

# Grundzüge der Informatik 1

Vorlesung 25



# Approximationsalgorithmen

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

- Die Aufgabenstellung vereinfachen!

## Approximationsalgorithmen

- Löst 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

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

# 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  $|U|$

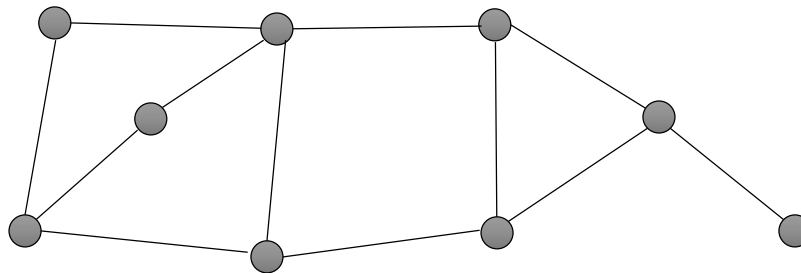
# Approximationsalgorithmen

## 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  $|U|$



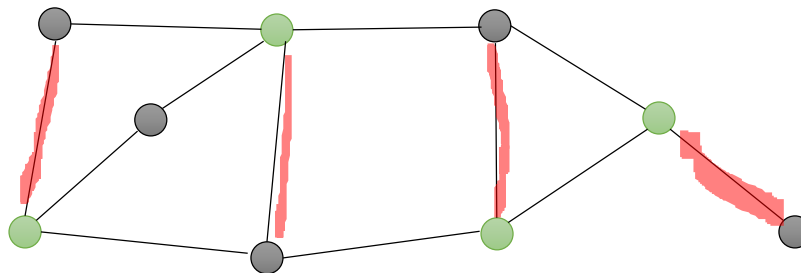
# Approximationsalgorithmen

## 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.  $U = \emptyset$
2. **while**  $E \neq \emptyset$  **do**
3.     wähle einen Knoten  $v$  mit maximalem Knotengrad
4.     Entferne alle an  $v$  anliegenden Kanten aus  $E$
5.      $U = U \cup \{v\}$

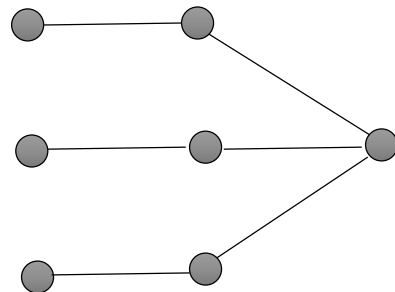
## Erste Frage

- Ist der Algorithmus optimal?

# Approximationsalgorithmen

## Erste Frage

- Ist der Algorithmus optimal?
- Nein! Gegenbeispiel:

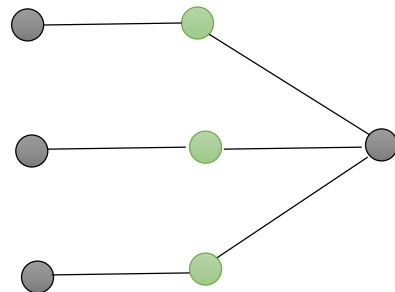




# Approximationsalgorithmen

## Erste Frage

- Ist der Algorithmus optimal?
- Nein! Gegenbeispiel:

