



Datenstrukturen, Algorithmen und Programmierung 2 (DAP2)

Approximationsalgorithmen

Was kann man tun, wenn man ein Problem nicht effizient lösen kann?

- Die Aufgabenstellung vereinfachen!

Approximationsalgorithmen

- Löse Problem nicht exakt, sondern nur approximativ
- Qualitätsgarantie in Abhängigkeit von optimaler Lösung
- Z.B.: jede berechnete Lösung ist nur doppelt so teuer, wie eine optimale Lösung

Heuristik

- Löse ein Problem nicht exakt
- Keine Qualitätsgarantie
- Können jedoch in der Praxis durchaus effizient sein

Approximationsalgorithmen

Approximationsalgorithmen

- Löse Problem nicht exakt, sondern nur approximativ
- Qualitätsgarantie in Abhängigkeit von optimaler Lösung
- Z.B.: jede berechnete Lösung ist nur doppelt so teuer, wie eine optimale Lösung

Beispiel (kürzeste Wege)

- Wir sind zufrieden mit Wegen, die maximal doppelt so lang sind, wie ein kürzester Weg
- Gilt dies für alle berechneten Wege, so haben wir einen 2-Approximationsalgorithmus

Approximationsalgorithmen

Definition (Approximationsalgorithmus)

- Ein Algorithmus A für ein Optimierungsproblem heißt $\alpha(n)$ -Approximationsalgorithmus, wenn für jedes n und jede Eingabe der Größe n gilt, dass

$$\max \left\{ \frac{C}{C^*}, \frac{C^*}{C} \right\} \leq \alpha(n)$$

wobei C die Kosten der von A berechneten Lösung für die gegebene Instanz bezeichnet und C^* die Kosten einer optimalen Lösung

- $\alpha(n)$ heißt auch Approximationsfaktor

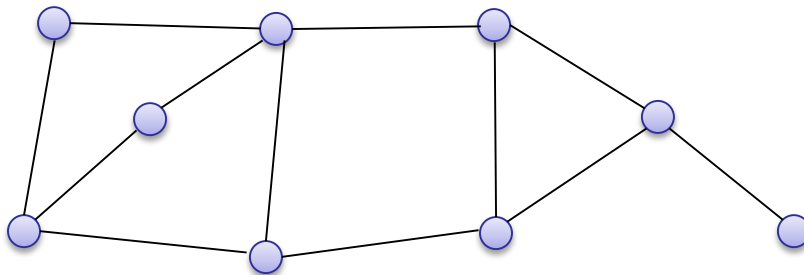
Approximationsalgorithmen

Knotenüberdeckung

- Sei $G = (V, E)$ ein ungerichteter Graph. Eine Menge $U \subseteq V$ heißt Knotenüberdeckung, wenn gilt, dass für jede Kante $(u, v) \in E$ mindestens einer der Endknoten u, v in U enthalten ist.

Problem minimale Knotenüberdeckung

- Gegeben ein Graph $G = (V, E)$
- Berechnen Sie eine Knotenüberdeckung U minimaler Größe $|U|$



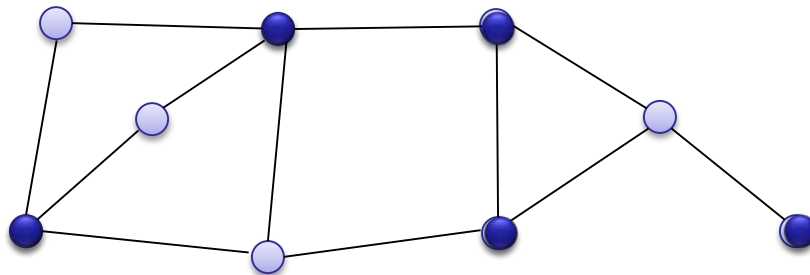
Approximationsalgorithmen

Knotenüberdeckung

- Sei $G = (V, E)$ ein ungerichteter Graph. Eine Menge $U \subseteq V$ heißt Knotenüberdeckung, wenn gilt, dass für jede Kante $(u, v) \in E$ mindestens einer der Endknoten u, v in U enthalten ist.

Problem minimale Knotenüberdeckung

- Gegeben ein Graph $G = (V, E)$
- Berechnen Sie eine Knotenüberdeckung U minimaler Größe $|U|$



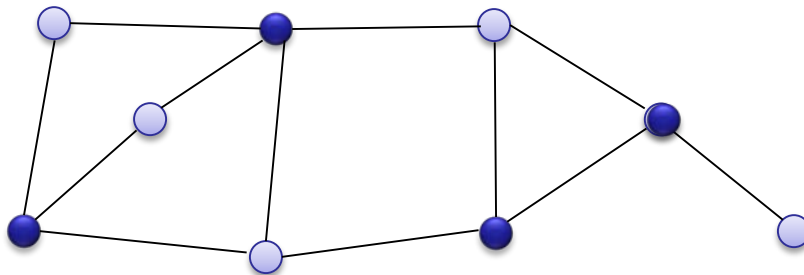
Approximationsalgorithmen

Knotenüberdeckung

- Sei $G = (V, E)$ ein ungerichteter Graph. Eine Menge $U \subseteq V$ heißt Knotenüberdeckung, wenn gilt, dass für jede Kante $(u, v) \in E$ mindestens einer der Endknoten u, v in U enthalten ist.

Problem minimale Knotenüberdeckung

- Gegeben ein Graph $G = (V, E)$
- Berechnen Sie eine Knotenüberdeckung U minimaler Größe $|U|$



Approximationsalgorithmen

Erste Idee

- Wähle immer Knoten mit maximalem Grad und entferne alle anliegenden Kanten

GreedyVertexCover1()

1. **while** $E \neq \emptyset$ **do**
2. wähle einen Knoten v mit maximalem Knotengrad
3. Entferne alle an v anliegenden Kanten aus E

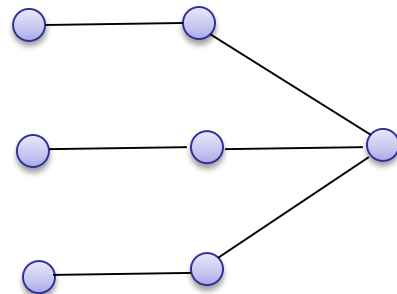
Erste Frage

- Ist der Algorithmus optimal?

Approximationsalgorithmen

Erste Frage

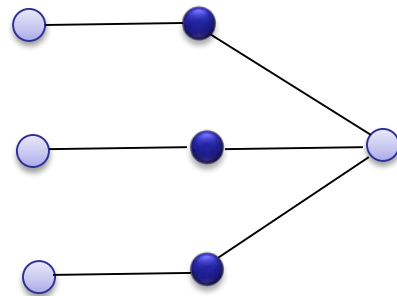
- Ist der Algorithmus optimal?
- Nein! Gegenbeispiel:



Approximationsalgorithmen

Erste Frage

- Ist der Algorithmus optimal?
- Nein! Gegenbeispiel:

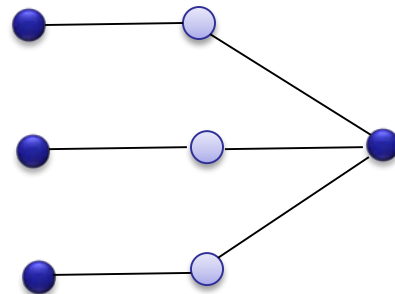


- Optimale Lösung hat Größe 3

Approximationsalgorithmen

Erste Frage

- Ist der Algorithmus optimal?
- Nein! Gegenbeispiel:



- Die von GreedyVertexCover1 berechnete Lösung hat Größe 4

Approximationsalgorithmen

Zweite Frage

- Hat der Algorithmus einen konstanten Approximationsfaktor?

Approximationsalgorithmen

Zweite Frage

- Hat der Algorithmus einen konstanten Approximationsfaktor?
- Nein!
- Wir entwickeln nun Konstruktion eines Gegenbeispiels

Approximationsalgorithmen

Zweite Frage

- Hat der Algorithmus einen konstanten Approximationsfaktor?
- Nein!
- Wir entwickeln nun Konstruktion eines Gegenbeispiels

Definition

- Ein Graph $G = (V, E)$ heißt **bipartit** (oder **2-färbbar**), wenn man V in zwei Mengen L und R partitionieren kann, so dass es keine Kante gibt, deren Endknoten beide in L oder beide in R liegen.
- Man schreibt auch häufig $G = (L \cup R, E)$, um die Partition direkt zu benennen.

Approximationsalgorithmen

Beobachtung

- Sei $G = (L \cup R, E)$ ein bipartiter Graph. Dann ist L bzw. R eine gültige Knotenüberdeckung (die aber natürlich nicht unbedingt minimale Größe hat)

Approximationsalgorithmen

Beobachtung

- Sei $G = (L \cup R, E)$ ein bipartiter Graph. Dann ist L bzw. R eine gültige Knotenüberdeckung (die aber natürlich nicht unbedingt minimale Größe hat)

Idee

- Wir konstruieren einen bipartiten Graph, bei dem $|L| = r$ ist und $|R| = \Omega(r \log r)$. Trotzdem wählt der Algorithmus GreedyVertexCover1 die Knoten der Seite R aus
- Damit ist für $r \rightarrow \infty$ der Approximationsfaktor nicht durch eine Konstante beschränkt

Approximationsalgorithmen

Die Konstruktion

- Sei $L = \{1, \dots, r\}$ eine Menge mit r Knoten

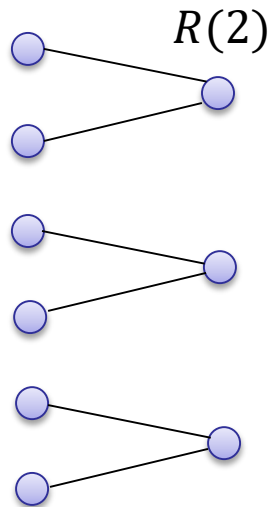


Die Menge L

Approximationsalgorithmen

Die Konstruktion

- Sei $L = \{1, \dots, r\}$ eine Menge mit r Knoten
- Wir wählen nun eine Menge $R(2)$ mit $\lfloor |L|/2 \rfloor$ Knoten
- Der j -te Knoten aus $R(2)$ wird mit Knoten $2j - 1$ und $2j$ verbunden

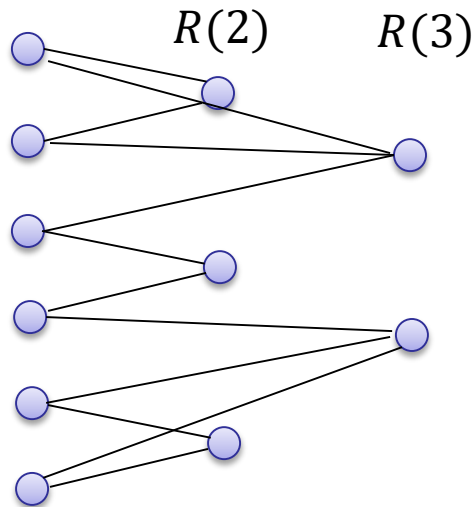


Die Menge L

Approximationsalgorithmen

Die Konstruktion

- Sei $L = \{1, \dots, r\}$ eine Menge mit r Knoten
- Im i -ten Schritt wählen wir Menge $R(i)$ mit $\lfloor |L|/i \rfloor$ Knoten
- Der j -te Knoten aus $R(i)$ wird mit Knoten $i(j-1) + 1, \dots, ij$ verbunden

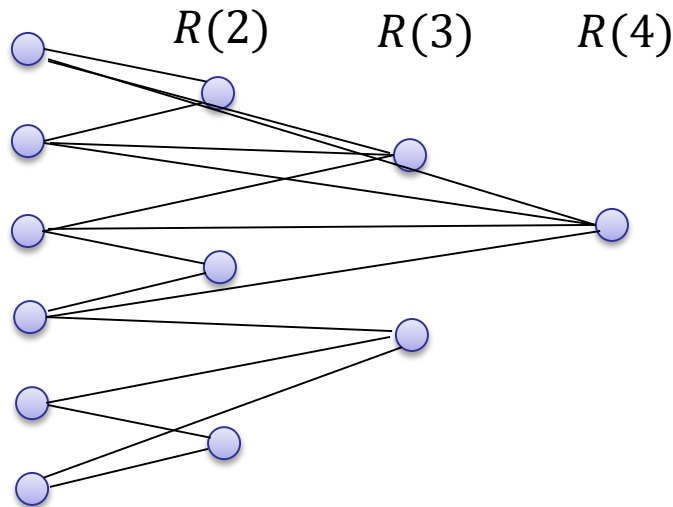


Die Menge L

Approximationsalgorithmen

Die Konstruktion

- Sei $L = \{1, \dots, r\}$ eine Menge mit r Knoten
- Im i -ten Schritt wählen wir Menge $R(i)$ mit $\lfloor |L|/i \rfloor$ Knoten
- Der j -te Knoten aus $R(i)$ wird mit Knoten $i(j-1) + 1, \dots, ij$ verbunden

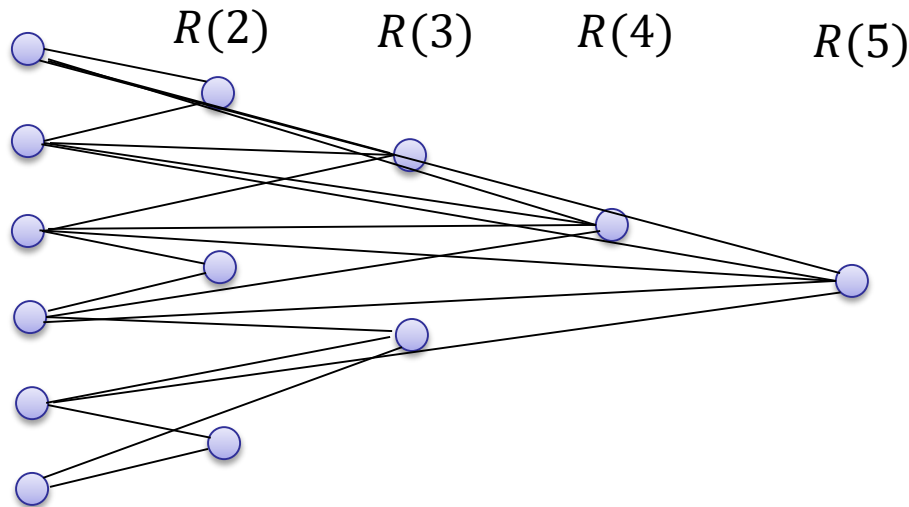


Die Menge L

Approximationsalgorithmen

Die Konstruktion

- Sei $L = \{1, \dots, r\}$ eine Menge mit r Knoten
- Im i -ten Schritt wählen wir Menge $R(i)$ mit $\lfloor |L|/i \rfloor$ Knoten
- Der j -te Knoten aus $R(i)$ wird mit Knoten $i(j-1) + 1, \dots, ij$ verbunden

