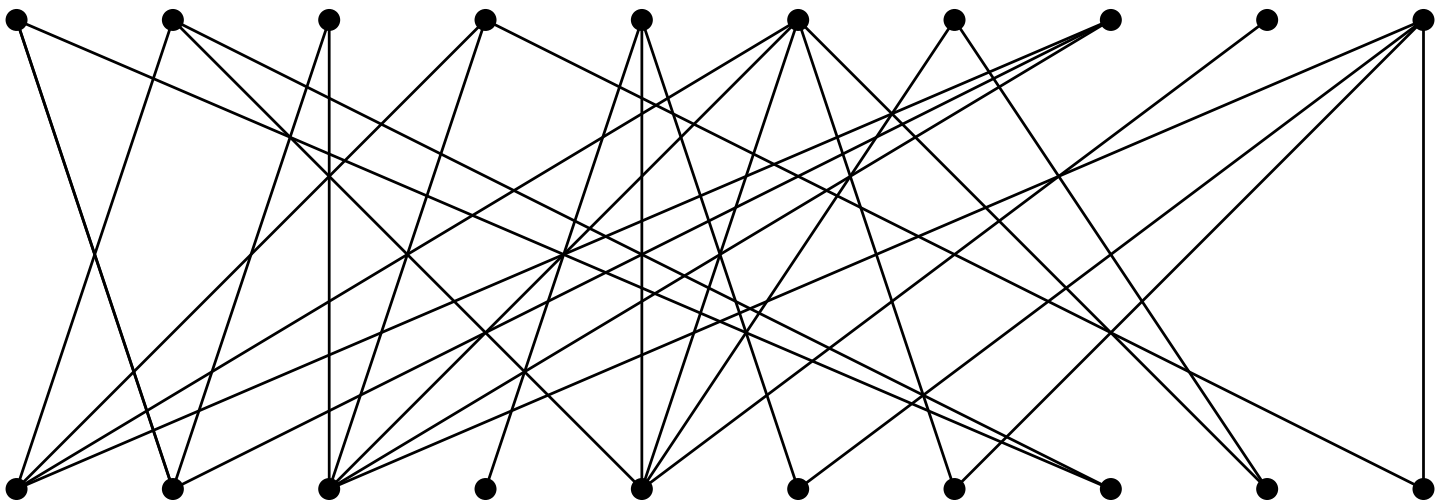
Bipartites Matching

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

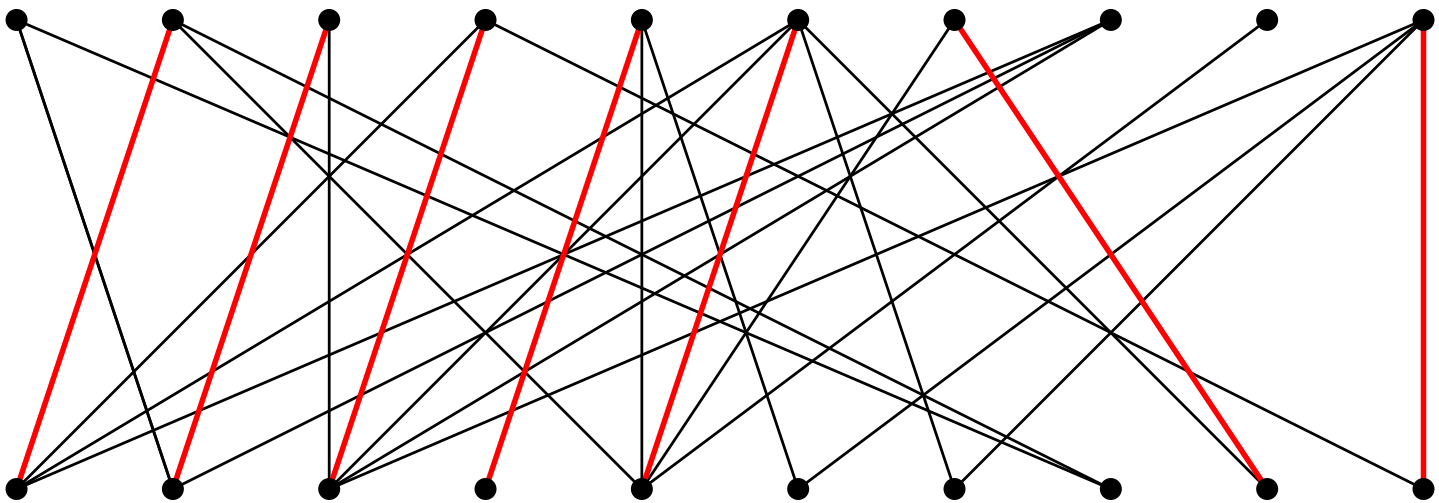
Gesucht:
Ein Matching (Paarung) maximaler Kardinalität.

