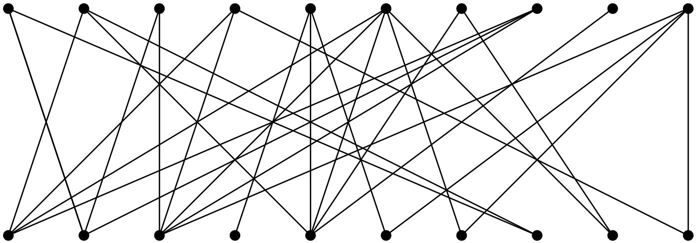
Bipartites Matching

Gegeben: Ein bipartiter, ungerichteter Graph (V_1, V_2, E).

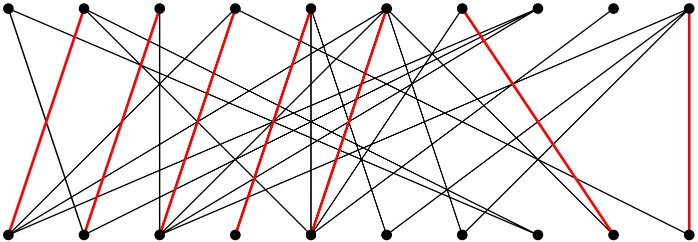
Gesucht:
Ein Matching (Paarung) maximaler Kardinalität.

Ein Matching ist eine Menge paarweise nicht inzidenter Kanten, also $M \subseteq E$ mit $m_1, m_2 \in M,\ m_1 \neq m_2 \Rightarrow m_1 \cap m_2 = \emptyset$.

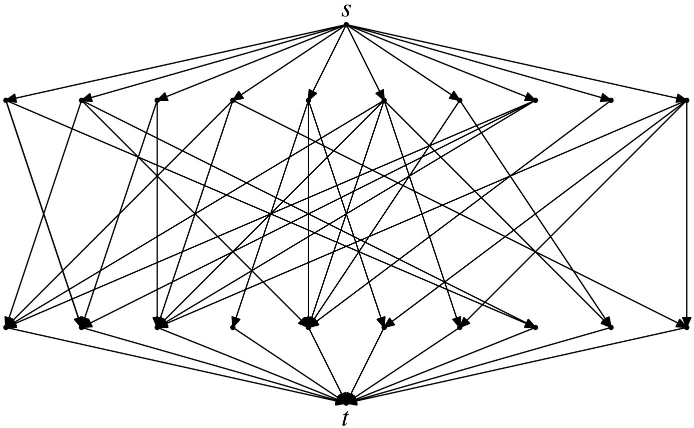
Beispiel



Beispiel

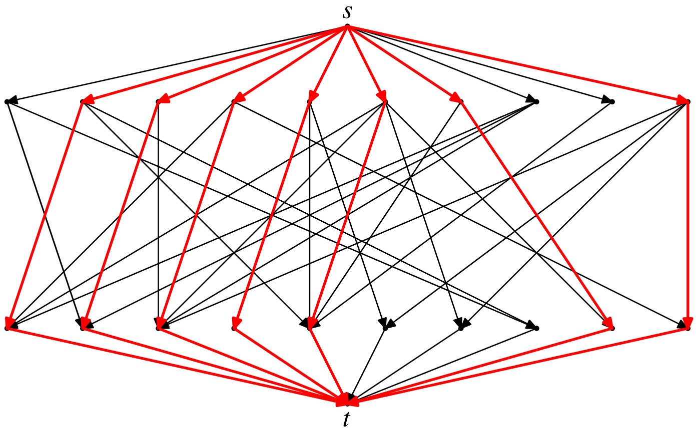


Lösung als Flußproblem

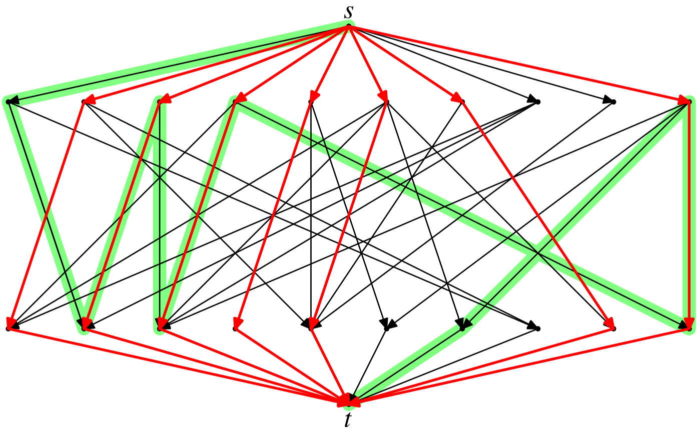


Alle Kapazitäten sind 1.
Maximaler ganzzahliger Fluß entspricht einem Matching maximaler Kardinalität.