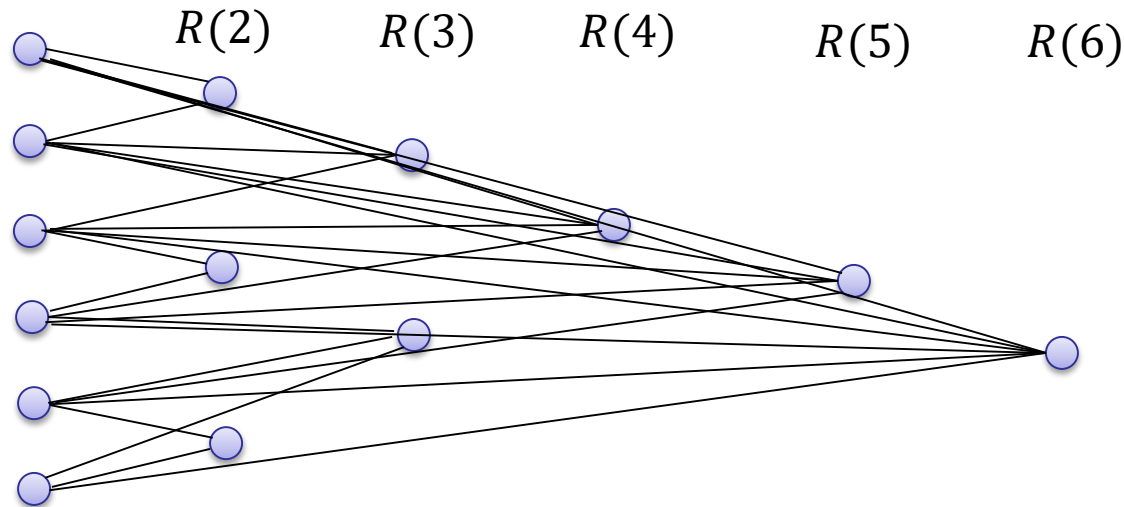


Die Menge L

Approximationsalgorithmen

Die Konstruktion

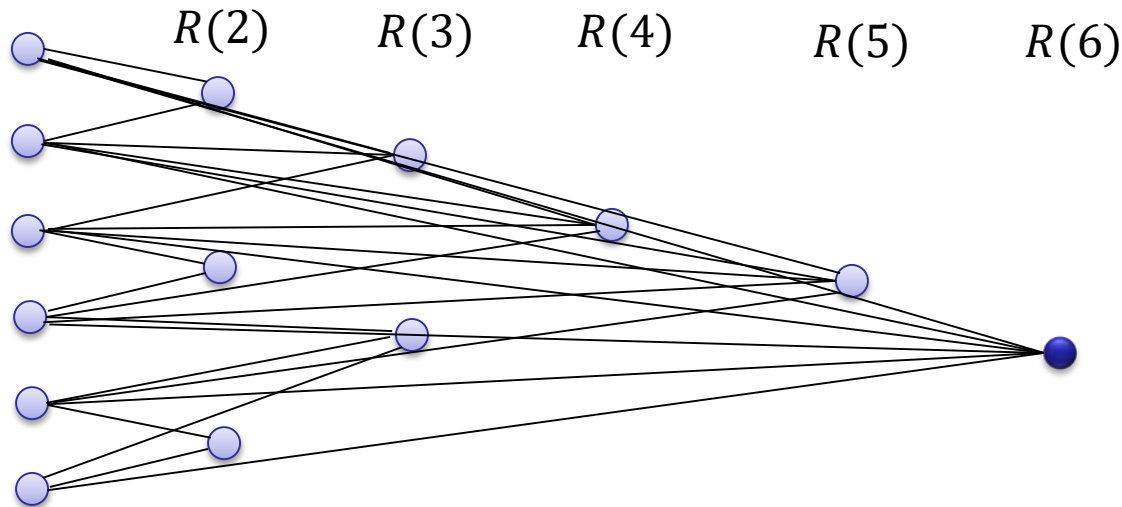
- Sei $L = \{1, \dots, r\}$ eine Menge mit r Knoten
- Im i -ten Schritt wählen wir Menge $R(i)$ mit $\lfloor |L|/i \rfloor$ Knoten
- Der j -te Knoten aus $R(i)$ wird mit Knoten $i(j-1) + 1, \dots, ij$ verbunden



Die Menge L

Approximationsalgorithmen

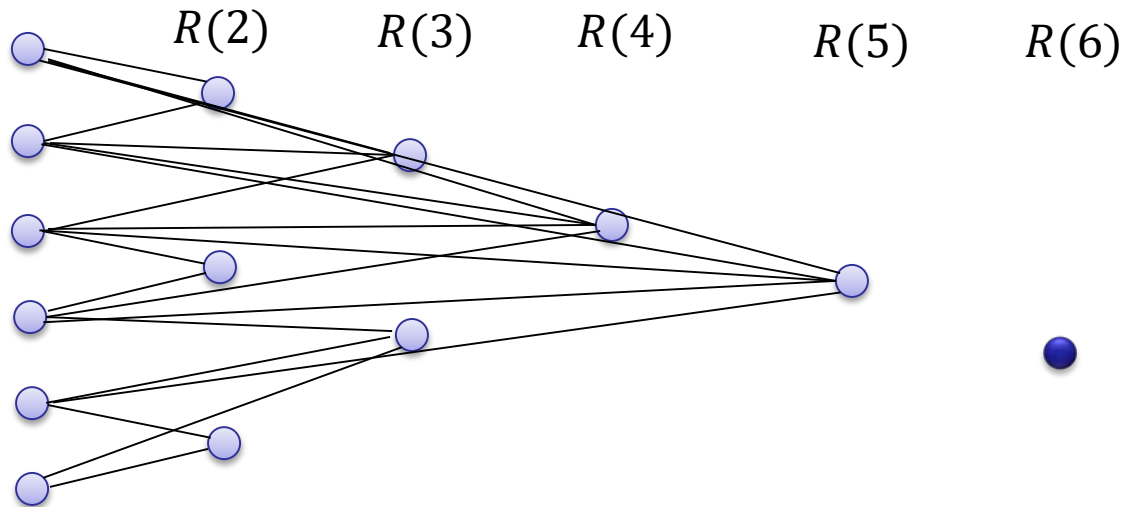
Was macht der Algorithmus?



Die Menge L

Approximationsalgorithmen

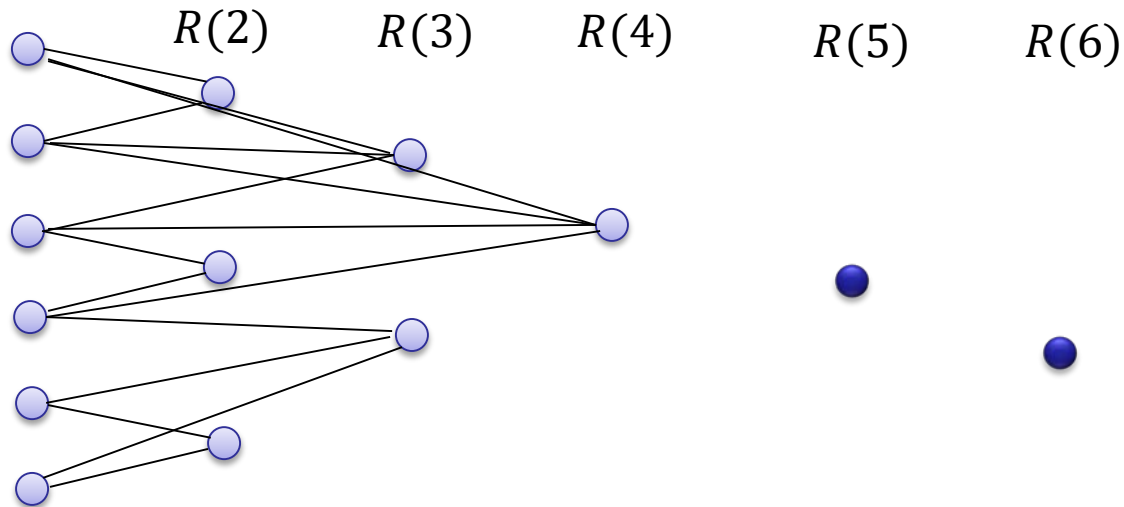
Was macht der Algorithmus?



Die Menge L

Approximationsalgorithmen

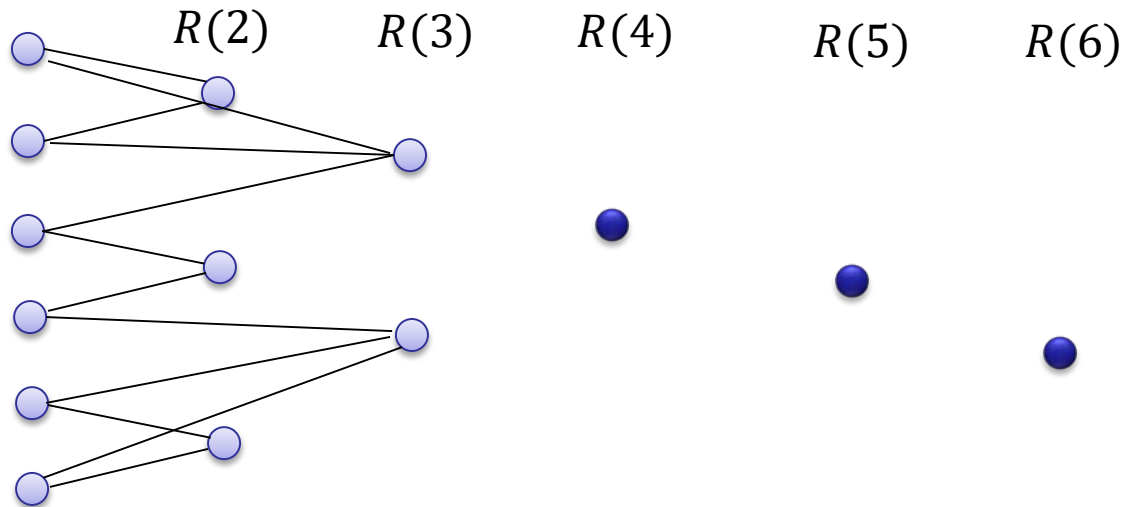
Was macht der Algorithmus?



Die Menge L

Approximationsalgorithmen

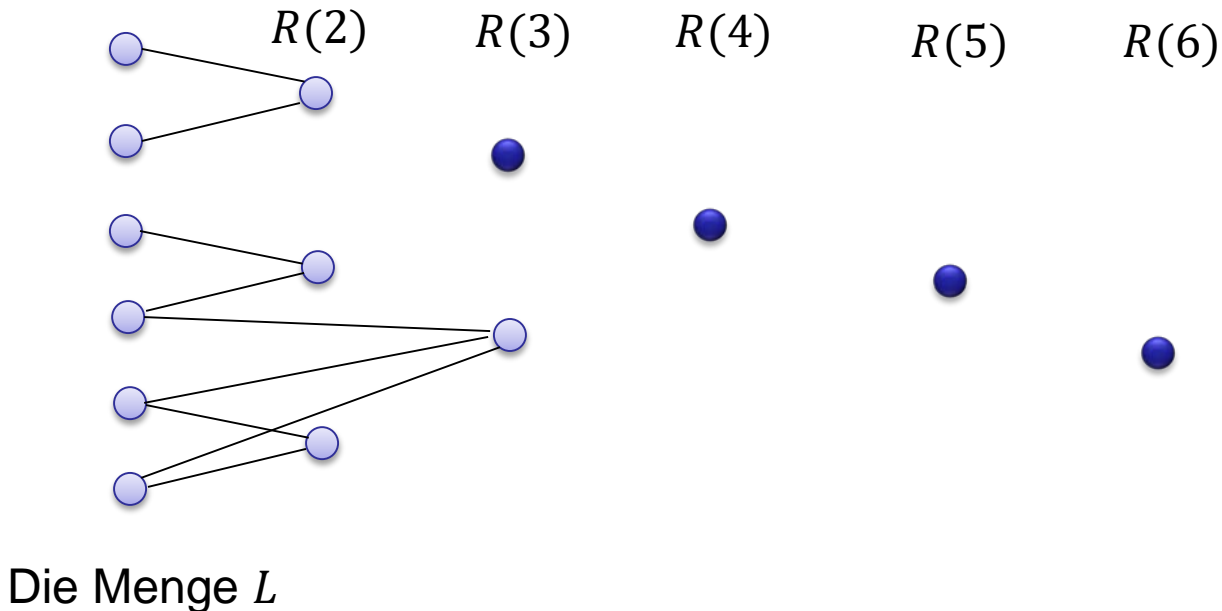
Was macht der Algorithmus?



Die Menge L

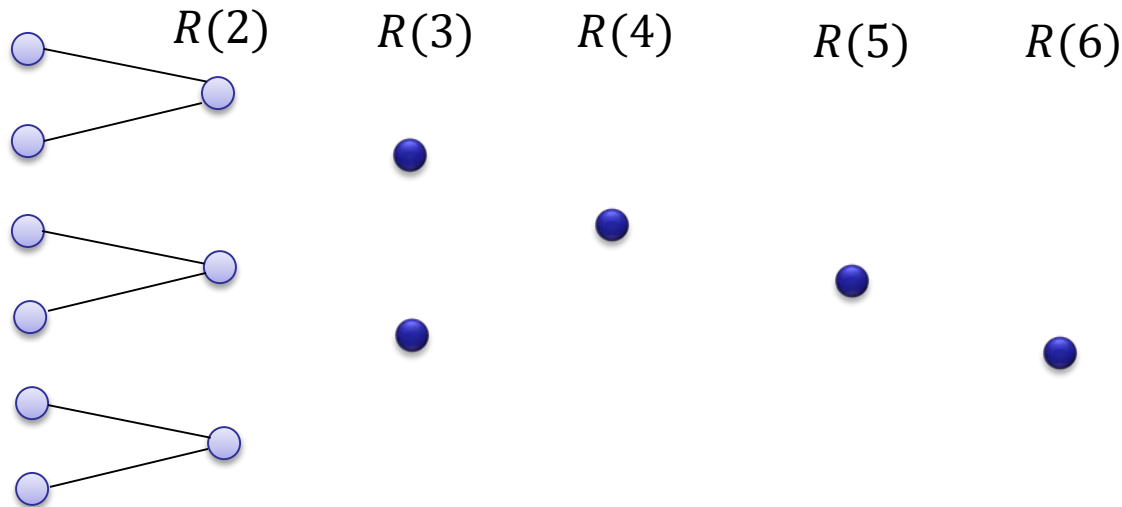
Approximationsalgorithmen

Was macht der Algorithmus?