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

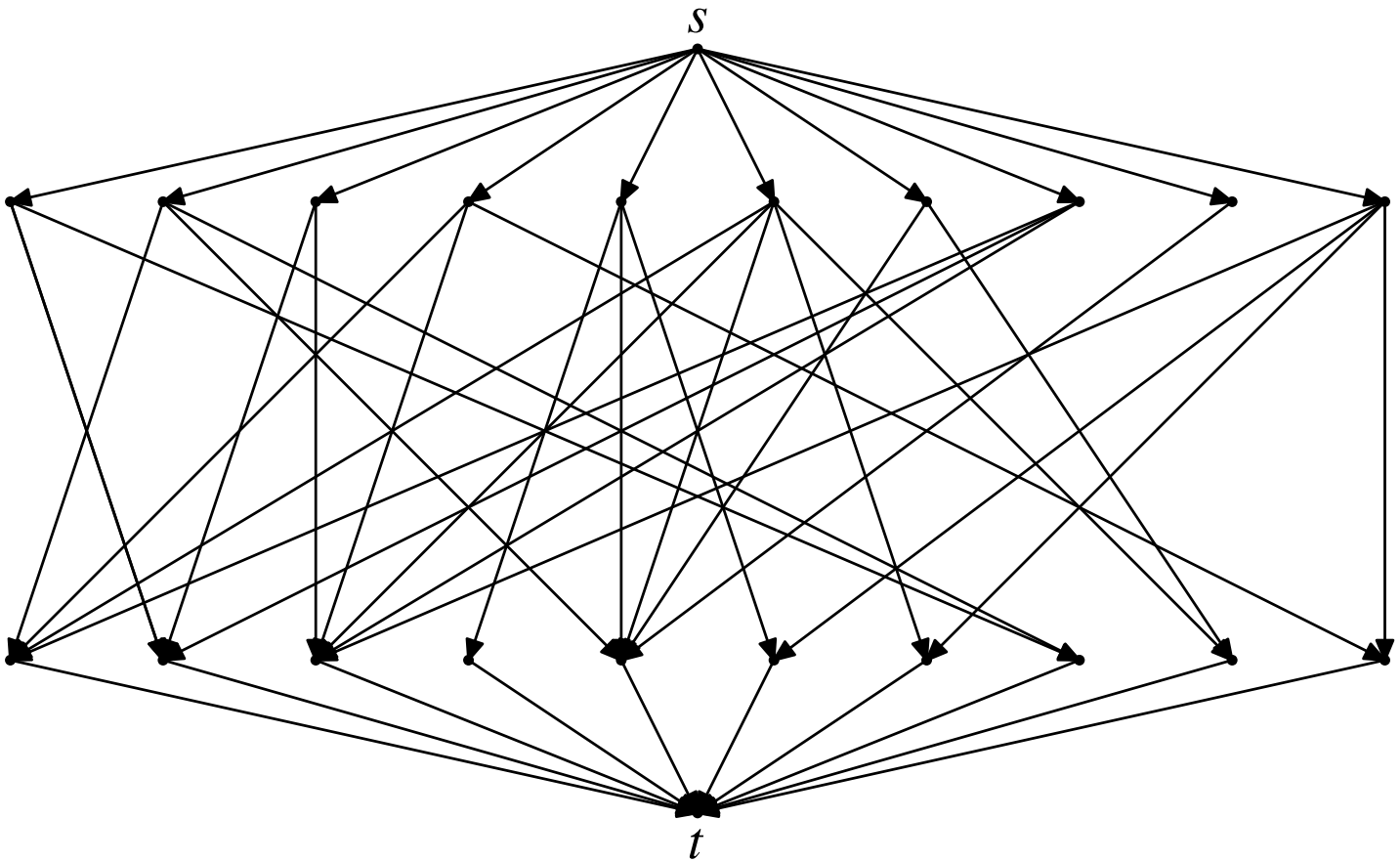
Beispiel



Beispiel



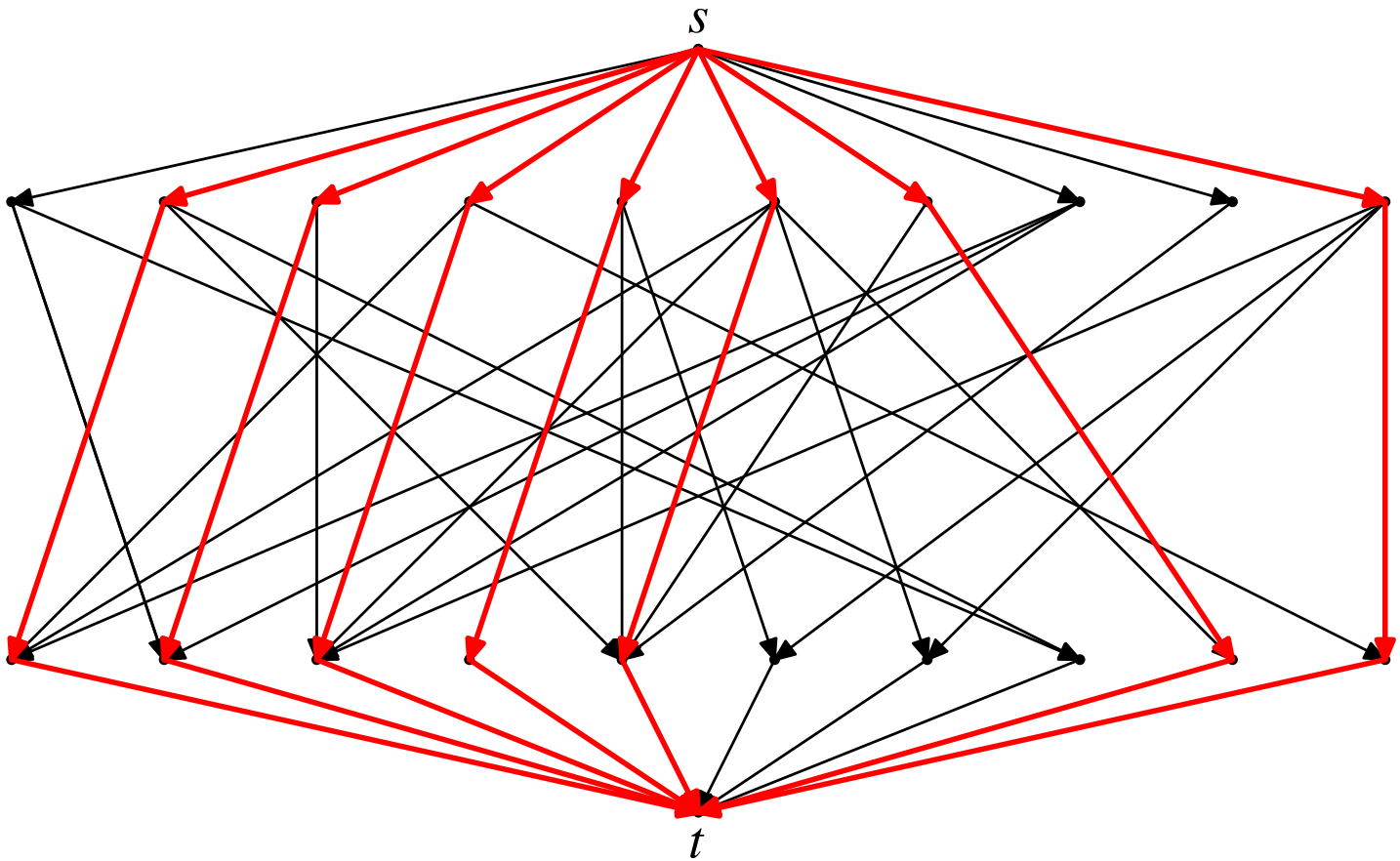
Lösung als Flußproblem



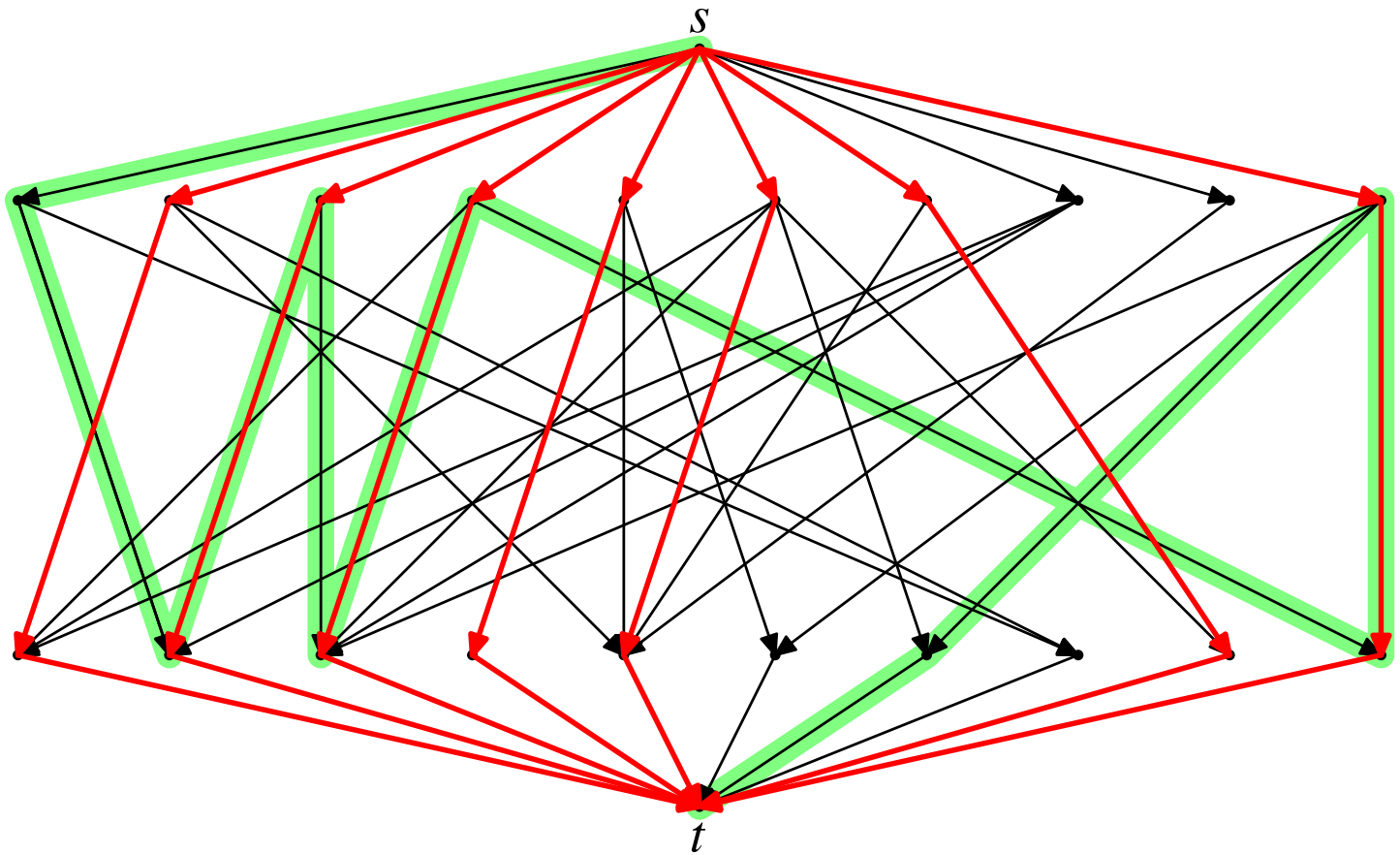
Alle Kapazitäten sind 1.

Maximaler ganzzahliger Fluß entspricht einem Matching maximaler Kardinalität.

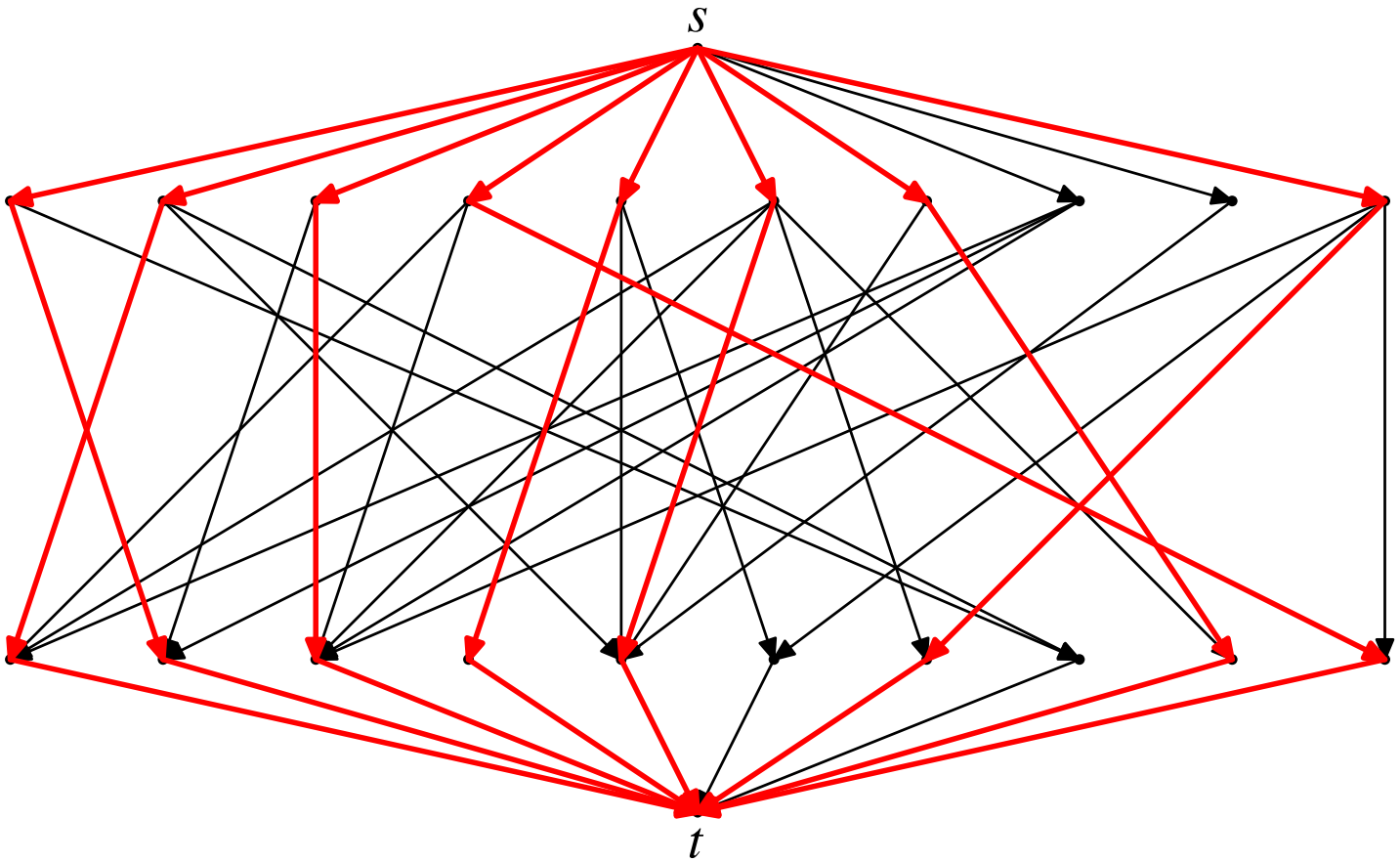
Gibt es einen augmentierenden Pfad?



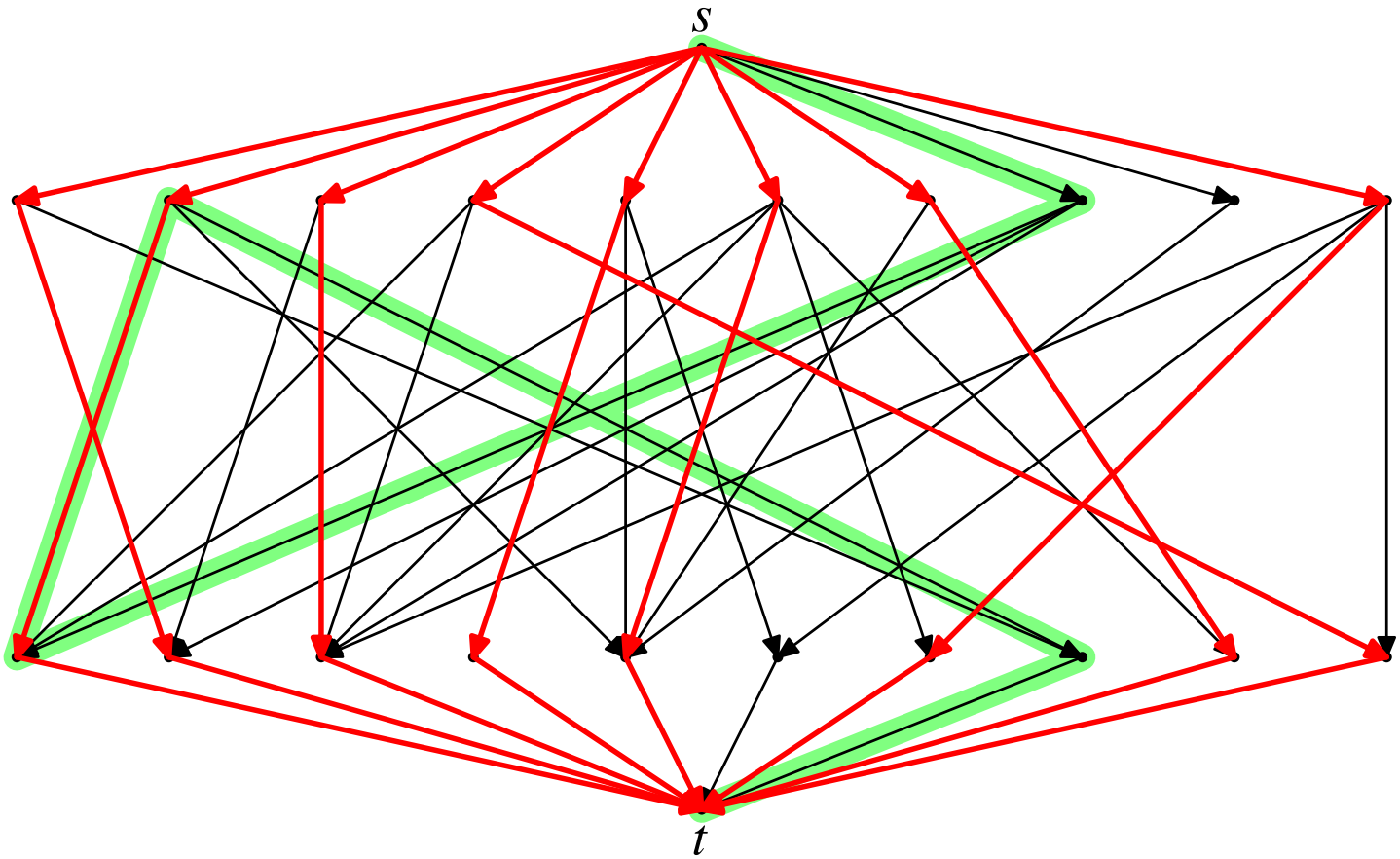
Gibt es einen augmentierenden Pfad? – Ja