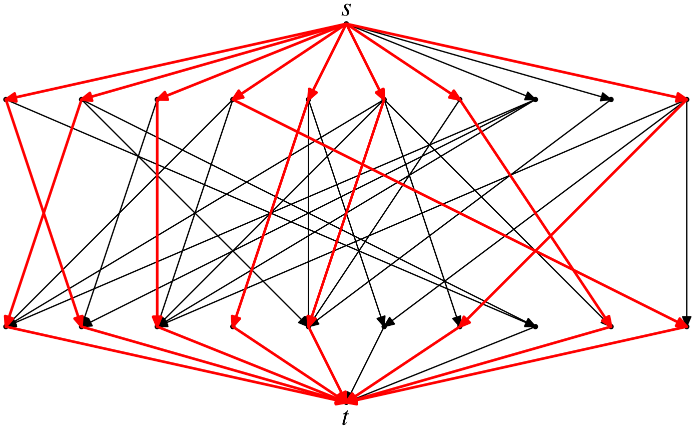
Gibt es einen augmentierenden Pfad?



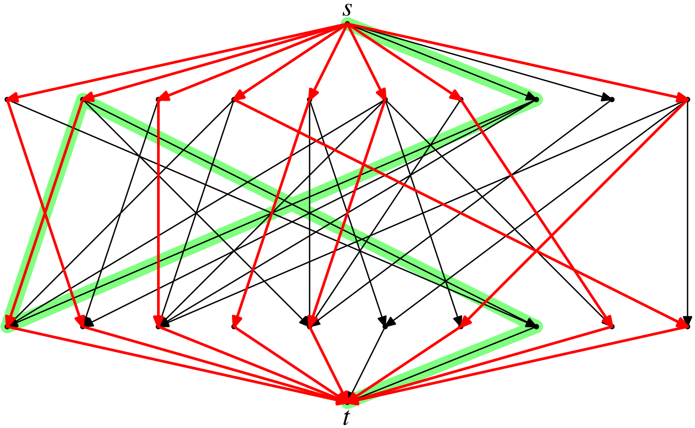
Gibt es einen augmentierenden Pfad? – Ja



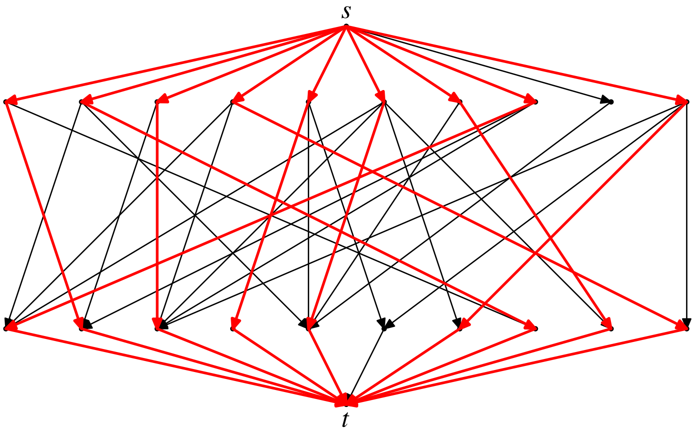
⇒ Neues Matching



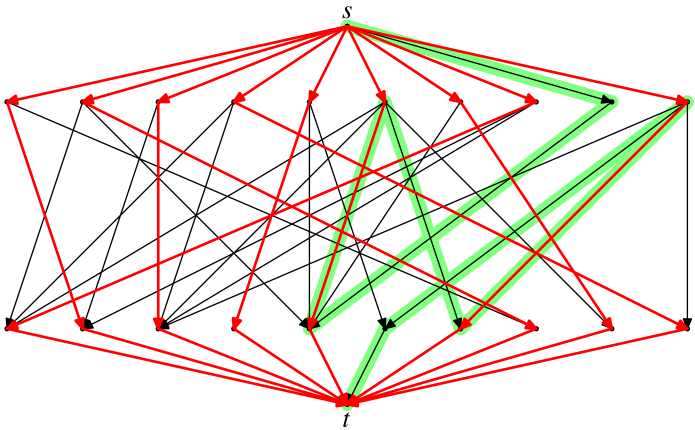
Es gibt wieder einen augmentierenden Pfad