Approximationsalgorithmen

Was macht der Algorithmus?

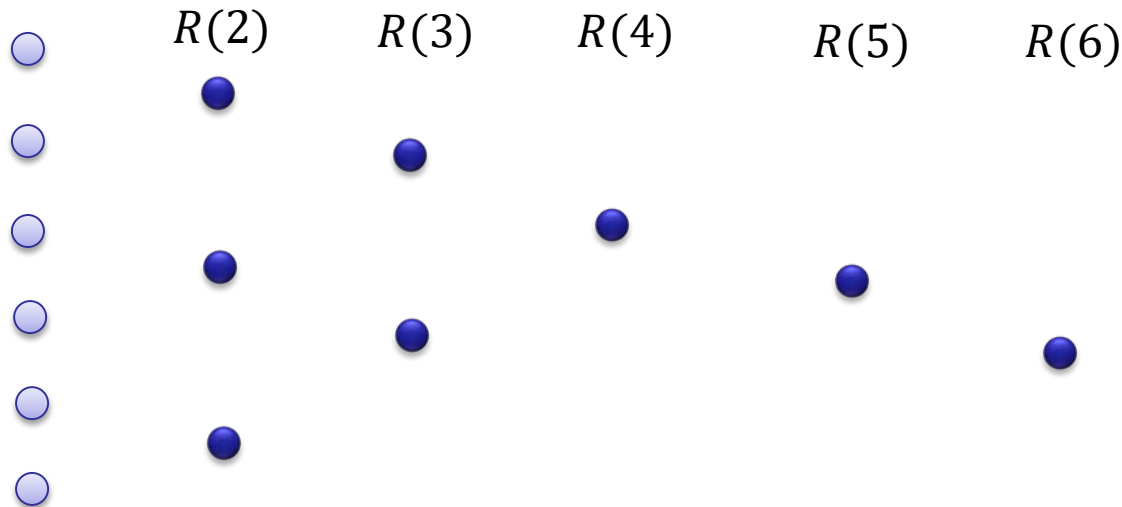


Die Menge L

Approximationsalgorithmen

Was macht der Algorithmus?

- Der Algorithmus wählt alle Knoten aus $R = \bigcup R(i)$



Die Menge L

Approximationsalgorithmen

Was macht der Algorithmus?

- Wie groß kann R werden?

Approximationsalgorithmen

Was macht der Algorithmus?

- Wie groß kann R werden?

$$|R| = \sum_{i=2}^r \left\lfloor \frac{|L|}{i} \right\rfloor \geq \sum_{i=2}^r \frac{|L|}{2i} = \frac{1}{2} \cdot \sum_{i=2}^r \frac{r}{i} = \frac{r}{2} \cdot \sum_{i=2}^r \frac{1}{i} \geq \frac{r}{2} (\ln r - 1) = \Omega(r \ln r)$$

Approximationsalgorithmen

Was macht der Algorithmus?

- Wie groß kann R werden?

$$|R| = \sum_{i=2}^r \left\lfloor \frac{|L|}{i} \right\rfloor \geq \sum_{i=2}^r \frac{|L|}{2i} = \frac{1}{2} \cdot \sum_{i=2}^r \frac{r}{i} = \frac{r}{2} \cdot \sum_{i=2}^r \frac{1}{i} \geq \frac{r}{2} (\ln r - 1) = \Omega(r \ln r)$$

- Damit ist das Approximationsverhältnis nicht konstant

Approximationsalgorithmen

Was macht der Algorithmus?

- Wie groß kann R werden?

$$|R| = \sum_{i=2}^r \left\lfloor \frac{|L|}{i} \right\rfloor \geq \sum_{i=2}^r \frac{|L|}{2i} = \frac{1}{2} \cdot \sum_{i=2}^r \frac{r}{i} = \frac{r}{2} \cdot \sum_{i=2}^r \frac{1}{i} \geq \frac{r}{2} (\ln r - 1) = \Omega(r \ln r)$$

- Damit ist das Approximationsverhältnis nicht konstant
- (Man kann zeigen, dass es für Graphen mit n Knoten $\mathbf{O}(\log n)$ ist)

Approximationsalgorithmen

Was macht der Algorithmus?

- Wie groß kann R werden?

$$|R| = \sum_{i=2}^r \left\lfloor \frac{|L|}{i} \right\rfloor \geq \sum_{i=2}^r \frac{|L|}{2i} = \frac{1}{2} \cdot \sum_{i=2}^r \frac{r}{i} = \frac{r}{2} \cdot \sum_{i=2}^r \frac{1}{i} \geq \frac{r}{2} (\ln r - 1) = \Omega(r \ln r)$$

- Damit ist das Approximationsverhältnis nicht konstant
- (Man kann zeigen, dass es für Graphen mit n Knoten $\mathbf{O}(\log n)$ ist)

Approximationsalgorithmen

Können wir einen besseren Algorithmus entwickeln?

- Wähle immer beide Endpunkte einer zufälligen Kante und entferne alle anliegenden Kanten

Approximationsalgorithmen

Können wir einen besseren Algorithmus entwickeln?

GreedyVertexCover2(G)

1. $C \leftarrow \emptyset$
2. $E' \leftarrow E(G)$
3. **while** $E' \neq \emptyset$ **do**
4. Sei (u, v) beliebige Kante aus E'
5. $C \leftarrow C \cup \{u, v\}$
6. Entferne aus E' jede Kante, die an u oder v anliegt
7. **return** C

Laufzeit: $\mathbf{O}(|V| + |E|)$

Approximationsalgorithmen

Satz 77

GreedyVertexCover2 ist ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

Approximationsalgorithmen

Satz 77

GreedyVertexCover2 ist ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

Beweis

- Die von GreedyVertexCover2 berechnete Menge C ist eine Knotenüberdeckung, da die **while**-Schleife solange durchlaufen wird, bis alle Kanten überdeckt sind

Approximationsalgorithmen

Satz 77

GreedyVertexCover2 ist ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

Beweis

- Die von GreedyVertexCover2 berechnete Menge C ist eine Knotenüberdeckung, da die **while**-Schleife solange durchlaufen wird, bis alle Kanten überdeckt sind
- Sei A die Menge der Kanten, die in Zeile 4 ausgewählt wurden

Approximationsalgorithmen

Satz 77

GreedyVertexCover2 ist ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

Beweis

- Die von GreedyVertexCover2 berechnete Menge C ist eine Knotenüberdeckung, da die **while**-Schleife solange durchlaufen wird, bis alle Kanten überdeckt sind
- Sei A die Menge der Kanten, die in Zeile 4 ausgewählt wurden
- Die Endpunkte der Kanten aus A sind disjunkt, da nach der Auswahl einer Kante alle an den Endpunkten anliegende Kanten gelöscht werden

Approximationsalgorithmen

Satz 77

GreedyVertexCover2 ist ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

Beweis

- Die von GreedyVertexCover2 berechnete Menge C ist eine Knotenüberdeckung, da die **while**-Schleife solange durchlaufen wird, bis alle Kanten überdeckt sind
- Sei A die Menge der Kanten, die in Zeile 4 ausgewählt wurden
- Die Endpunkte der Kanten aus A sind disjunkt, da nach der Auswahl einer Kante alle an den Endpunkten anliegende Kanten gelöscht werden
- Es gilt somit $|C| = 2|A|$

Approximationsalgorithmen

Satz 77

GreedyVertexCover2 ist ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

Beweis

- Die von GreedyVertexCover2 berechnete Menge C ist eine Knotenüberdeckung, da die **while**-Schleife solange durchlaufen wird, bis alle Kanten überdeckt sind
- Sei A die Menge der Kanten, die in Zeile 4 ausgewählt wurden
- Die Endpunkte der Kanten aus A sind disjunkt, da nach der Auswahl einer Kante alle an den Endpunkten anliegende Kanten gelöscht werden
- Es gilt somit $|C| = 2|A|$
- Jede Knotenüberdeckung (insbesondere eine optimale Überdeckung C^*) muss die Kanten aus A überdecken und somit mindestens einen Endpunkt jeder Kante enthalten

Approximationsalgorithmen

Satz 77

GreedyVertexCover2 ist ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

Beweis

- Die von GreedyVertexCover2 berechnete Menge C ist eine Knotenüberdeckung, da die **while**-Schleife solange durchlaufen wird, bis alle Kanten überdeckt sind
- Sei A die Menge der Kanten, die in Zeile 4 ausgewählt wurden
- Die Endpunkte der Kanten aus A sind disjunkt, da nach der Auswahl einer Kante alle an den Endpunkten anliegende Kanten gelöscht werden
- Es gilt somit $|C| = 2|A|$
- Jede Knotenüberdeckung (insbesondere eine optimale Überdeckung C^*) muss die Kanten aus A überdecken und somit mindestens einen Endpunkt jeder Kante enthalten

Approximationsalgorithmen

Satz 77

GreedyVertexCover2 ist ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

Beweis

- Da keine zwei Kanten aus A einen gemeinsamen Endpunkt haben, liegt kein Knoten aus der Überdeckung C^* an mehr als einer Kante aus A an

Approximationsalgorithmen

Satz 77

GreedyVertexCover2 ist ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

Beweis

- Da keine zwei Kanten aus A einen gemeinsamen Endpunkt haben, liegt kein Knoten aus der Überdeckung C^* an mehr als einer Kante aus A an
- Somit gilt $|A| \leq |C^*|$ und damit folgt $|C| \leq 2 |C^*|$

Approximationsalgorithmen

Satz 77

GreedyVertexCover2 ist ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

Beweis

- Da keine zwei Kanten aus A einen gemeinsamen Endpunkt haben, liegt kein Knoten aus der Überdeckung C^* an mehr als einer Kante aus A an
- Somit gilt $|A| \leq |C^*|$ und damit folgt $|C| \leq 2 |C^*|$

Approximationsalgorithmen

Travelling Salesman Problem (TSP)

- Sei $G = (V, E)$ ein ungerichteter vollständiger Graph mit positiven Kantengewichten $w(u, v)$ für alle $(u, v) \in E$; o.B.d.A. $V = \{1, \dots, n\}$
- Gesucht ist eine Reihenfolge $\pi(1), \dots, \pi(n)$ der Knoten aus V , so dass die Länge der Rundreise $\pi(1), \dots, \pi(n), \pi(1)$ minimiert wird
- Die Länge der Rundreise ist dabei gegeben durch

$$\sum_{i=1}^n w(\pi(i), \pi(i + 1 \bmod n))$$

Approximationsalgorithmen

Travelling Salesman Problem (TSP) mit Dreiecksungleichung

- Sei $G = (V, E)$ ein ungerichteter vollständiger Graph mit positiven Kantengewichten $w(u, v)$ für alle $(u, v) \in E$; o.B.d.A. $V = \{1, \dots, n\}$
- Gesucht ist eine Reihenfolge $\pi(1), \dots, \pi(n)$ der Knoten aus V , so dass die Länge der Rundreise $\pi(1), \dots, \pi(n), \pi(1)$ minimiert wird
- Die Länge der Rundreise ist dabei gegeben durch

$$\sum_{i=1}^n w(\pi(i), \pi(i+1 \bmod n))$$

Dreiecksungleichung

- Für je drei Knoten u, v, x gilt $w(u, x) \leq w(u, v) + w(v, x)$

Beispiel:

Finde eine möglichst kurze Rundreise durch alle deutschen Bundeshauptstädte.

Approximationsalgorithmen

ApproxTSP(G, w)

1. Berechne minimalen Spannbaum T von G
2. Sei π die Liste der Knoten von G in der Reihenfolge eines Preorder-Tree-Walk von einem beliebigen Knoten v
3. **return** π

Laufzeit

- $\mathbf{O}(|E| \log |E|)$ für die Spannbaumberechnung

Approximationsalgorithmen

ApproxTSP(G, w)

1. Berechne minimalen Spannbaum T von G
2. Sei π die Liste der Knoten von G in der Reihenfolge eines Preorder-Tree-Walk von einem beliebigen Knoten v
3. **return** π

Preorder-Tree-Walk

- Besucht rekursiv alle Knoten von T und gibt jeden Knoten sofort aus, wenn er besucht wird
- Dann erst finden die rekursiven Aufrufe für die Kinder statt

Traversierung eines Binärbaums mit Tiefensuche

Inorder-Tree-Walk(x)

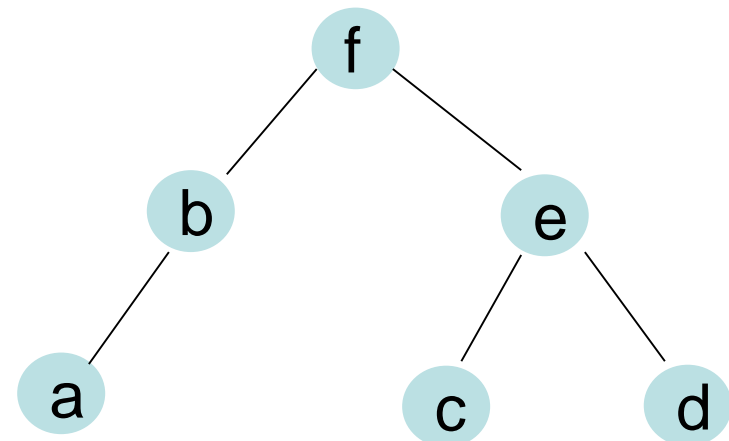
1. **if $x \neq \text{nil}$ then**
2. Inorder-Tree-Walk(lc[x])
3. Ausgabe key[x]
4. Inorder-Tree-Walk(rc[x])

Postorder-Tree-Walk(x)

1. **if $x \neq \text{nil}$ then**
2. Postorder-Tree-Walk(lc[x])
3. Postorder-Tree-Walk(rc[x])
4. Ausgabe key[x]

Preorder-Tree-Walk(x)

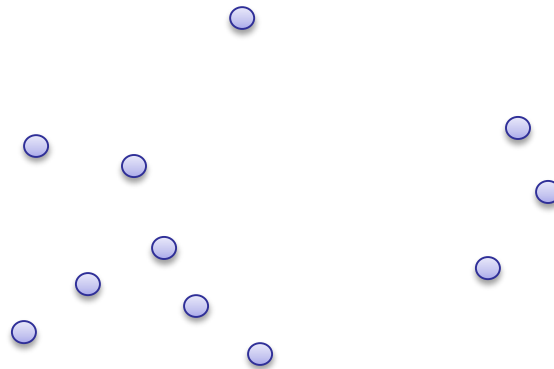
1. **if $x \neq \text{nil}$ then**
2. Ausgabe key[x]
3. Preorder-Tree-Walk(lc[x])
4. Preorder-Tree-Walk(rc[x])



Frage: Welche Traversierung erzeugt a,b,c,d,e,f?

