Algorithmen auf Graphen

Andreas Hohenauer

Algorithmen und Datenstrukturen

14. September 2025

Traversierung von Bäumen

Wir betrachten im Folgenden Algorithmen zur Traversierung von Bäumen. Diese können jedoch auch zur Traversierung von Graphen im Allgemeinen verwendet werden.

Ziel: Wir wollen die Knoten eines Baumes systematisch durchlaufen, mit dem Ziel einen bestimmten Knoten zu finden.

- **Breitensuche (Breadth-First-Search (BFS)):** In jedem Schritt werden zunächst alle Nachbarknoten eines Knoten besucht, bevor von dort aus weitere Pfade gebildet werden.
- **Tiefensuche (Depth-First-Search (DFS)):** Ein Pfad wird vollständig in die Tiefe durchlaufen, bevor etwaige Abzweigungen verwendet werden.

Traversierung von Bäumen

Zur Beschreibung der Suchverfahren werden folgende Begriffe benötigt:

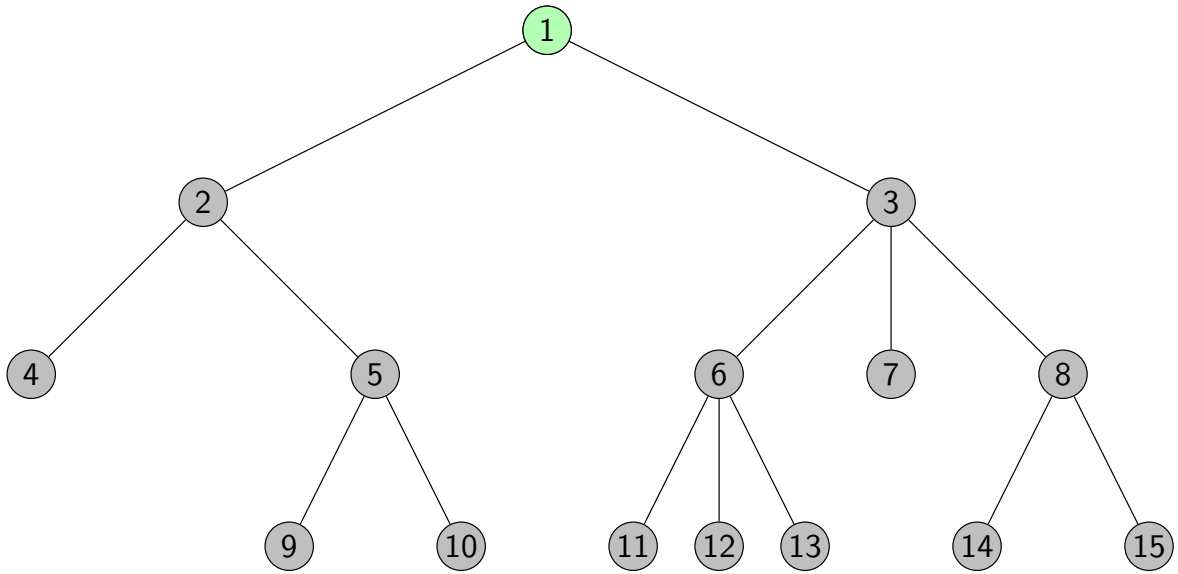
- Ein Knoten wird **entdeckt**, wenn er das erste Mal besucht wird.
- Ein Knoten wird **fertiggestellt/abgeschlossen**, wenn er das letzte Mal verlassen wird.

Für manche Anwendungen ist es wichtig festzuhalten, wann ein Knoten *entdeckt*, bzw. abgeschlossen wurde. Dazu führen wir einen Zähler τ mit, der in jedem Schritt um 1 erhöht wird.

- Wird ein Knoten v das erste mal besucht ("entdeckt"), so setzen wir $\tau_d(v) = \tau$
- Wird ein Knoten v das letzte mal verlassen ("abgeschlossen"), so setzen wir $\tau_f(v) = \tau$
- Unmittelbar danach wird der Zähler jeweils erhöht: $\tau = \tau + 1$

Breitensuche

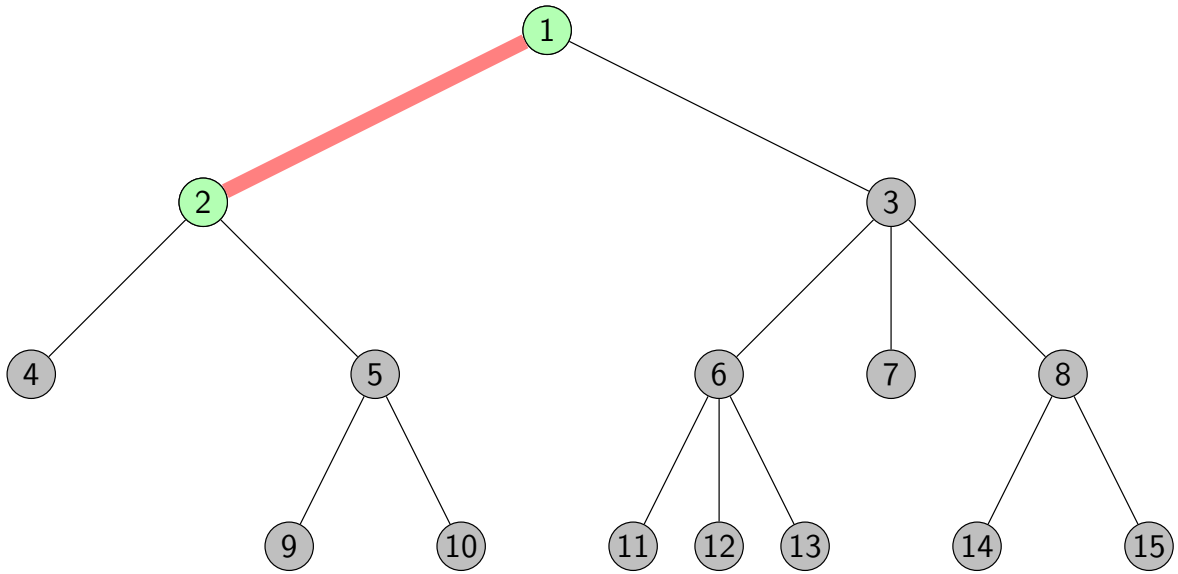
Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.



$\tau_d(1) = 1,$

Breitensuche

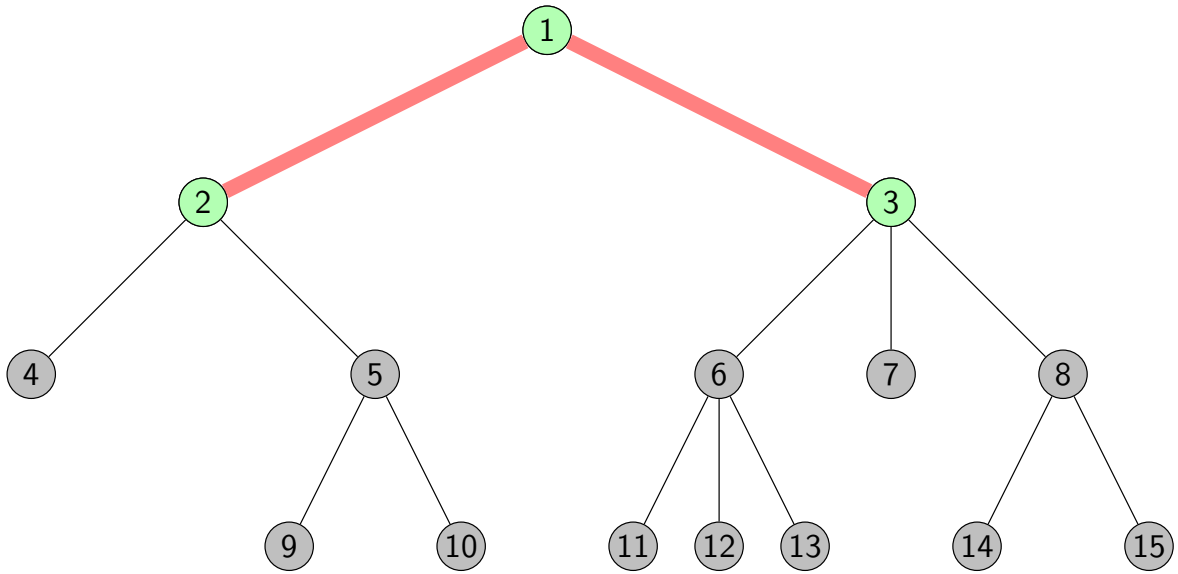
Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.



$\tau_d(1) = 1, \tau_d(2) = 2,$

Breitensuche

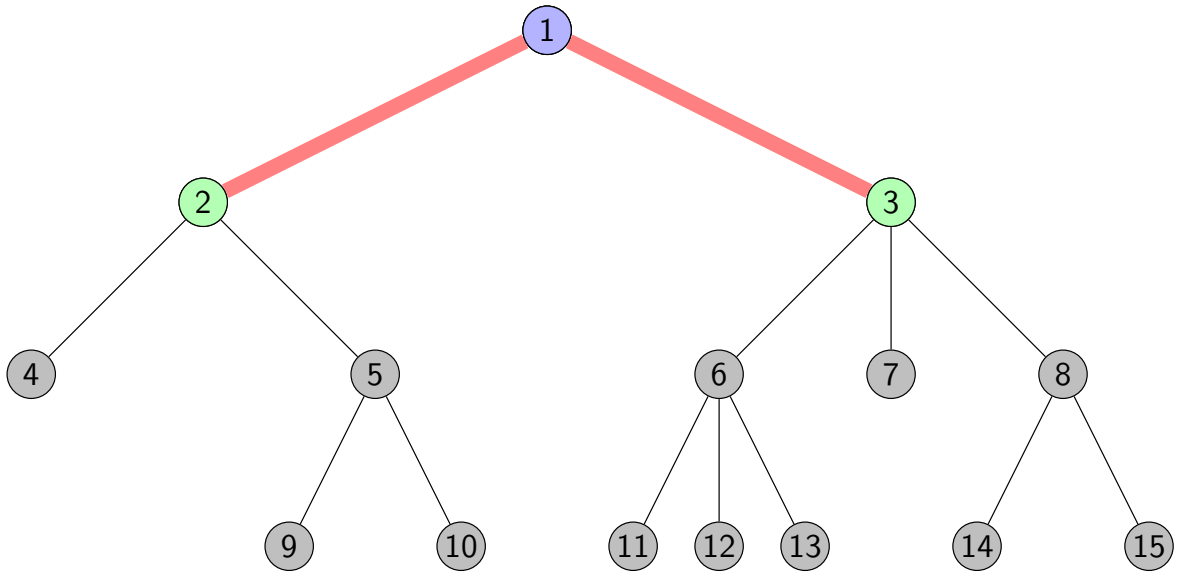
Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.



$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3,$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