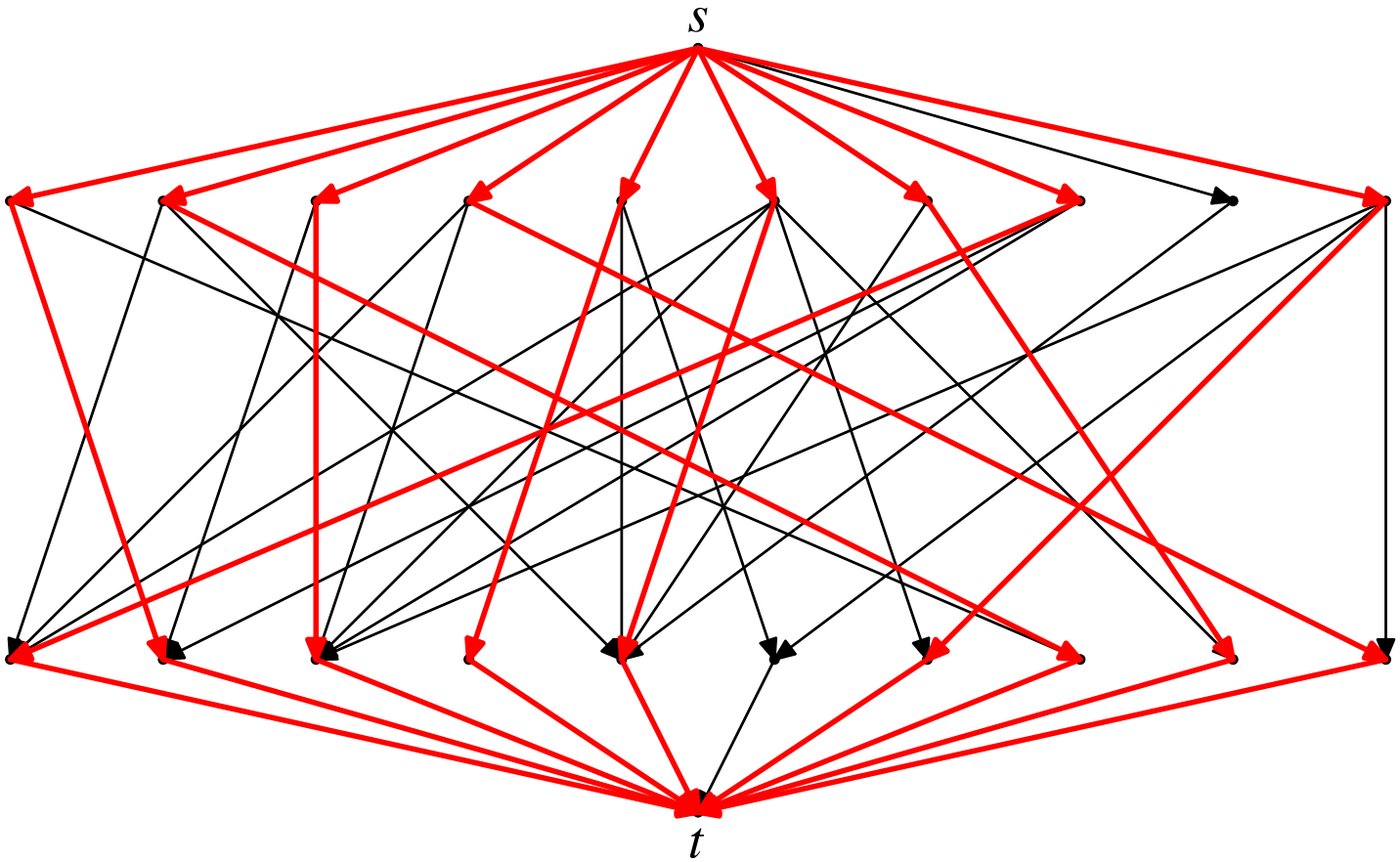
$\Rightarrow$ Neues Matching



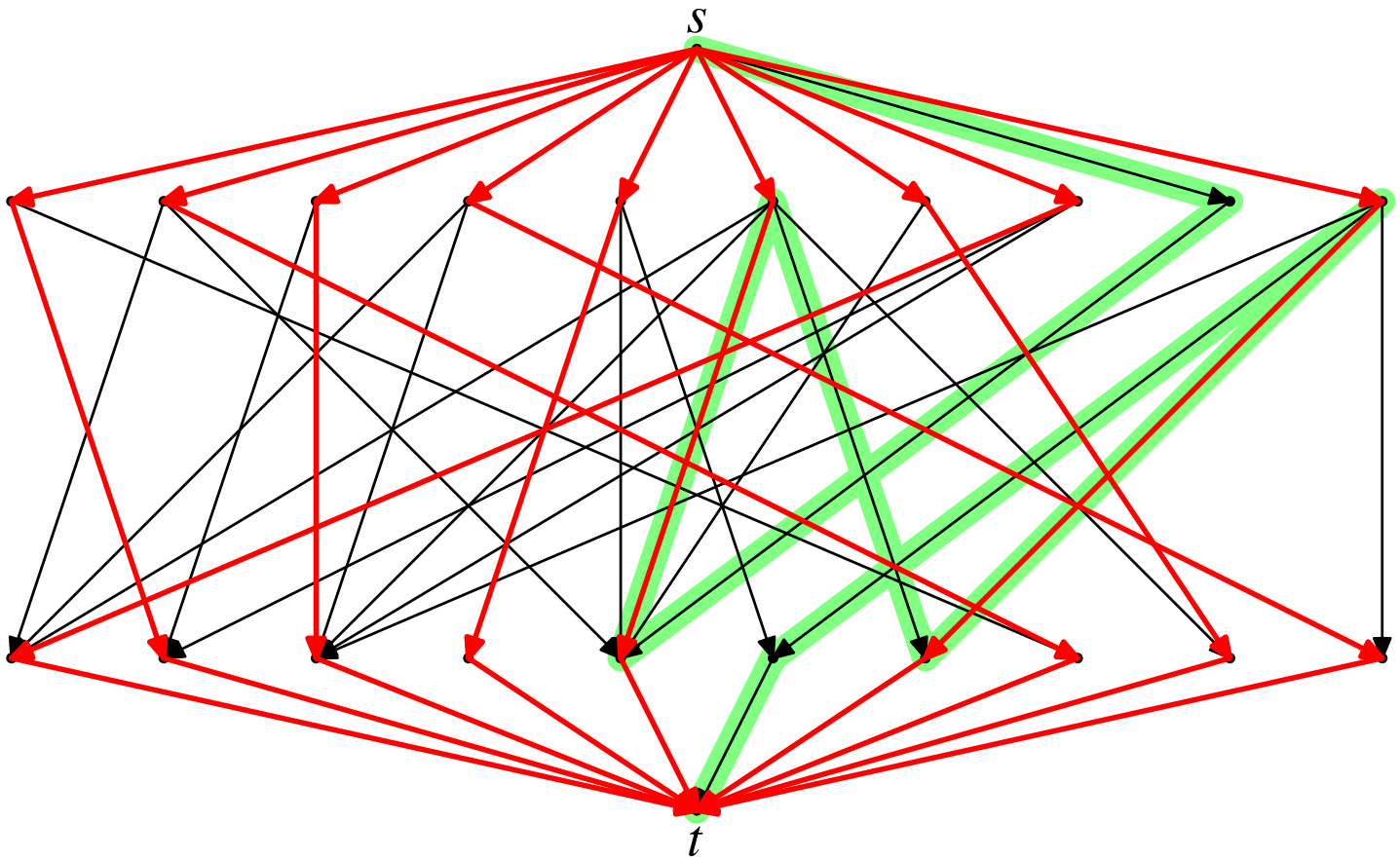
Es gibt wieder einen augmentierenden Pfad



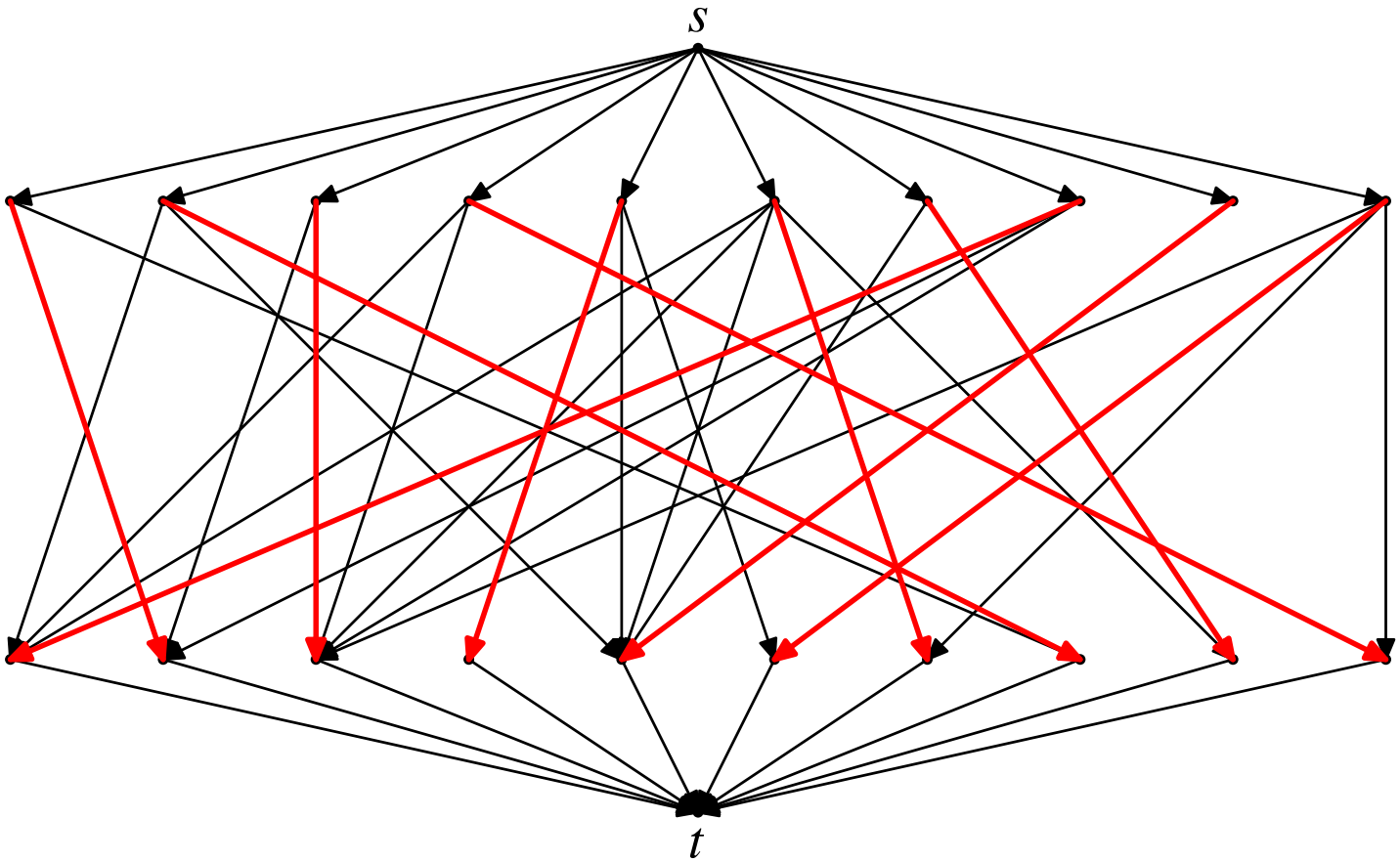
$\Rightarrow$ Neues Matching



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 $O(|E|)$.