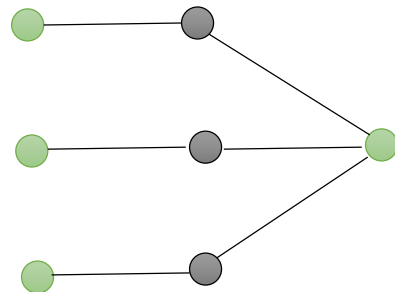


- Optimale Lösung hat Größe 3

# 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



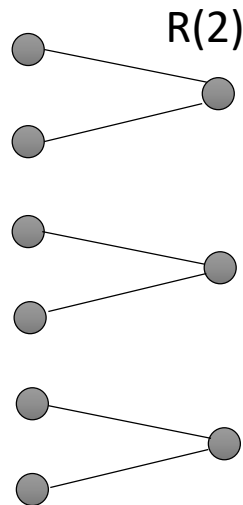
Die Menge  $L$



# 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

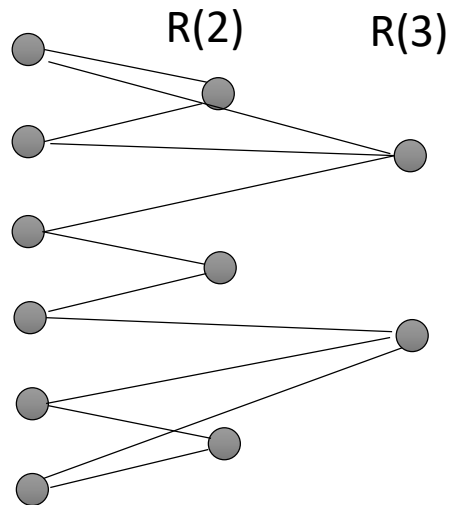


Die Menge  $L$

# 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

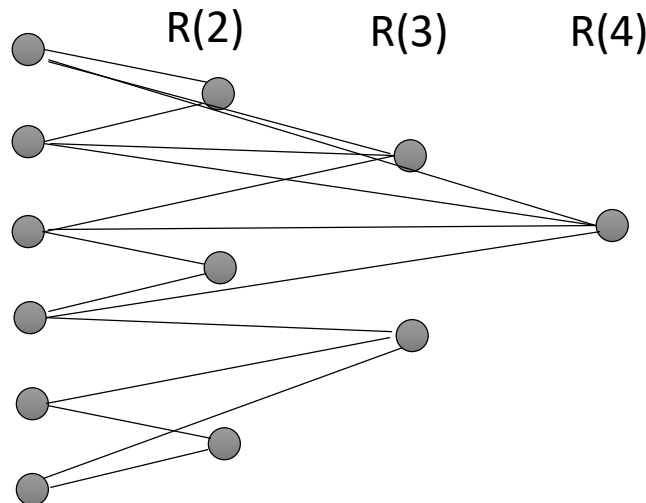


Die Menge L

# 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

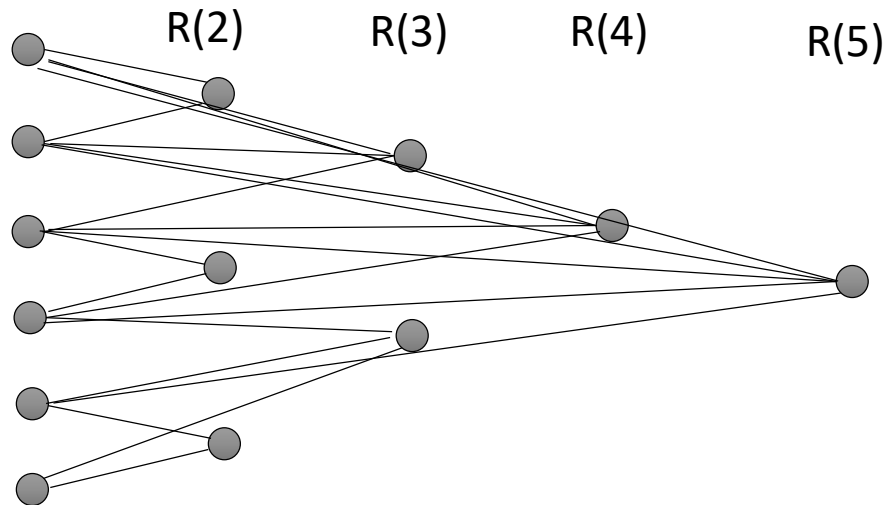