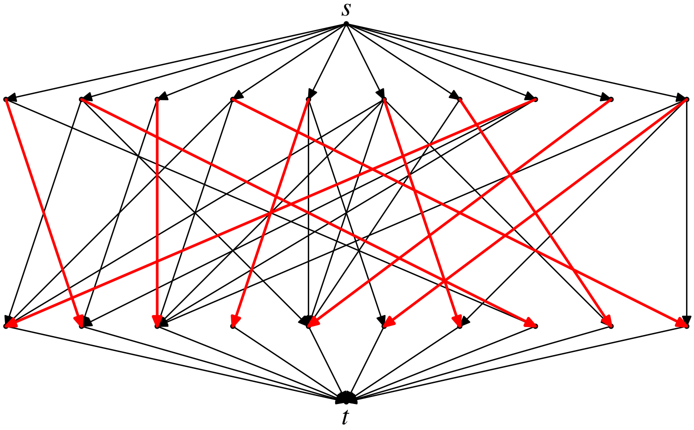
$\Rightarrow$ Neues Matching



Es gibt wieder einen augmentierenden Pfad



Ergebnis: Perfektes Matching



Laufzeit

Finden eines Matchings maximaler Kardinalität dauert nur $O(|E| \cdot \min\{|V_1|, |V_2|\})$ mit der Ford–Fulkerson–Methode.

- Der Fluß ist höchstens $f^* = \min\{|V_1|, |V_2|\}$.
- Finden eines Pfads dauert $O(|E|)$.

Variante der Ford–Fulkerson–Methode

Dieser Algorithmus funktioniert bei ganzzahligen Kapazitäten:

Algorithmus

$K \leftarrow 2^{\lfloor \log_2(\max\{\, c(u,v) \mid (u,v) \in E \,\}) \rfloor}$
$f \leftarrow 0$
while $K \geq 1$ **do**
  while es gibt einen augmentierenden
    Pfad p mit $c_f(p) \geq K$ **do**
    augmentiere f entlang p
  $K \leftarrow K/2$
return f

Die Laufzeit ist $O(|E|^2 \log K)$.
$\Rightarrow$ vergleiche mit $O(|E| f^*)$.

Variante der Ford–Fulkerson–Methode

Theorem

Die Laufzeit dieser Variante beträgt $O(|E|^2 \log C)$, *wobei* $C = \max\{\, c(u,v) \mid (u,v) \in E \,\}$.

Beweis.

- Die Restkapazität eines minimalen Schnitts ist stets höchstens $2K|E|$.
- Für jedes K gibt es nur $|E|$ Augmentierungen
- Es gibt $O(\log C)$ verschiedene K

$\square$

Der Edmonds–Karp–Algorithmus

Die Ford–Fulkerson–Methode kann sehr langsam sein, auch wenn das Netzwerk klein ist.

Der Edmonds–Karp–Algorithmus ist polynomiell in der Größe des Netzwerks.

Algorithmus

Initialisiere Fluß f zu 0
while es gibt einen augmentierenden Pfad **do**
  finde einen kürzesten augmentierenden Pfad p
  augmentiere f entlang p
return f