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

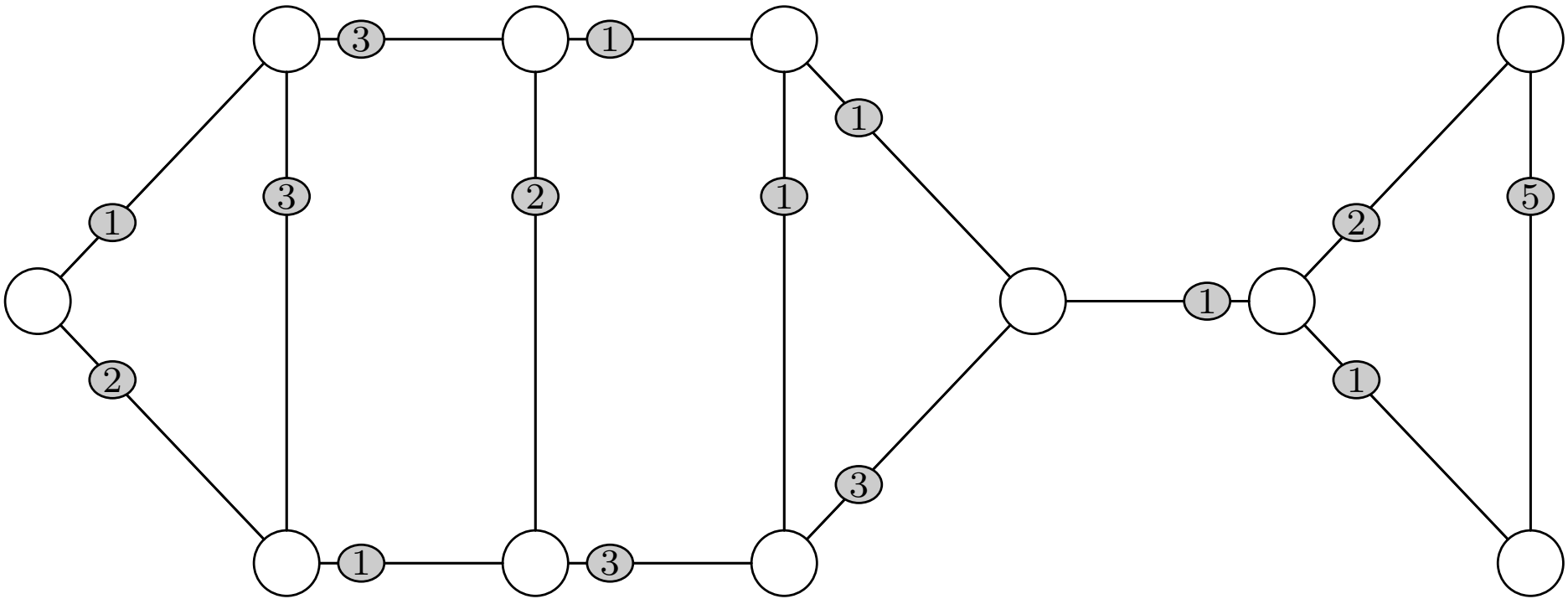
Der Edmonds–Karp–Algorithmus

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

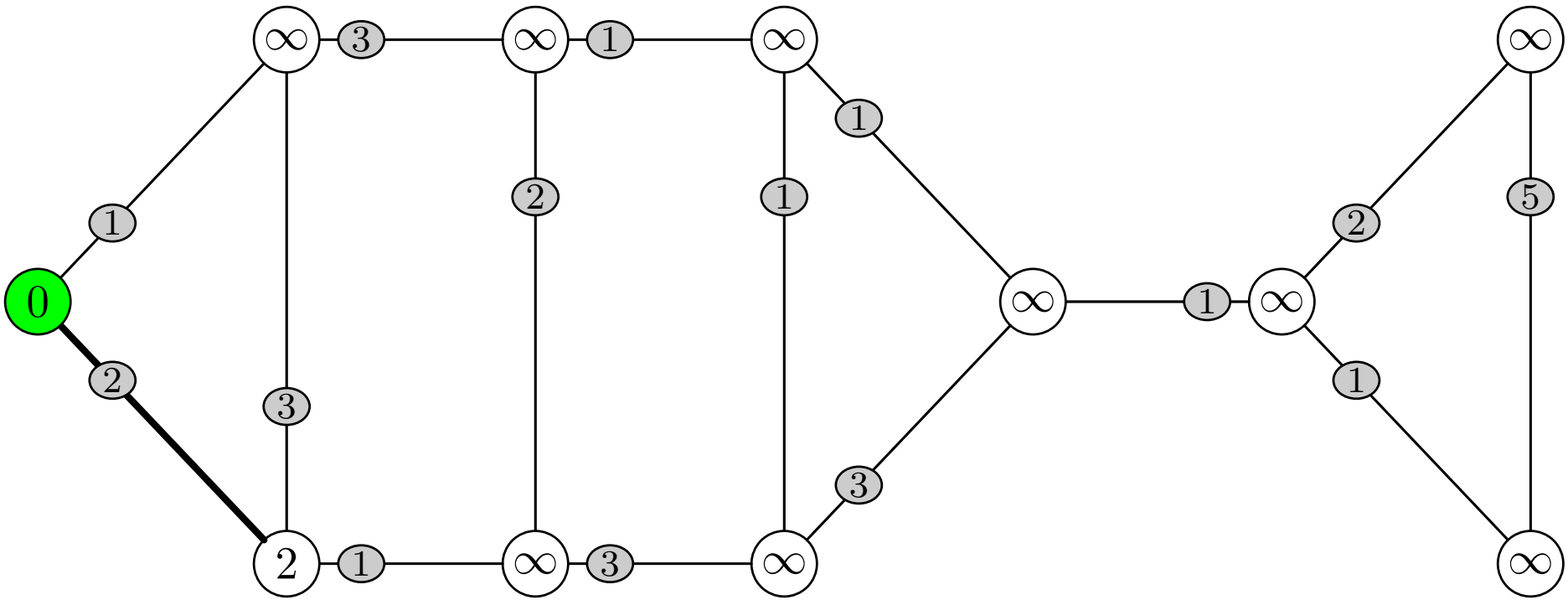
Minimale Spannbäume



Eingabe: Ungerichteter Graph mit Kantengewichten

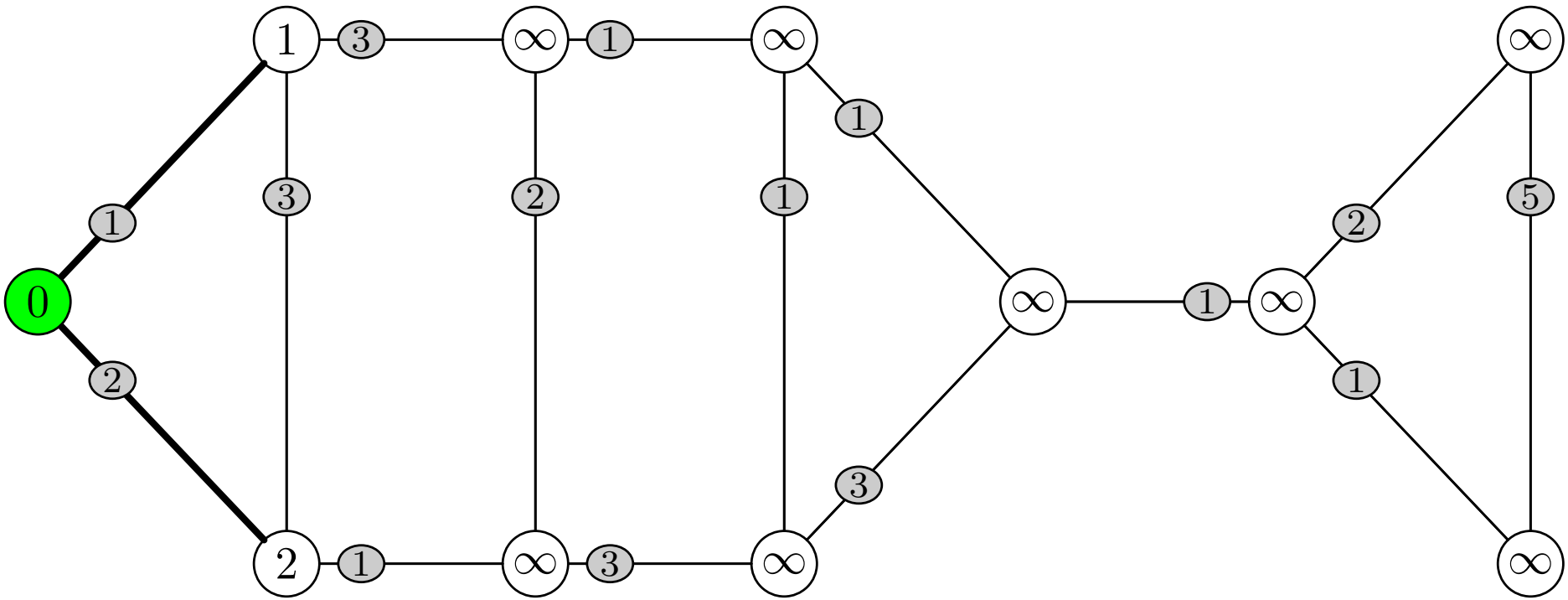
Ausgabe: Ein Baum, der alle Knoten enthält und minimales Kantengewicht hat

Der Algorithmus von Prim – Beispiel



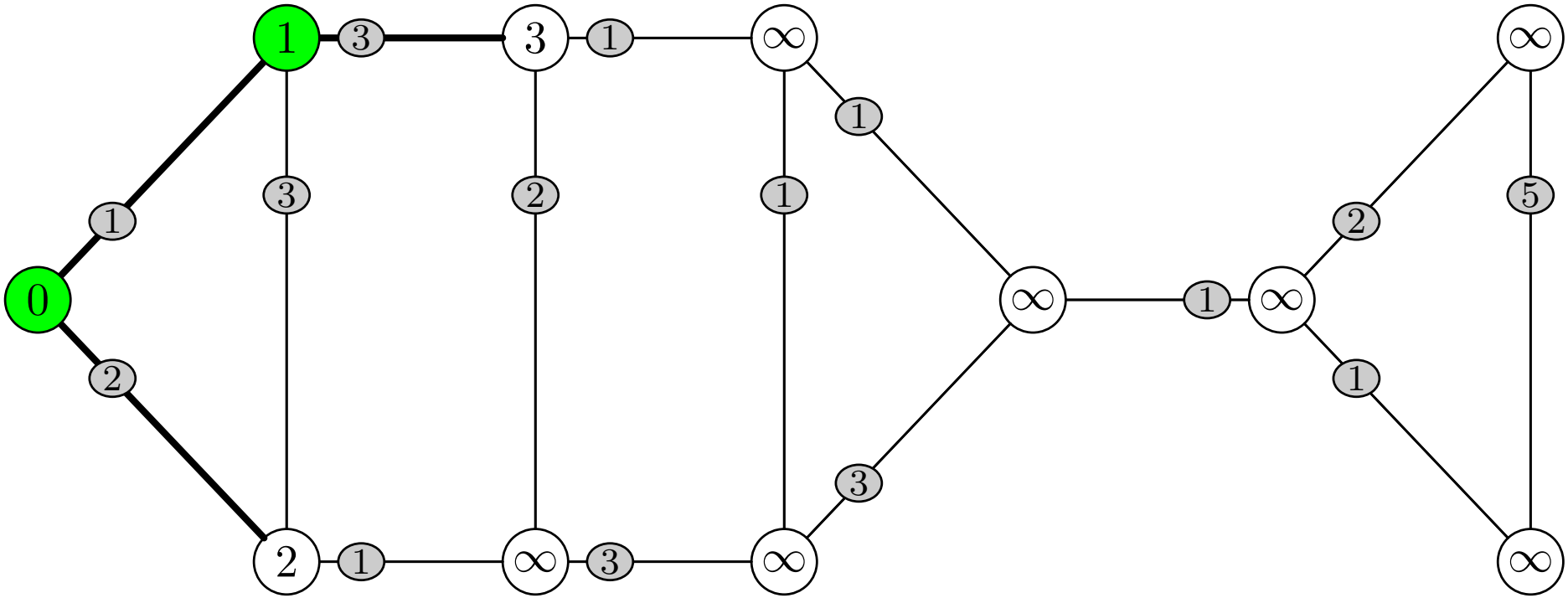
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



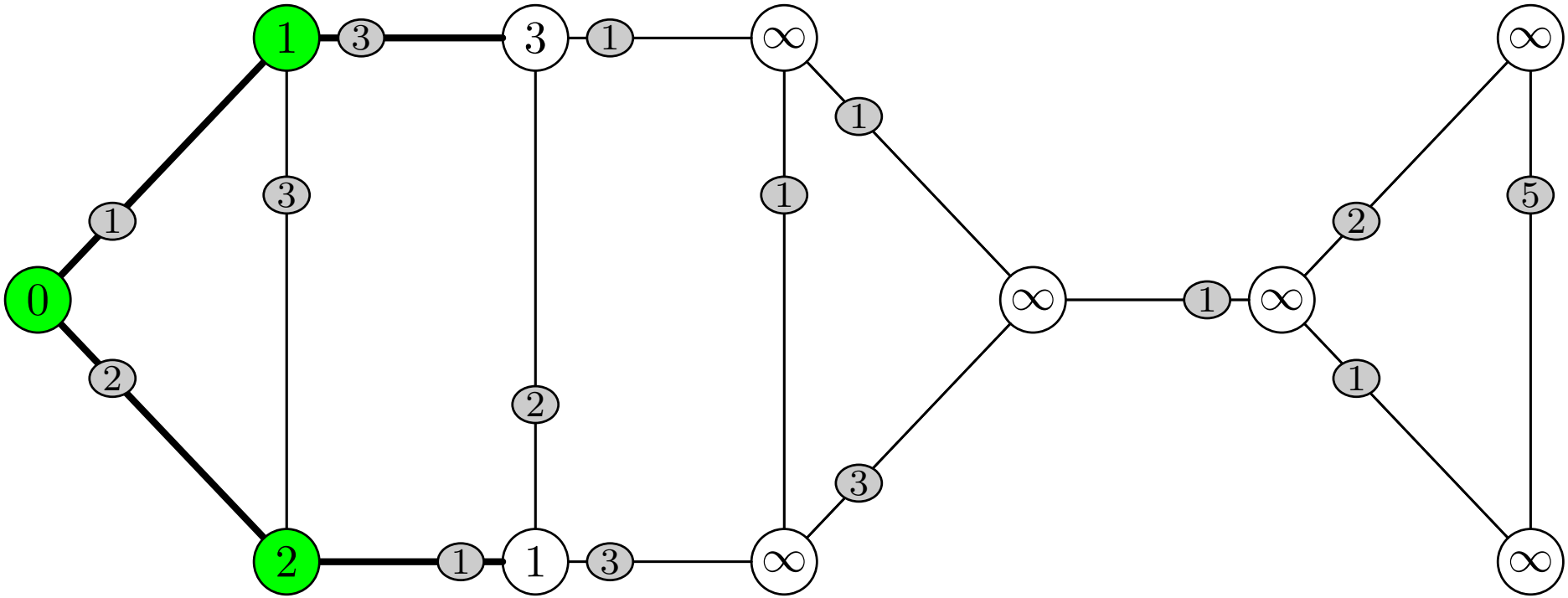
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



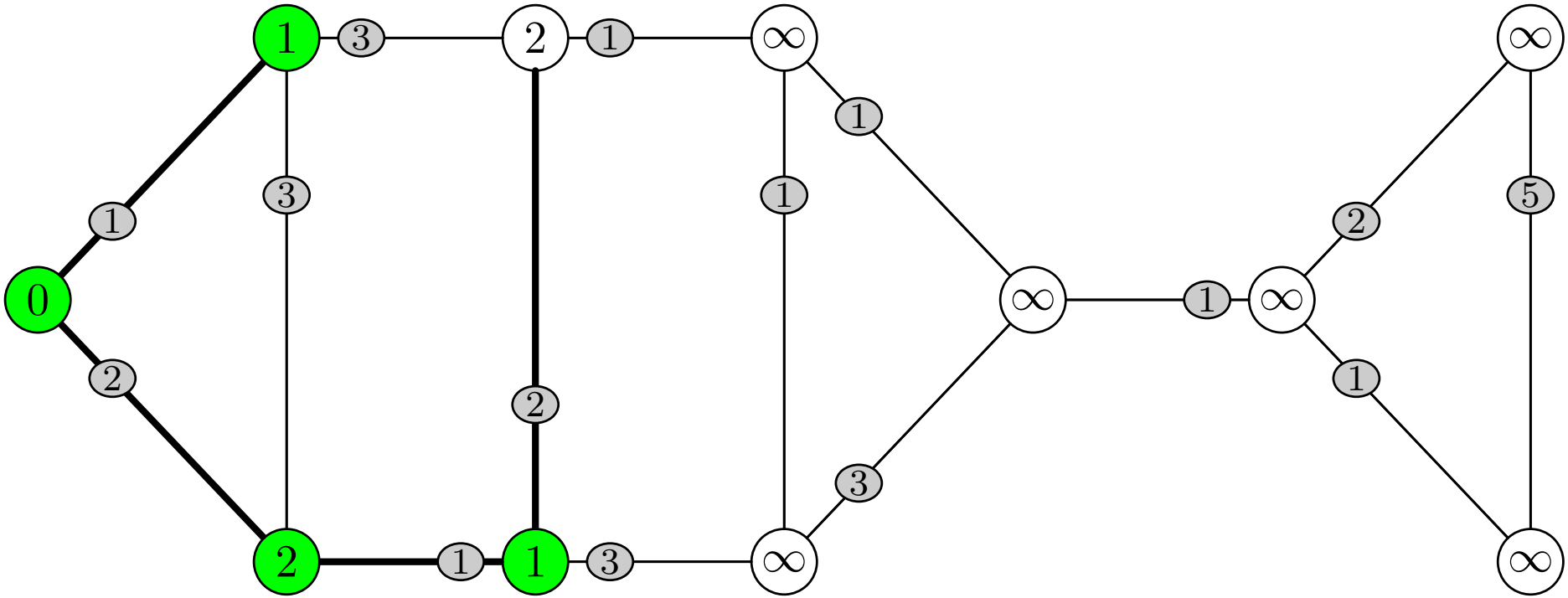
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