Die Menge L

# 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

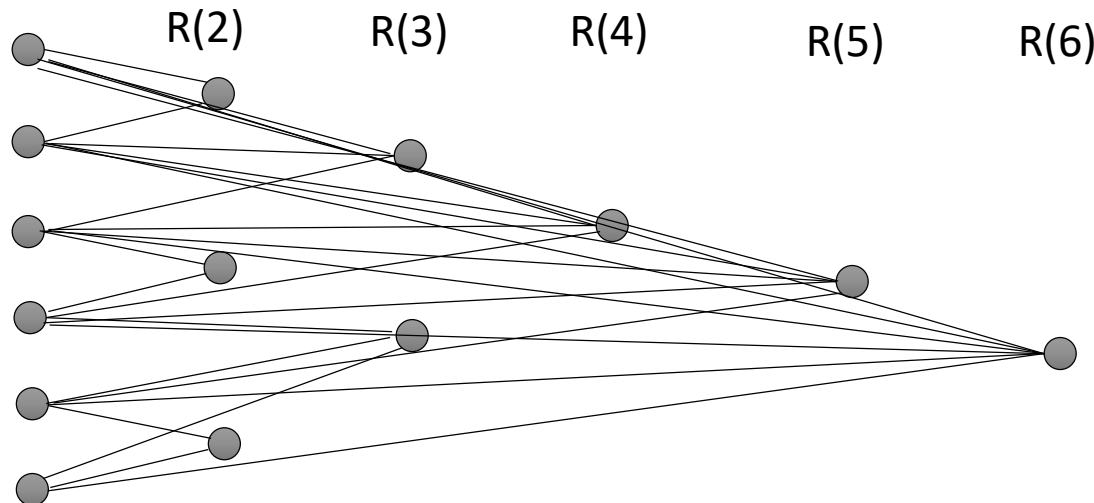


Die Menge  $L$

# Approximationsalgorithmen

## Die Konstruktion

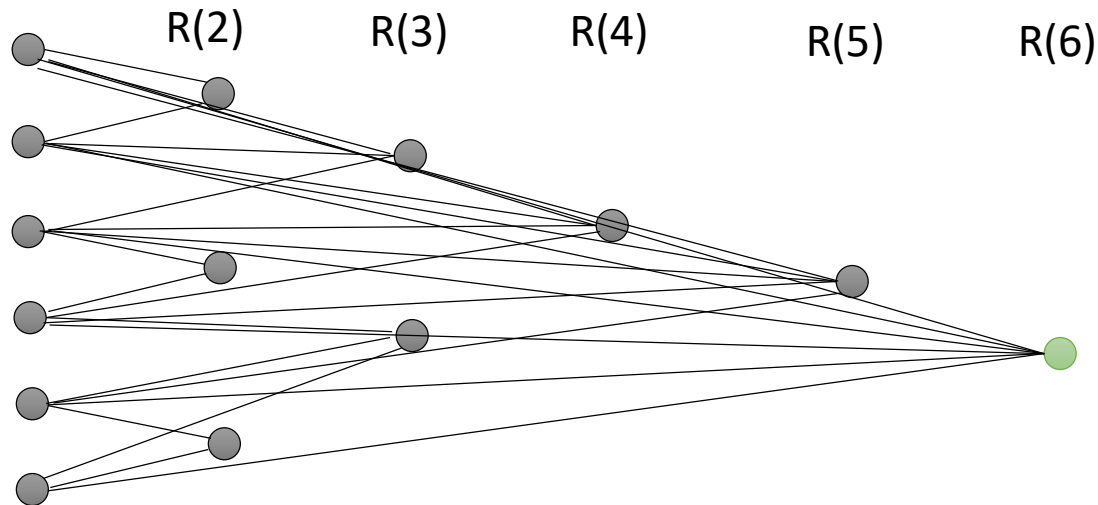
- Sei  $L=\{1,\dots,r\}$  eine Menge mit  $r$  Knoten
- Im  $i$ -ten Schritt wählen wir Menge  $R(i)$  mit  $\lfloor |L|/i \rfloor$  Knoten
- Der  $j$ -te Knoten aus  $R(i)$  wird mit Knoten  $i(j-1)+1, \dots, ij$  verbunden