Unterschied: Es wird ein kürzester Pfad gewählt

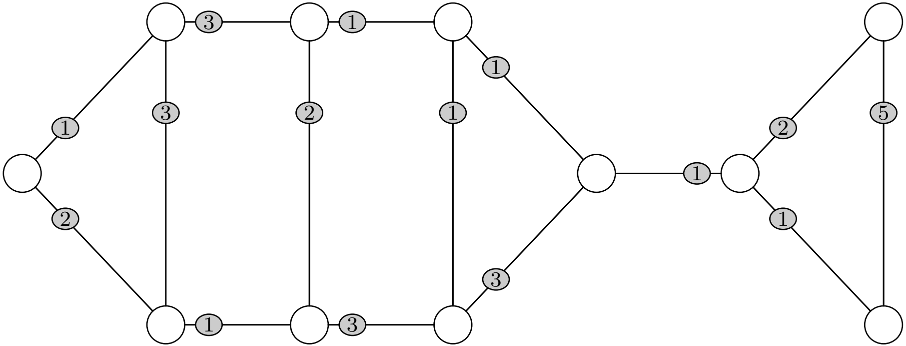
Der Edmonds–Karp–Algorithmus

Algorithmus

for each edge $(u, v) \in E$ **do**
  $f(u, v) \leftarrow 0$
  $f(v, u) \leftarrow 0$
while there exists a path from s to t in G_f **do**
  $p \leftarrow$ a shortest path from s to t in G_f
  $c_f(p) \leftarrow \min\{\, c_f(u, v) \mid (u, v) \text{ is in } p\,\}$
  for each edge (u, v) in p **do**
    $f(u, v) \leftarrow f(u, v) + c_f(p)$
    $f(v, u) \leftarrow -f(u, v)$
return f

Übersicht

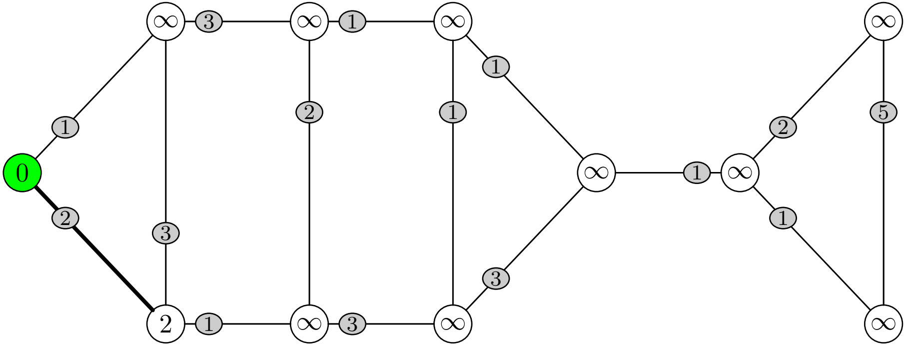
Minimale Spannbäume



Eingabe: Ungerichteter Graph mit Kantengewichten

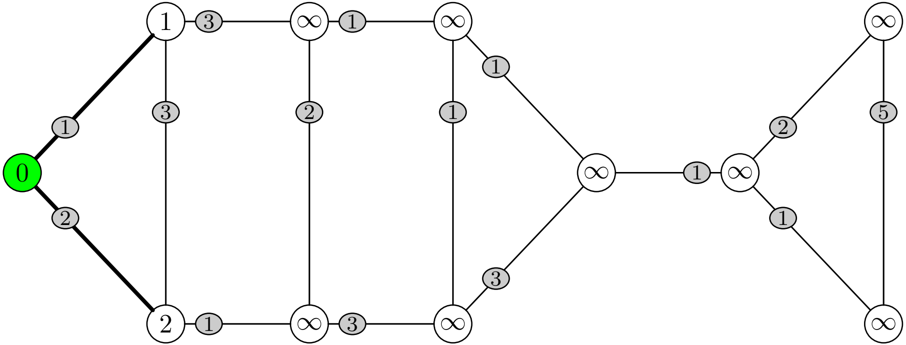
Ausgabe: Ein Baum, der alle Knoten enthält und minimales Kantengewicht hat

Der Algorithmus von Prim – Beispiel



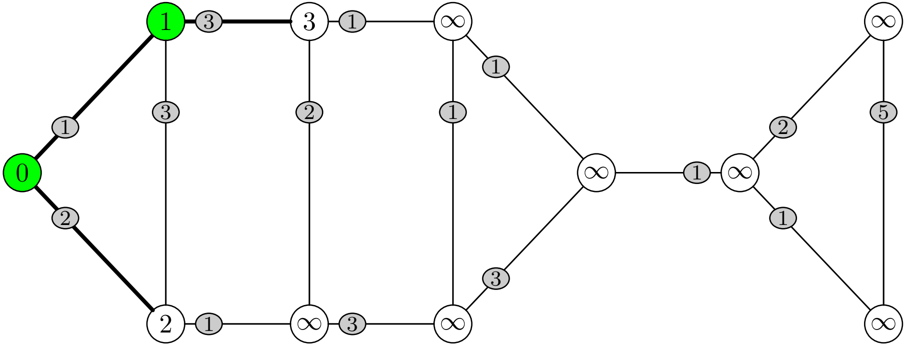
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