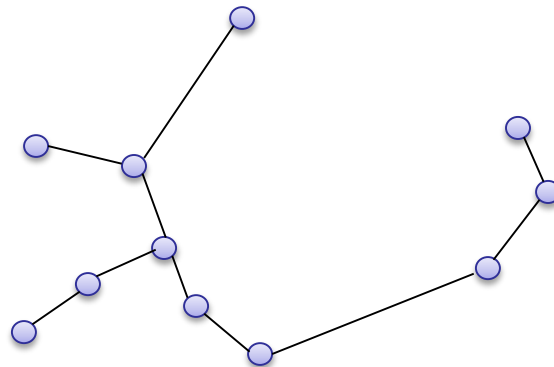
Approximationsalgorithmen

Beispiel (euklidische Entfernung)



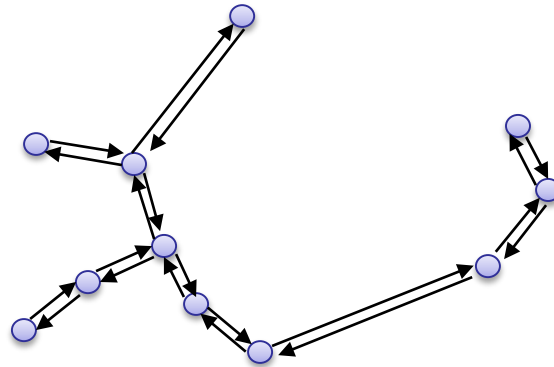
Approximationsalgorithmen

Beispiel (euklidische Entfernung)



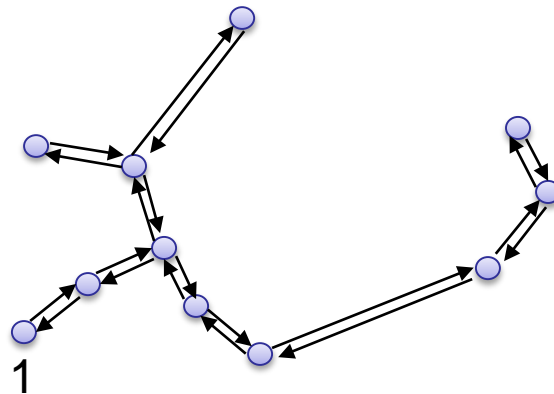
Approximationsalgorithmen

Beispiel (euklidische Entfernung)



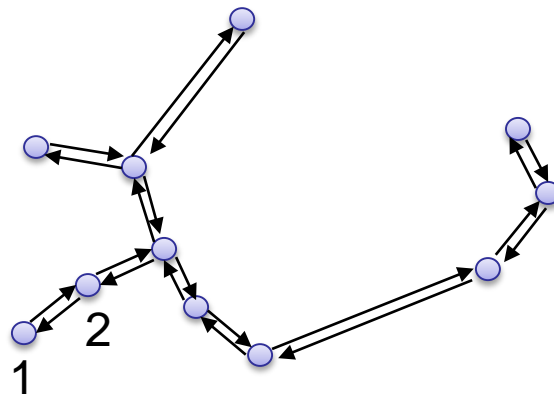
Approximationsalgorithmen

Beispiel (euklidische Entfernung)



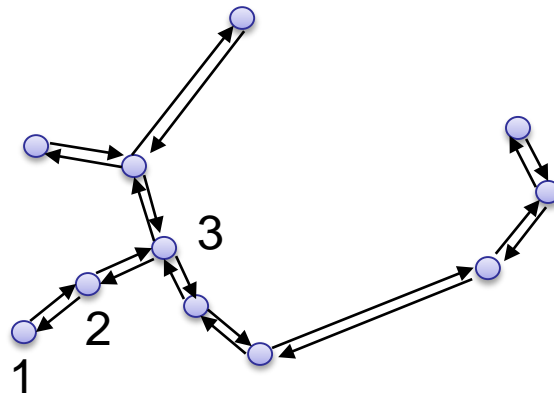
Approximationsalgorithmen

Beispiel (euklidische Entfernung)



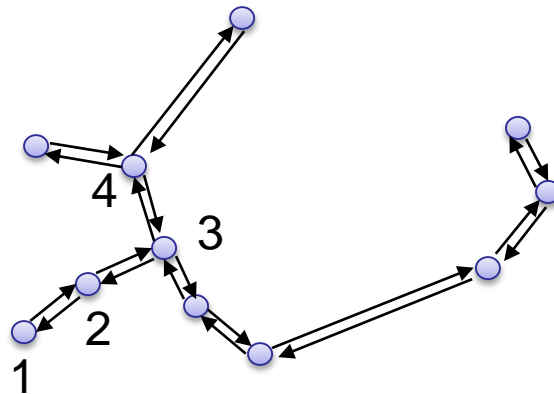
Approximationsalgorithmen

Beispiel (euklidische Entfernung)



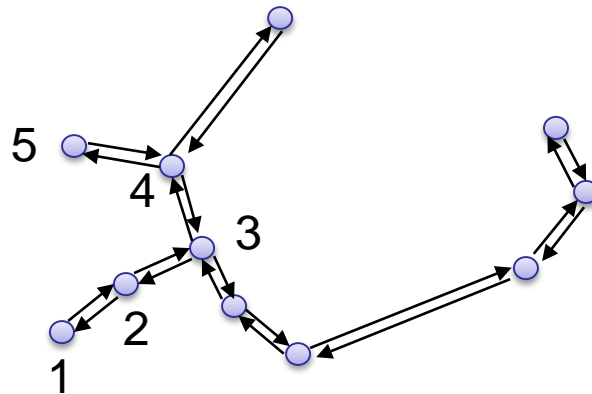
Approximationsalgorithmen

Beispiel (euklidische Entfernung)



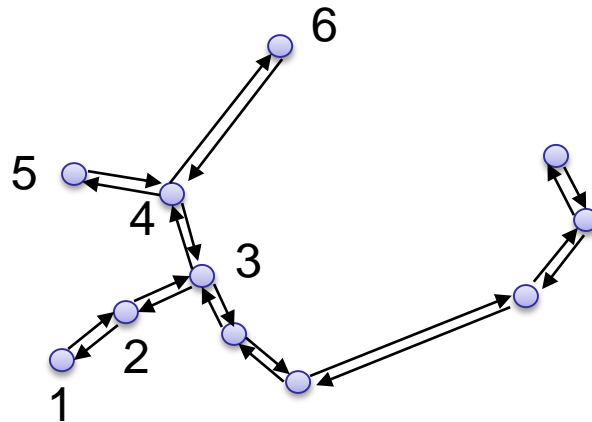
Approximationsalgorithmen

Beispiel (euklidische Entfernung)



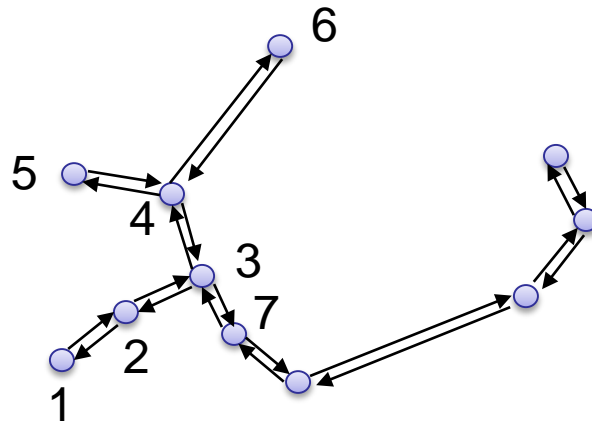
Approximationsalgorithmen

Beispiel (euklidische Entfernung)



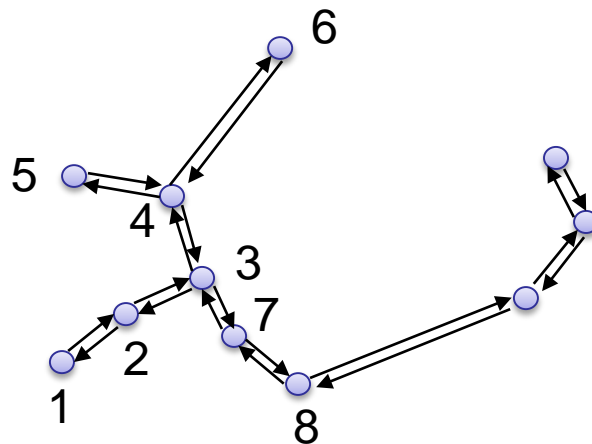
Approximationsalgorithmen

Beispiel (euklidische Entfernung)



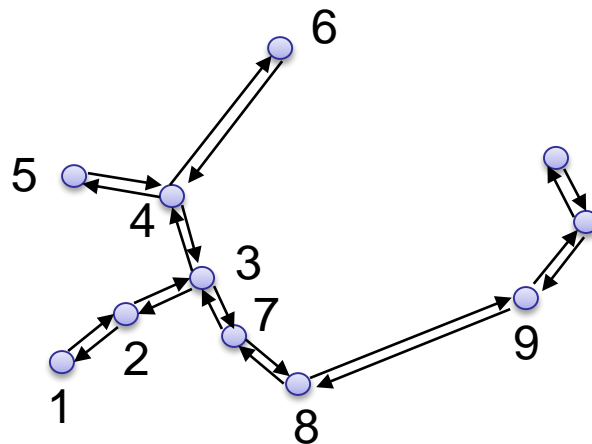
Approximationsalgorithmen

Beispiel (euklidische Entfernung)



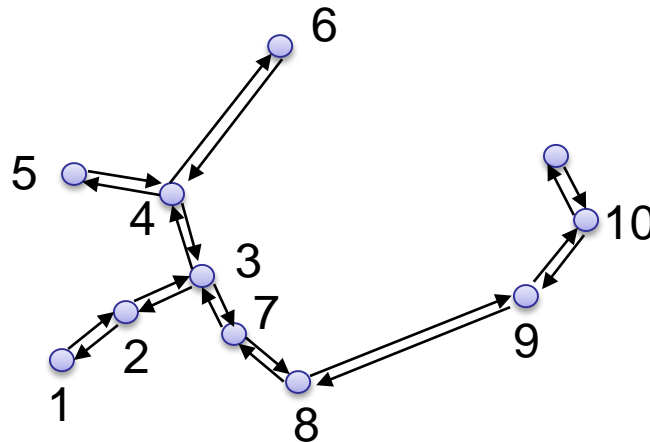
Approximationsalgorithmen

Beispiel (euklidische Entfernung)



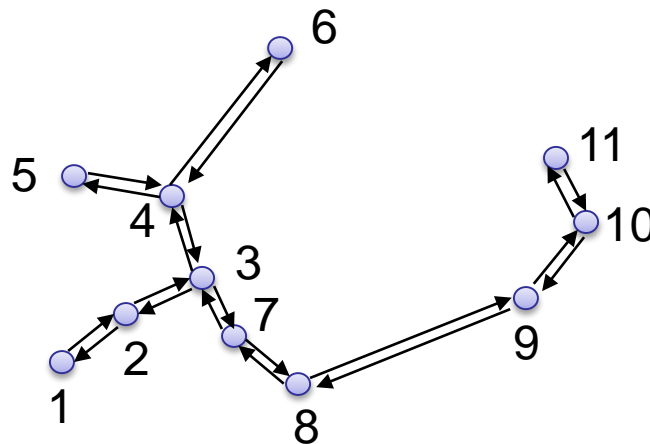
Approximationsalgorithmen

Beispiel (euklidische Entfernung)



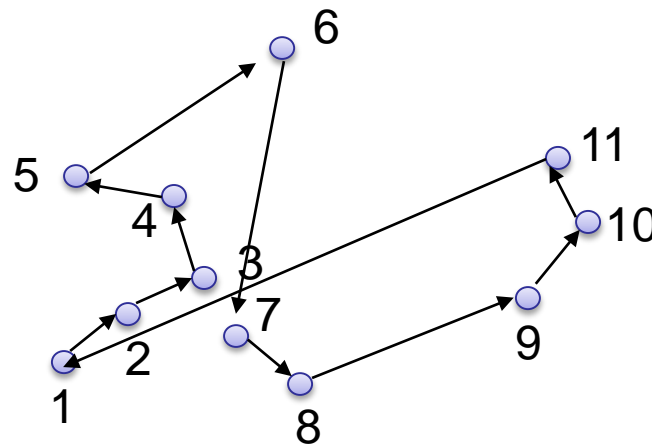
Approximationsalgorithmen

Beispiel (euklidische Entfernung)



Approximationsalgorithmen

Beispiel (euklidische Entfernung)



Approximationsalgorithmen

Satz 78

Algorithmus ApproxTSP ist ein 2-Approximationsalgorithmus für das Travelling Salesman Problem mit Dreiecksungleichung.

Beweis

- Sei H^* eine optimale Rundreise und bezeichne $w(H^*)$ ihre Kosten

Approximationsalgorithmen

Satz 78

Algorithmus ApproxTSP ist ein 2-Approximationsalgorithmus für das Travelling Salesman Problem mit Dreiecksungleichung.

Beweis

- Sei H^* eine optimale Rundreise und bezeichne $w(H^*)$ ihre Kosten
- Z.z.: $w(H) \leq 2 \cdot w(H^*)$, wobei H die von ApproxTSP zurückgegebene Rundreise ist und $w(H)$ ihre Kosten bezeichnet

Approximationsalgorithmen

Satz 78

Algorithmus ApproxTSP ist ein 2-Approximationsalgorithmus für das Travelling Salesman Problem mit Dreiecksungleichung.