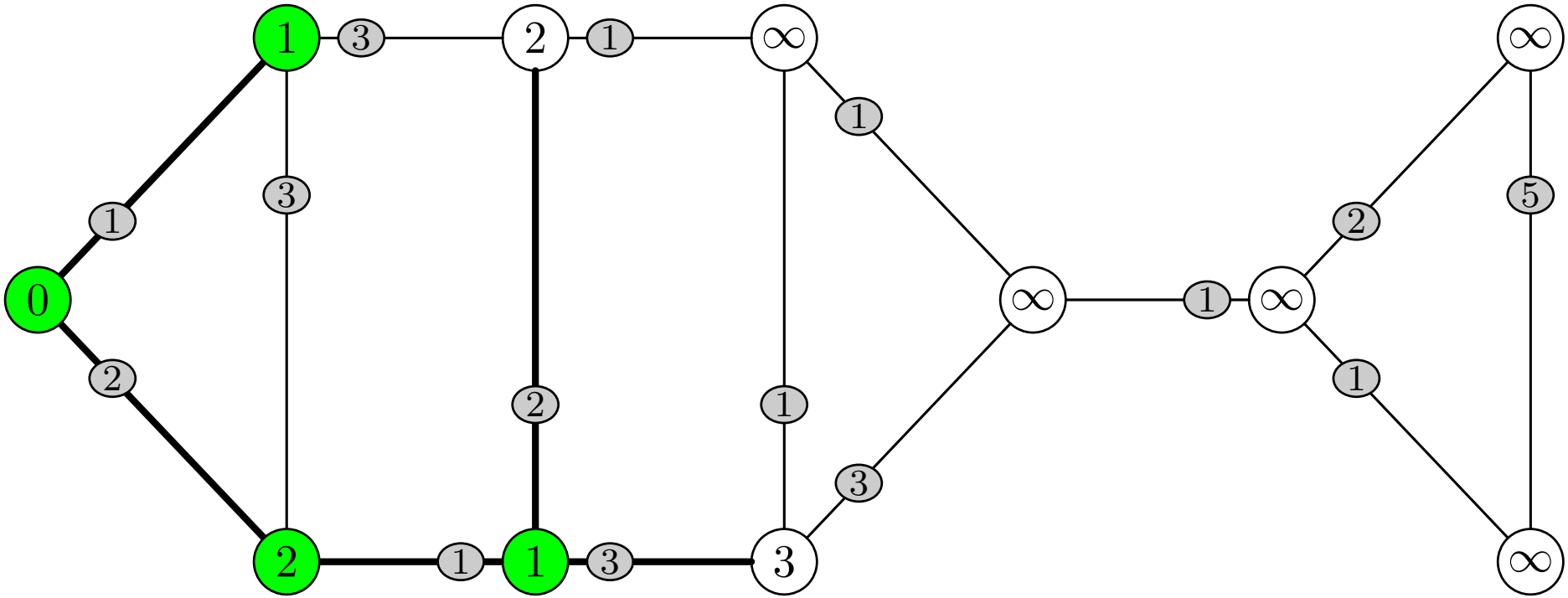
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



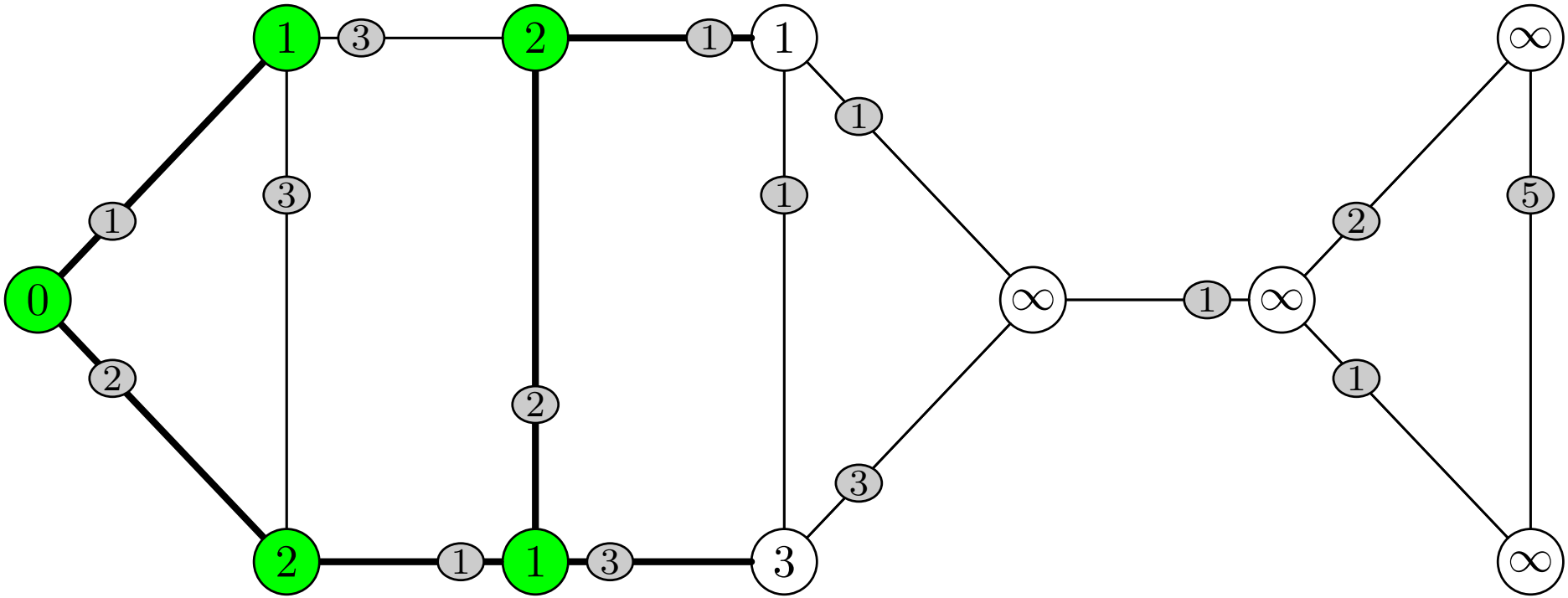
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



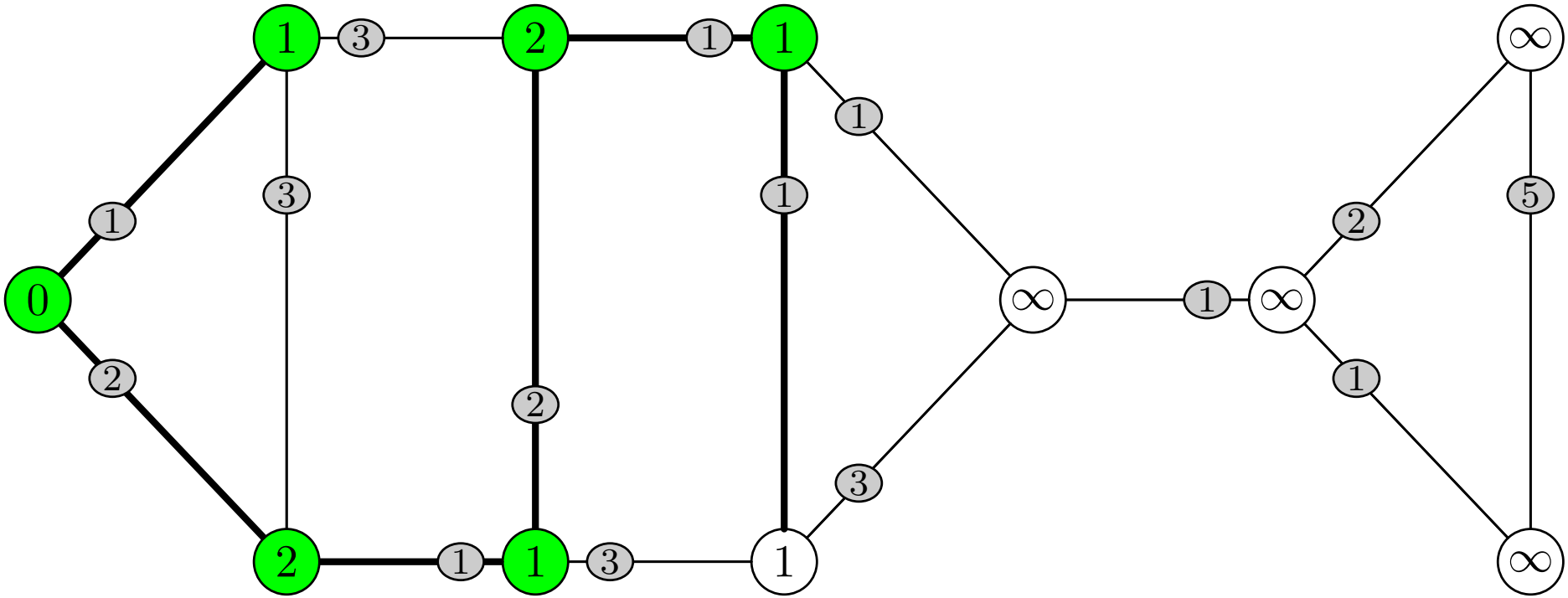
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



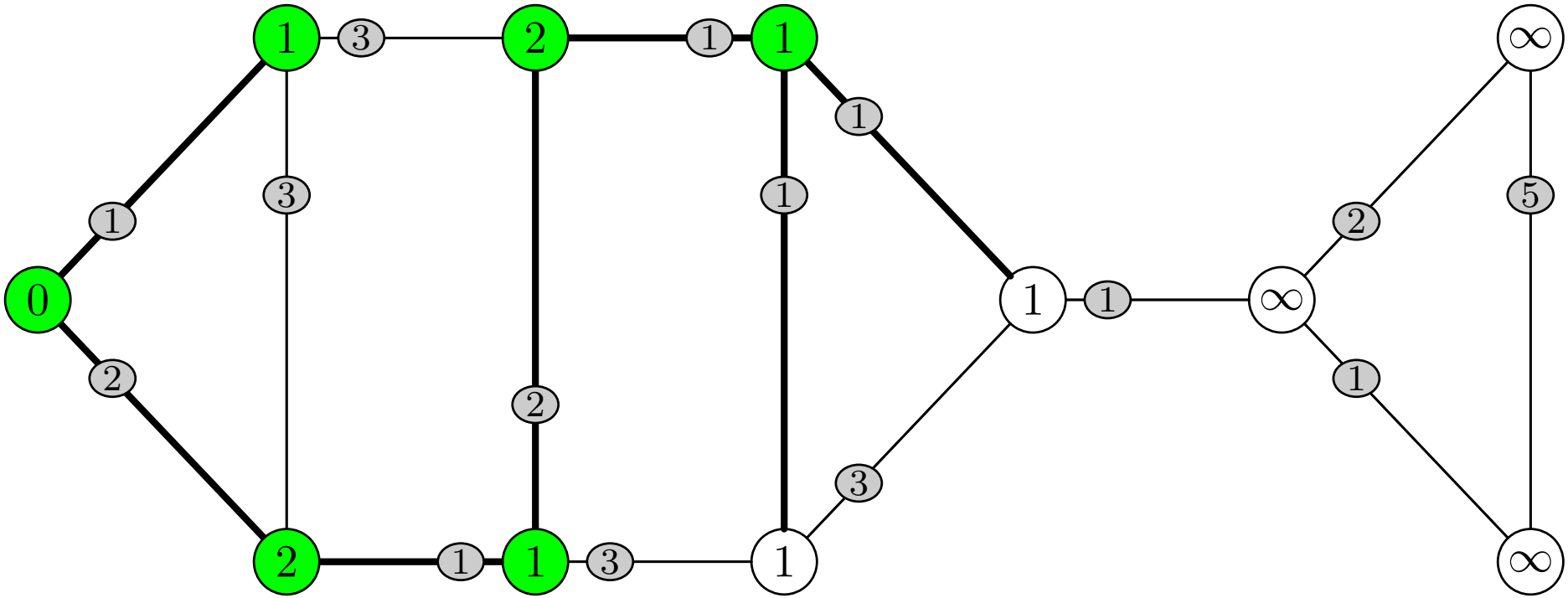
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



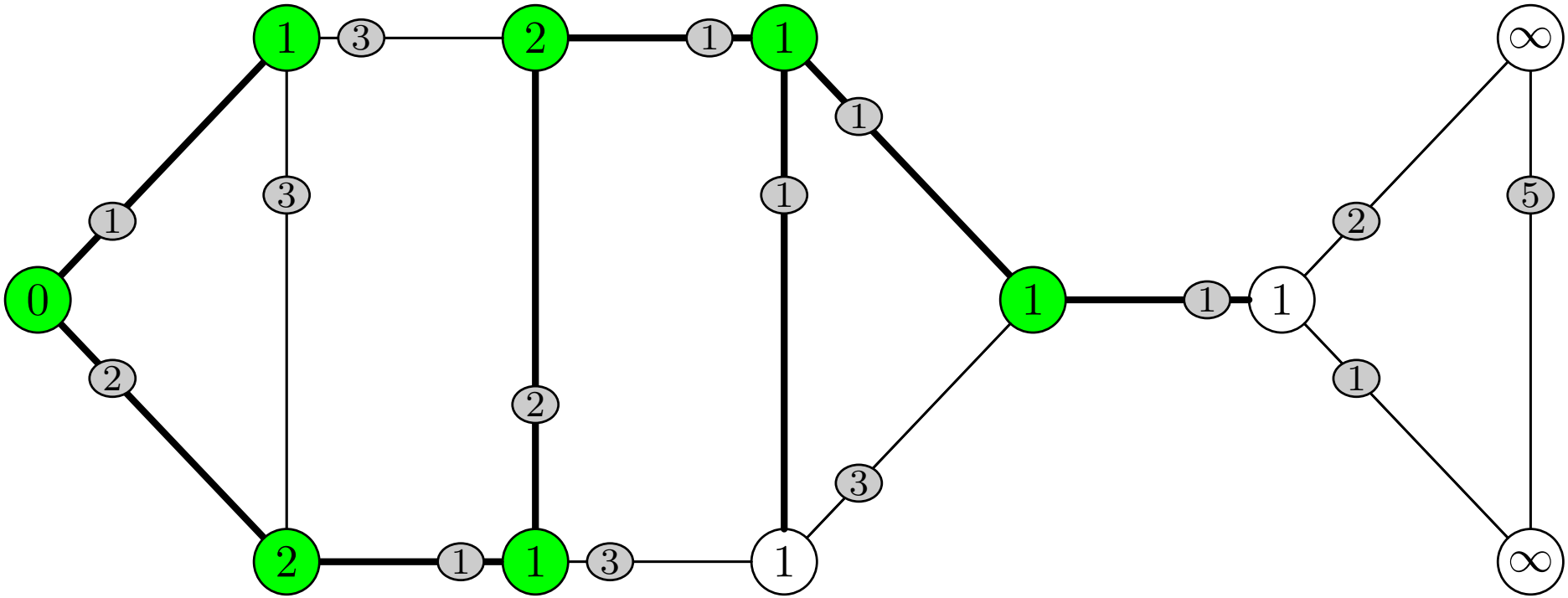
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