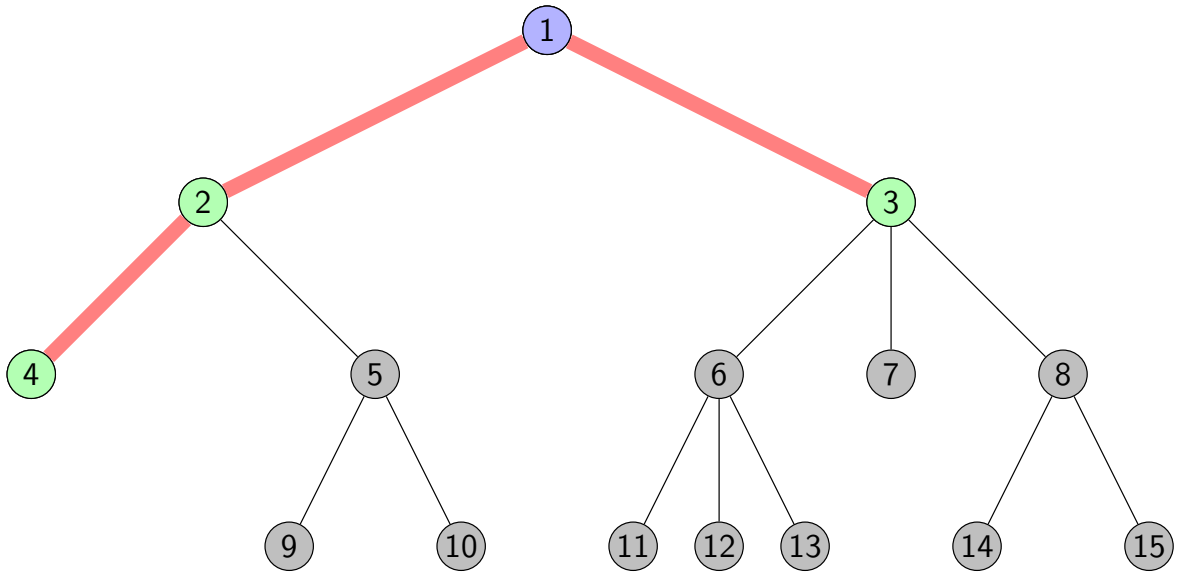


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3,$

$\tau_f(1) = 4,$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

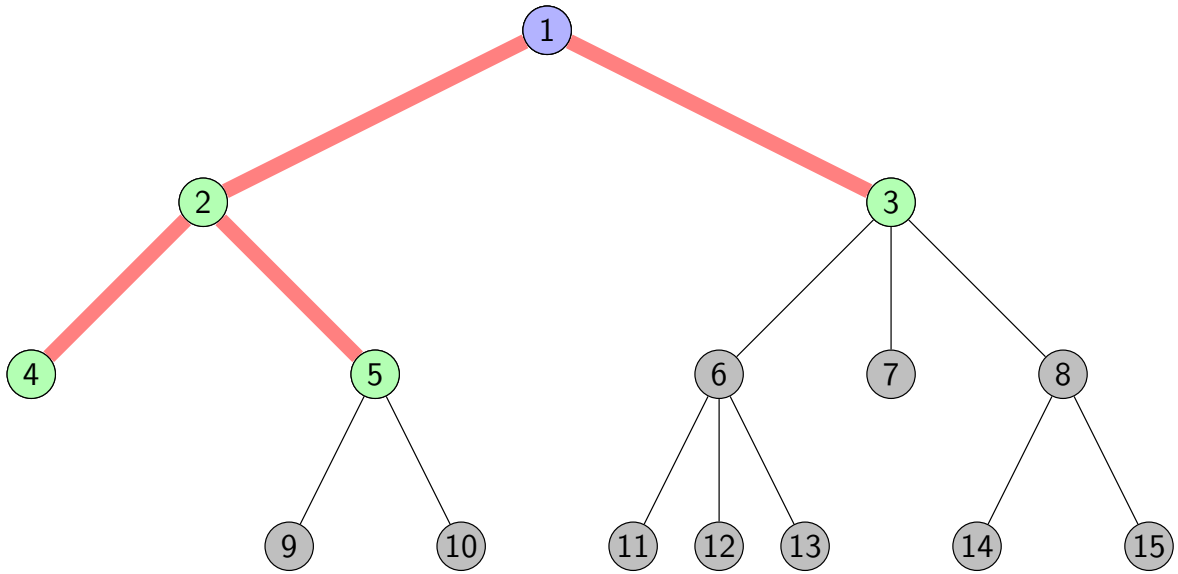


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5,$

$\tau_f(1) = 4,$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

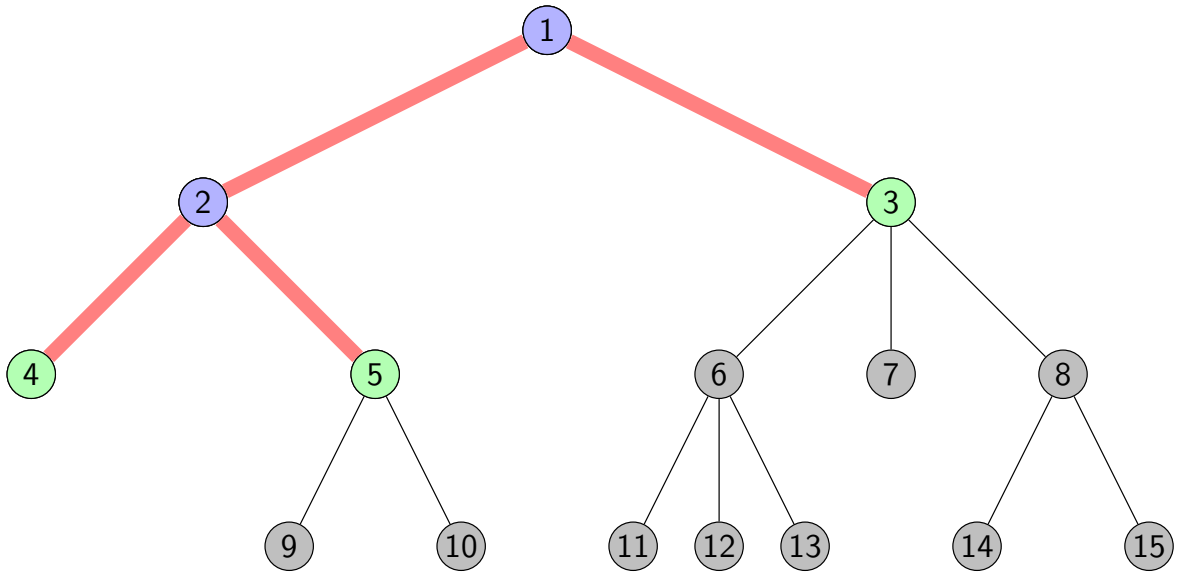


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6,$

$\tau_f(1) = 4,$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

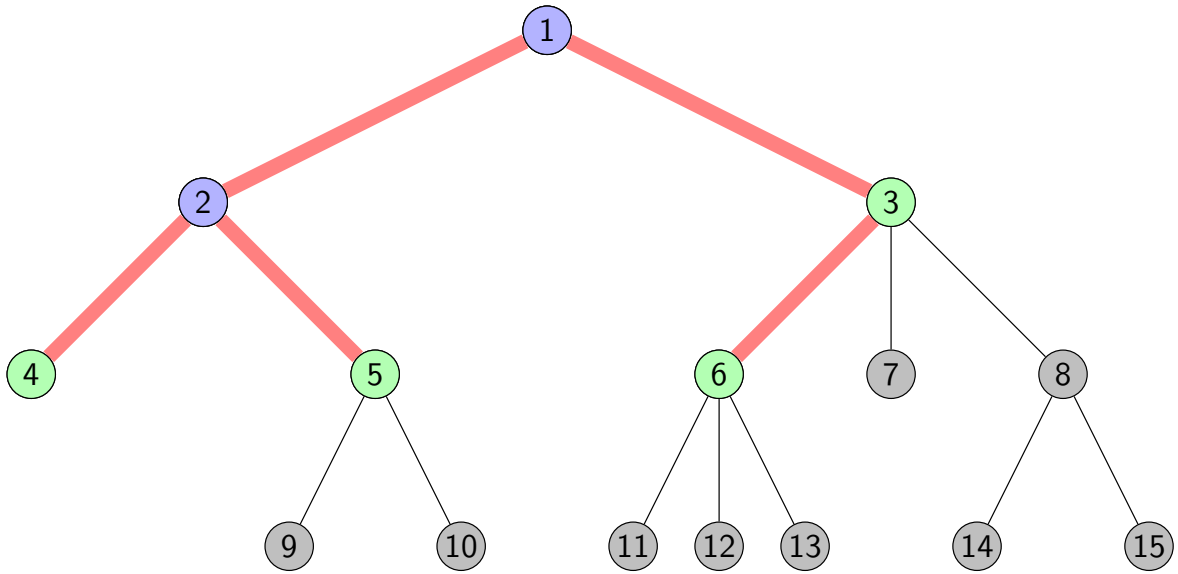


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6,$

$\tau_f(1) = 4, \tau_f(2) = 7,$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

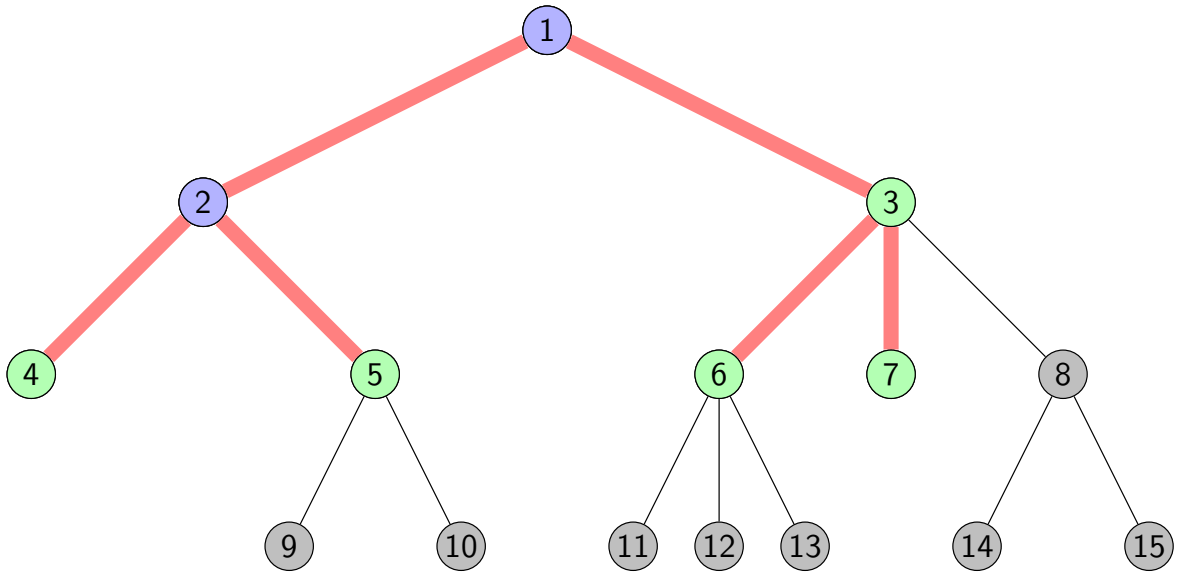


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8,$

$\tau_f(1) = 4, \tau_f(2) = 7,$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

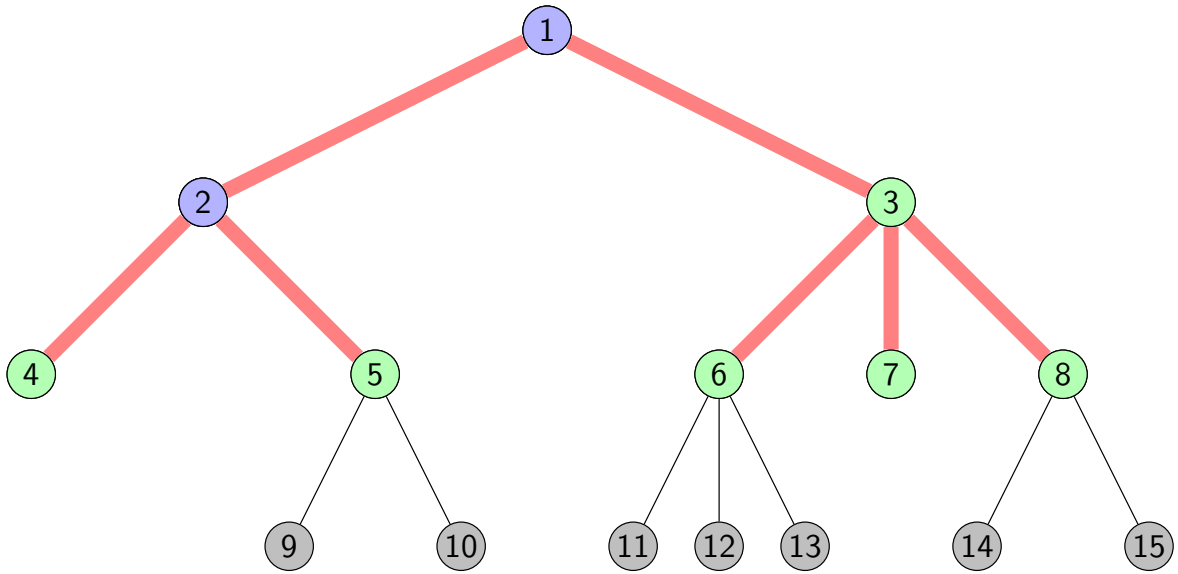


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9,$

$\tau_f(1) = 4, \tau_f(2) = 7,$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

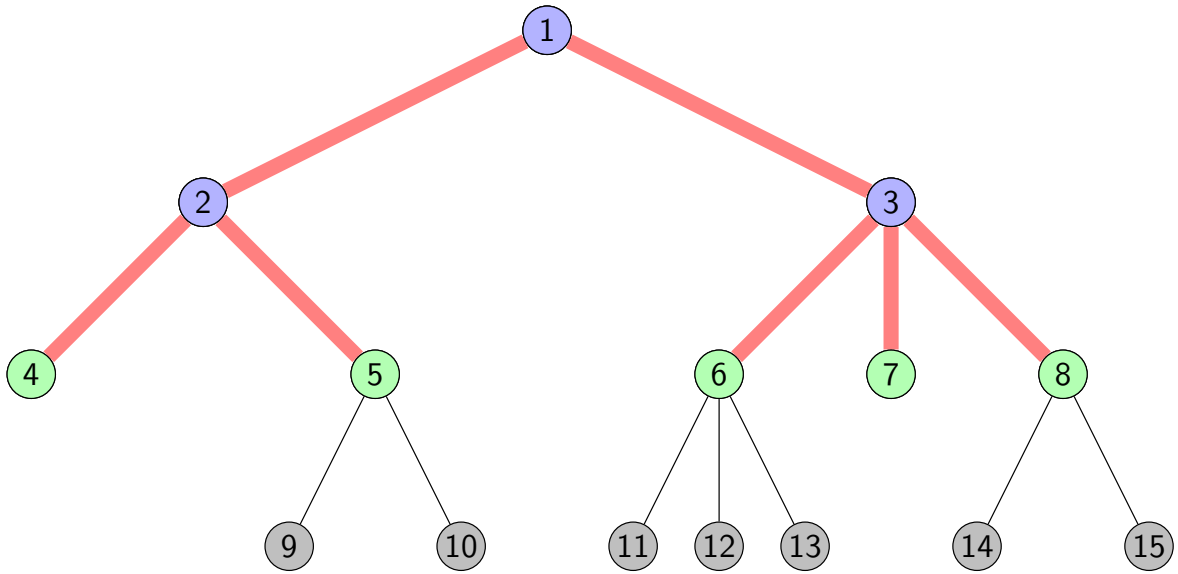


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10,$

$\tau_f(1) = 4, \tau_f(2) = 7,$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

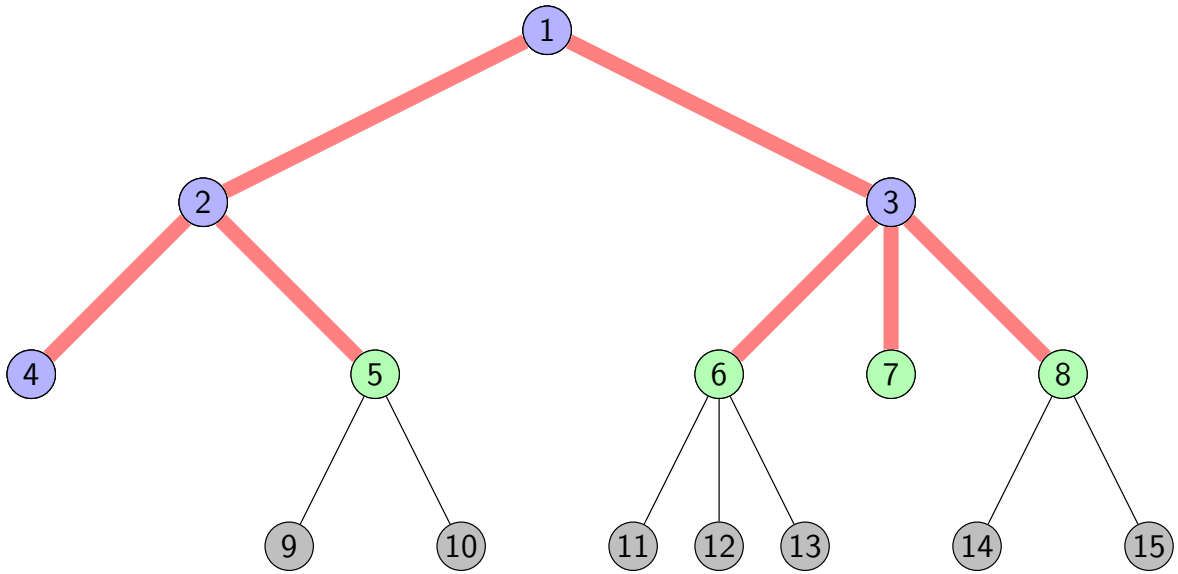


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10,$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11,$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

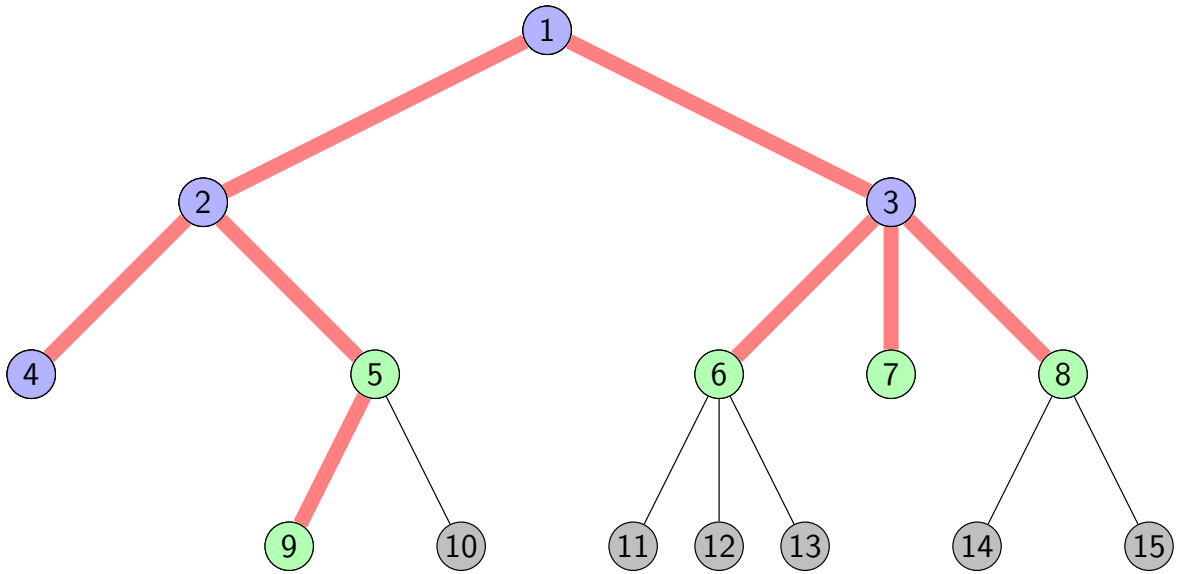


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10,$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12,$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