Beweis

- Sei H^* eine optimale Rundreise und bezeichne $w(H^*)$ ihre Kosten
- Z.z.: $w(H) \leq 2 \cdot w(H^*)$, wobei H die von ApproxTSP zurückgegebene Rundreise ist und $w(H)$ ihre Kosten bezeichnet
- Sei T ein min. Spannbaum und $w(T)$ seine Kosten

Approximationsalgorithmen

Satz 78

Algorithmus ApproxTSP ist ein 2-Approximationsalgorithmus für das Travelling Salesman Problem mit Dreiecksungleichung.

Beweis

- Sei H^* eine optimale Rundreise und bezeichne $w(H^*)$ ihre Kosten
- Z.z.: $w(H) \leq 2 \cdot w(H^*)$, wobei H die von ApproxTSP zurückgegebene Rundreise ist und $w(H)$ ihre Kosten bezeichnet
- Sei T ein min. Spannbaum und $w(T)$ seine Kosten
- Es gilt $w(T) \leq w(H^*)$, da man durch Löschen einer Kante aus H^* einen Spannbaum bekommen kann. Dieser hat Gewicht mind. $w(T)$

Approximationsalgorithmen

Satz 78

Algorithmus ApproxTSP ist ein 2-Approximationsalgorithmus für das Travelling Salesman Problem mit Dreiecksungleichung.

Beweis

- Sei H^* eine optimale Rundreise und bezeichne $w(H^*)$ ihre Kosten
- Z.z.: $w(H) \leq 2 \cdot w(H^*)$, wobei H die von ApproxTSP zurückgegebene Rundreise ist und $w(H)$ ihre Kosten bezeichnet
- Sei T ein min. Spannbaum und $w(T)$ seine Kosten
- Es gilt $w(T) \leq w(H^*)$, da man durch Löschen einer Kante aus H^* einen Spannbaum bekommen kann. Dieser hat Gewicht mind. $w(T)$

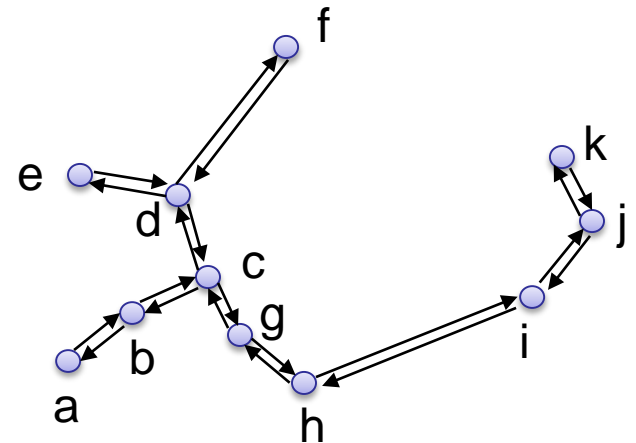
Approximationsalgorithmen

Beweis

- Ein FullWalk gibt die Knoten bei jedem ersten Besuch aus und auch immer, wenn der Algorithmus zu ihnen zurückkehrt

In unserem Beispiel:

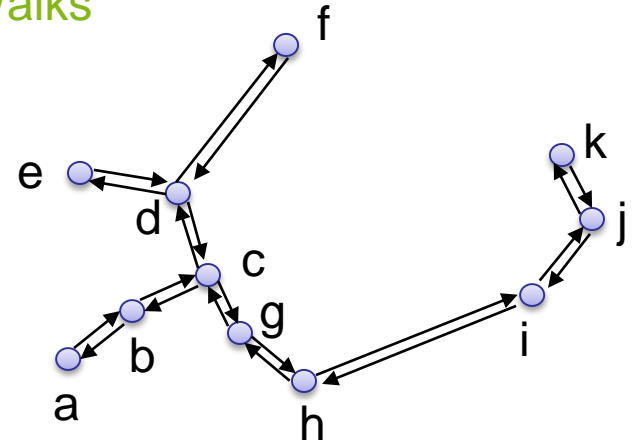
- a, b, c, d, e, d, f, d, c, g, h, i, j, k, j, i, h, g, c, b, a



Approximationsalgorithmen

Beweis

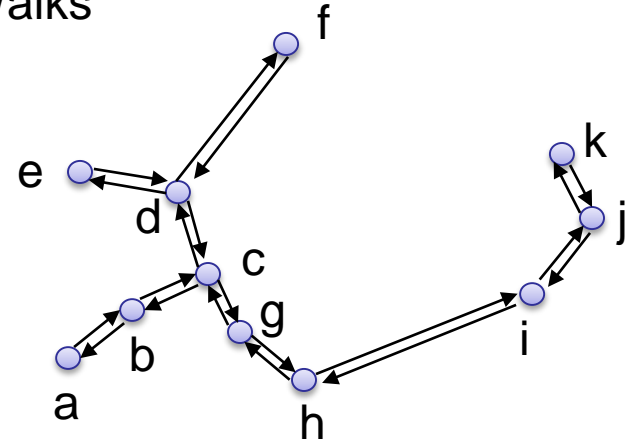
- Ein FullWalk gibt die Knoten bei jedem ersten Besuch aus und auch immer, wenn der Algorithmus zu ihnen zurückkehrt
- Da der FullWalk F jede Kante von T genau zweimal durchquert, gilt $w(F) = 2 w(T)$, wobei $w(F)$ die Kosten des FullWalks bezeichnet



Approximationsalgorithmen

Beweis

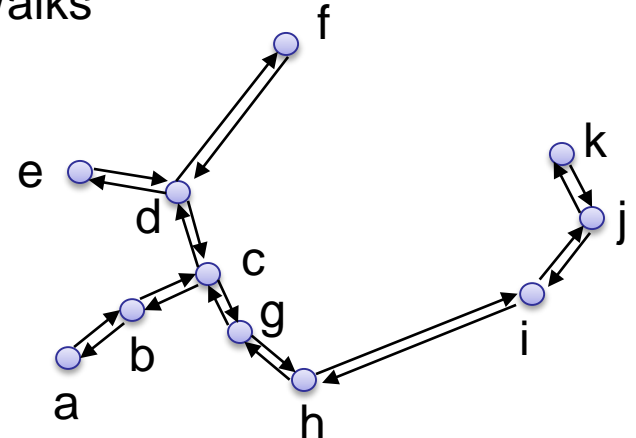
- Ein FullWalk gibt die Knoten bei jedem ersten Besuch aus und auch immer, wenn der Algorithmus zu ihnen zurückkehrt
- Da der FullWalk F jede Kante von T genau zweimal durchquert, gilt $w(F) = 2 w(T)$, wobei $w(F)$ die Kosten des FullWalks bezeichnet
- Also folgt $w(F) \leq 2 w(H^*)$



Approximationsalgorithmen

Beweis

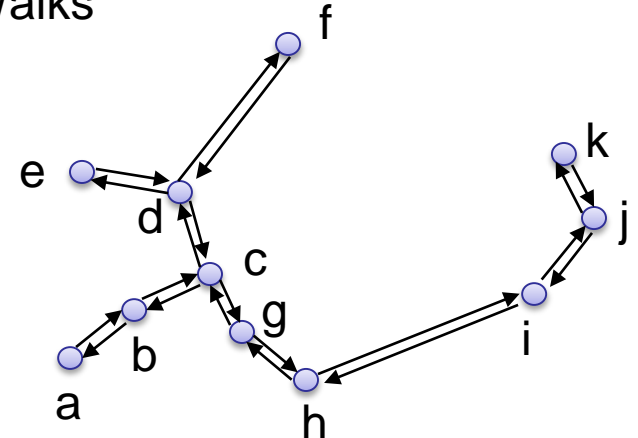
- Ein FullWalk gibt die Knoten bei jedem ersten Besuch aus und auch immer, wenn der Algorithmus zu ihnen zurückkehrt
- Da der FullWalk F jede Kante von T genau zweimal durchquert, gilt $w(F) = 2 w(T)$, wobei $w(F)$ die Kosten des FullWalks bezeichnet
- Also folgt $w(F) \leq 2 w(H^*)$
- F ist jedoch keine Rundreise (und nicht die von ApproxTSP berechnete Ausgabe)



Approximationsalgorithmen

Beweis

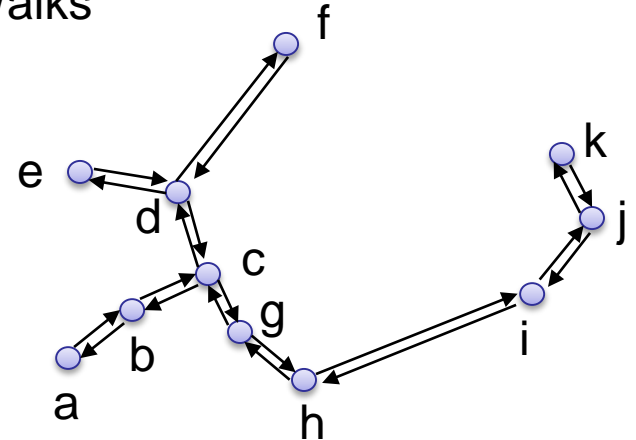
- Ein FullWalk gibt die Knoten bei jedem ersten Besuch aus und auch immer, wenn der Algorithmus zu ihnen zurückkehrt
- Da der FullWalk F jede Kante von T genau zweimal durchquert, gilt $w(F) = 2 w(T)$, wobei $w(F)$ die Kosten des FullWalks bezeichnet
- Also folgt $w(F) \leq 2 w(H^*)$
- F ist jedoch keine Rundreise (und nicht die von ApproxTSP berechnete Ausgabe)
- Wir formen nun F in diese Ausgabe um, ohne die Kosten zu erhöhen



Approximationsalgorithmen

Beweis

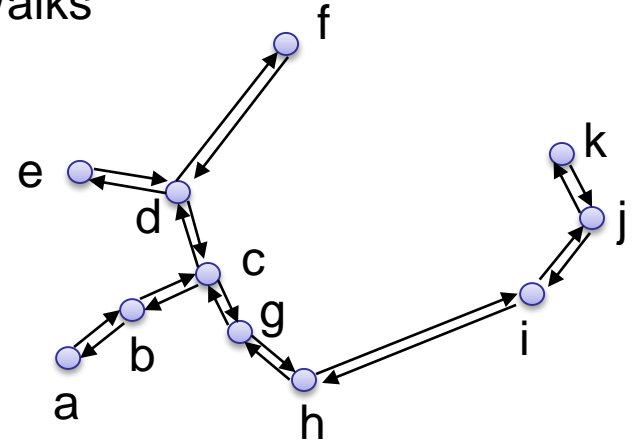
- Ein FullWalk gibt die Knoten bei jedem ersten Besuch aus und auch immer, wenn der Algorithmus zu ihnen zurückkehrt
- Da der FullWalk F jede Kante von T genau zweimal durchquert, gilt $w(F) = 2 w(T)$, wobei $w(F)$ die Kosten des FullWalks bezeichnet
- Also folgt $w(F) \leq 2 w(H^*)$
- F ist jedoch keine Rundreise (und nicht die von ApproxTSP berechnete Ausgabe)
- Wir formen nun F in diese Ausgabe um, ohne die Kosten zu erhöhen
- Beobachtung: Aufgrund der Dreiecksungleichung können wir den Besuch eines Knotens aus F löschen, ohne die Kosten der Rundreise zu erhöhen (wird v zwischen u und x gelöscht, so werden die Kanten (u, v) und (v, x) durch (u, x) ersetzt)



Approximationsalgorithmen

Beweis

- Ein FullWalk gibt die Knoten bei jedem ersten Besuch aus und auch immer, wenn der Algorithmus zu ihnen zurückkehrt
- Da der FullWalk F jede Kante von T genau zweimal durchquert, gilt $w(F) = 2 w(T)$, wobei $w(F)$ die Kosten des FullWalks bezeichnet
- Also folgt $w(F) \leq 2 w(H^*)$
- F ist jedoch keine Rundreise (und nicht die von ApproxTSP berechnete Ausgabe)
- Wir formen nun F in diese Ausgabe um, ohne die Kosten zu erhöhen
- Beobachtung: Aufgrund der Dreiecksungleichung können wir den Besuch eines Knotens aus F löschen, ohne die Kosten der Rundreise zu erhöhen (wird v zwischen u und x gelöscht, so werden die Kanten (u, v) und (v, x) durch (u, x) ersetzt)



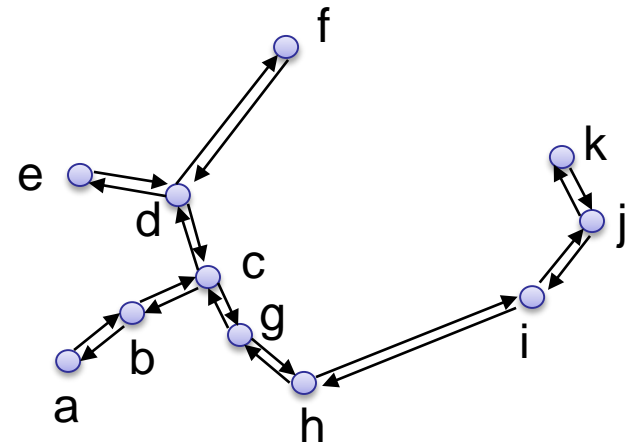
Approximationsalgorithmen

Beweis

- Auf diese Weise können wir alle Besuche außer den ersten aus unserer Liste entfernen

In unserem Beispiel:

- a, b, c, d, e, d, f, d, c, g, h, i, j, k, j, i, h, g, c, b, a



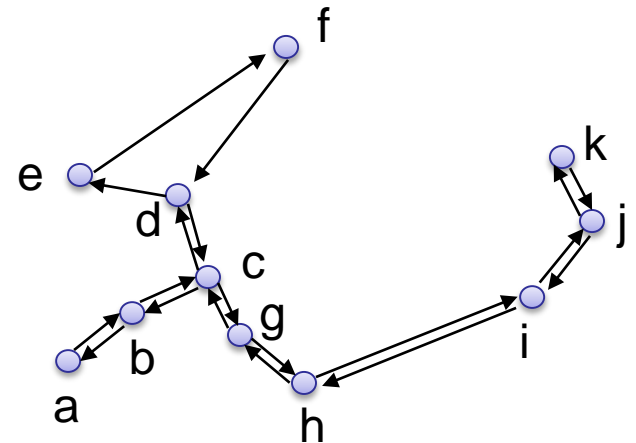
Approximationsalgorithmen

Beweis

- Auf diese Weise können wir alle Besuche außer den ersten aus unserer Liste entfernen

In unserem Beispiel:

- a, b, c, d, e, f, d, c, g, h, i, j, k, j, i, h, g, c, b, a



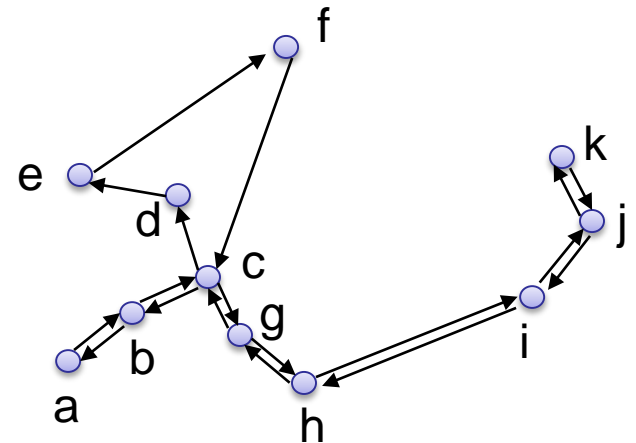
Approximationsalgorithmen

Beweis

- Auf diese Weise können wir alle Besuche außer den ersten aus unserer Liste entfernen

In unserem Beispiel:

- a, b, c, d, e, f, c, g, h, i, j, k, j, i, h, g, c, b, a



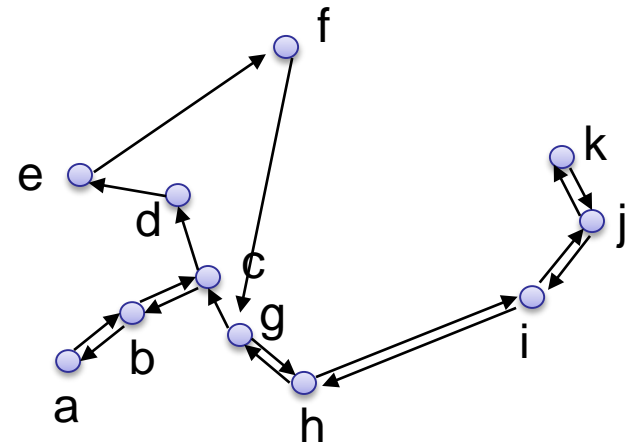
Approximationsalgorithmen

Beweis

- Auf diese Weise können wir alle Besuche außer den ersten aus unserer Liste entfernen

In unserem Beispiel:

- a, b, c, d, e, f, g, h, i, j, k, j, i, h, g, c, b, a



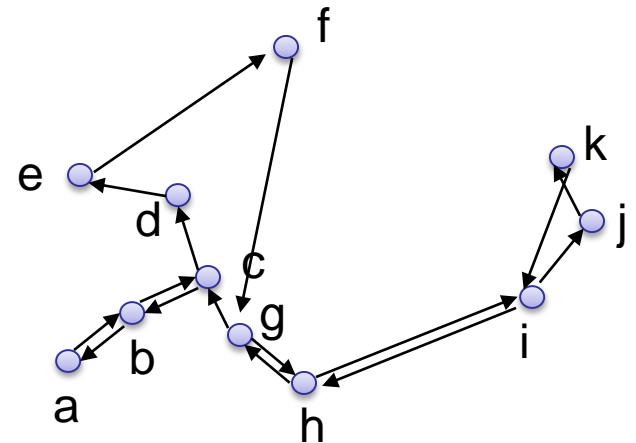
Approximationsalgorithmen

Beweis

- Auf diese Weise können wir alle Besuche außer den ersten aus unserer Liste entfernen

In unserem Beispiel:

- a, b, c, d, e, f, g, h, i, j, k, i, h, g, c, b, a



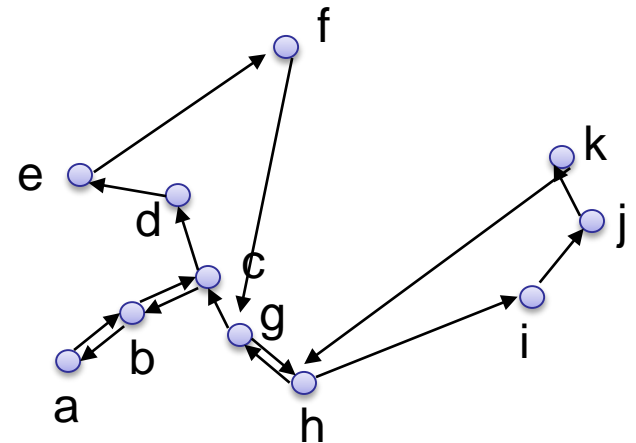
Approximationsalgorithmen

Beweis

- Auf diese Weise können wir alle Besuche außer den ersten aus unserer Liste entfernen

In unserem Beispiel:

- a, b, c, d, e, f, g, h, i, j, k, h, g, c, b, a



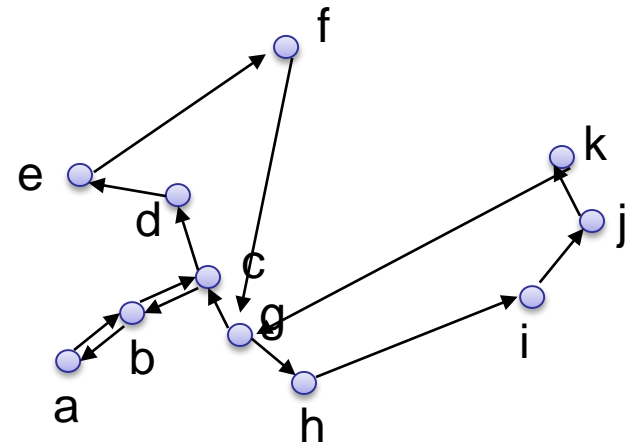
Approximationsalgorithmen

Beweis

- Auf diese Weise können wir alle Besuche außer den ersten aus unserer Liste entfernen

In unserem Beispiel:

- a, b, c, d, e, f, g, h, i, j, k, g, c, b, a



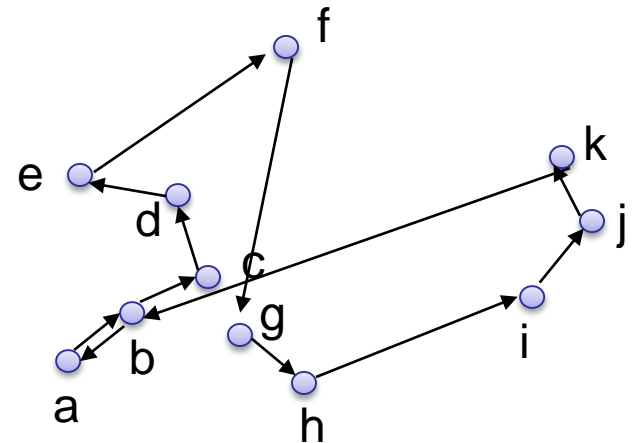
Approximationsalgorithmen

Beweis

- Auf diese Weise können wir alle Besuche außer den ersten aus unserer Liste entfernen

In unserem Beispiel:

- a, b, c, d, e, f, g, h, i, j, k, b, a



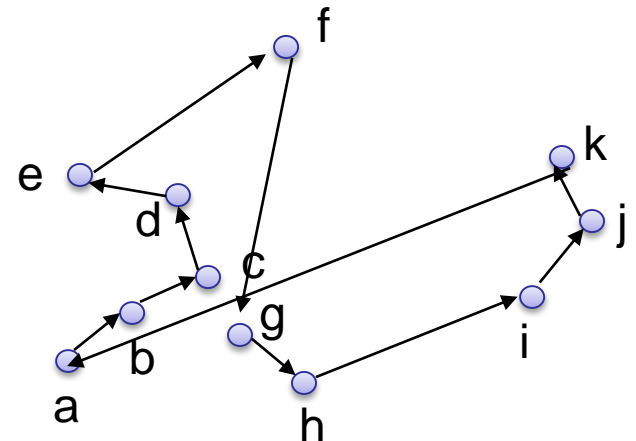
Approximationsalgorithmen

Beweis

- Auf diese Weise können wir alle Besuche außer den ersten aus unserer Liste entfernen

In unserem Beispiel:

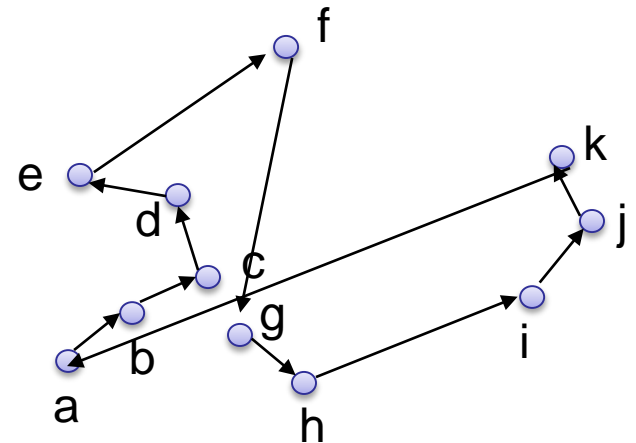
- a, b, c, d, e, f, g, h, i, j, k, a



Approximationsalgorithmen

Beweis

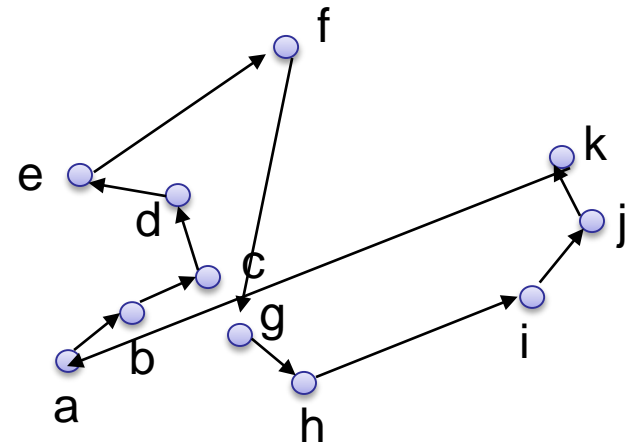
- Auf diese Weise können wir alle Besuche außer den ersten aus unserer Liste entfernen
- Wir erhalten dieselbe Rundreise wie bei Preorder-Tree-Walk



Approximationsalgorithmen

Beweis

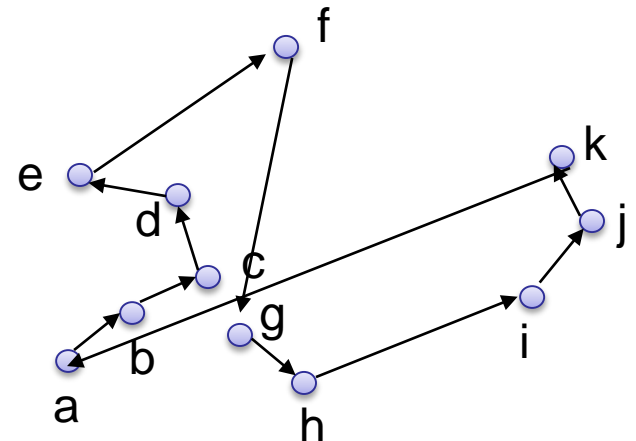
- Auf diese Weise können wir alle Besuche außer den ersten aus unserer Liste entfernen
- Wir erhalten dieselbe Rundreise wie bei Preorder-Tree-Walk
- Da diese nur durch „abkürzen“ von F zu H gekommen ist, gilt $w(H) \leq w(F)$



Approximationsalgorithmen

Beweis

- Auf diese Weise können wir alle Besuche außer den ersten aus unserer Liste entfernen
- Wir erhalten dieselbe Rundreise wie bei Preorder-Tree-Walk
- Da diese nur durch „abkürzen“ von F zu Stande gekommen ist, gilt $w(H) \leq w(F)$
- Somit folgt $w(H) \leq w(F) \leq 2 w(H^*)$



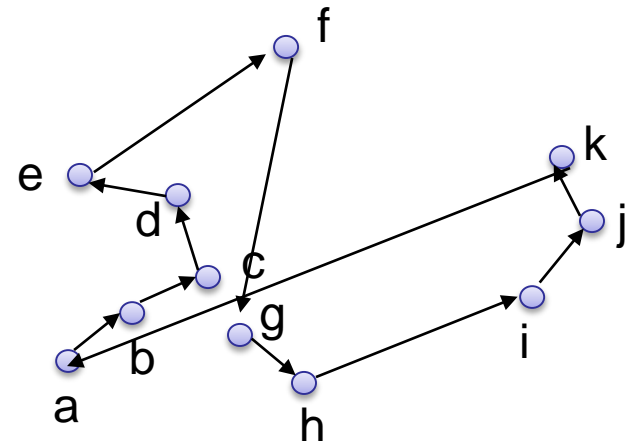
Approximationsalgorithmen

Beweis

- Auf diese Weise können wir alle Besuche außer den ersten aus unserer Liste entfernen
- Wir erhalten dieselbe Rundreise wie bei Preorder-Tree-Walk
- Da diese nur durch „abkürzen“ von F zu Stande gekommen ist, gilt $w(H) \leq w(F)$
- Somit folgt $w(H) \leq w(F) \leq 2 w(H^*)$

Zusammenfassung:

ApproxTSP berechnet in $\mathbf{O}(|E| \log |E|)$ Zeit
eine 2-Approximation



Approximationsalgorithmen

Last Balanzierung

- m identische Maschinen $\{1, \dots, m\}$
- n Aufgabe $\{1, \dots, n\}$
- Aufgabe j hat Länge $t(j)$
- Problem: Platziere die Aufgaben auf den Maschinen, so dass diese möglichst „balanciert“ sind
- Sei $A(i)$ die Menge der Aufgaben auf Maschine i
- Sei $T(i) = \sum_{j \in A(i)} t(j)$
- Makespan: $\max T(i)$
- Präzises Problem : Minimiere Makespan

Frage: Was ist der minimale Makespan von $n = 5$ Aufgaben der Länge 1,2,3,4,5 auf $m = 3$ Maschinen?

Approximationsalgorithmen

Gieriger Ansatz:

- verteile die Aufgaben der Reihe nach
- wähle immer eine Maschine mit kleinster Belastung

Approximationsalgorithmen

GreedyLoadBalancing

1. Setze $T(i) \leftarrow 0$ und $A(i) \leftarrow \emptyset$ für alle Maschinen $i \in \{1, \dots, m\}$
2. **for** $j = 1$ **to** n **do**
3. Sei $M(i)$ eine Maschine mit $T(i) = \min_{k \in \{1, \dots, m\}} T(k)$
4. Weise Aufgabe j Maschine i zu
5. $A(i) \leftarrow A(i) \cup \{j\}$
6. $T(i) \leftarrow T(i) + t(j)$

Approximationsalgorithmen

Satz 79

Algorithmus GreedyLoadBalancing hat ein Approximationsverhältnis von mindestens $2 - 1/m$.