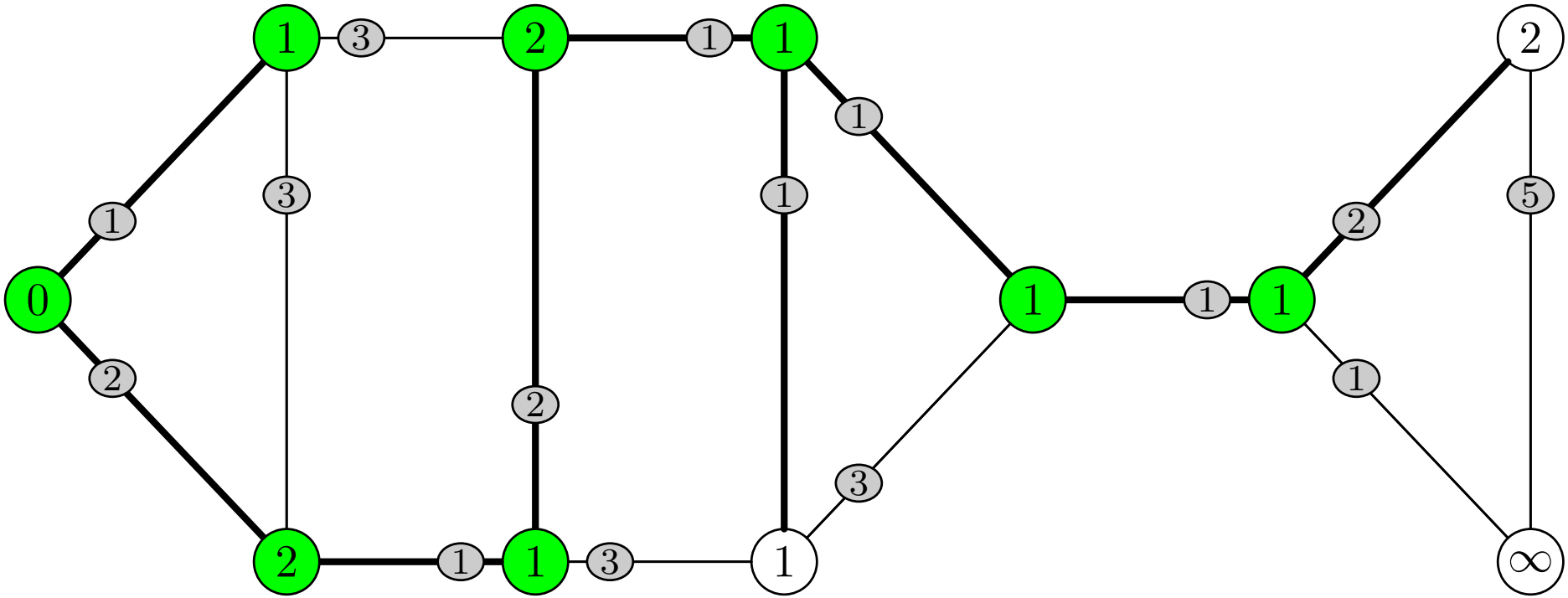
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

Der Algorithmus von Prim – Beispiel



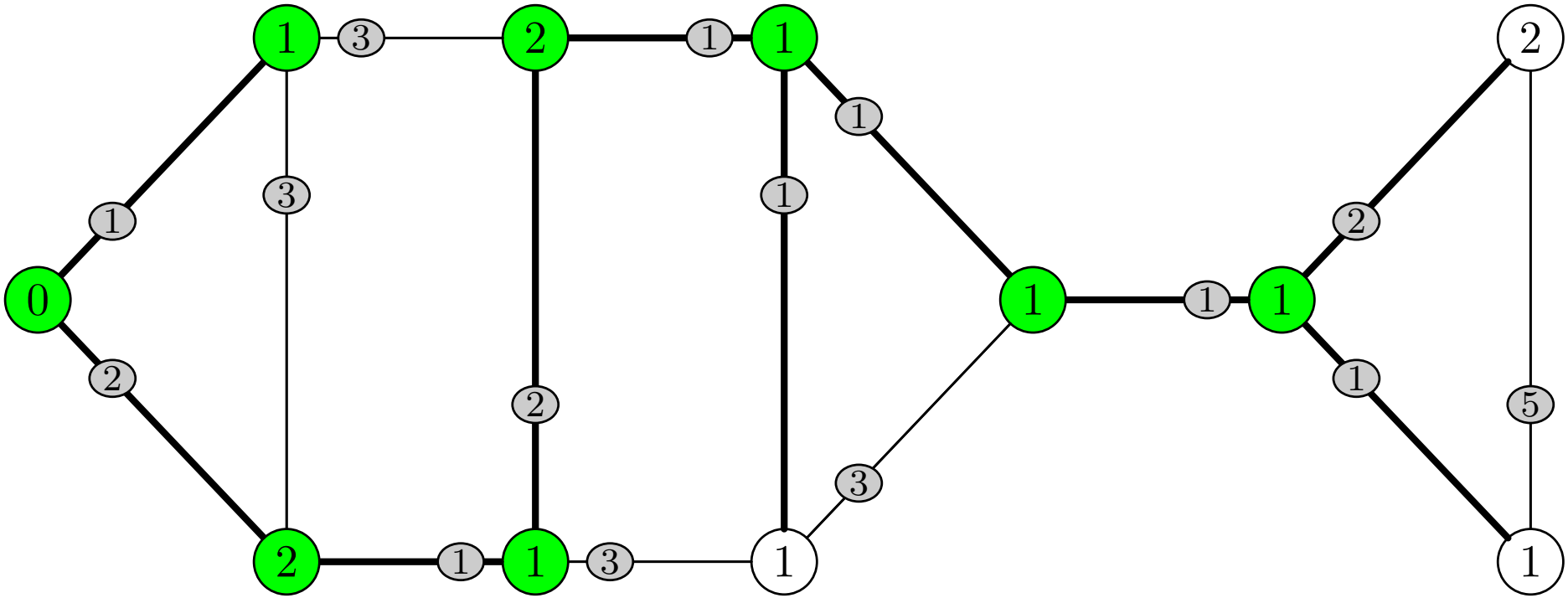
- Beginne mit leerem Baum (nur Wurzel)
- Hänge wiederholt billigste Kante an ohne einen Kreis zu schließen

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

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 cup { 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 $100 \leq v < 50(k+1)$ gelten. Es folgt $k \geq 2$, 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. □

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, $C_1, C_2, \ldots, C_n$ 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. □

Briefmarkensammeln

Funktioniert der Greedy-Algorithmus auch für Briefmarken aus Manchukuo?

Welche Briefmarken für einen 20 fen–Brief?

Der Greedy-Algorithmus führt nicht zu einer optimalen Lösung und findet manchmal gar keine!

Briefmarkensammeln

Funktioniert der Greedy-Algorithmus auch für Briefmarken aus Manchukuo?

Welche Briefmarken für einen 20 fen–Brief?

Der Greedy-Algorithmus führt nicht zu einer optimalen Lösung und findet manchmal gar keine!

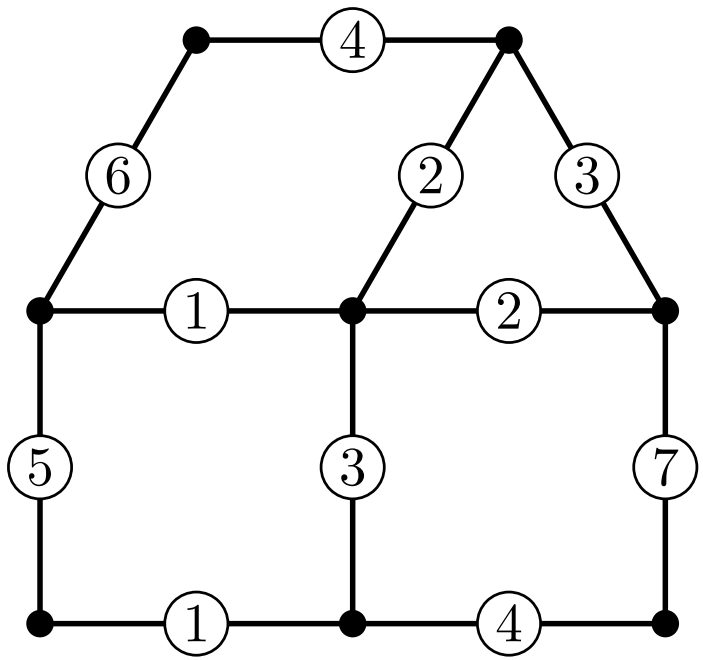
Briefmarkensammeln

Funktioniert der Greedy-Algorithmus auch für Briefmarken aus Manchukuo?

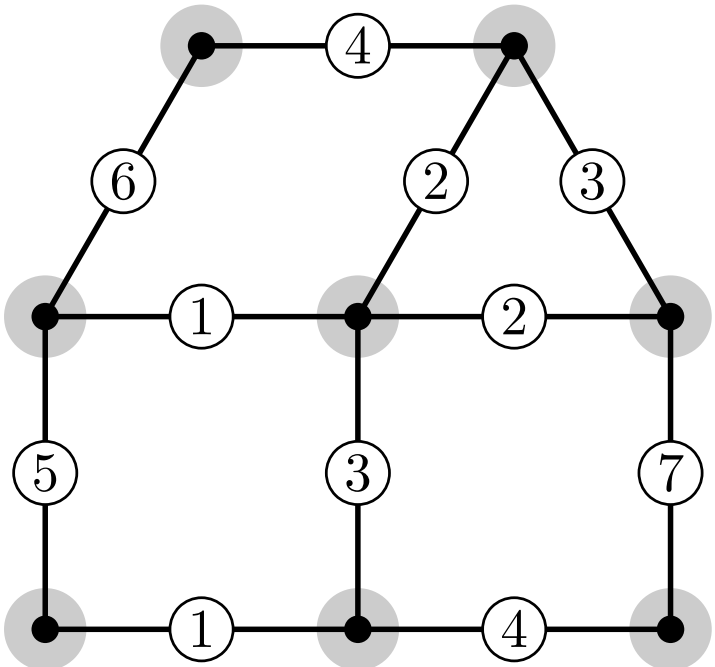
Welche Briefmarken für einen 20 fen–Brief?

Der Greedy-Algorithmus führt nicht zu einer optimalen Lösung und findet manchmal gar keine!