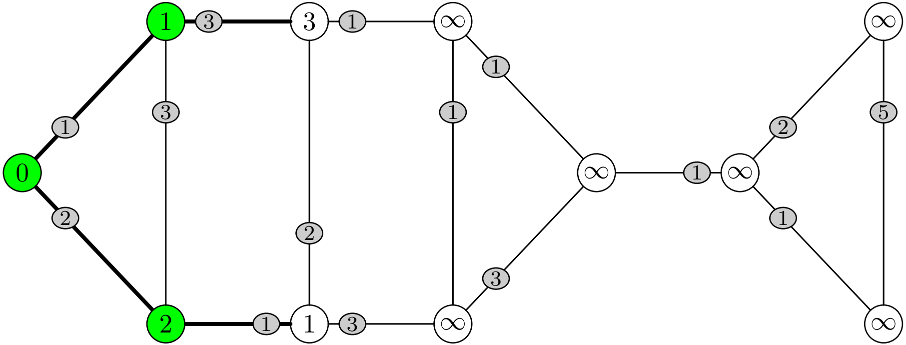
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



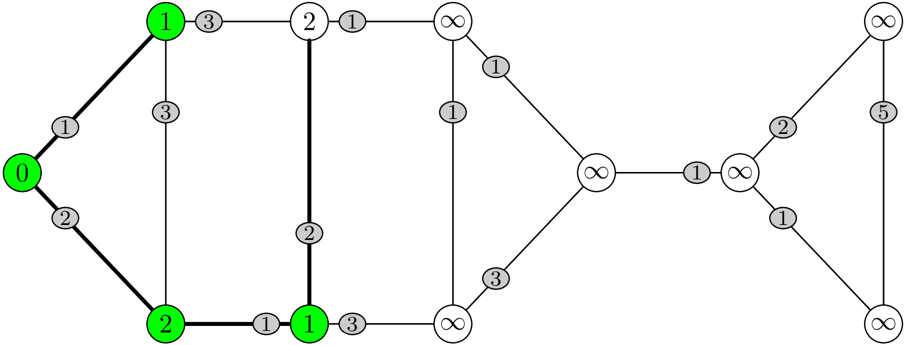
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



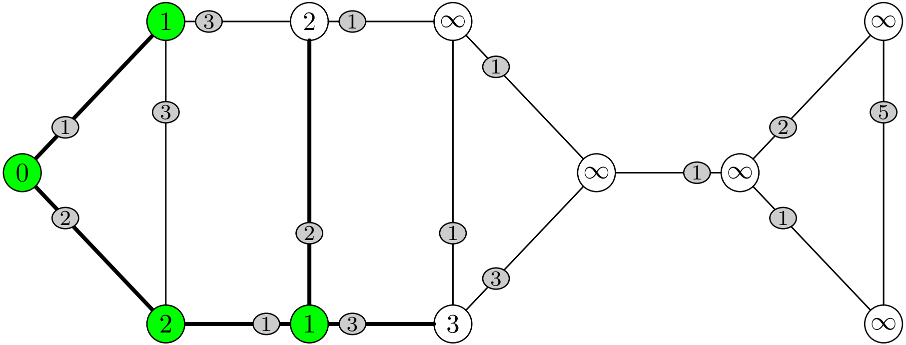
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



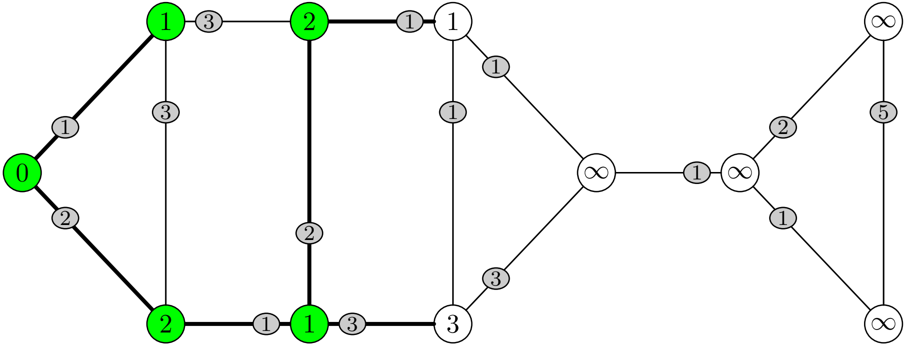
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