Beweis

- Eingabe: m ($m - 1$) Aufgaben der Länge 1 und eine Aufgabe der Länge m

Approximationsalgorithmen

Satz 79

Algorithmus GreedyLoadBalancing hat ein Approximationsverhältnis von mindestens $2 - 1/m$.

Beweis

- Eingabe: m ($m - 1$) Aufgaben der Länge 1 und eine Aufgabe der Länge m
- Optimale Lösung:
- Die Aufgabe der Länge m wird einer Maschine zugeteilt
- Die anderen Aufgaben werden gleichmäßig auf die übrigen $m - 1$ Maschinen verteilt; resultierender Makespan: m

Maschine 1: 

Maschine 2: 

Maschine 3:

Approximationsalgorithmen

Satz 79

Algorithmus GreedyLoadBalancing hat ein Approximationsverhältnis von mindestens $2 - 1/m$.

Beweis

- Eingabe: m ($m - 1$) Aufgaben der Länge 1 und eine Aufgabe der Länge m
- Optimale Lösung:
- Die Aufgabe der Länge m wird einer Maschine zugeteilt
- Die anderen Aufgaben werden gleichmäßig auf die übrigen $m - 1$ Maschinen verteilt; resultierender Makespan: m

Maschine 1: 

Maschine 2: 

Maschine 3: 

Approximationsalgorithmen

Satz 79

Algorithmus GreedyLoadBalancing hat ein Approximationsverhältnis von mindestens $2 - 1/m$.

Beweis

- GreedyLoadBalancing verteilt zunächst die kurzen Aufgaben gleichmäßig

Maschine 1: ■■

Maschine 2: ■■

Maschine 3: ■■

Approximationsalgorithmen

Satz 79


Algorithmus GreedyLoadBalancing hat ein Approximationsverhältnis von mindestens $2 - 1/m$.

Beweis

- GreedyLoadBalancing verteilt zunächst die kurzen Aufgaben gleichmäßig
- Danach wird die lange Aufgabe zugewiesen

Maschine 1: 

Maschine 2: 

Maschine 3: 

Approximationsalgorithmen

Satz 79


Algorithmus GreedyLoadBalancing hat ein Approximationsverhältnis von mindestens $2 - 1/m$.

Beweis

- GreedyLoadBalancing verteilt zunächst die kurzen Aufgaben gleichmäßig
- Danach wird die lange Aufgabe zugewiesen
- Makespan: $2m - 1$

Maschine 1: 

Maschine 2: 

Maschine 3: 

Approximationsalgorithmen

Satz 79


Algorithmus GreedyLoadBalancing hat ein Approximationsverhältnis von mindestens $2 - 1/m$.

Beweis

- GreedyLoadBalancing verteilt zunächst die kurzen Aufgaben gleichmäßig
- Danach wird die lange Aufgabe zugewiesen
- Makespan: $2m - 1$

Maschine 1: 

Maschine 2: 

Maschine 3: 

Approximationsalgorithmen

Satz 79

Algorithmus GreedyLoadBalancing hat ein Approximationsverhältnis von mindestens $2 - 1/m$.

Beweis

- Damit ist das Approximationsverhältnis mindestens $(2m - 1)/m = 2 - 1/m$.

Approximationsalgorithmen

Beobachtung 80

- Für jede Problemistanz ist der optimale Makespan mindestens

$$T^* = \frac{1}{m} \sum_{j=1}^n t(j)$$

- Begründung: Bestenfalls können wir die Aufgaben genau auf die m Maschinen aufteilen und jede Maschine hat Last Gesamtlast/Anzahl Maschinen

Approximationsalgorithmen

Satz 81

Algorithmus GreedyLoadBalancing ist ein 2-Approximationsalgorithmus für das Lastbalancierungsproblem.

Approximationsalgorithmen

Satz 81

Algorithmus GreedyLoadBalancing ist ein 2-Approximationsalgorithmus für das Lastbalancierungsproblem.

Beweis

- Sei i^* die Maschine, die maximale Last in der vom Algorithmus berechneten Zuteilung erhält

Approximationsalgorithmen

Satz 81

Algorithmus GreedyLoadBalancing ist ein 2-Approximationsalgorithmus für das Lastbalancierungsproblem.

Beweis

- Sei i^* die Maschine, die maximale Last in der vom Algorithmus berechneten Zuteilung erhält
- Sei j^* die Aufgabe, die Maschine i^* als letzte zugewiesen wurde

Approximationsalgorithmen

Satz 81

Algorithmus GreedyLoadBalancing ist ein 2-Approximationsalgorithmus für das Lastbalancierungsproblem.

Beweis

- Sei i^* die Maschine, die maximale Last in der vom Algorithmus berechneten Zuteilung erhält
- Sei j^* die Aufgabe, die Maschine i^* als letzte zugewiesen wurde
- Es gilt: $T(k) \geq T(i^*) - t(j^*)$ für alle Maschinen k , da zum Zeitpunkt der Zuweisung von j^* , $T(i^*)$ Minimum der $T(k)$ war

Approximationsalgorithmen

Satz 81

Algorithmus GreedyLoadBalancing ist ein 2-Approximationsalgorithmus für das Lastbalancierungsproblem.

Beweis

- Sei i^* die Maschine, die maximale Last in der vom Algorithmus berechneten Zuteilung erhält
- Sei j^* die Aufgabe, die Maschine i^* als letzte zugewiesen wurde
- Es gilt: $T(k) \geq T(i^*) - t(j^*)$ für alle Maschinen k , da zum Zeitpunkt der Zuweisung von j^* , $T(i^*)$ Minimum der $T(k)$ war
- Somit folgt für die Kosten Opt einer optimalen Zuweisung:

Approximationsalgorithmen

Satz 81

Algorithmus GreedyLoadBalancing ist ein 2-Approximationsalgorithmus für das Lastbalancierungsproblem.

Beweis

- Sei i^* die Maschine, die maximale Last in der vom Algorithmus berechneten Zuteilung erhält
- Sei j^* die Aufgabe, die Maschine i^* als letzte zugewiesen wurde
- Es gilt: $T(k) \geq T(i^*) - t(j^*)$ für alle Maschinen k , da zum Zeitpunkt der Zuweisung von j^* , $T(i^*)$ Minimum der $T(k)$ war
- Somit folgt für die Kosten Opt einer optimalen Zuweisung:

$$\text{Opt} \geq \frac{1}{m} \sum_{k=1}^m T(k) \geq T(i^*) - t(j^*)$$

Approximationsalgorithmen

Satz 81

Algorithmus GreedyLoadBalancing ist ein 2-Approximationsalgorithmus für das Lastbalancierungsproblem.

Beweis

- Sei i^* die Maschine, die maximale Last in der vom Algorithmus berechneten Zuteilung erhält
- Sei j^* die Aufgabe, die Maschine i^* als letzte zugewiesen wurde
- Es gilt: $T(k) \geq T(i^*) - t(j^*)$ für alle Maschinen k , da zum Zeitpunkt der Zuweisung von j^* , $T(i^*)$ Minimum der $T(k)$ war
- Somit folgt für die Kosten Opt einer optimalen Zuweisung:

$$\text{Opt} \geq \frac{1}{m} \sum_{k=1}^m T(k) \geq T(i^*) - t(j^*)$$

Approximationsalgorithmen

Satz 81

Algorithmus GreedyLoadBalancing ist ein 2-Approximationsalgorithmus für das Lastbalancierungsproblem.

Beweis

- Außerdem gilt sicher $\text{Opt} \geq t(j^*)$

Approximationsalgorithmen

Satz 81

Algorithmus GreedyLoadBalancing ist ein 2-Approximationsalgorithmus für das Lastbalancierungsproblem.

Beweis

- Außerdem gilt sicher $\text{Opt} \geq t(j^*)$
- Es folgt

$$T(i^*) = (T(i^*) - t(j^*)) + t(j^*) \leq 2 \cdot \text{Opt}$$

Approximationsalgorithmen

Das (diskrete) k -Center Clustering Problem

- Gegeben: Menge P von n Punkten in der Ebene
- Gesucht: Menge $C \subseteq P$ von k Zentren, so dass die maximale Distanz der Punkte zum nächstgelegenen Zentrum, d.h.
$$\text{cost}(P, C) = \max_{p \in P} d(p, C)$$
 minimiert wird, wobei
- $\text{dist}(p, C) = \min_{q \in C} \text{dist}(p, q)$ und
- $\text{dist}(p, q)$ bezeichnet den Abstand (Euklidische Distanz) von p und q

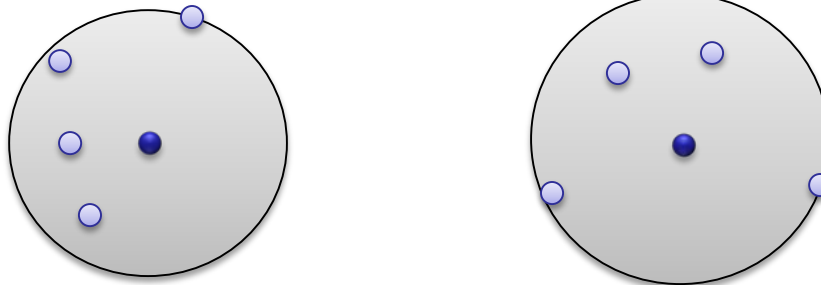
Approximationsalgorithmen

Beispiel



Approximationsalgorithmen

Beispiel



Alternative Formulierung

Finde k Scheiben mit Zentrum aus P , die alle Punkte abdecken und deren Maximaler Radius minimiert wird.

Approximationsalgorithmen

Typische Anwendung

- Punkte symbolisieren Städte
- Will Mobilfunkmasten mit möglichst geringer Leistung aufstellen, so dass alle Städte versorgt sind

Allgemeiner

- Punkte (typischerweise in höheren Dimensionen) sind „Beschreibungen von Objekten“
- Will Objekte in Gruppen von ähnlichen Objekten unterteilen

Approximationsalgorithmen

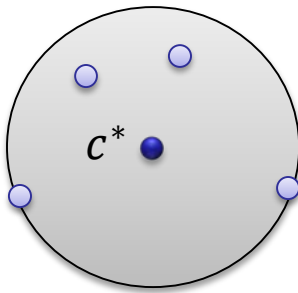
Ein Gedankenexperiment

- Nehmen wir an, wir kennen die Kosten r einer optimalen Lösung, d.h. wir wissen, dass man mit Scheiben mit Radius r und Zentrum aus P die Punkte abdecken kann
- Wir werden zeigen, dass wir dann einen einfachen 2-Approximationsalgorithmus finden können

Approximationsalgorithmen

Idee

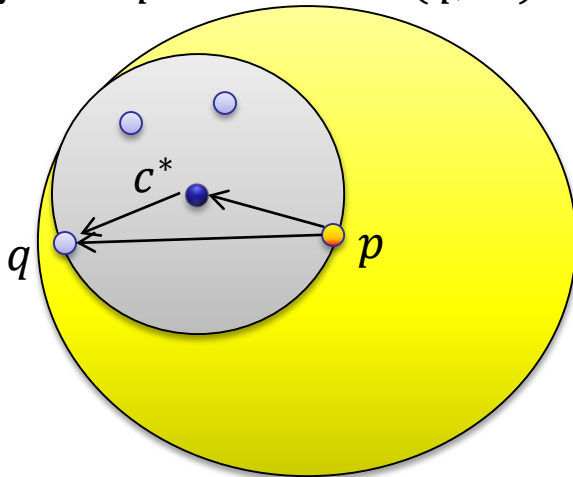
- Wir nutzen Existenz von Lösung C^* mit Radius (Kosten) r
- Betrachte Punkt $p \in P$
- Dann gibt es Zentrum $c^* \in C^*$ mit $\text{dist}(p, c^*) \leq r$
- Nehmen wir nun p als Zentrum anstelle von c^* und verdoppeln wir den Radius, so decken wir jeden Punkt q ab, der von c^* mit Radius r abgedeckt wurde, d.h.
- für jedes $q \in P$ mit $\text{dist}(q, c^*) \leq r$ gilt $\text{dist}(p, q) \leq \text{dist}(p, c^*) + \text{dist}(c^*, q) \leq 2r$



Approximationsalgorithmen

Idee

- Wir nutzen Existenz von Lösung C^* mit Radius (Kosten) r
- Betrachte Punkt $p \in P$
- Dann gibt es Zentrum $c^* \in C^*$ mit $\text{dist}(p, c^*) \leq r$
- Nehmen wir nun p als Zentrum anstelle von c^* und verdoppeln wir den Radius, so decken wir jeden Punkt q ab, der von c^* mit Radius r abgedeckt wurde, d.h.
- für jedes $q \in P$ mit $\text{dist}(q, c^*) \leq r$ gilt $\text{dist}(p, q) \leq \text{dist}(p, c^*) + \text{dist}(c^*, q) \leq 2r$



Approximationsalgorithmen

k -Center1(P, k)

1. $C \leftarrow \emptyset; P' \leftarrow P$
2. **while** $P' \neq \emptyset$ **do**
3. Wähle beliebigen Punkt $p \in P'$
4. $C \leftarrow C \cup \{p\}$
5. Lösche alle Punkte aus P' mit Distanz höchstens $2r$ von p
6. **if** $|C| \leq k$ **then return** C
7. **else return** „Es gibt keine Menge von k Zentren mit Radius r “

Approximationsalgorithmen

k -Center1(P, k)

1. $C \leftarrow \emptyset; P' \leftarrow P$
2. **while** $P' \neq \emptyset$ **do**
3. Wähle beliebigen Punkt $p \in P'$
4. $C \leftarrow C \cup \{p\}$
5. Lösche alle Punkte aus P' mit Distanz höchstens $2r$ von p
6. **if** $|C| \leq k$ **then return** C
7. **else return** „Es gibt keine Menge von k Zentren mit Radius r “

Offensichtlich gilt

Jede Menge von k Zentren, die der Algorithmus zurückgibt, hat Kosten $\leq 2r$.

Approximationsalgorithmen

Lemma 82

Wenn Algorithmus k -Center1 mehr als k Zentren auswählt, dann gilt für jede Menge $C^* \subseteq P$ von k Zentren, dass $\text{cost}(P, C^*) > r$ ist.