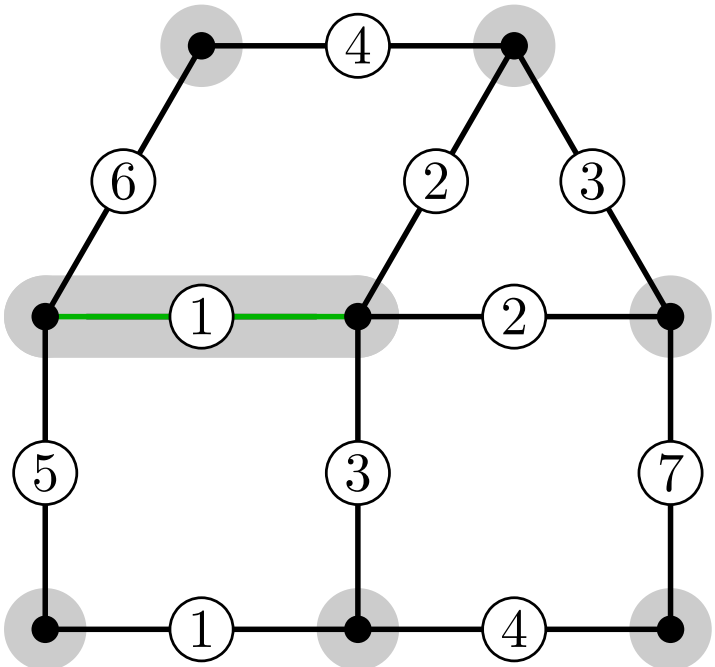
Kruskal: Minimaler Spannbaum



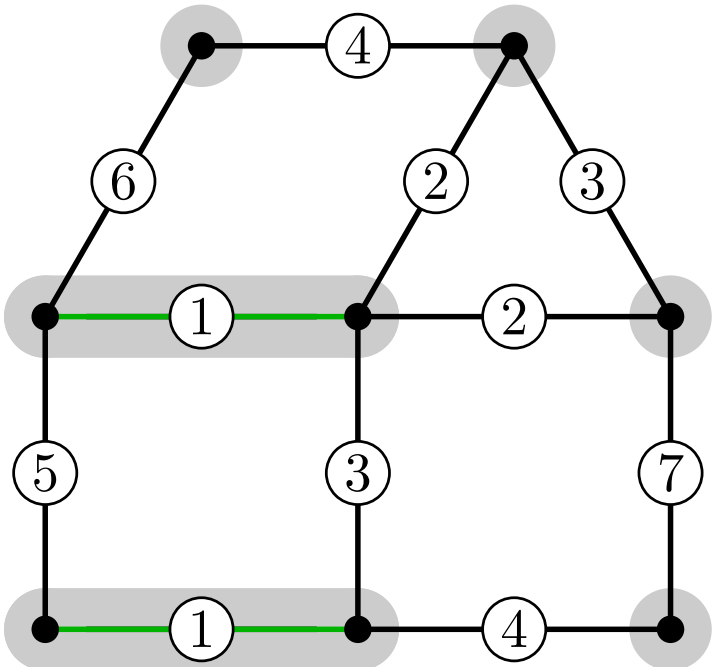
Kruskal: Minimaler Spannbaum



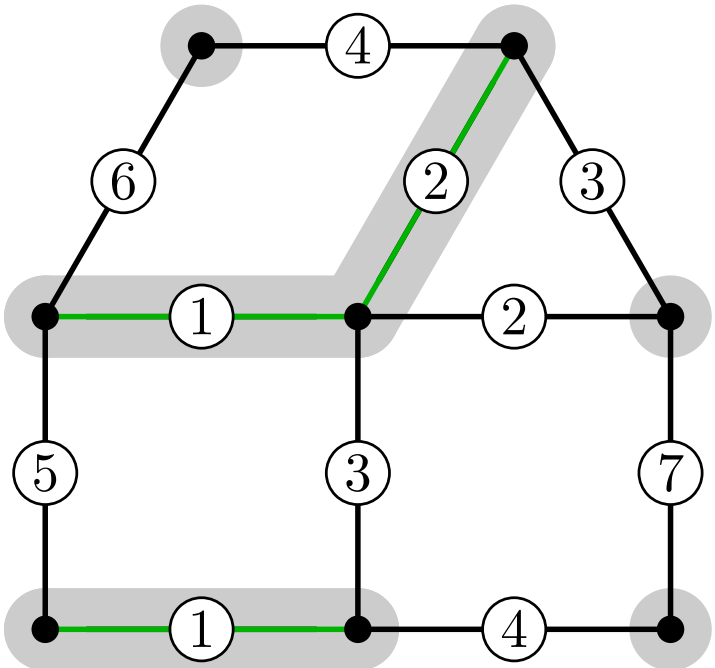
Kruskal: Minimaler Spannbaum



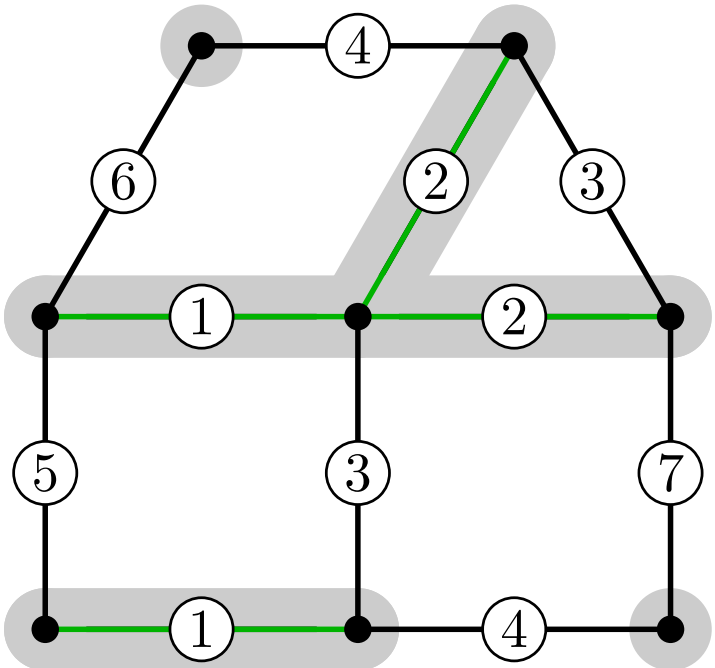
Kruskal: Minimaler Spannbaum



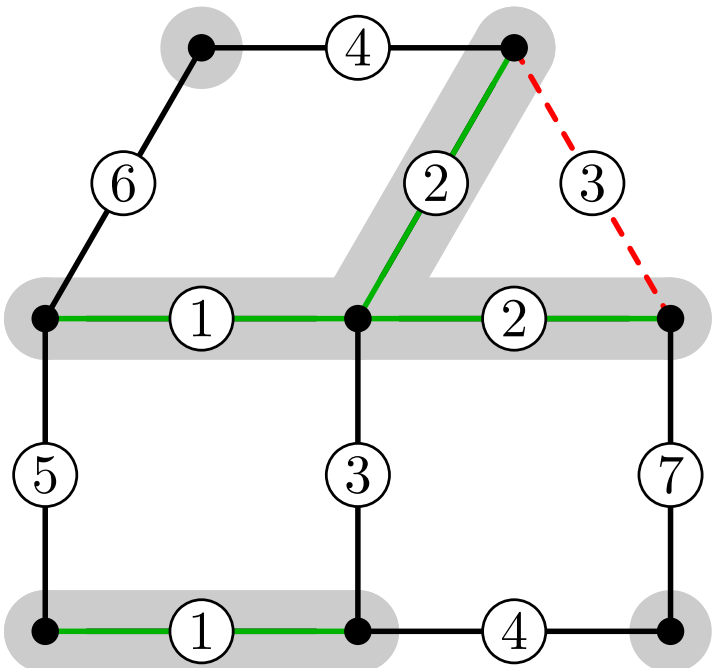
Kruskal: Minimaler Spannbaum



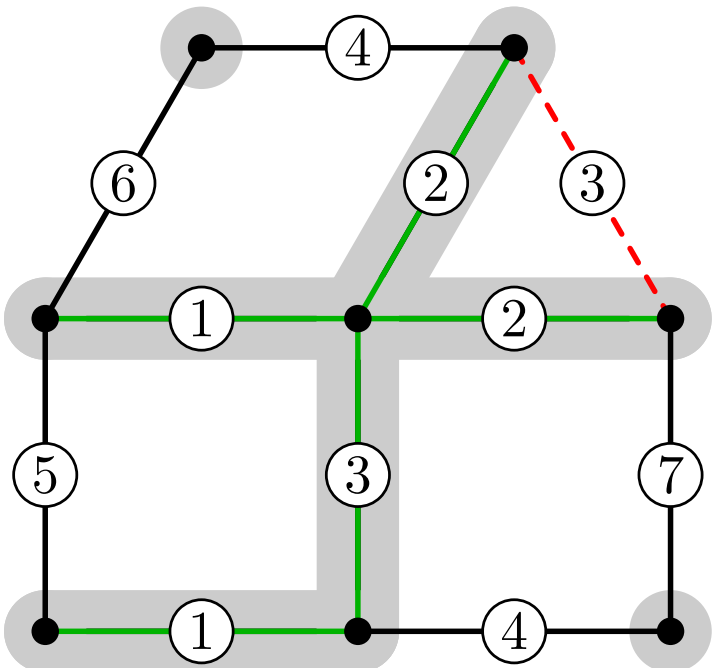
Kruskal: Minimaler Spannbaum



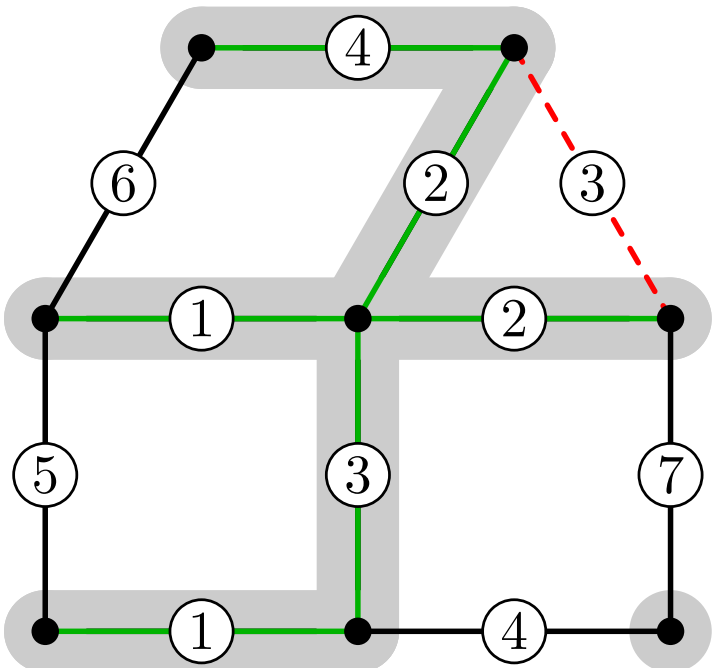
Kruskal: Minimaler Spannbaum



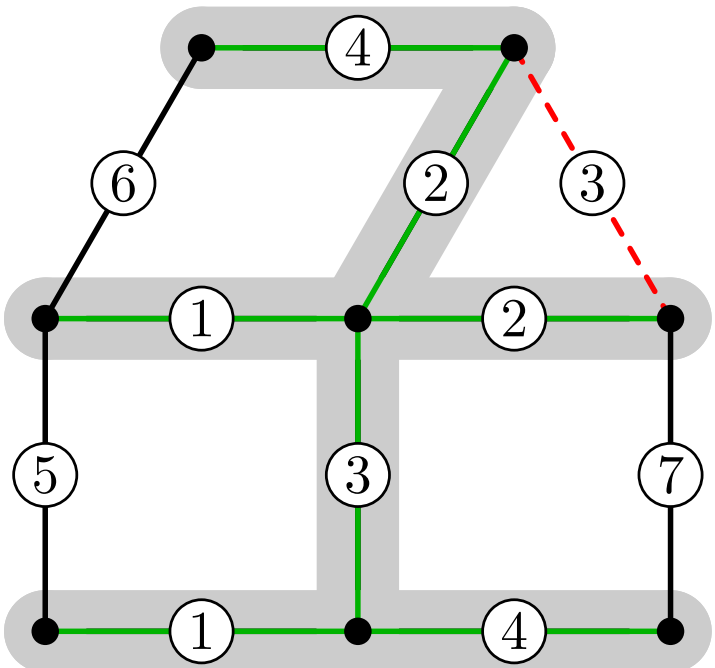
Kruskal: Minimaler Spannbaum



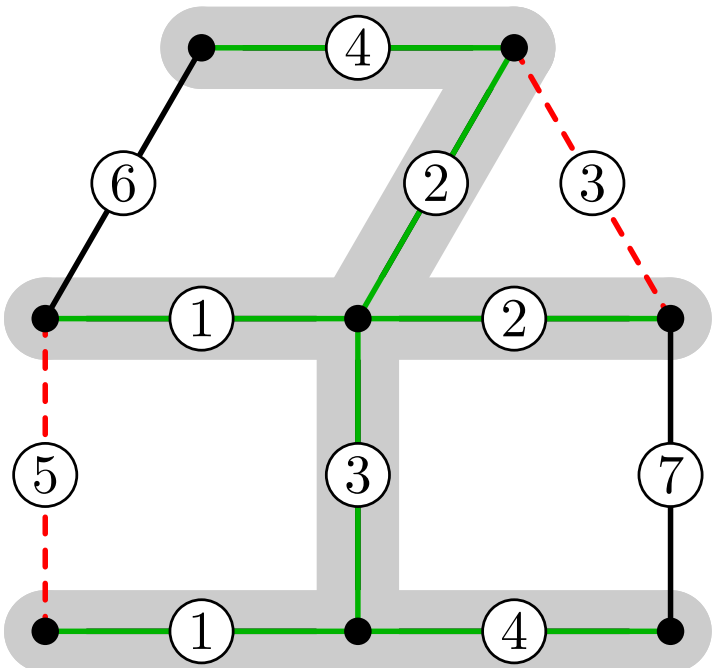
Kruskal: Minimaler Spannbaum



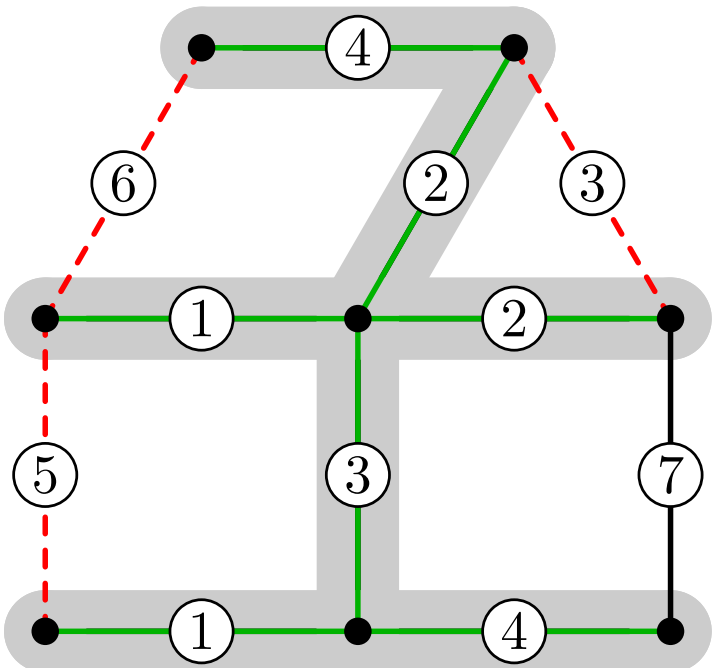
Kruskal: Minimaler Spannbaum



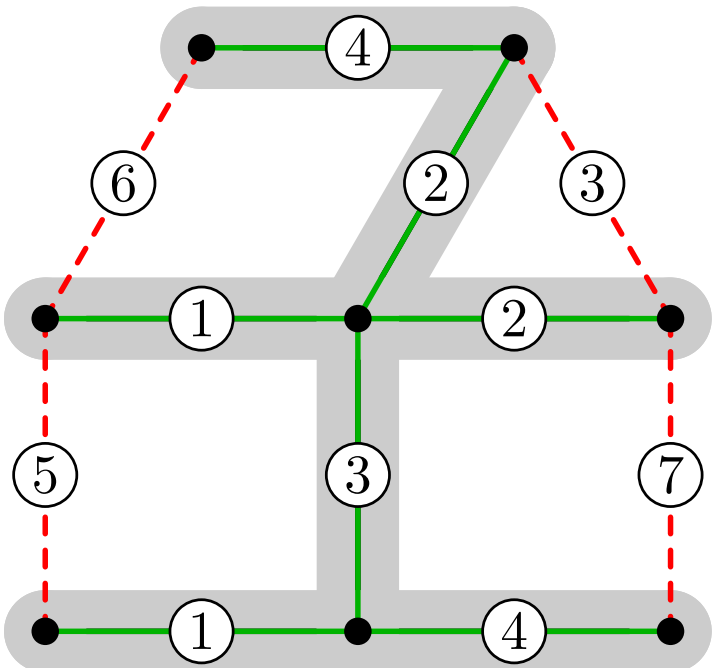
Kruskal: Minimaler Spannbaum



Kruskal: Minimaler Spannbaum



Kruskal: Minimaler Spannbaum



Kruskals Algorithmus – Implementierung

Algorithmus

```
function Kruskal(G, w) :
A := emptyset;
for each vertex v in V[G] do
  Make_Set(v)
od;
sort the edges of E into nondecreasing order by weight;
for each edge { u, v}in E, nondecreasingly do
  if Find_Set(u) ≠ Find_Set(v) then
     A := A cup { { u, v}};
     Union(u, v)
  fi
od;
return A
```

Korrektheit und Laufzeit der Implementierung

Die Korrektheit folgt als Korollar aus der Korrektheit des allgemeinen Greedy-Algorithmus und der Beobachtung zum graphischen Matroid.