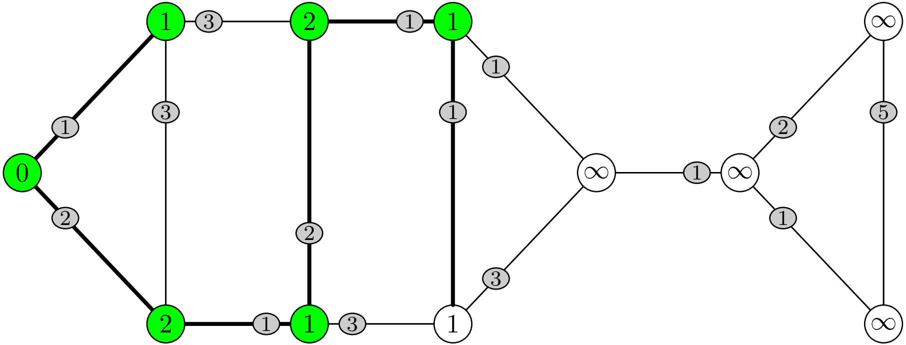
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



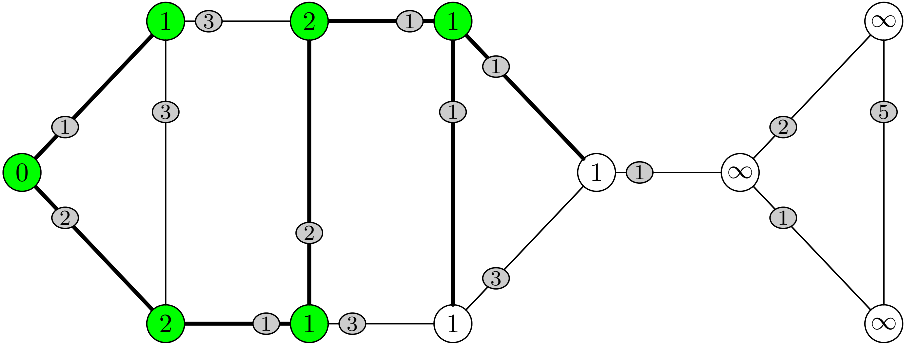
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



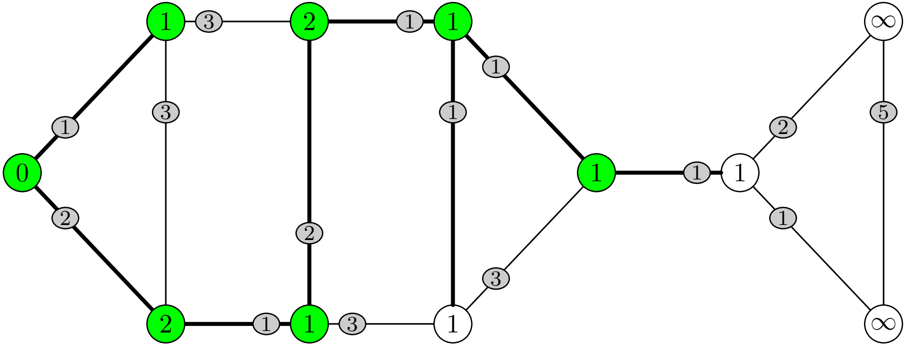
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



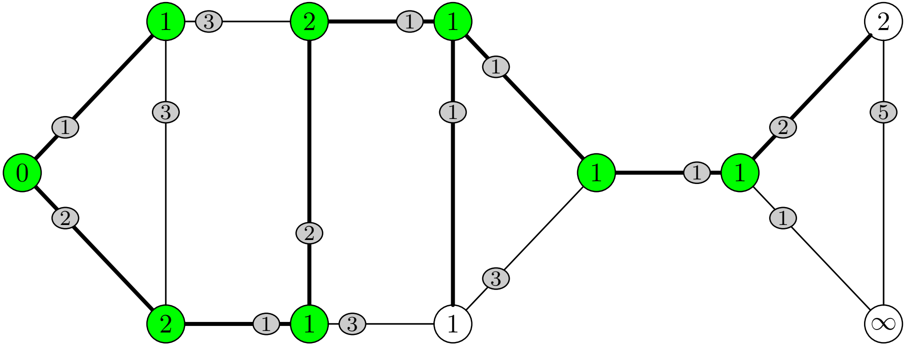
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