Die Menge  $L$

# Approximationsalgorithmen

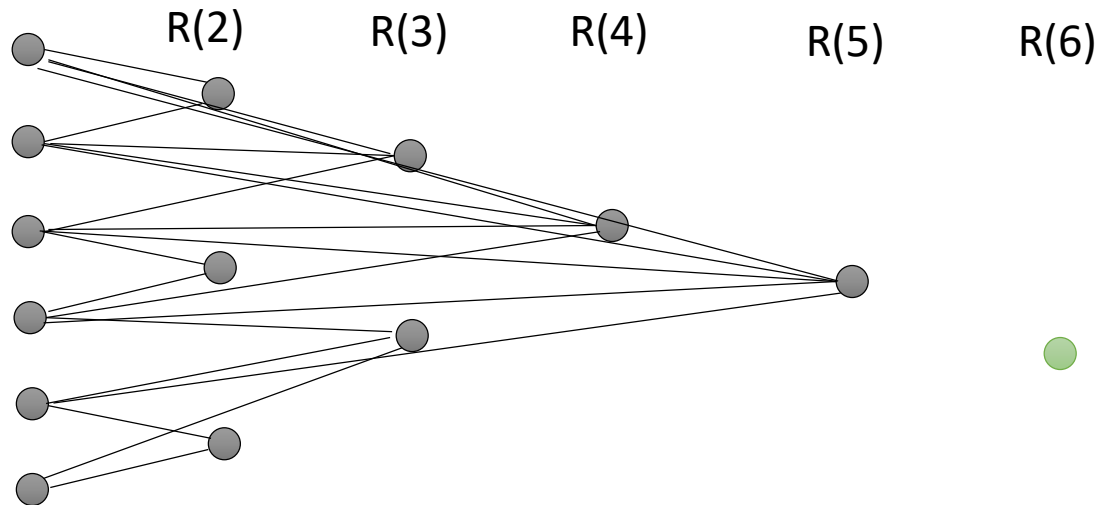
## Was macht der Algorithmus?



Die Menge  $L$

# Approximationsalgorithmen

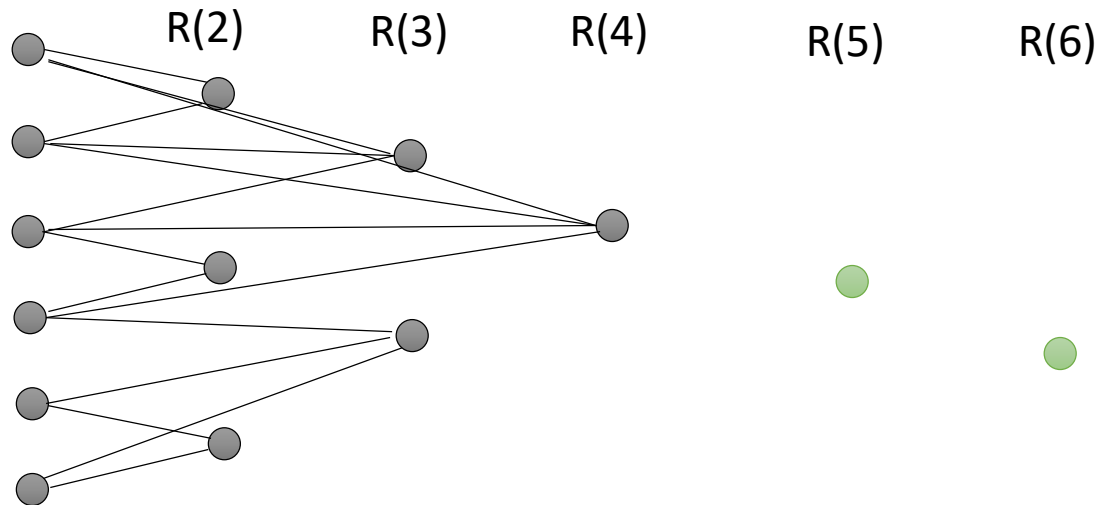
## Was macht der Algorithmus?



