinen polynomiellen ε-Approximationsalgorithmus für das symmetrische TSP gibt.
Das symmetrische TSP ist „nicht approximierbar“.

Beweis: Wie auf den vorherigen Folien, nur gilt jetzt $c(u, v) = \varepsilon \cdot n + 1$ wenn $(u, v) \notin E$. $\square$

Frage: Wozu dann der ganze Aufwand, wenn eine beliebige Approximation nicht effizient möglich ist?

Antwort: Unter einer einfachen Voraussetzung wird das Ergebnis besser.

Spanning-Tree-Heuristik (ST): Metrisches TSP

Metrisches TSP: Ein TSP heißt metrisch, wenn für die Distanzmatrix C die Dreiecksungleichung gilt, d.h. für alle Knoten i, j, k gilt

$$c_{ik} \leq c_{ij} + c_{jk}.$$

Hinweis: Insbesondere ist auch das Euklidische TSP, bei dem den Knoten Punkte in der euklidischen Ebene entsprechen und die Distanzen die euklidischen Distanzen sind, metrisch.

Theorem: Das metrische TSP besitzt einen polynomiellen Approximationsalgorithmus mit einer Gütegarantie von 2, d.h.

$$\frac{c_{\mathrm{ST}}(x)}{c_{\mathrm{opt}}(x)} \leq 2 \quad \text{für alle TSP-Instanzen } x$$

Spanning-Tree-Heuristik (ST): Metrisches TSP

Laufzeit: Die Spanning-Tree-Heuristik läuft in polynomieller Zeit (aufwendigster Schritt ist Ermitteln des MST im ersten Schritt).

Beweis für Gütegarantie: Es gilt

$$c_{\mathrm{ST}}(x) \leq c_{\mathrm{B}_2}(x) = 2c_{\mathrm{B}}(x) \leq 2c_{\mathrm{opt}}(x)$$

- *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.*

Lastverteilung (Load Balancing)

Lastverteilung

Eingabe: m identische Maschinen; n Jobs (Aufgaben), Job j hat Bearbeitungszeit t_j.

- Jeder Job j muss ununterbrochen auf einer Maschine ausgeführt werden.
- Eine Maschine kann nur einen Job auf einmal ausführen.

Definition: Sei J_i die Teilmenge von Jobs, die Maschine i zugewiesen wurde.
Die Last der Maschine i ist $L_i = \sum_{j \in J_i} t_j$.

Definition: Die Bearbeitungsdauer (*makespan*) ist die maximale Last auf irgendeiner Maschine $L = \max_{i=1,\ldots,n} L_i$.

Lastverteilung: Teile jeden Job einer Maschine so zu, dass die Bearbeitungsdauer minimiert wird.

Lastverteilung: List-Scheduling

List-Scheduling-Algorithmus:

- Berücksichtige n Jobs mit einer fixen Ordnung.
- Greedy-Algorithmus: Teile Job j einer Maschine mit der aktuell kleinsten Last zu.
- $L_i =$ Last auf Maschine i.
- $J_i =$ Jobs, die Maschine i zugewiesen wurden.

```
List-Scheduling(m,n,t_1,t_2,...,t_n):
for i ← 1 bis m
    J_i ← ∅
    L_i ← 0
for j ← 1 bis n
    i = argmin_{k=1,...,m} L_k
    J_i ← J_i ∪ {j}
    L_i ← L_i + t_j
return J_1,...,J_m
```

$\square$ *Maschine i hat geringste Last*

Laufzeit?

Lastverteilung: List-Scheduling

List-Scheduling-Algorithmus:

- Berücksichtige n Jobs mit einer fixen Ordnung.
- Greedy-Algorithmus: Teile Job j einer Maschine mit der aktuell kleinsten Last zu.
- $L_i =$ Last auf Maschine i.
- $J_i =$ Jobs, die Maschine i zugewiesen wurden.

```
List-Scheduling(m,n,t_1,t_2,...,t_n):
for i ← 1 bis m
    J_i ← ∅
    L_i ← 0
for j ← 1 bis n
    i = argmin_{k=1,...,m} L_k
    J_i ← J_i ∪ {j}
    L_i ← L_i + t_j
return J_1, ..., J_m
```

□ *Maschine i hat geringste Last*

Laufzeit: $O(n \log m)$ mit einer Priority-Queue.

Lastverteilung: Analyse von List-Scheduling

Theorem: [Graham, 1966] List-Scheduling ist ein 2-Approximationsalgorithmus.