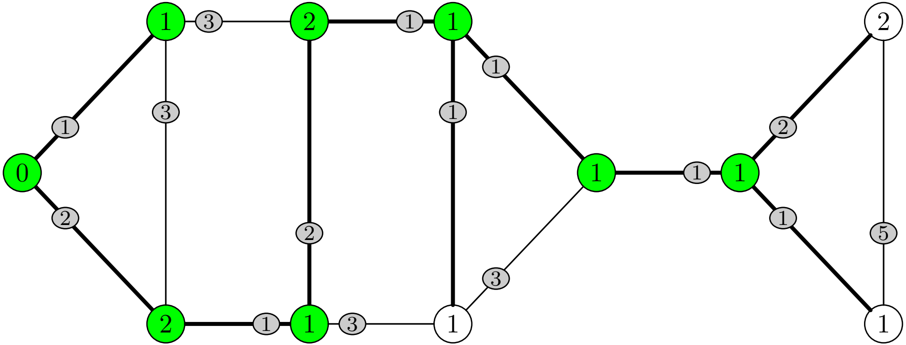
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Java

```
public static<V extends Comparable<V>> Map<V, V>
Prim(Graph<V> G, V s, Map<Edge<V>, Double> length) {
  Map<V, Double> dist = new HashMap<V, Double>();
  Map<V, V> pred = new HashMap<V, V>();
  Map<V, Integer> color = new HashMap<V, Integer>();
  PriorityQueue<V, Double> queue = new SplayPriorityQueue<V, Double>();
  for(V u : G.allNodes()) { dist.put(u, Double.MAX_VALUE); color.put(u, WHITE); }
  dist.put(s, 0.0); color.put(s, GRAY); queue.insert(s, 0.0);
  while(!queue.isEmpty()) { V u = queue.extractMin();
    for(V v : G.neighbors(u)) { Double l = length.get(G.edge(u, v));
      if(color.get(v) == WHITE) { queue.insert(v, l); dist.put(v, l);
        color.put(v, GRAY); pred.put(v, u);
      } else if(color.get(v) == GRAY && l < dist.get(v)) {
        queue.decreaseKey(v, l); dist.put(v, l); pred.put(v, u);
      }
    }
    color.put(u, BLACK);
  }
  return pred;
}
```

Der Algorithmus von Prim – Laufzeit

Laufzeit für einen Graphen $G = (V, E)$.

- Anzahl von extract_min: $|V|$
- Anzahl von decrease_key: $|E|$

Laufzeit ist $O((|V| + |E|) \log |V|)$, falls wir einen Heap als Prioritätswarteschlange verwenden.

Laufzeit ist $O(|V| \log |V| + |E|)$, falls wir stattdessen einen Fibonacci-Heap nehmen.

Korrektheit des Algorithmus folgt später.

Der Algorithmus von Prim – Laufzeit

Laufzeit für einen Graphen $G = (V, E)$.

- Anzahl von extract_min: $|V|$
- Anzahl von decrease_key: $|E|$

Laufzeit ist $O((|V| + |E|) \log |V|)$, falls wir einen Heap als Prioritätswarteschlange verwenden.

Laufzeit ist $O(|V| \log |V| + |E|)$, falls wir stattdessen einen Fibonacci-Heap nehmen.

Korrektheit des Algorithmus folgt später.

Der Algorithmus von Prim – Laufzeit

Laufzeit für einen Graphen $G = (V, E)$.

- Anzahl von extract_min: $|V|$
- Anzahl von decrease_key: $|E|$

Laufzeit ist $O((|V| + |E|) \log |V|)$, falls wir einen Heap als Prioritätswarteschlange verwenden.

Laufzeit ist $O(|V| \log |V| + |E|)$, falls wir stattdessen einen Fibonacci-Heap nehmen.

Korrektheit des Algorithmus folgt später.

Der Algorithmus von Prim – Laufzeit

Laufzeit für einen Graphen $G = (V, E)$.

- Anzahl von extract_min: $|V|$
- Anzahl von decrease_key: $|E|$

Laufzeit ist $O((|V| + |E|) \log |V|)$, falls wir einen Heap als Prioritätswarteschlange verwenden.

Laufzeit ist $O(|V| \log |V| + |E|)$, falls wir stattdessen einen Fibonacci-Heap nehmen.

Korrektheit des Algorithmus folgt später.

Greedy Algorithmen – Münzen wechseln

Es gibt diese acht Euromünzen:

Was ist die minimale Zahl von Münzen um 3.34 Euro zu zahlen?