Laufzeit mit $m = |E|$ und $n = |V|$ und $m \geq n - 1$:
n Make-Set-Operationen,
$O(m \log m)$ Zeit für das Sortieren der Kanten, und
$O(m)$ Find-Set- und Union-Operationen.

Mit einer geeigneten Union-Find-Implementierung zusammen $O(m \log m)$.

Korrektheit und Laufzeit der Implementierung

Die Korrektheit folgt als Korollar aus der Korrektheit des allgemeinen Greedy-Algorithmus und der Beobachtung zum graphischen Matroid.
Laufzeit mit $m = |E|$ und $n = |V|$ und $m \geq n - 1$:
n Make-Set-Operationen,
$O(m \log m)$ Zeit für das Sortieren der Kanten, und
$O(m)$ Find-Set- und Union-Operationen.

Mit einer geeigneten Union-Find-Implementierung zusammen $O(m \log m)$.

Union-Find: naiv

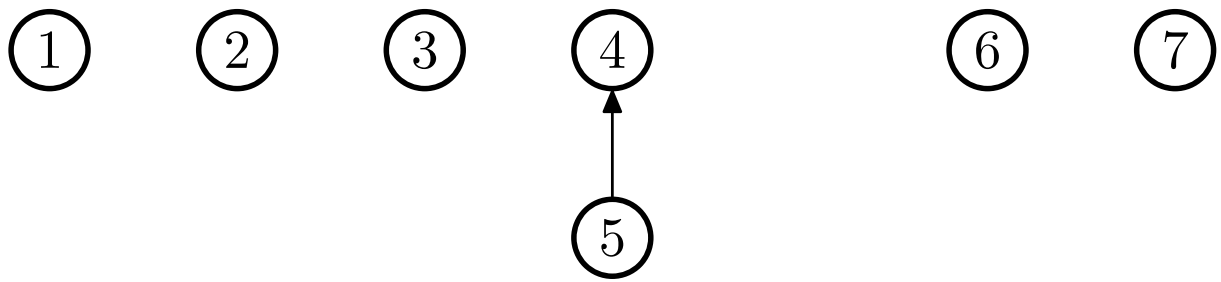
union(a, b): Hänge *find*(a) bei *find*(b) ein

Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$...

Union-Find: naiv

union(a, b): Hänge *find*(a) bei *find*(b) ein

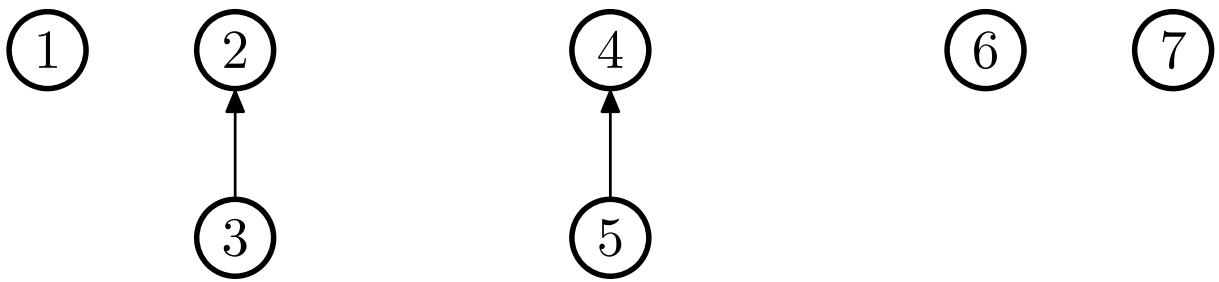
Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)\ldots$



Union-Find: naiv

union(a, b): Hänge *find*(a) bei *find*(b) ein

Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)\ldots$



Union-Find: naiv

union(a, b): Hänge *find*(a) bei *find*(b) ein

Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)\ldots$

Union-Find: naiv

union(a, b): Hänge *find*(a) bei *find*(b) ein

Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)\ldots$

Union-Find: union by rank

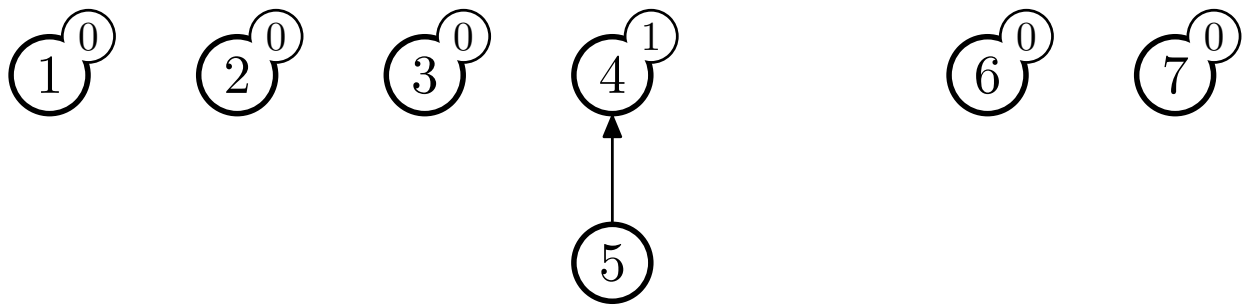
union(a, b): Verwende die ranghöhere Wurzel

Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$, *union*$(3, 4)$

Union-Find: union by rank

union(a, b): Verwende die ranghöhere Wurzel

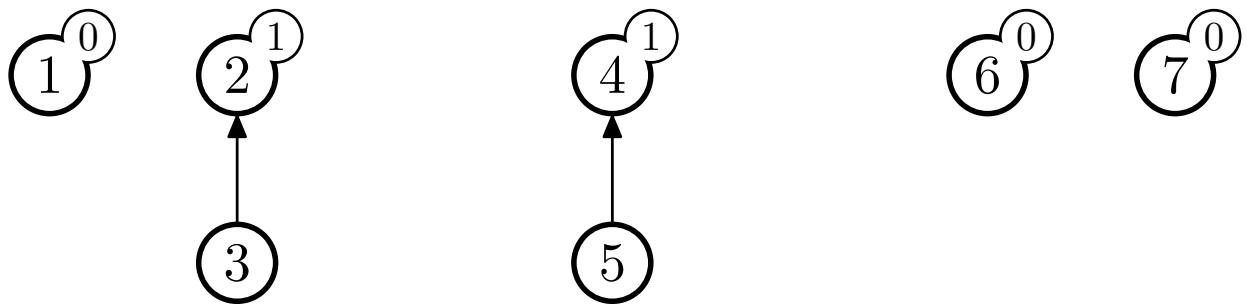
Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$, *union*$(3, 4)$



Union-Find: union by rank

union(a, b): Verwende die ranghöhere Wurzel

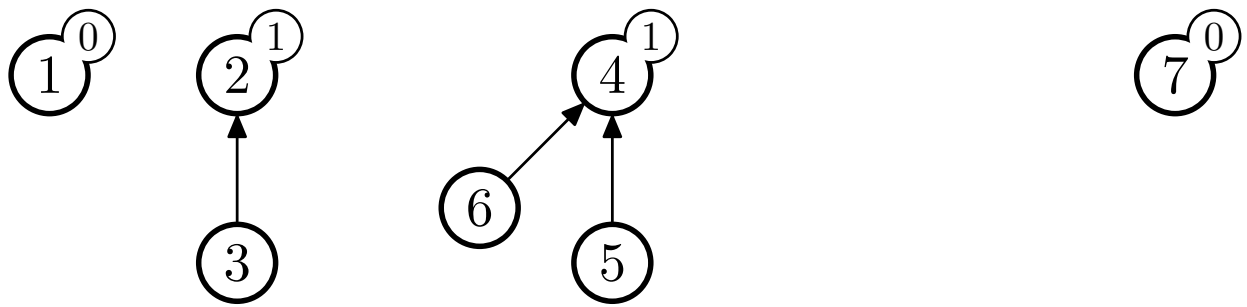
Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$, *union*$(3, 4)$



Union-Find: union by rank

union(a, b): Verwende die ranghöhere Wurzel

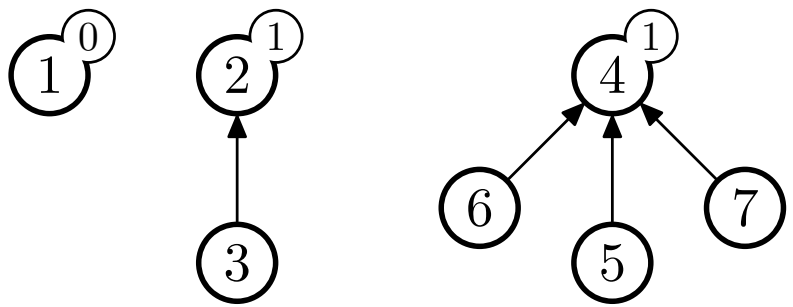
Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$, *union*$(3, 4)$



Union-Find: union by rank

union(a, b): Verwende die ranghöhere Wurzel

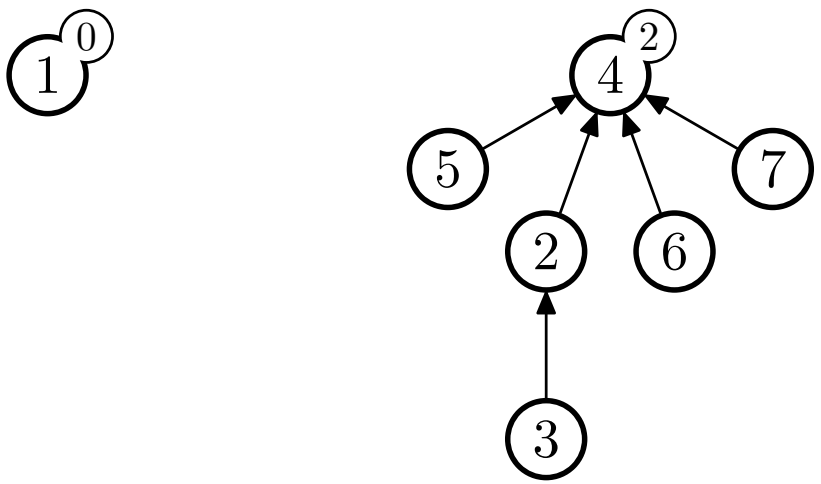
Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$, *union*$(3, 4)$



Union-Find: union by rank

union(a, b): Verwende die ranghöhere Wurzel

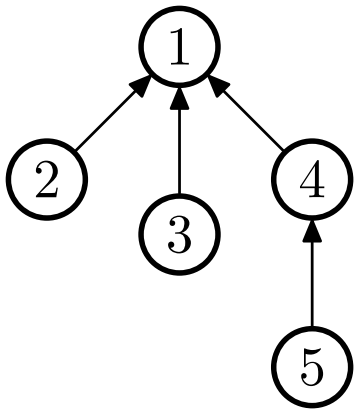
Beispiel: *union*$(5, 4)$, *union*$(3, 2)$, *union*$(5, 6)$, *union*$(4, 7)$, *union*$(3, 4)$



Union-Find: Pfadkompression

find(*a*): Komprimiere durchlaufene Pfade

Beispiel: *find*(4)



Union-Find

Algorithmus

procedure Make_Set(x) :
p[x] := x;
rank[x] := 0

- Wir betrachten jeweils eine Menge $\{0, \ldots, n-1\}$
- Ein Array für die Eltern eines für den Rang
- Ein Baum pro Menge repräsentiert durch die Wurzel
- Wurzel hier kurzgeschlossen

Union-Find

Algorithmus

```
function Find_Set(x) :
if x ≠ p[x] then p[x] := Find − Set(p[x]) fi;
return p[x];
```

Algorithmus

```
procedure Union(x, y) :
x := Find_Set(x);
y := Find_Set(y);
if rank[x] > rank[y] then p[y] := x
  else p[x] := y;
    if rank[x] = rank[y] then rank[y]++fi;
fi;
```