Die Menge L

# Approximationsalgorithmen

## Was macht der Algorithmus?

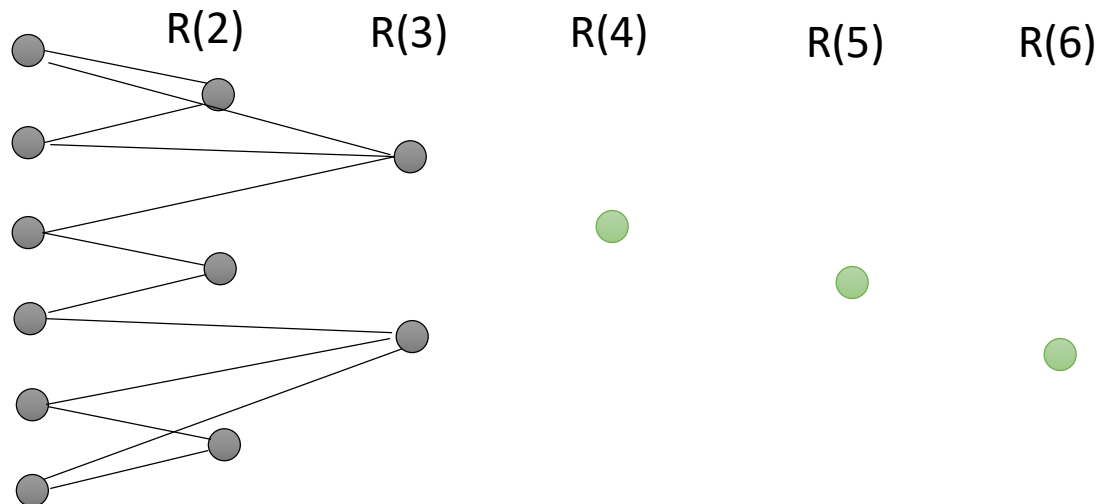


Die Menge L



# Approximationsalgorithmen

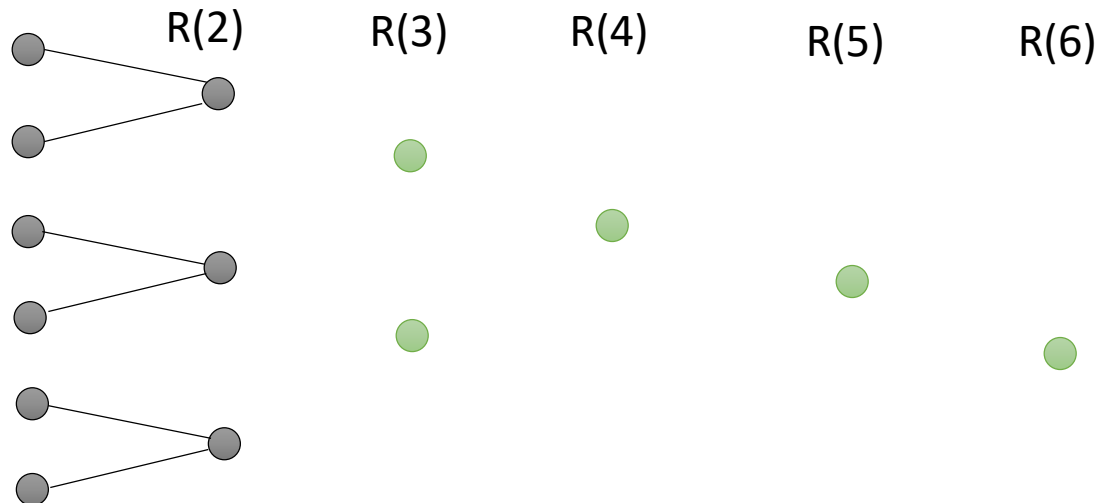
## Was macht der Algorithmus?