Antwort:
Wir brauchen sechs Münzen:

Es ist unmöglich, weniger Münzen zu verwenden.

Greedy Algorithmen – Münzen wechseln

Es gibt diese acht Euromünzen:

Was ist die minimale Zahl von Münzen um 3.34 Euro zu zahlen?

Antwort:
Wir brauchen sechs Münzen:

Es ist unmöglich, weniger Münzen zu verwenden.

Münzwechsel – Ein Greedy-Algorithmus

Wir wollen den Betrag n wechseln:

Algorithmus

r := n;
while r > 0 **do**
  Choose biggest coin c with value(c) ≤ r;
  S := S ∪ { c} ;
  r := r − value(c)
od;
return S

Korrektheit

Lemma A

Sei C eine Münze und v ein Betrag, der mindestens so groß ist wie der Wert von C. Dann ist es suboptimal, v mit Münzen kleiner als C auszudrücken.

Beweise das Lemma für jede Münze von der kleinsten bis zur größten.

Nimm als Beispiel.

Sei v mindestens 1 EUR. Nehmen wir an, v kann mit genau k und kleineren Münzen für die verbleibenden $v - 50k$ Cents optimal bezahlt werden.

Da diese $v - 50k$ Cents optimal ausgezahlt werden, muß $v - 50k < 50$ und somit auch $100 \le v < 50(k+1)$ gelten. Es folgt $k \ge 2$, ein Widerspruch zur Optimalität.

Korrektheit

Lemma A

Sei C eine Münze und v ein Betrag, der mindestens so groß ist wie der Wert von C. Dann ist es suboptimal, v mit Münzen kleiner als C auszudrücken.

Beweise das Lemma für jede Münze von der kleinsten bis zur größten.

Nimm  als Beispiel.

Sei v mindestens 1 EUR. Nehmen wir an, v kann mit genau k  und kleineren Münzen für die verbleibenden $v - 50k$ Cents optimal bezahlt werden.

Da diese $v - 50k$ Cents optimal ausgezahlt werden, muß $v - 50k < 50$ und somit auch $100 \leq v < 50(k+1)$ gelten. Es folgt $k \geq 2$, ein Widerspruch zur Optimalität.

Korrektheit

Theorem

Der Greedy-Algorithmus für den Münzwechsel ist optimal.

Beweis.

Nimm an, $C_1, C_2, \ldots, C_n$ ist eine Greedy-Lösung (mit $C_i \geq C_{i+1}$).
Zeige mit Induktion über k, daß eine optimale Lösung mit $C_1, C_2, \ldots, C_k$ beginnt.

Falls dem nicht so wäre, gäbe es eine optimale Lösung $C_1, C_2, \ldots, C_{k-1}, C'_k, \ldots, C'_m$ wobei $C_k > C'_i$ für $i = k, \ldots, m$.
Da $C'_k + \cdots + C'_m \geq C_k$ ist dies ein Widerspruch zu Lemma A. $\square$

Korrektheit

Theorem

Der Greedy-Algorithmus für den Münzwechsel ist optimal.

Beweis.

Nimm an, $C_1, C_2, \dots, C_n$ ist eine Greedy-Lösung (mit $C_i \geq C_{i+1}$).
Zeige mit Induktion über k, daß eine optimale Lösung mit $C_1, C_2, \dots, C_k$ beginnt.

Falls dem nicht so wäre, gäbe es eine optimale Lösung $C_1, C_2, \dots, C_{k-1}, C'_k, \dots, C'_m$ wobei $C_k > C'_i$ für $i = k, \dots, m$.
Da $C'_k + \cdots + C'_m \geq C_k$ ist dies ein Widerspruch zu Lemma A. □

Korrektheit

Theorem

Der Greedy-Algorithmus für den Münzwechsel ist optimal.

Beweis.

Nimm an, $C_1, C_2, \dots, C_n$ ist eine Greedy-Lösung (mit $C_i \geq C_{i+1}$).
Zeige mit Induktion über k, daß eine optimale Lösung mit $C_1, C_2, \dots, C_k$ beginnt.

Falls dem nicht so wäre, gäbe es eine optimale Lösung $C_1, C_2, \dots, C_{k-1}, C'_k, \dots, C'_m$ wobei $C_k > C'_i$ für $i = k, \dots, m$.
Da $C'_k + \cdots + C'_m \geq C_k$ ist dies ein Widerspruch zu Lemma A. $\square$