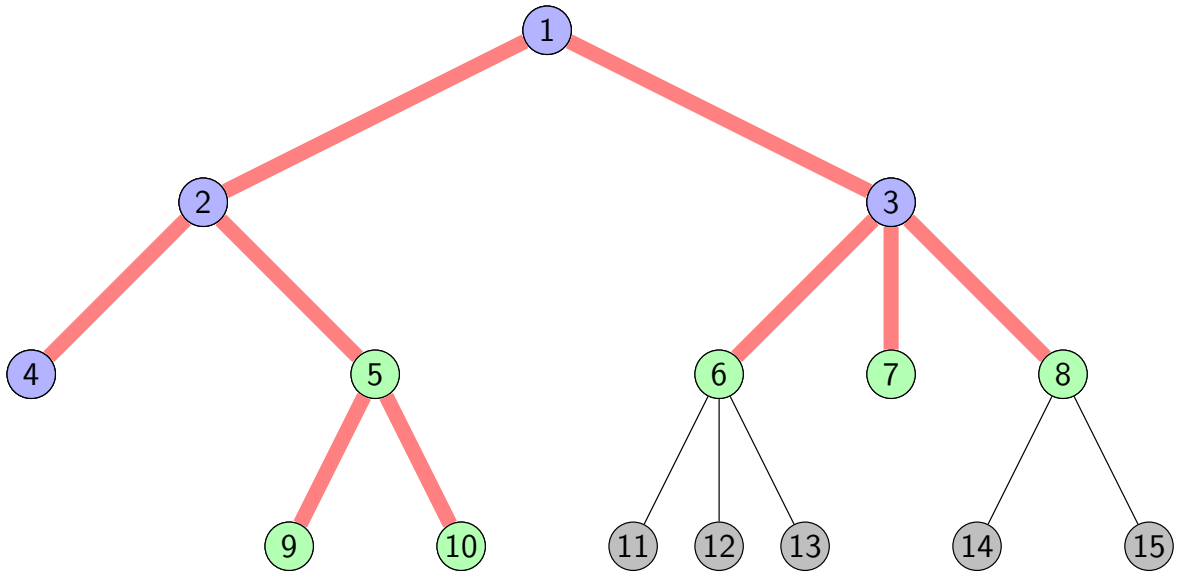


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13,$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12,$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

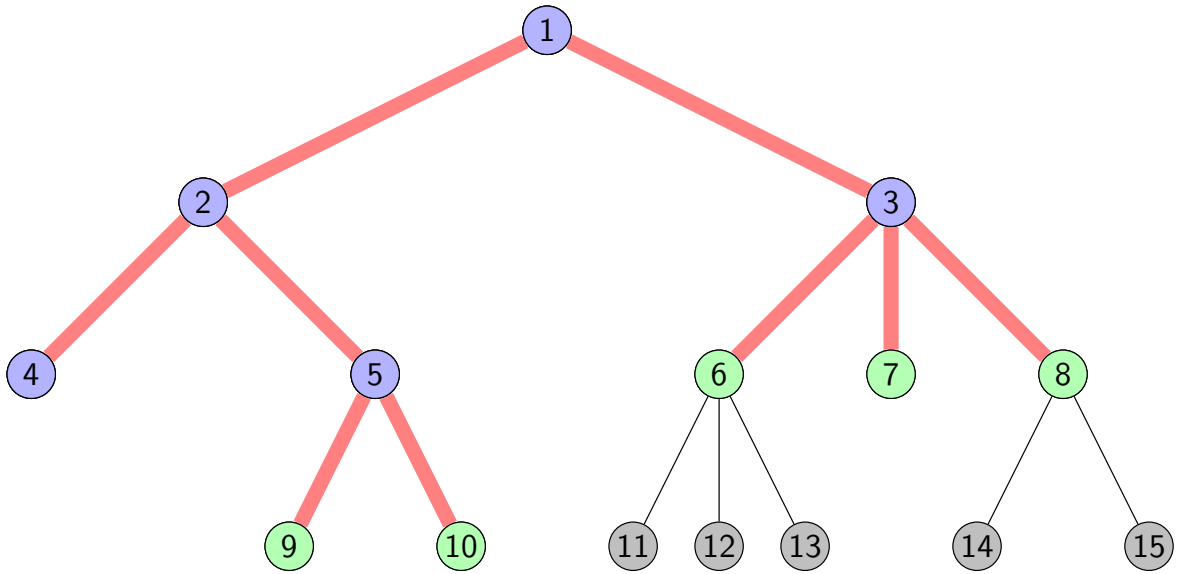


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14,$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12,$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

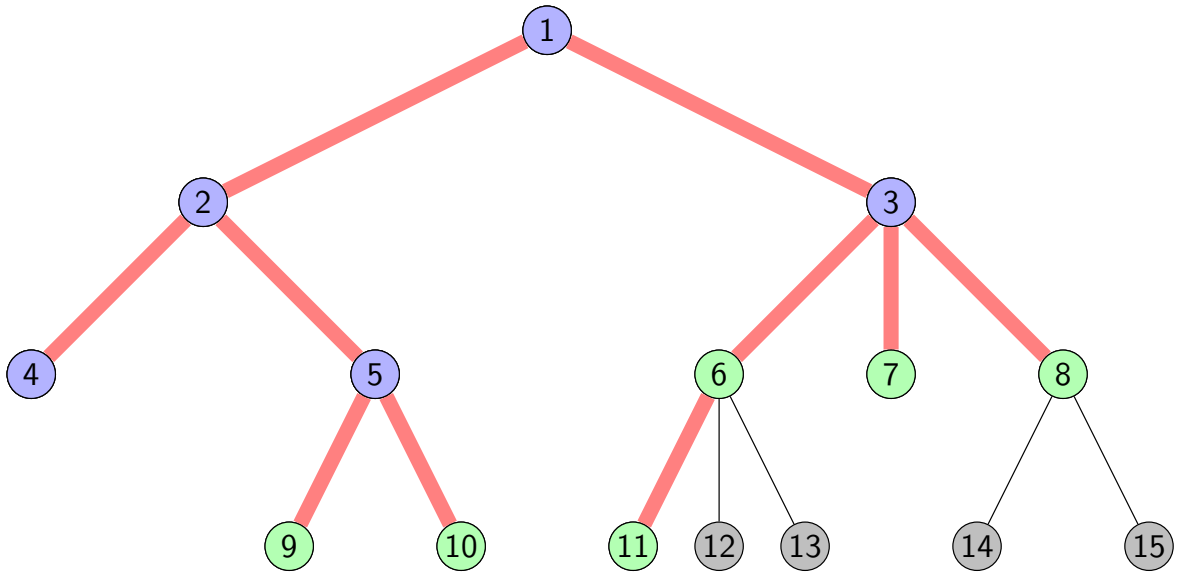


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14,$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15,$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