Approximationsalgorithmen

Lemma 82

Wenn Algorithmus k -Center1 mehr als k Zentren auswählt, dann gilt für jede Menge $C^* \subseteq P$ von k Zentren, dass $\text{cost}(P, C^*) > r$ ist.

Beweis (durch Widerspruch)

- Annahme: Es gibt C^* mit $\text{cost}(P, C^*) \leq r$ und $|C^*| \leq k$ und $|C| > k$.

Approximationsalgorithmen

Lemma 82

Wenn Algorithmus k -Center1 mehr als k Zentren auswählt, dann gilt für jede Menge $C^* \subseteq P$ von k Zentren, dass $\text{cost}(P, C^*) > r$ ist.

Beweis (durch Widerspruch)

- Annahme: Es gibt C^* mit $\text{cost}(P, C^*) \leq r$ und $|C^*| \leq k$ und $|C| > k$.
- Sei C die Menge der Zentren, die k -Center1 auswählt

Approximationsalgorithmen

Lemma 82

Wenn Algorithmus k -Center1 mehr als k Zentren auswählt, dann gilt für jede Menge $C^* \subseteq P$ von k Zentren, dass $\text{cost}(P, C^*) > r$ ist.

Beweis (durch Widerspruch)

- Annahme: Es gibt C^* mit $\text{cost}(P, C^*) \leq r$ und $|C^*| \leq k$ und $|C| > k$.
- Sei C die Menge der Zentren, die k -Center1 auswählt
- Da $C \subseteq P$ gibt es für jedes $p \in C$ (mindestens) ein $c^* \in C^*$ mit $\text{dist}(p, c^*) \leq r$
- Wir nennen c^* nah zu p

Approximationsalgorithmen

Lemma 82

Wenn Algorithmus k -Center1 mehr als k Zentren auswählt, dann gilt für jede Menge $C^* \subseteq P$ von k Zentren, dass $\text{cost}(P, C^*) > r$ ist.

Beweis (durch Widerspruch)

- Annahme: Es gibt C^* mit $\text{cost}(P, C^*) \leq r$ und $|C^*| \leq k$ und $|C| > k$.
- Sei C die Menge der Zentren, die k -Center1 auswählt
- Da $C \subseteq P$ gibt es für jedes $p \in C$ (mindestens) ein $c^* \in C^*$ mit $\text{dist}(p, c^*) \leq r$
- Wir nennen c^* nah zu p
- **Behauptung:** Kein c^* kann nah zu zwei $p \in C$ sein (Beweis später)

Approximationsalgorithmen

Lemma 82

Wenn Algorithmus k -Center1 mehr als k Zentren auswählt, dann gilt für jede Menge $C^* \subseteq P$ von k Zentren, dass $\text{cost}(P, C^*) > r$ ist.

Beweis (durch Widerspruch)

- Annahme: Es gibt C^* mit $\text{cost}(P, C^*) \leq r$ und $|C^*| \leq k$ und $|C| > k$.
- Sei C die Menge der Zentren, die k -Center1 auswählt
- Da $C \subseteq P$ gibt es für jedes $p \in C$ (mindestens) ein $c^* \in C^*$ mit $\text{dist}(p, c^*) \leq r$
- Wir nennen c^* nah zu p
- Behauptung: Kein c^* kann nah zu zwei $p \in C$ sein (Beweis später)
- Damit folgt: $|C^*| \geq |C|$ und somit Widerspruch zu $|C^*| \leq k$ und $|C| > k$.

Approximationsalgorithmen

Lemma 82

Wenn Algorithmus k -Center1 mehr als k Zentren auswählt, dann gilt für jede Menge $C^* \subseteq P$ von k Zentren, dass $\text{cost}(P, C^*) > r$ ist.

Beweis (durch Widerspruch)

- Annahme: Es gibt C^* mit $\text{cost}(P, C^*) \leq r$ und $|C^*| \leq k$ und $|C| > k$.
- Sei C die Menge der Zentren, die k -Center1 auswählt
- Da $C \subseteq P$ gibt es für jedes $p \in C$ (mindestens) ein $c^* \in C^*$ mit $\text{dist}(p, c^*) \leq r$
- Wir nennen c^* nah zu p
- Behauptung: Kein c^* kann nah zu zwei $p \in C$ sein (Beweis später)
- Damit folgt: $|C^*| \geq |C|$ und somit Widerspruch zu $|C^*| \leq k$ und $|C| > k$.

Approximationsalgorithmen

Lemma 82

Wenn Algorithmus k -Center1 mehr als k Zentren auswählt, dann gilt für jede Menge $C^* \subseteq P$ von k Zentren, dass $\text{cost}(P, C^*) > r$ ist.

Beweis (durch Widerspruch)

- Behauptung: Kein c^* kann nah zu zwei $p \in C$ sein
- Beweis der Behauptung:

Approximationsalgorithmen

Lemma 82

Wenn Algorithmus k -Center1 mehr als k Zentren auswählt, dann gilt für jede Menge $C^* \subseteq P$ von k Zentren, dass $\text{cost}(P, C^*) > r$ ist.

Beweis (durch Widerspruch)

- Behauptung: Kein c^* kann nah zu zwei $p \in C$ sein
- Beweis der Behauptung:
- Alle Paare von Zentren p, q aus C haben Abstand $> 2r$

Approximationsalgorithmen

Lemma 82

Wenn Algorithmus k -Center1 mehr als k Zentren auswählt, dann gilt für jede Menge $C^* \subseteq P$ von k Zentren, dass $\text{cost}(P, C^*) > r$ ist.

Beweis (durch Widerspruch)

- Behauptung: Kein c^* kann nah zu zwei $p \in C$ sein
- Beweis der Behauptung:
- Alle Paare von Zentren p, q aus C haben Abstand $> 2r$
- Wäre nun für ein Zentrum c^* $\text{dist}(p, c^*) \leq r$ und $\text{dist}(q, c^*) \leq r$, so würde $\text{dist}(p, q) \leq \text{dist}(p, c^*) + \text{dist}(c^*, q) = \text{dist}(p, c^*) + \text{dist}(c^*, q) \leq 2r$ gelten.
Widerspruch!

Approximationsalgorithmen

Lemma 82

Wenn Algorithmus k -Center1 mehr als k Zentren auswählt, dann gilt für jede Menge $C^* \subseteq P$ von k Zentren, dass $\text{cost}(P, C^*) > r$ ist.

Beweis (durch Widerspruch)

- Behauptung: Kein c^* kann nah zu zwei $p \in C$ sein
- Beweis der Behauptung:
- Alle Paare von Zentren p, q aus C haben Abstand $> 2r$
- Wäre nun für ein Zentrum c^* $\text{dist}(p, c^*) \leq r$ und $\text{dist}(q, c^*) \leq r$, so würde $\text{dist}(p, q) \leq \text{dist}(p, c^*) + \text{dist}(c^*, q) = \text{dist}(p, c^*) + \text{dist}(c^*, q) \leq 2r$ gelten. Widerspruch!

Approximationsalgorithmen

Was, wenn wir r nicht kennen?

- Wir wissen nicht, welche Punkte Distanz größer als $2r$ von den bisher ausgewählten Zentren haben
- Idee: Wähle immer den am weitesten entfernten Punkt, d.h. der $\text{dist}(p, C)$ maximiert
- Gibt es einen Punkt mit $\text{dist}(p, C) > 2r$, dann ist es dieser

Approximationsalgorithmen

k-Center2(P, k)

1. Wähle beliebigen Punkt $p \in P$ und setze $C \leftarrow \{p\}$
3. **while** $|C| < k$ **do**
3. Wähle Punkt $p \in P$, der $\text{dist}(p, C)$ maximiert
4. $C \leftarrow C \cup \{p\}$
5. **return** C

Approximationsalgorithmen

Satz 83

Algorithmus k-Center2 ist ein 2-Approximationsalgorithmus für das diskrete k -Center Clustering Problem. Algorithmus k-Center2 kann mit Laufzeit $\mathbf{O}(nk)$ implementiert werden.

Beweis

- Zunächst zur Laufzeit:

Approximationsalgorithmen

Satz 83

Algorithmus k-Center2 ist ein 2-Approximationsalgorithmus für das diskrete k -Center Clustering Problem. Algorithmus k-Center2 kann mit Laufzeit $\mathbf{O}(nk)$ implementiert werden.

Beweis

- Zunächst zur Laufzeit:
- Um den Algorithmus in $\mathbf{O}(nk)$ Zeit zu implementieren, müssen wir jeden Schleifendurchlauf in $\mathbf{O}(n)$ Zeit erledigen können. Dazu speichern wir uns für jeden Punkt p den Wert $\text{dist}(p, C)$.

Approximationsalgorithmen

Satz 83

Algorithmus k-Center2 ist ein 2-Approximationsalgorithmus für das diskrete k -Center Clustering Problem. Algorithmus k-Center2 kann mit Laufzeit $\mathbf{O}(nk)$ implementiert werden.

Beweis

- Zunächst zur Laufzeit:
- Um den Algorithmus in $\mathbf{O}(nk)$ Zeit zu implementieren, müssen wir jeden Schleifendurchlauf in $\mathbf{O}(n)$ Zeit erledigen können. Dazu speichern wir uns für jeden Punkt p den Wert $\text{dist}(p, C)$.
- Wird nun ein neues Zentrum c in C eingefügt, so müssen wir nur für jeden Punkt überprüfen, ob $\text{dist}(p, c) < \text{dist}(p, C)$ ist und ggf. $\text{dist}(p, C)$ aktualisieren. Dies geht insgesamt in $\mathbf{O}(n)$ Zeit.

Approximationsalgorithmen

Satz 83

Algorithmus k-Center2 ist ein 2-Approximationsalgorithmus für das diskrete k -Center Clustering Problem. Algorithmus k-Center2 kann mit Laufzeit $\mathbf{O}(nk)$ implementiert werden.

Beweis

- Zunächst zur Laufzeit:
- Um den Algorithmus in $\mathbf{O}(nk)$ Zeit zu implementieren, müssen wir jeden Schleifendurchlauf in $\mathbf{O}(n)$ Zeit erledigen können. Dazu speichern wir uns für jeden Punkt p den Wert $\text{dist}(p, C)$.
- Wird nun ein neues Zentrum c in C eingefügt, so müssen wir nur für jeden Punkt überprüfen, ob $\text{dist}(p, c) < \text{dist}(p, C)$ ist und ggf. $\text{dist}(p, C)$ aktualisieren. Dies geht insgesamt in $\mathbf{O}(n)$ Zeit.

Approximationsalgorithmen

Satz 83

Algorithmus k-Center2 ist ein 2-Approximationsalgorithmus für das diskrete k -Center Clustering Problem. Algorithmus k-Center2 kann mit Laufzeit $\mathbf{O}(nk)$ implementiert werden.

Beweis

- Wir bezeichnen nun mit r die Kosten einer optimalen Lösung C^* .
Annahme: Algorithmus liefert Menge C mit Kosten $> 2r$.

Approximationsalgorithmen

Satz 83

Algorithmus k-Center2 ist ein 2-Approximationsalgorithmus für das diskrete k -Center Clustering Problem. Algorithmus k-Center2 kann mit Laufzeit $\mathbf{O}(nk)$ implementiert werden.

Beweis

- Wir bezeichnen nun mit r die Kosten einer optimalen Lösung C^* .
Annahme: Algorithmus liefert Menge C mit Kosten $> 2r$.
- Dann gibt es einen Punkt p mit $\text{dist}(p, C) > 2r$.

Approximationsalgorithmen

Satz 83

Algorithmus k-Center2 ist ein 2-Approximationsalgorithmus für das diskrete k -Center Clustering Problem. Algorithmus k-Center2 kann mit Laufzeit $\mathbf{O}(nk)$ implementiert werden.

Beweis

- Wir bezeichnen nun mit r die Kosten einer optimalen Lösung C^* .
Annahme: Algorithmus liefert Menge C mit Kosten $> 2r$.
- Dann gibt es einen Punkt p mit $\text{dist}(p, C) > 2r$.
- Da der Algorithmus immer den Punkt auswählt, der maximalen Abstand zu den bisher ausgewählten Zentren hat, haben alle ausgewählten Zentren Abstand $> 2r$ zur den bisher ausgewählten Zentren

Approximationsalgorithmen

Satz 83

Algorithmus k-Center2 ist ein 2-Approximationsalgorithmus für das diskrete k -Center Clustering Problem. Algorithmus k-Center2 kann mit Laufzeit $\mathbf{O}(nk)$ implementiert werden.

Beweis

- Wir bezeichnen nun mit r die Kosten einer optimalen Lösung C^* .
Annahme: Algorithmus liefert Menge C mit Kosten $> 2r$.
- Dann gibt es einen Punkt p mit $\text{dist}(p, C) > 2r$.
- Da der Algorithmus immer den Punkt auswählt, der maximalen Abstand zu den bisher ausgewählten Zentren hat, haben alle ausgewählten Zentren Abstand $> 2r$ zur den bisher ausgewählten Zentren
- Somit würde Algorithmus k-Center1 auf dieser Eingabe mehr als k Zentren zurückgeben. Damit gilt nach Lemma 82 $\text{cost}(P, C^*) > r$. Widerspruch!

Approximationsalgorithmen

Satz 83

Algorithmus k-Center2 ist ein 2-Approximationsalgorithmus für das diskrete k -Center Clustering Problem. Algorithmus k-Center2 kann mit Laufzeit $\mathbf{O}(nk)$ implementiert werden.

Beweis

- Wir bezeichnen nun mit r die Kosten einer optimalen Lösung C^* .
Annahme: Algorithmus liefert Menge C mit Kosten $> 2r$.
- Dann gibt es einen Punkt p mit $\text{dist}(p, C) > 2r$.
- Da der Algorithmus immer den Punkt auswählt, der maximalen Abstand zu den bisher ausgewählten Zentren hat, haben alle ausgewählten Zentren Abstand $> 2r$ zur den bisher ausgewählten Zentren
- Somit würde Algorithmus k-Center1 auf dieser Eingabe mehr als k Zentren zurückgeben. Damit gilt nach Lemma 82 $\text{cost}(P, C^*) > r$. Widerspruch!

Approximationsalgorithmen

Zusammenfassung & Kommentare

- Man kann viele Probleme approximativ schneller lösen als exakt (für die drei Beispiele sind keine Algorithmen mit Laufzeit $\mathbf{O}(n^c)$ für eine Konstante c bekannt)
- Gierige Algorithmen sind häufig Approximationsalgorithmen