- Erste Worst-Case-Analyse eines Approximationsalgorithmus.
- Dazu muss man die resultierende Lösung mit der optimalen Bearbeitungsdauer L^* vergleichen.

Lemma 1: Für die optimale Dauer gilt in jedem Fall $L^* \geq \max_j t_j$.

Beweis: Eine Maschine muss den aufwendigsten Job verarbeiten. $\square$

Lemma 2: Für die optimale Dauer gilt in jedem Fall $L^* \geq \frac{1}{m} \sum_j t_j$.

Beweis:

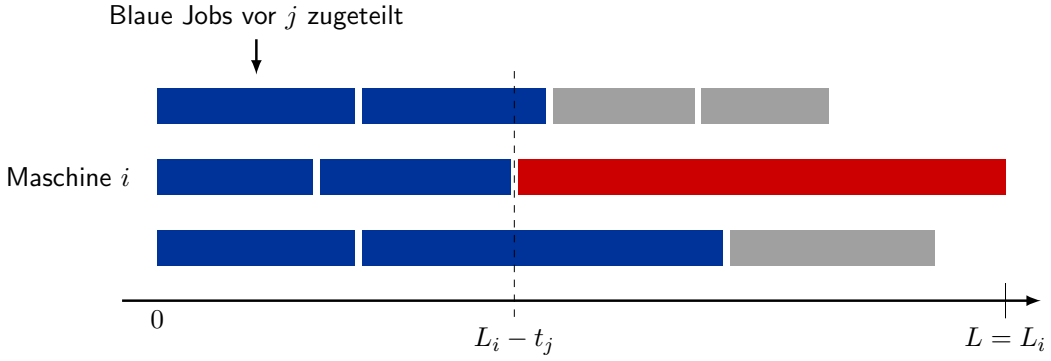
- Die gesamte Verarbeitungszeit ist $\sum_j t_j$.
- Eine der m Maschinen muss zumindest den $1/m$-ten Teil der Arbeit machen. $\square$

Lastverteilung: Analyse von List-Scheduling

Theorem: List-Scheduling ist ein 2-Approximationsalgorithmus.

Beweis: Wir betrachten die Last L_i einer Maschine i, die einen Flaschenhals darstellt.

- Sei j der letzte zugeteilte Job auf Maschine i.
- Wenn Job j Maschine i zugewiesen wird, hat i die geringste Last.
  Die Last vor der Zuteilung ist $L_i - t_j \Rightarrow L_i - t_j \leq L_k$ für alle $1 \leq k \leq m$.



Lastverteilung: Analyse von List-Scheduling

Theorem: List-Scheduling ist ein 2-Approximationsalgorithmus.

Beweis: Wir betrachten die Last L_i einer Maschine i, die einen Flaschenhals darstellt.

- Sei j der letzte zugeteilte Job auf Maschine i.
- Wenn Job j Maschine i zugewiesen wird, hat i die geringste Last.
  Die Last vor der Zuteilung ist $L_i - t_j \Rightarrow L_i - t_j \leq L_k$ für alle $1 \leq k \leq m$.
- Wir summieren alle Ungleichungen über alle 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 $L_i = \underbrace{(L_i - t_j)}_{\leq L^*} + \underbrace{t_j}_{\leq L^*} \leq 2L^*$. □

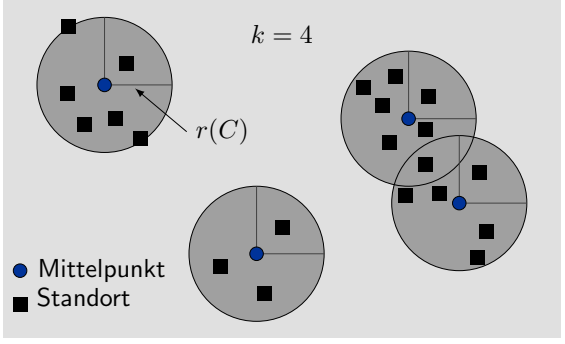
Lemma 2 Lemma 1

Center Selection

Center Selection

Eingabe: Menge von n Standorten $s_1, \ldots, s_n$ und ganze Zahl $k > 0$.

Center-Selection-Problem: Wähle k Mittelpunkte C, so dass die maximale Distanz von einem Standort zu einem nächsten Mittelpunkt minimiert wird.



Center Selection

Eingabe: Menge von n Standorten $s_1, \ldots, s_n$ und ganze Zahl $k > 0$.