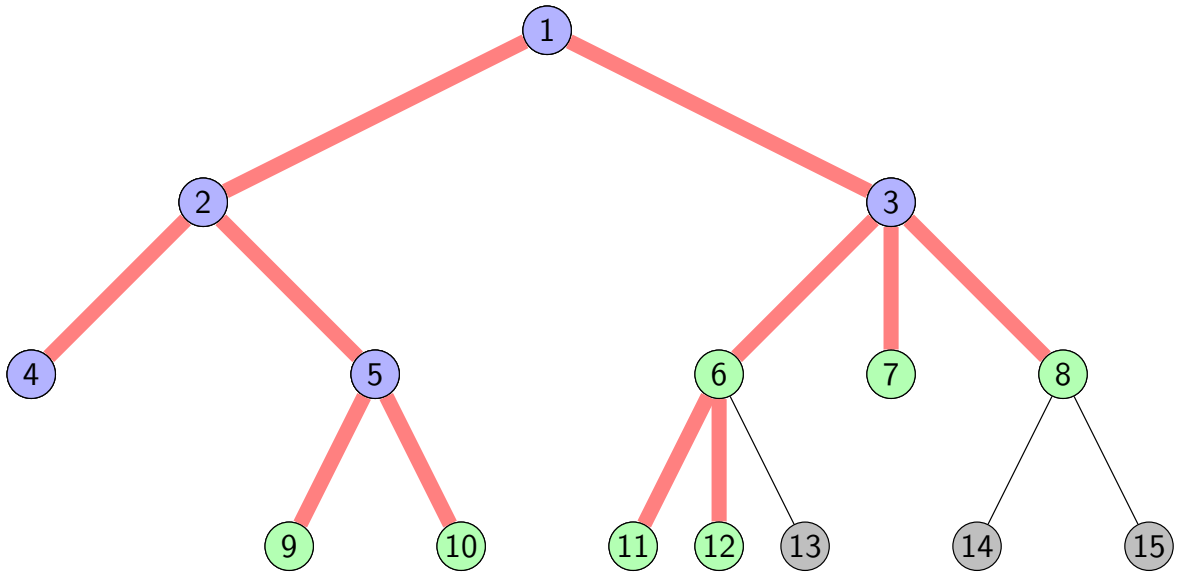


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

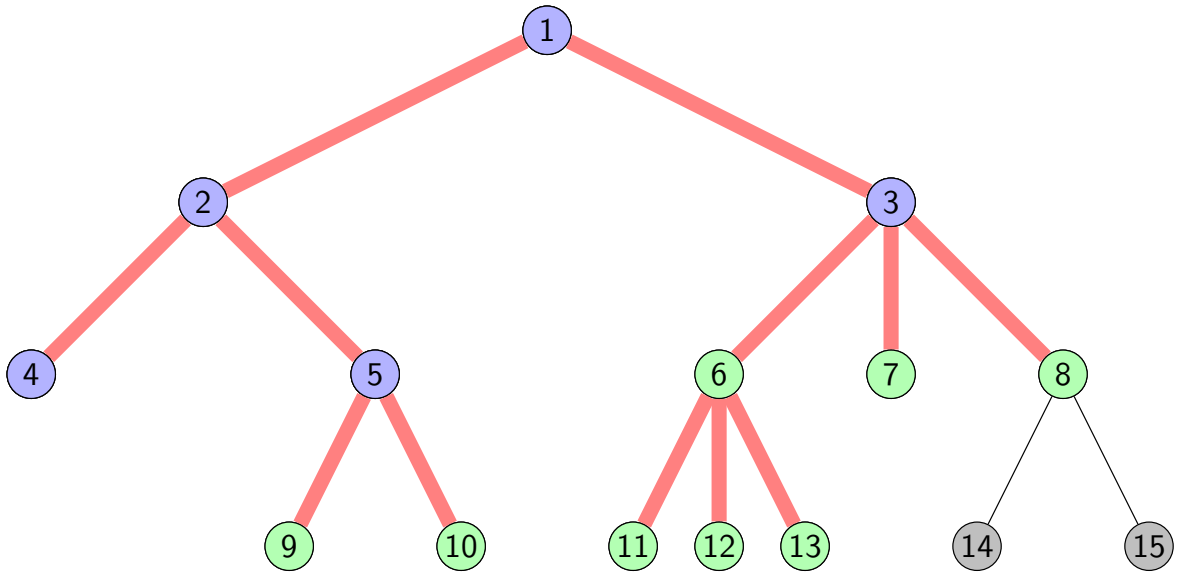


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

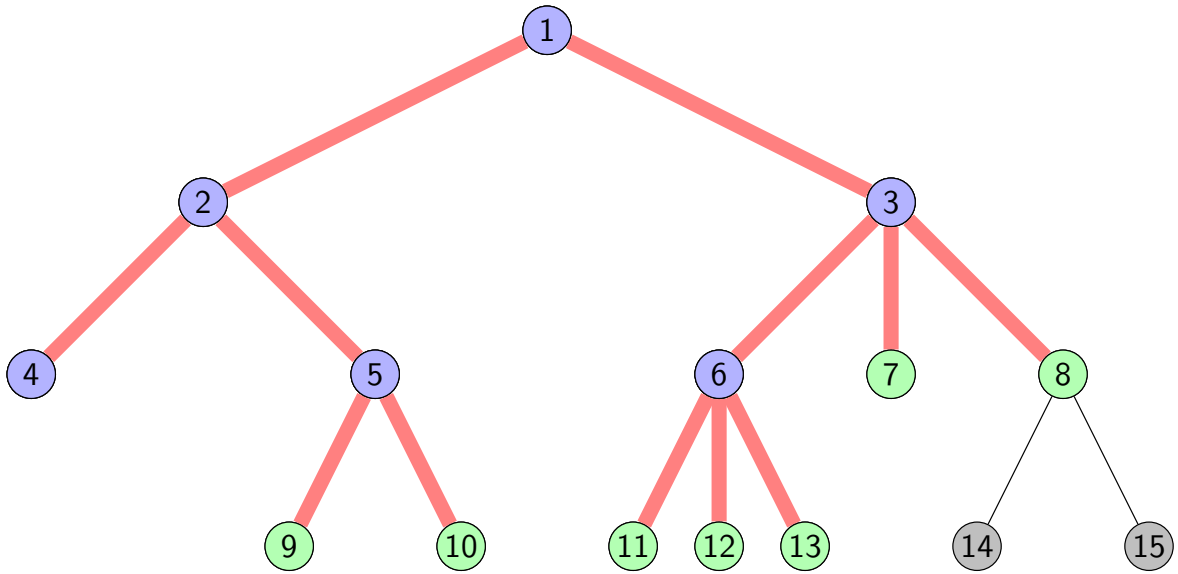


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

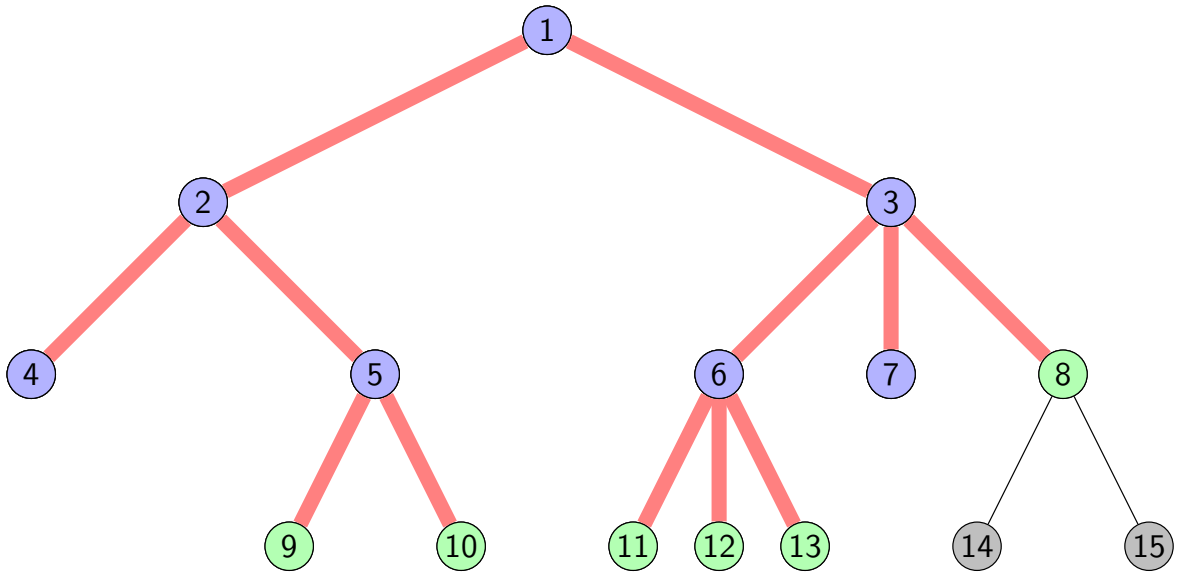


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

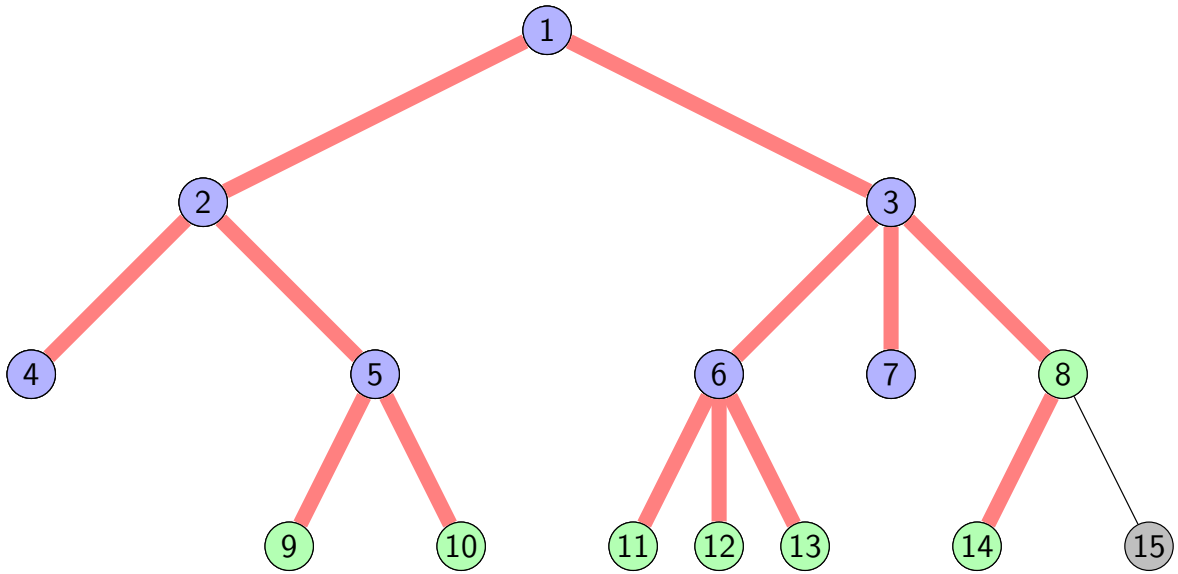


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

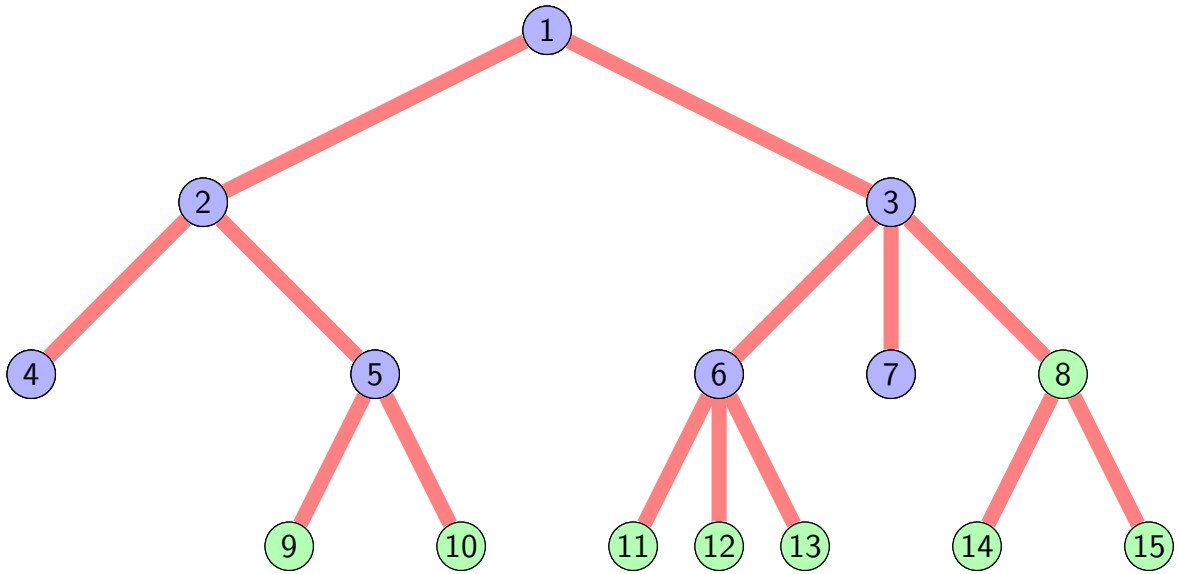


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

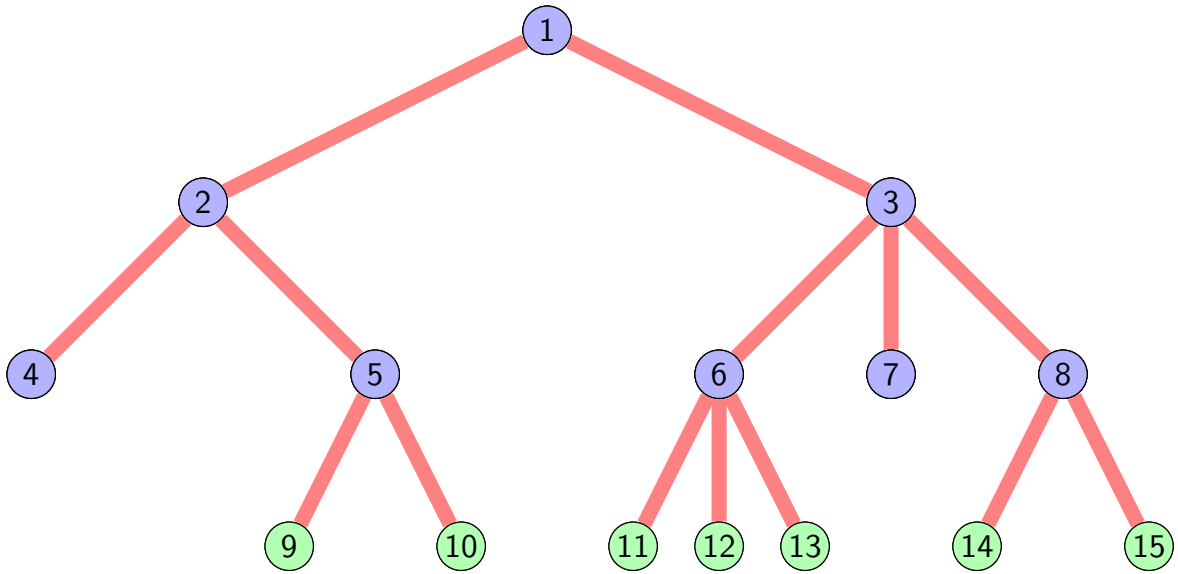


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

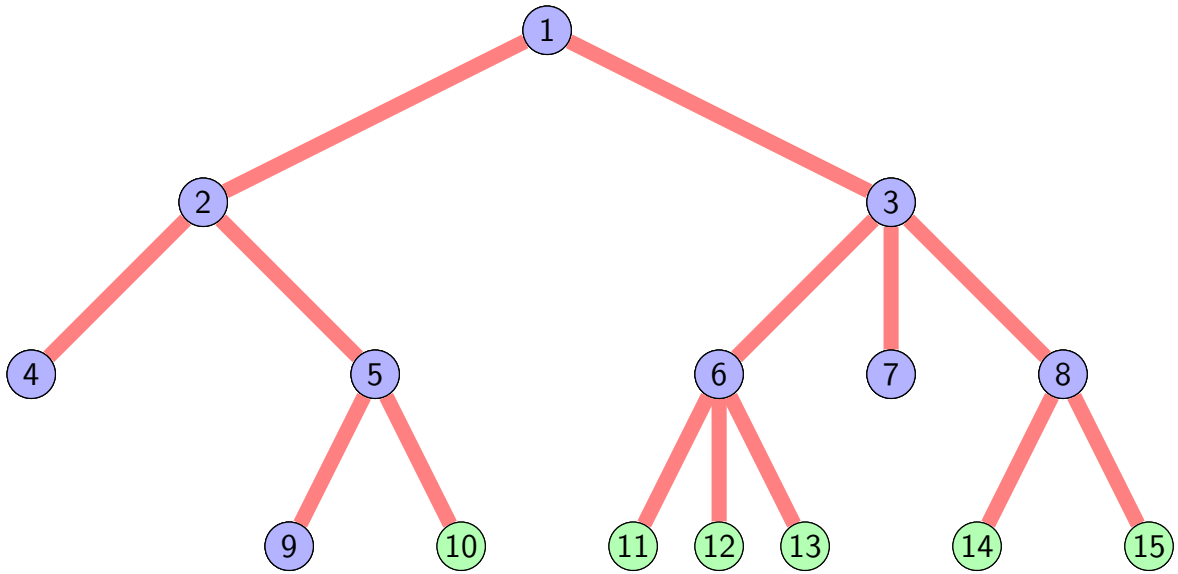


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

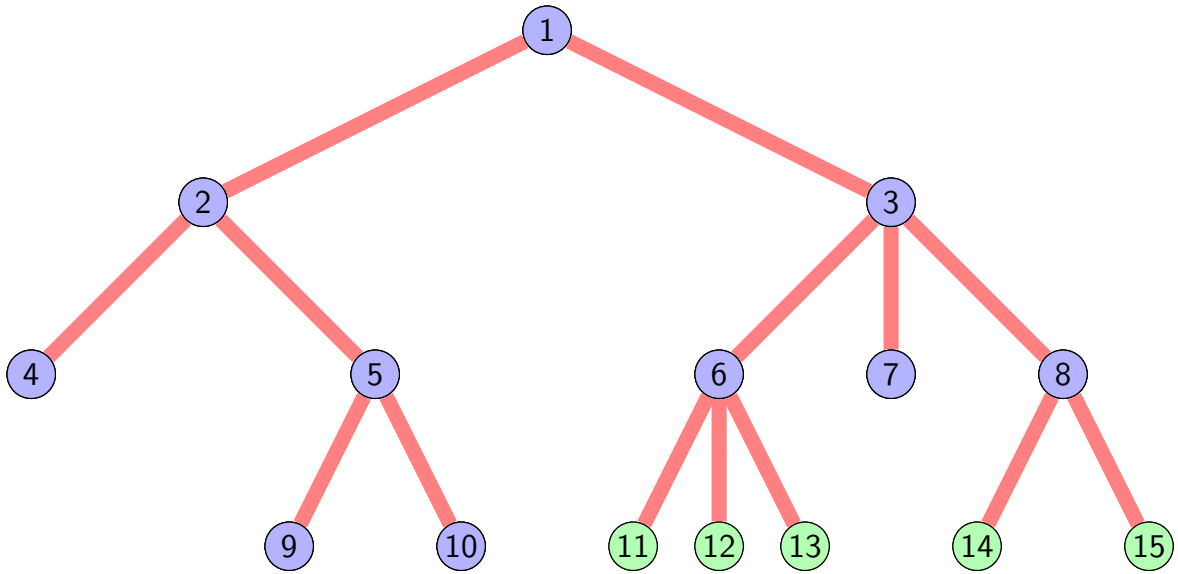


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

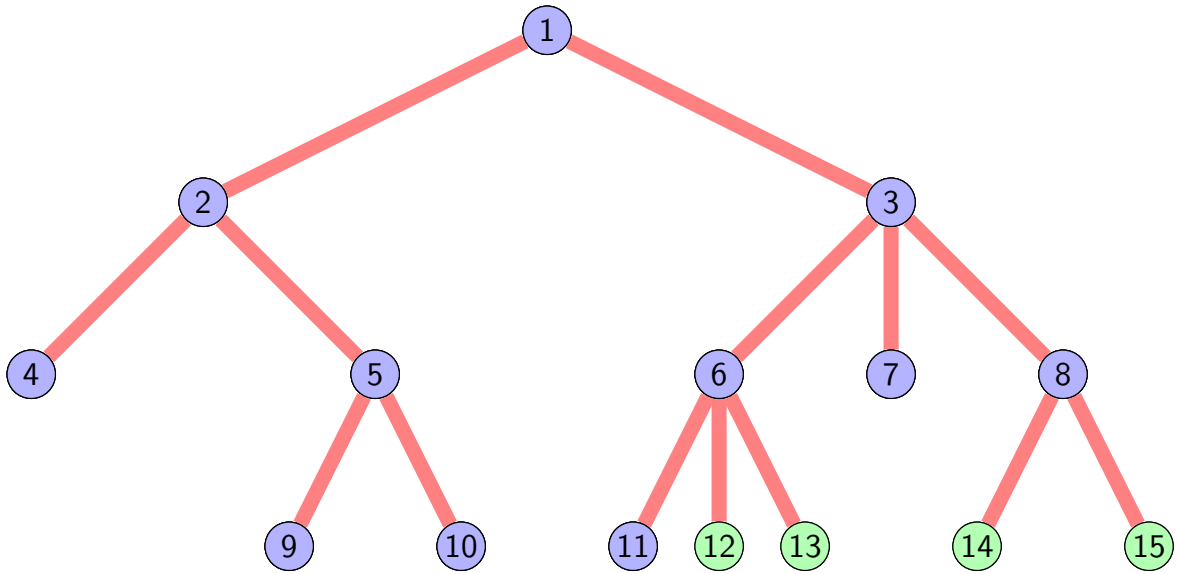


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

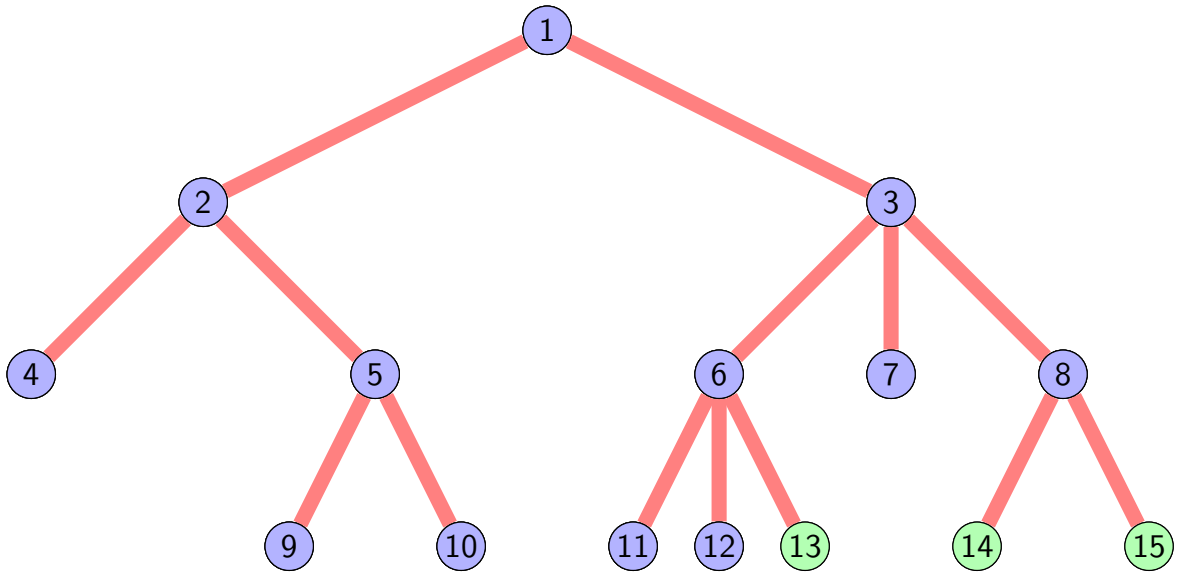


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

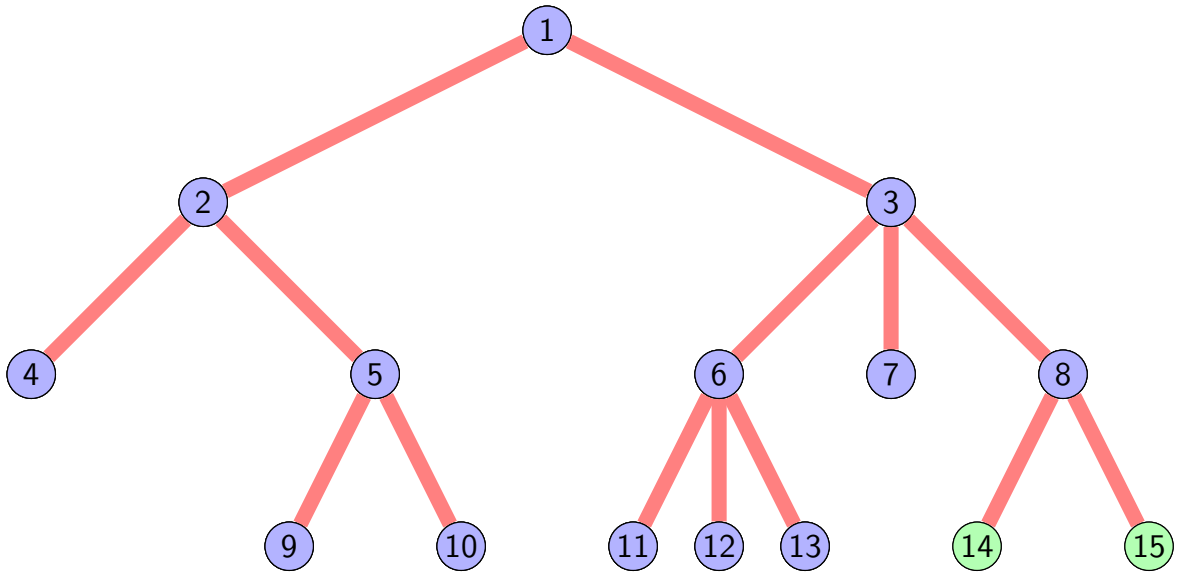


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

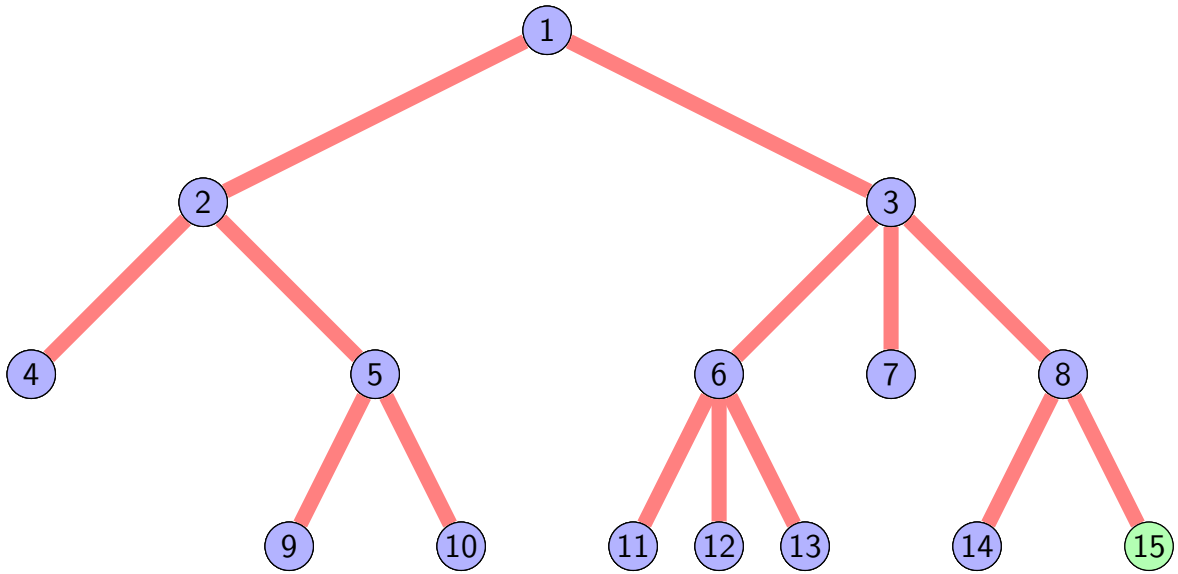


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.