Die Menge  $L$

# Approximationsalgorithmen

## Was macht der Algorithmus?

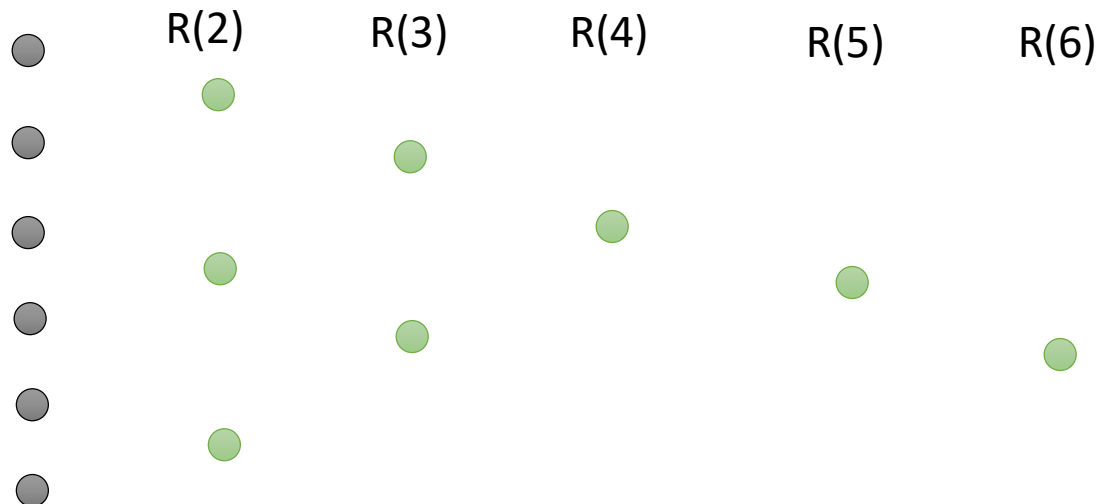


Die Menge  $L$

# 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  $O(\log n)$  ist)

# 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:  $O(|V|+|E|)$



# Approximationsalgorithmen

## Satz

- ApproxVertexCover2 ist ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

# Approximationsalgorithmen

## Satz

- ApproxVertexCover2 ist ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

## Beweis

# Approximationsalgorithmen

## Satz

- ApproxVertexCover2 ist ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

## Beweis

- Die von ApproxVertexCover2 berechnete Menge  $C$  ist eine Knotenüberdeckung, da die while-Schleife solange durchlaufen wird, bis alle Kanten überdeckt sind

# Approximationsalgorithmen

## Satz

- `ApproxVertexCover2` ist ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

## Beweis

- Die von `ApproxVertexCover2` 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

- `ApproxVertexCover2` ist ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

## Beweis

- Die von `ApproxVertexCover2` 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

- `ApproxVertexCover2` ist ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

## Beweis

- Die von `ApproxVertexCover2` 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

- ApproxVertexCover2 ist ein 2-Approximationsalgorithmus für das Knotenüberdeckungsproblem.

## Beweis

- Die von ApproxVertexCover2 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

- ApproxVertexCover2 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

- ApproxVertexCover2 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

- ApproxVertexCover2 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

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

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

# 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

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

Dreiecksungleichung

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

# Approximationsalgorithmen

## ApproxTSP( $G, w$ )

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

## 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

# Approximationsalgorithmen

## ApproxTSP( $G, w$ )

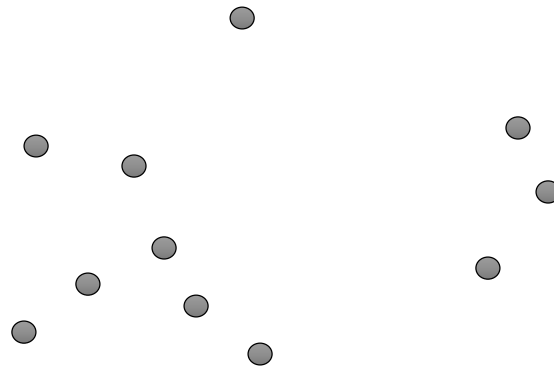
1. Berechne minimalen Spannbaum  $T$  von  $G$
2. Sei  $\pi$  die Liste der Knoten von  $G$  in der Reihenfolge eines Preorder-Tree-Walk von einem beliebigen Knoten  $v$
3. **return**  $L$