Center-Selection-Problem: Wähle k Mittelpunkte C, so dass die maximale Distanz von einem Standort zu einem nächsten Mittelpunkt minimiert wird.

Notation:

- $\text{dist}(x, y) =$ Distanz zwischen x und y.
- $\text{dist}(s_i, C) = \min_{c \in C} \text{dist}(s_i, c) =$ Distanz von s_i zu einem nächsten Mittelpunkt.
- $r(C) = \max_i \text{dist}(s_i, C) =$ kleinster überdeckender Radius.

Ziel: Finde eine Menge von Mittelpunkten C, die $r(C)$ minimiert, unter Berücksichtigung von $|C| = k$.

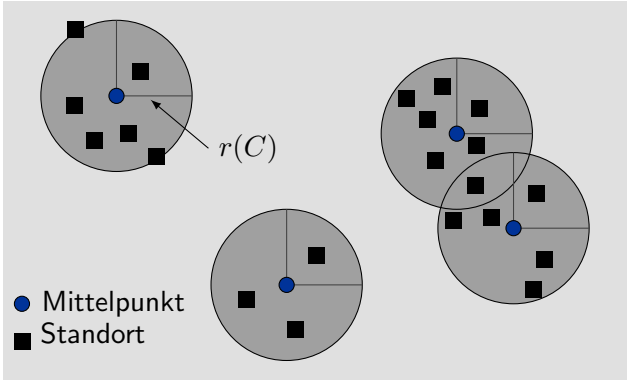
Eigenschaften der Distanzfunktion:

- $\text{dist}(x, x) = 0$ (Identität)
- $\text{dist}(x, y) = \text{dist}(y, x)$ (Symmetrie)
- $\text{dist}(x, y) \leq \text{dist}(x, z) + \text{dist}(z, y)$ (Dreiecksungleichung)

Center Selection: Beispiel

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

Anmerkung: Es gibt unendlich viele potentielle Lösungen!



Greedy-Algorithmus: Falscher Ansatz

Greedy-Algorithmus: Wähle für den ersten Mittelpunkt einen besten Platz für einen Mittelpunkt. Danach füge iterativ Mittelpunkte so hinzu, dass der Abdeckradius immer am meisten reduziert wird.

Anmerkung: Kann beliebig schlecht sein!

Center Selection: Greedy-Algorithmus

Greedy-Algorithmus: Wähle den ersten Mittelpunkt beliebig aus der Menge aller Standorte. Wähle wiederholt als nächsten Mittelpunkt einen Standort, der am weitesten von jedem existierenden Mittelpunkt entfernt ist.

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

- *Standort am weitesten von jedem Mittelpunkt.*

Beobachtung: Nach der Terminierung sind alle Mittelpunkte in C paarweise zumindest $r(C)$ voneinander entfernt.

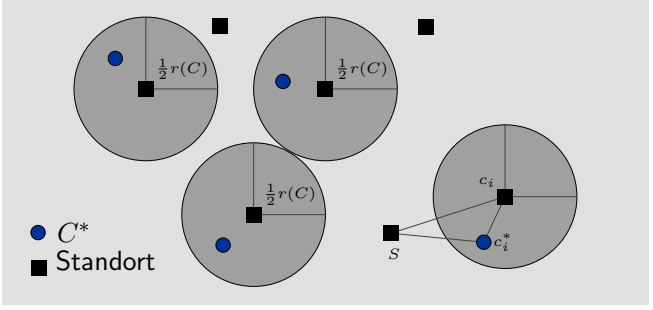
Beweis: Durch Konstruktion des Algorithmus.

Greedy-Algorithmus: Analyse

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 $r(C) \leq 2r(C^*)$. $\square$

Δ-*Ungleichung* $\leq r(C^*)$ *da* c_i^* *der nächste Mittelpunkt ist.*



Center Selection

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

Theorem: Greedy-Algorithmus ist ein 2-Approximationsalgorithmus für das Center-Selection-Problem.

Anmerkung: Greedy-Algorithmus platziert Mittelpunkte immer auf Standorten aber erreicht trotzdem eine Gütegarantie von 2 gegenüber einer optimalen Lösung, bei der Mittelpunkte überall platziert werden dürfen.

- *z.B., Punkte in einer Ebene*

Frage: Gibt es Hoffnung auf einen 3/2-Approximationsalgorithmus? 4/3? Eher nein!

Theorem: Es existiert ein ε-Approximationsalgorithmus für das Center-Selection-Problem für ein beliebiges $\varepsilon < 2$ nur dann wenn P = NP gilt.