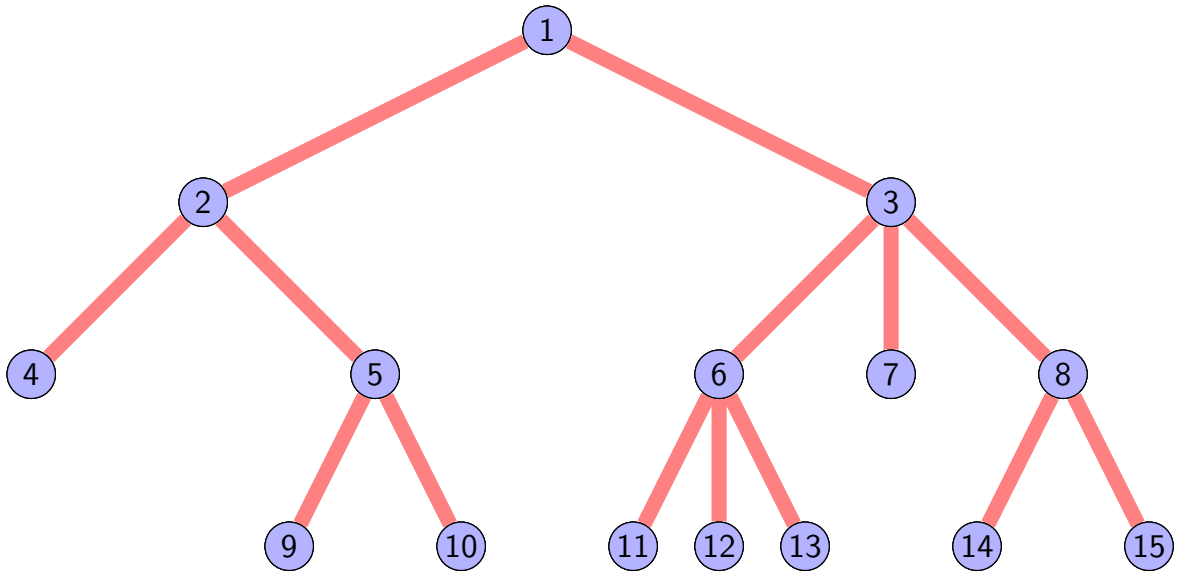


$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche

Beispiel: Beispiel einer Breitensuche. Die Suche kann abgebrochen werden, sobald der gesuchte Knoten gefunden ("entdeckt") wurde.



$\tau_d(1) = 1, \tau_d(2) = 2, \tau_d(3) = 3, \tau_d(4) = 5, \tau_d(5) = 6, \tau_d(6) = 8, \tau_d(7) = 9, \tau_d(8) = 10, \tau_d(9) = 13, \tau_d(10) = 14, \ldots$

$\tau_f(1) = 4, \tau_f(2) = 7, \tau_f(3) = 11, \tau_f(4) = 12, \tau_f(5) = 15, \ldots$

Breitensuche: Algorithmus

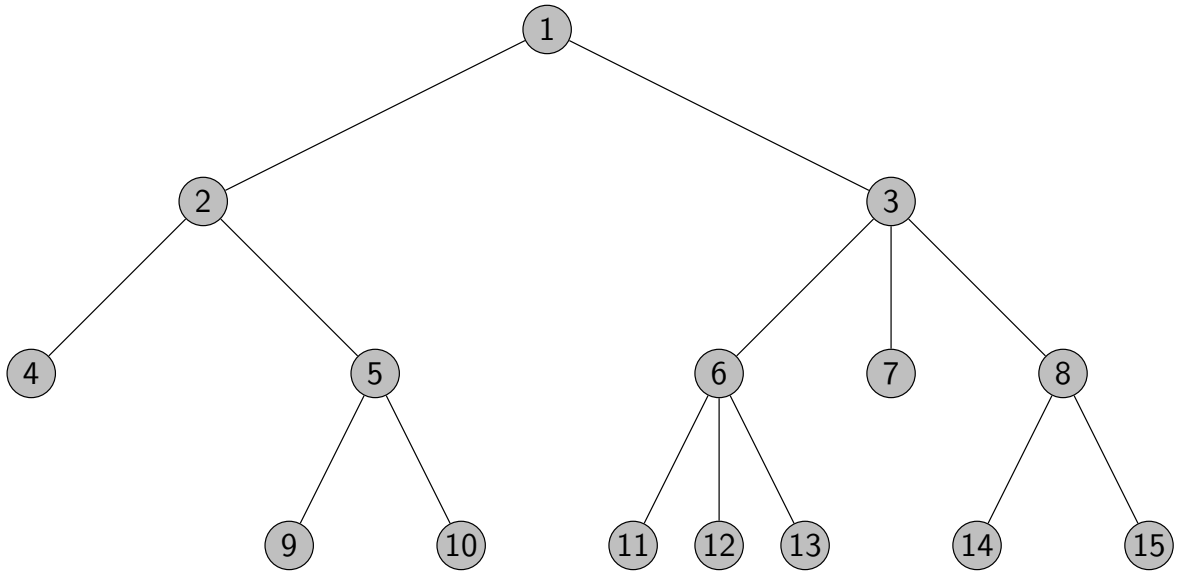
Algorithmus 1: Breitensuche (BFS)

Input: $G = (V, E)$, Startknoten v, Gesuchter Knoten s
Result: vertex $s \in V$, if exists

```
1  Function BFS(G, v, s)
2      Queue Q
3      markiere v als besucht
4      Q.enqueue(v)
5      while Q not empty do
6          u = Q.dequeue()
7          if u = s then
8              return u
9          for alle Nachbarn w von u do
10             if w noch nicht besucht then
11                 markiere w als besucht
12                 Q.enqueue(w)
13     return nicht gefunden
```

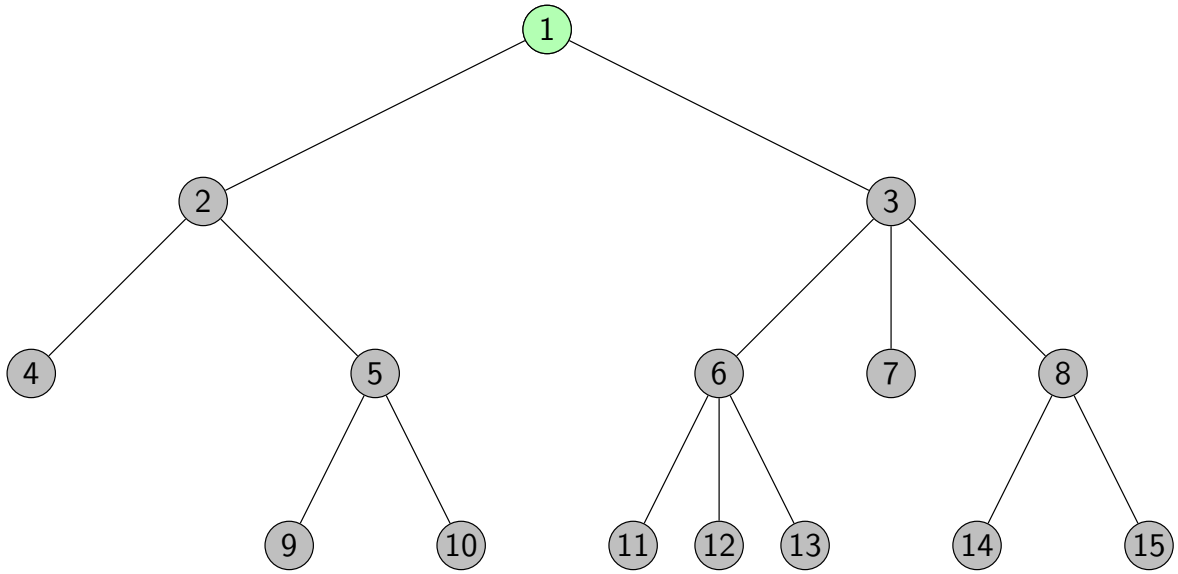
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



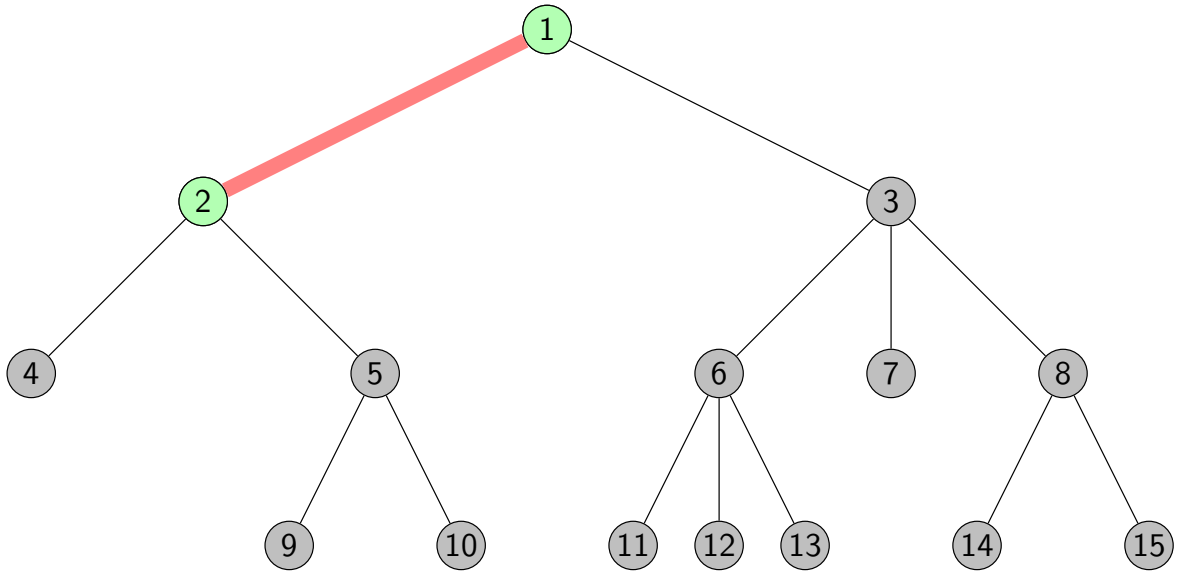
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