## Laufzeit

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

# Approximationsalgorithmen

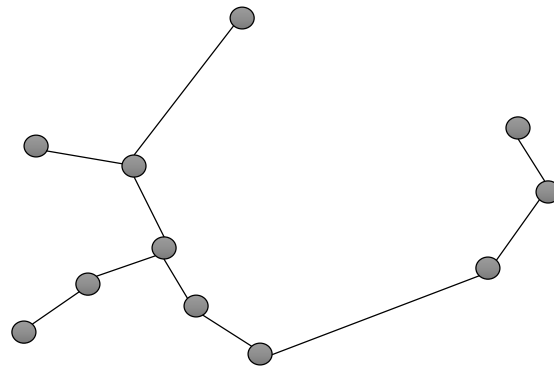
*Beispiel (euklidische Entfernung)*





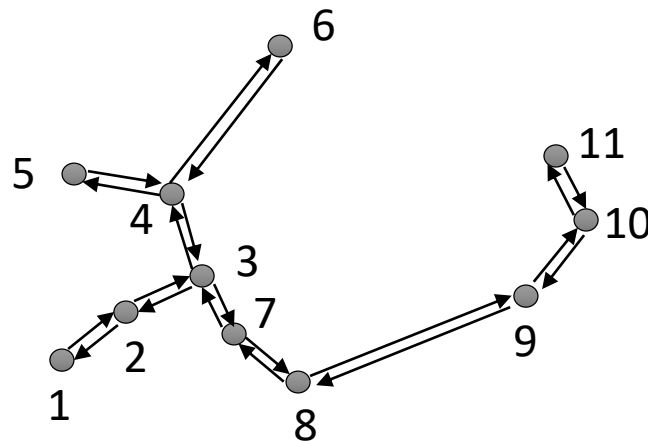
# Approximationsalgorithmen

*Beispiel (euklidische Entfernung)*



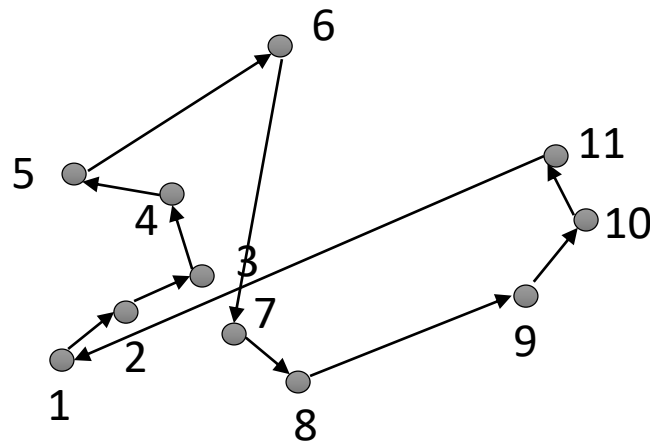
# Approximationsalgorithmen

*Beispiel (euklidische Entfernung)*



# Approximationsalgorithmen

*Beispiel (euklidische Entfernung)*



# Approximationsalgorithmen

## Satz

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

# Approximationsalgorithmen

## Satz

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

## Beweis

# Approximationsalgorithmen

## Satz

- 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

- 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

- 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

- 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

# Approximationsalgorithmen

## Beweis

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

# 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  $w(F)=2w(T)$ , 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  $w(F)=2w(T)$ , 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  $w(F)=2w(T)$ , 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  $w(F)=2w(T)$ , 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  $w(F)=2w(T)$ , 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

# Approximationsalgorithmen

## Beweis

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

# Approximationsalgorithmen

## Beweis

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

# Approximationsalgorithmen

## Beweis

- Auf diese Weise können wir alle Besuche außer dem 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  $w(H) \leq w(F)$

# Approximationsalgorithmen

## Beweis

- Auf diese Weise können wir alle Besuche außer dem 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 dem 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

## Last Balanzierung

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

# Approximationsalgorithmen

## GreedyLoadBalancing

1. Setze  $T(i)=0$  und  $A(i)=\emptyset$  für alle Maschinen  $i \in \{1, \dots, m\}$
2. **for**  $j=1$  **to**  $n$  **do**
3.   Sei  $i$  eine Maschine mit  $T(i) = \min T(k)$
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

- 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


- 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

- 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

- 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

- 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

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

## Beweis

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

# Approximationsalgorithmen

## Beobachtung

- Für jede Probleminstanz 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

- Algorithmus GreedyLoadBalancing ist ein 2-Approximationsalgorithmus für das Lastbalanzierungsproblem.

## Beweis

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



# Approximationsalgorithmen

## Satz

- Algorithmus GreedyLoadBalancing ist ein 2-Approximationsalgorithmus für das Lastbalanzierungsproblem.

## 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

- Algorithmus GreedyLoadBalancing ist ein 2-Approximationsalgorithmus für das Lastbalanzierungsproblem.

## 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

- Algorithmus GreedyLoadBalancing ist ein 2-Approximationsalgorithmus für das Lastbalanzierungsproblem.

## 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:

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

# Approximationsalgorithmen

## Satz

- Algorithmus GreedyLoadBalancing ist ein 2-Approximationsalgorithmus für das Lastbalanzierungsproblem.

## Beweis

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

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

# Approximationsalgorithmen

## Zusammenfassung & Kommentare

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