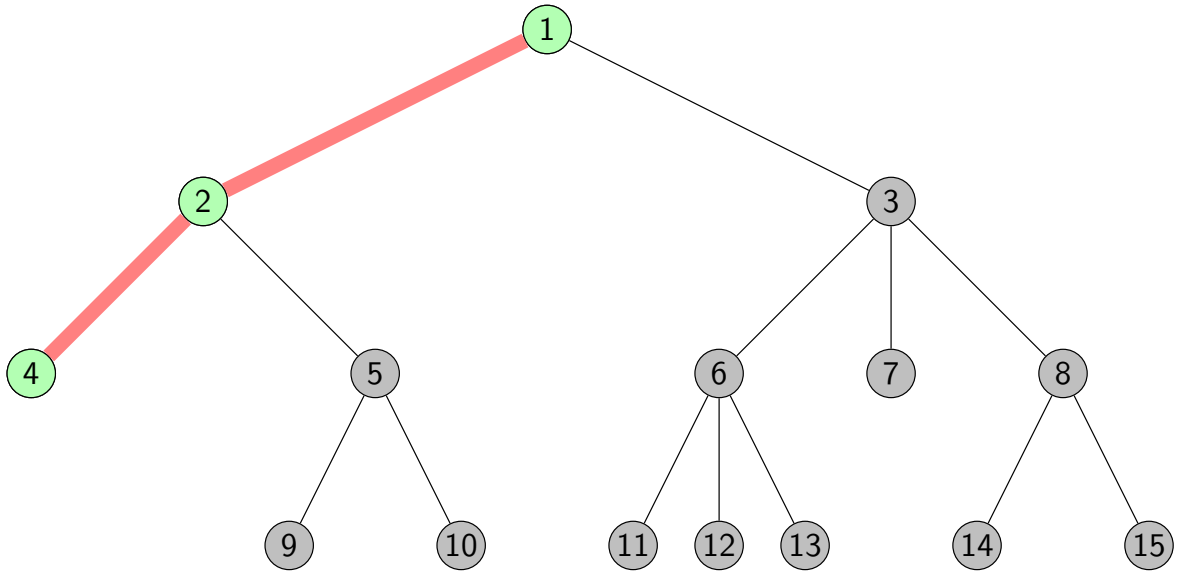
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



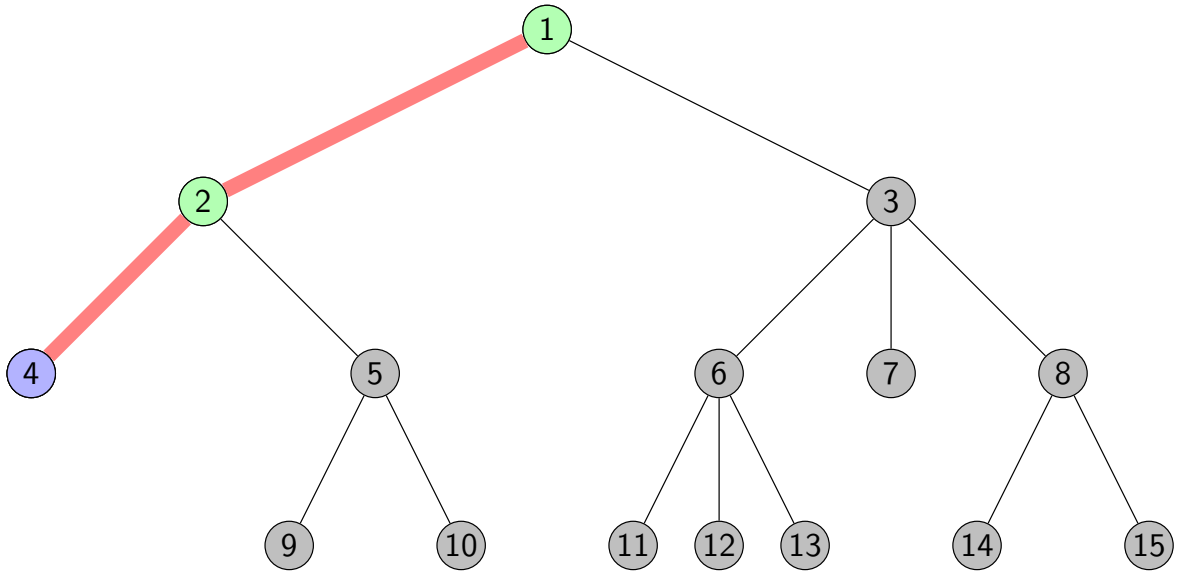
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



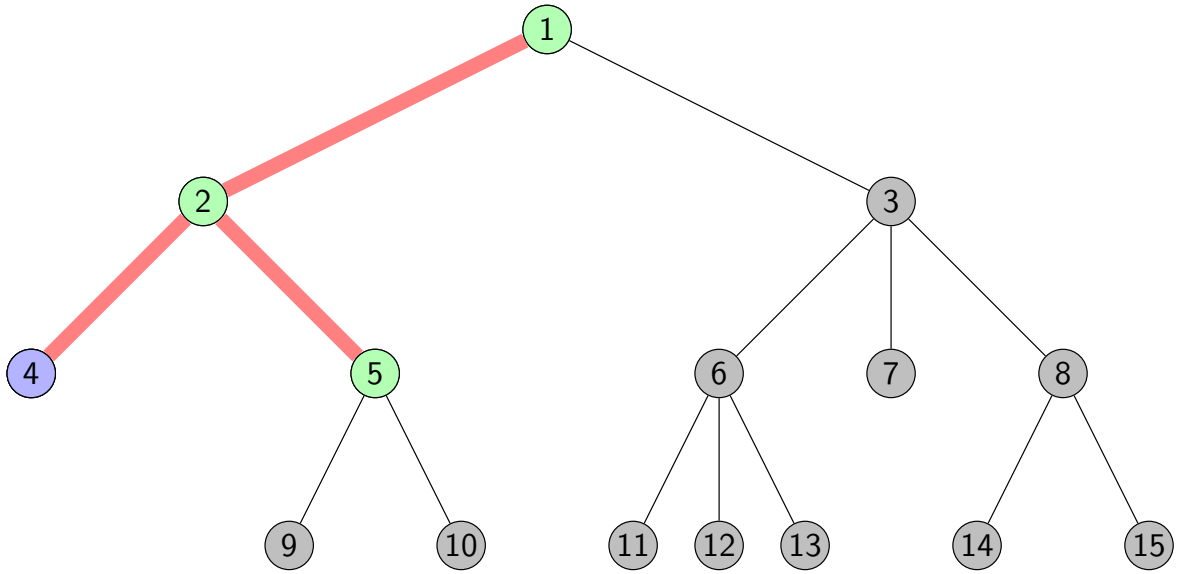
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



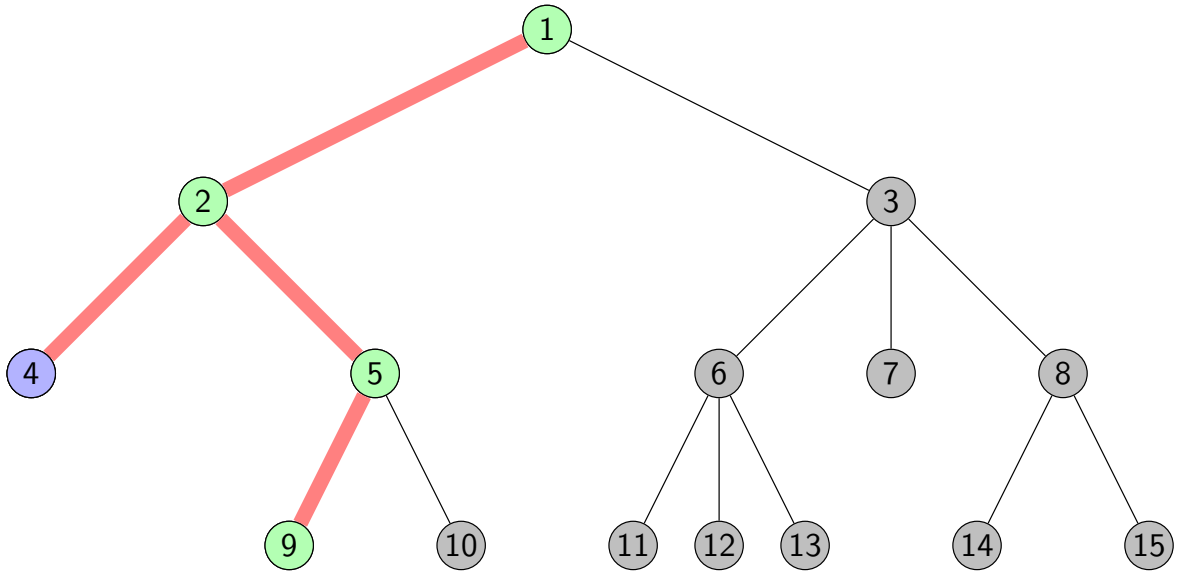
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



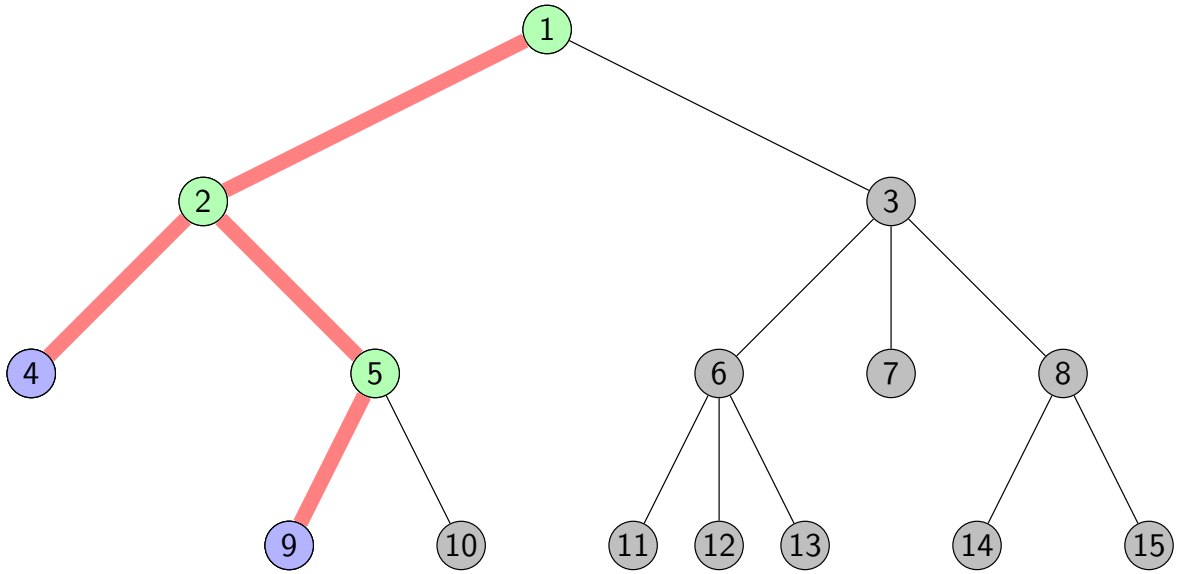
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



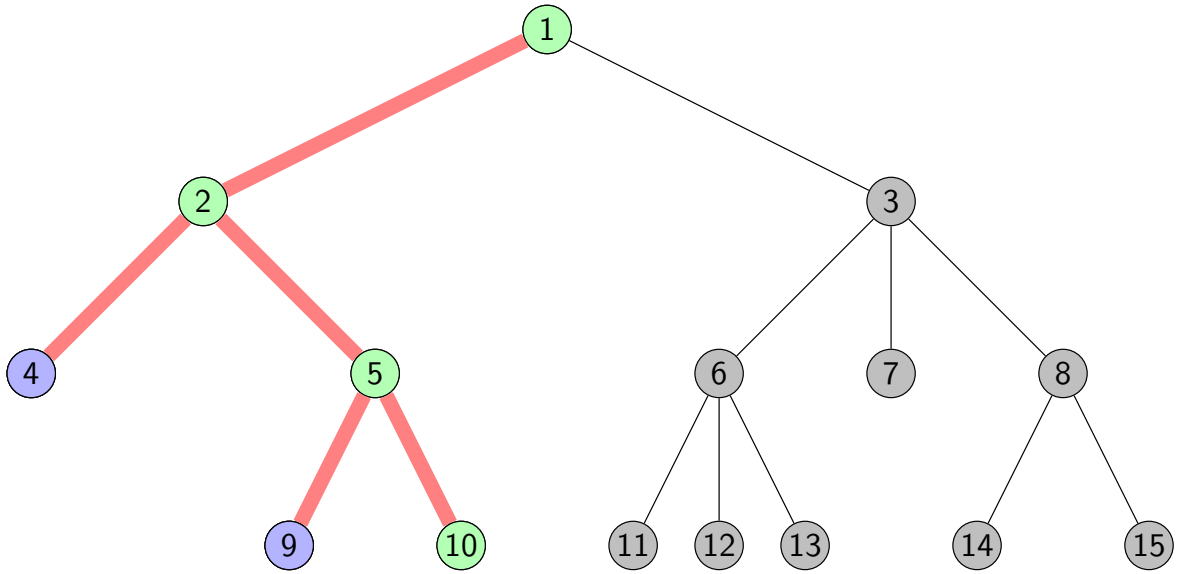
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



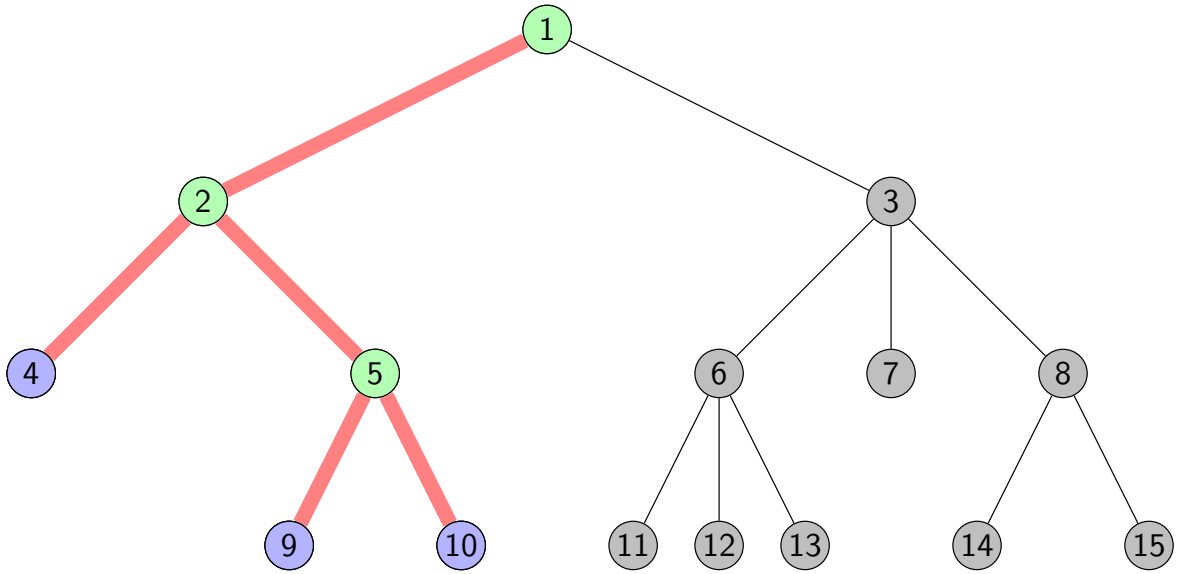
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



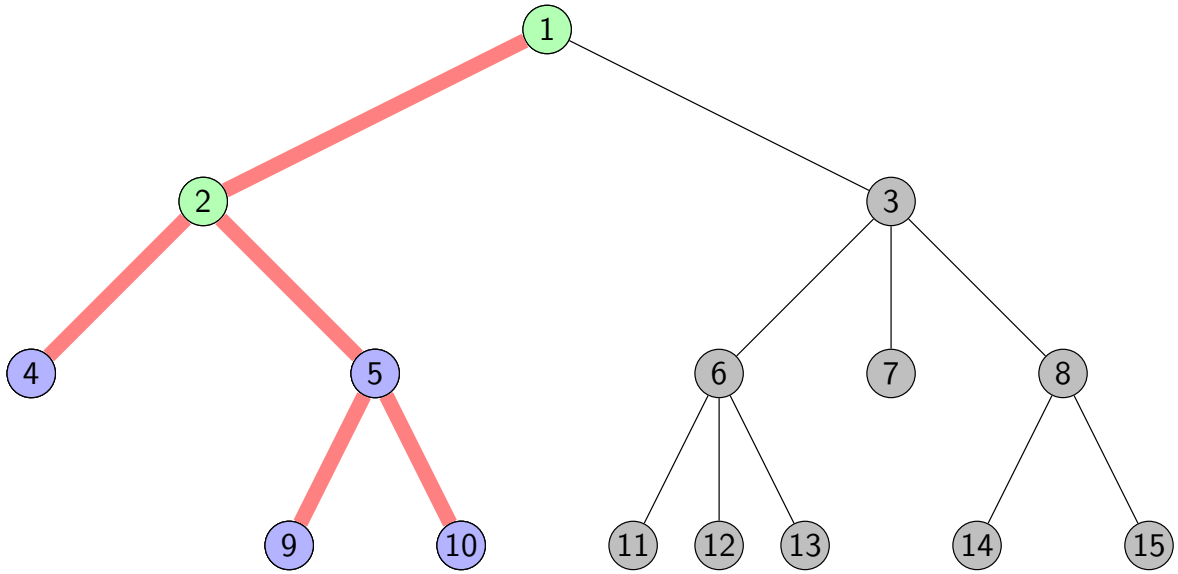
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



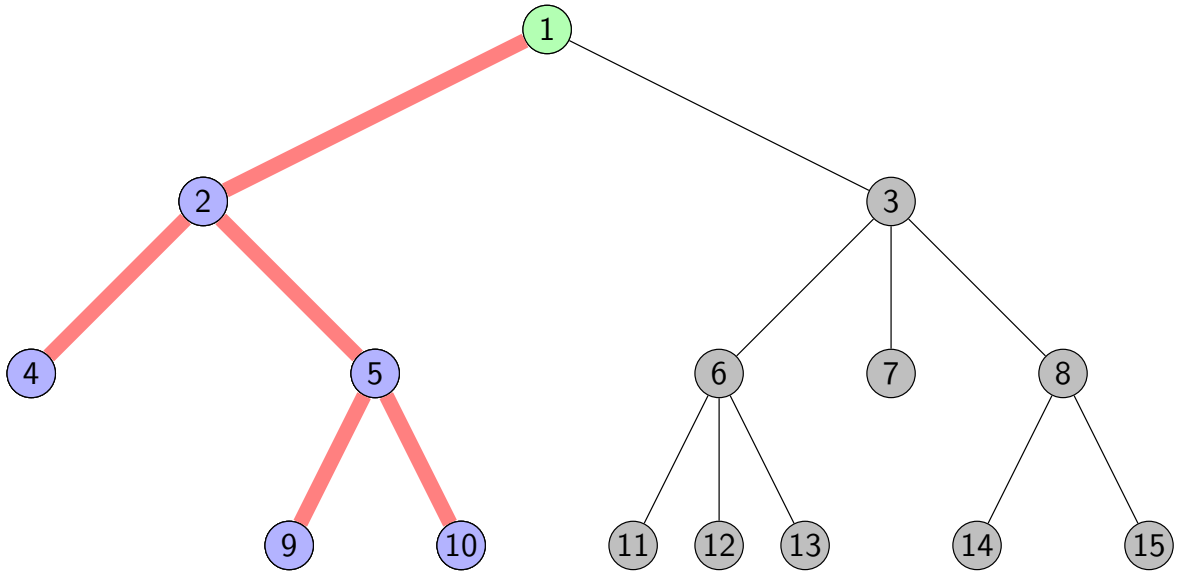
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



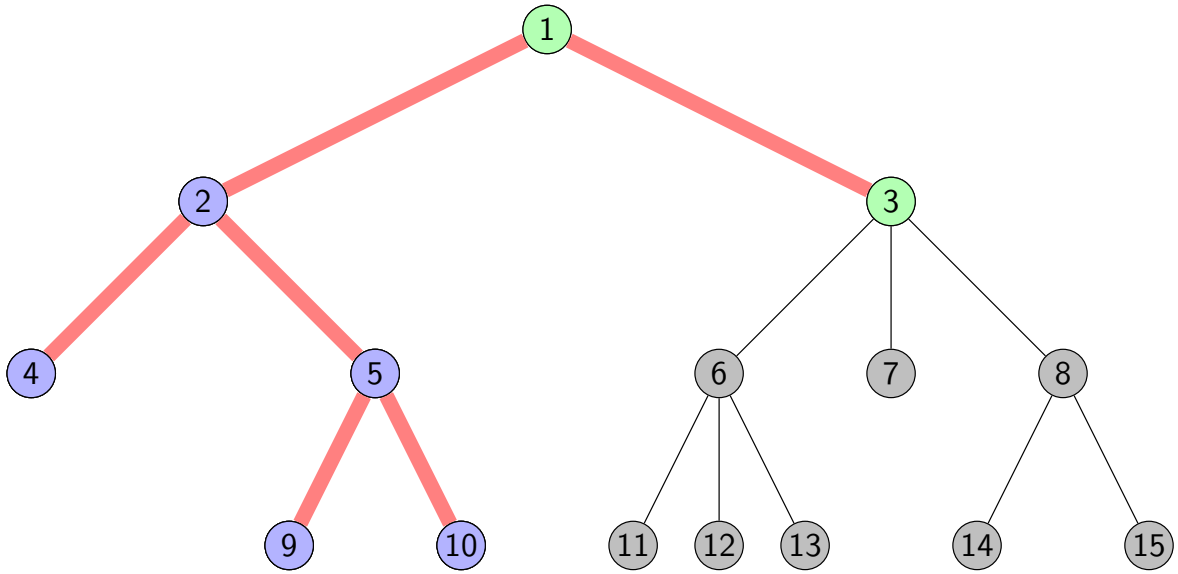
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



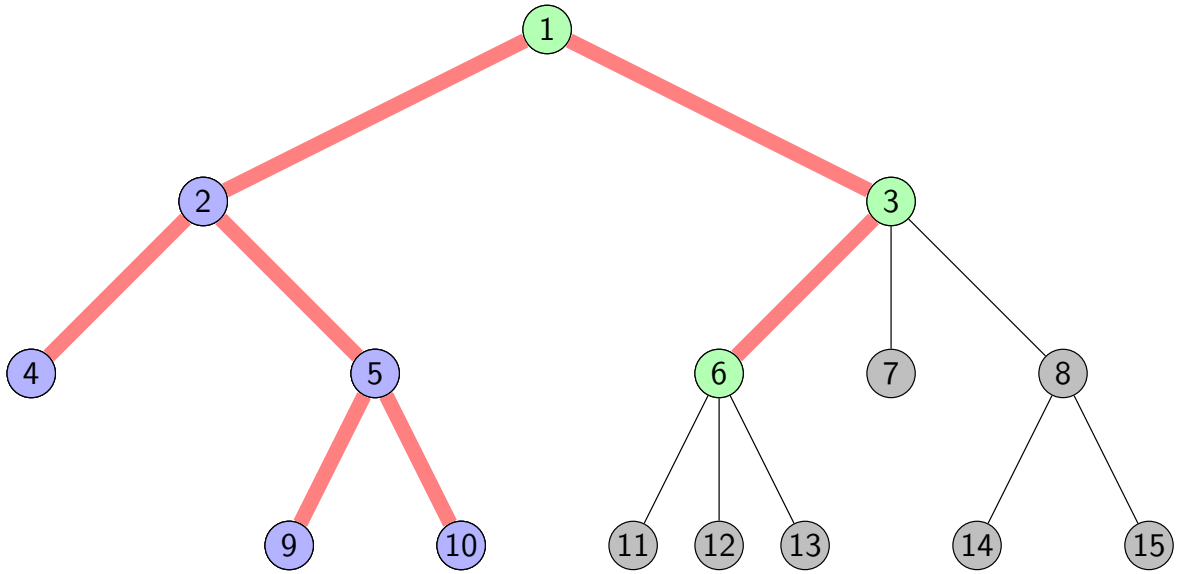
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



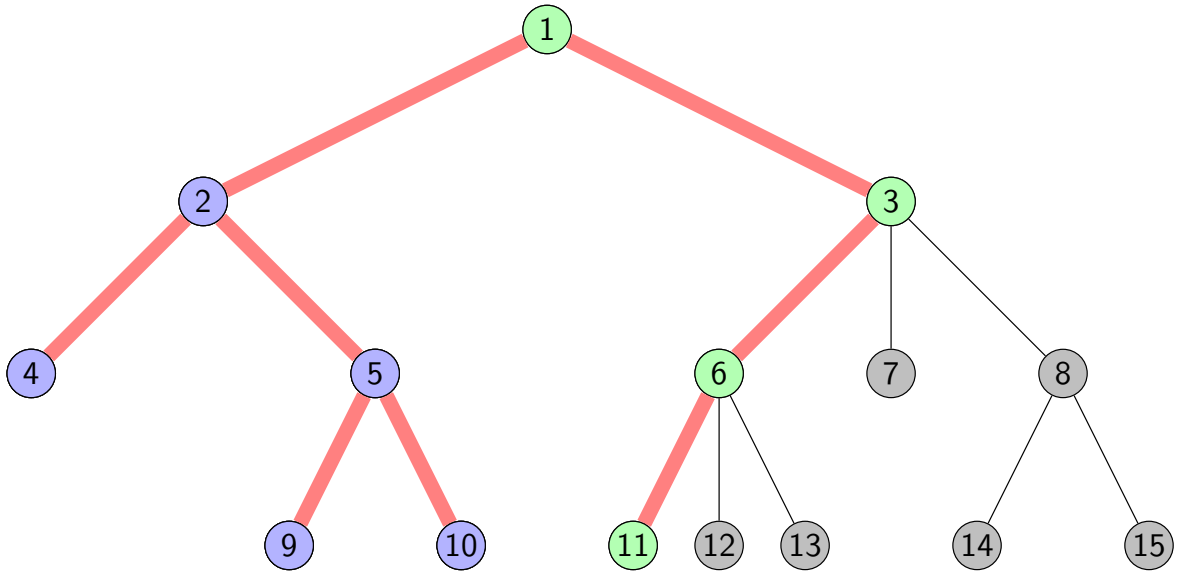
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



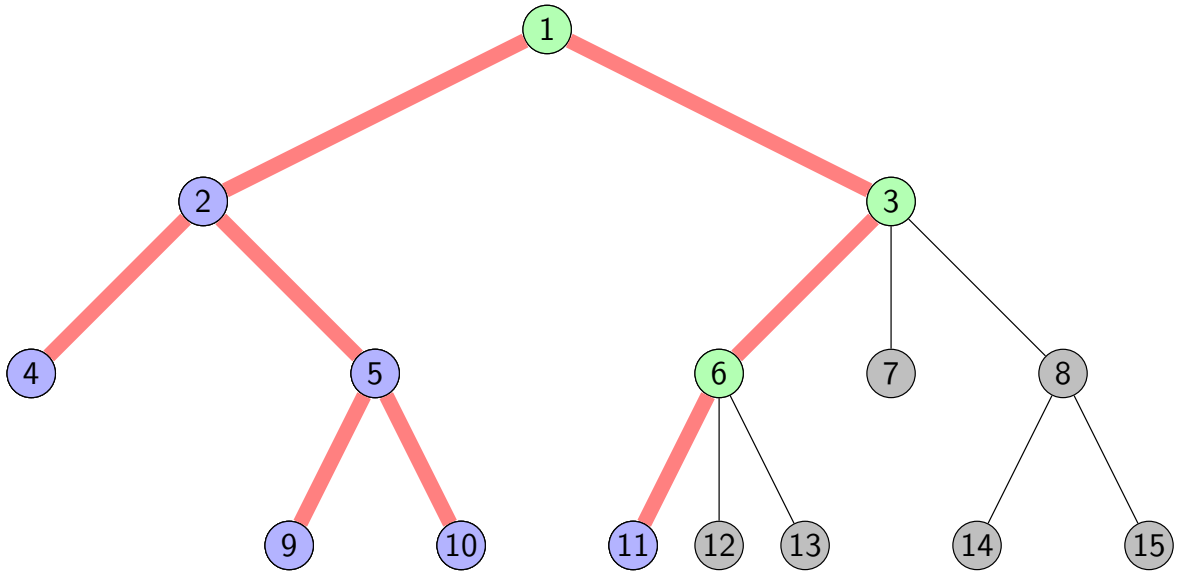
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



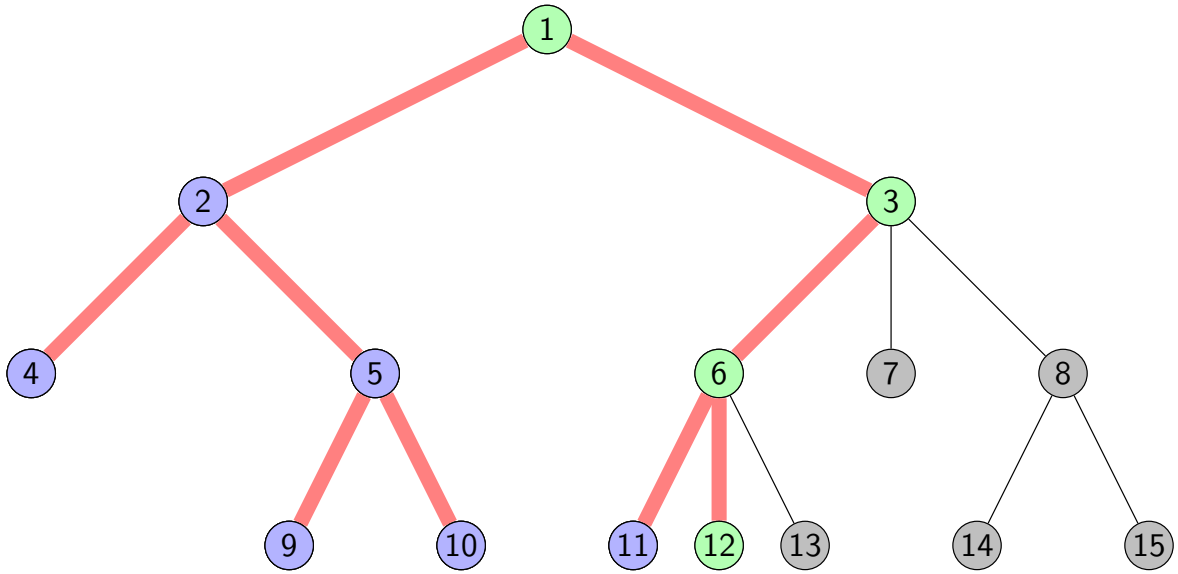
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



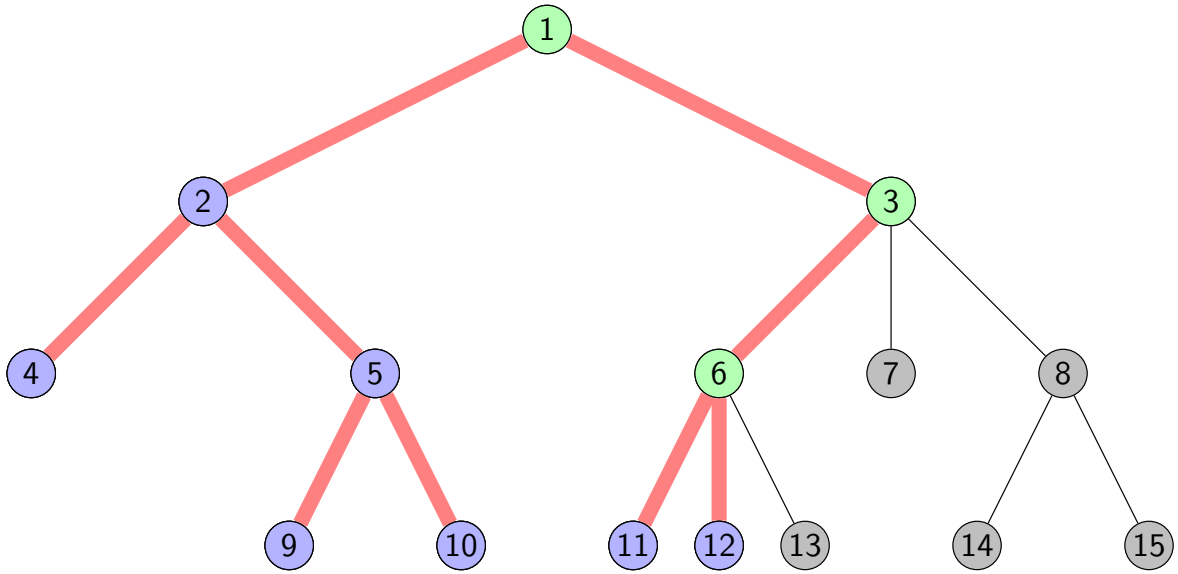
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



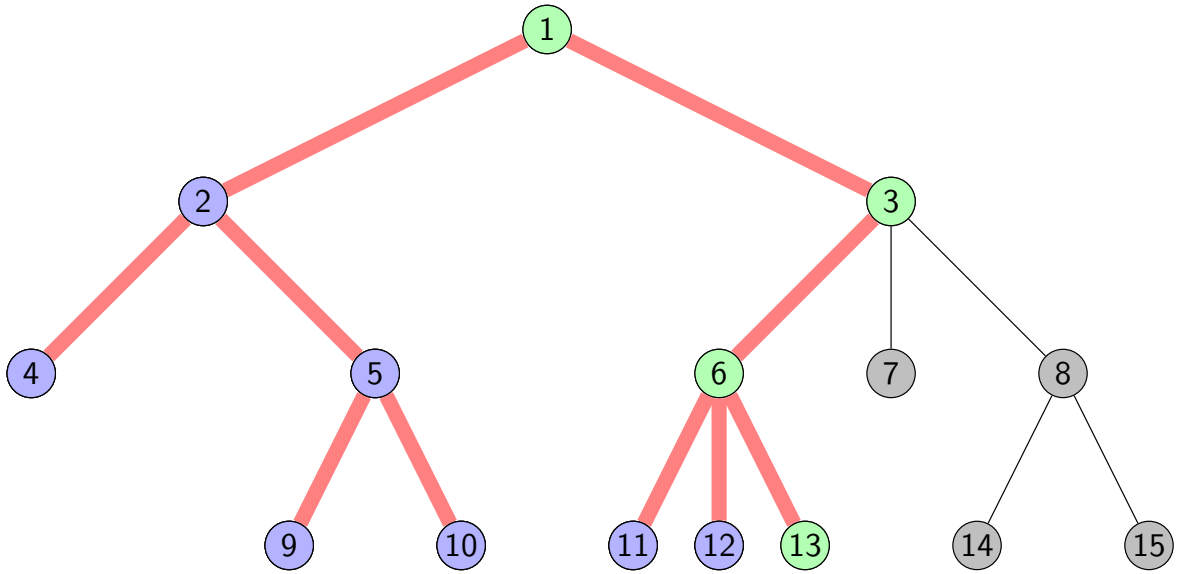
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



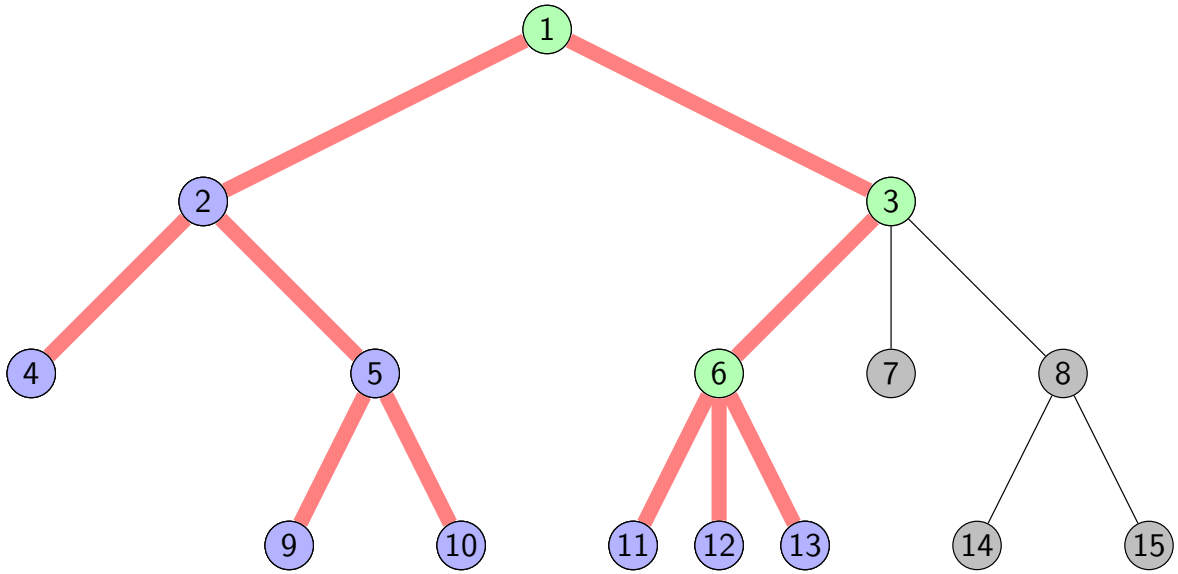
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



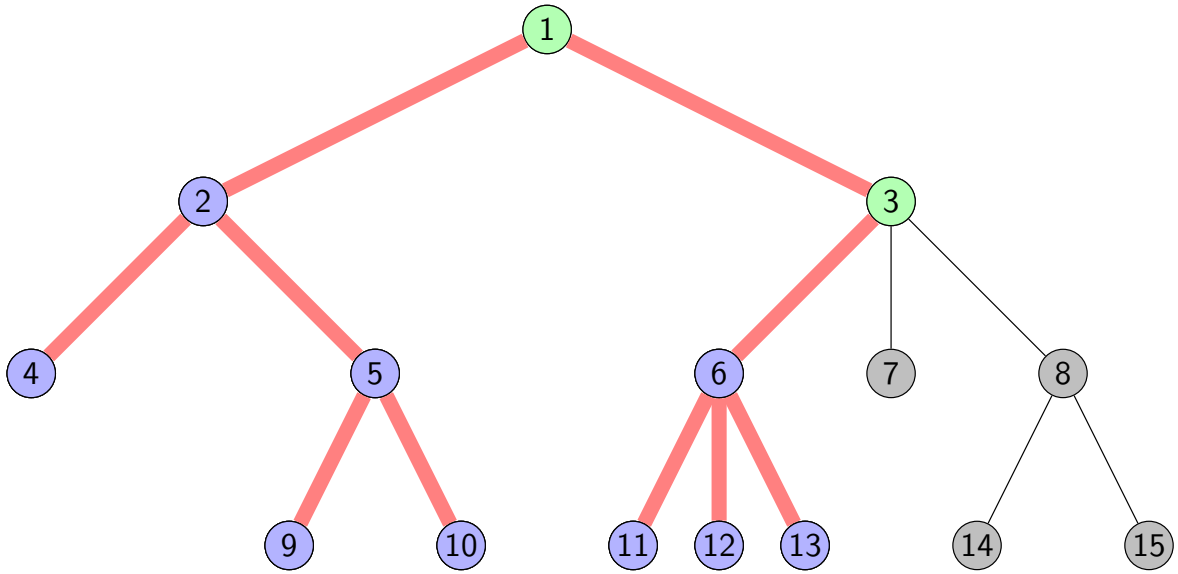
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



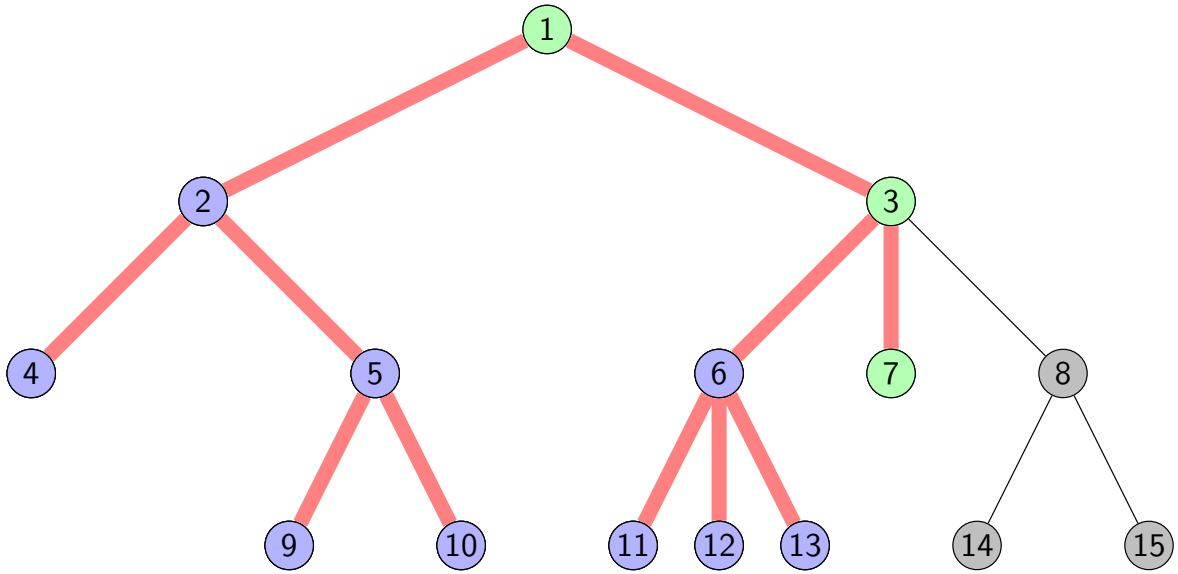
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



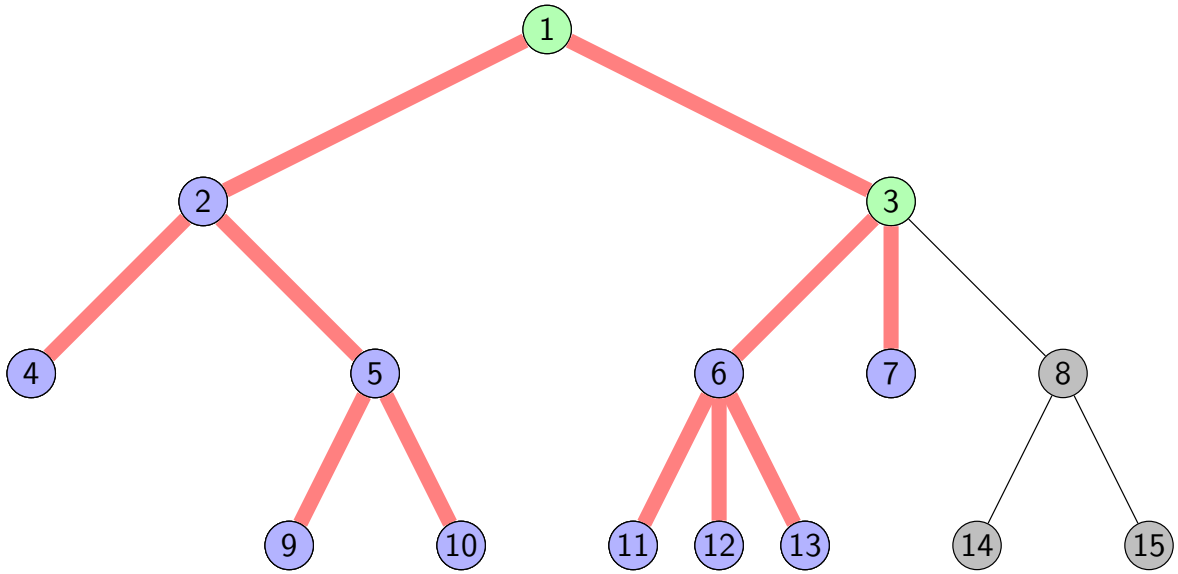
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



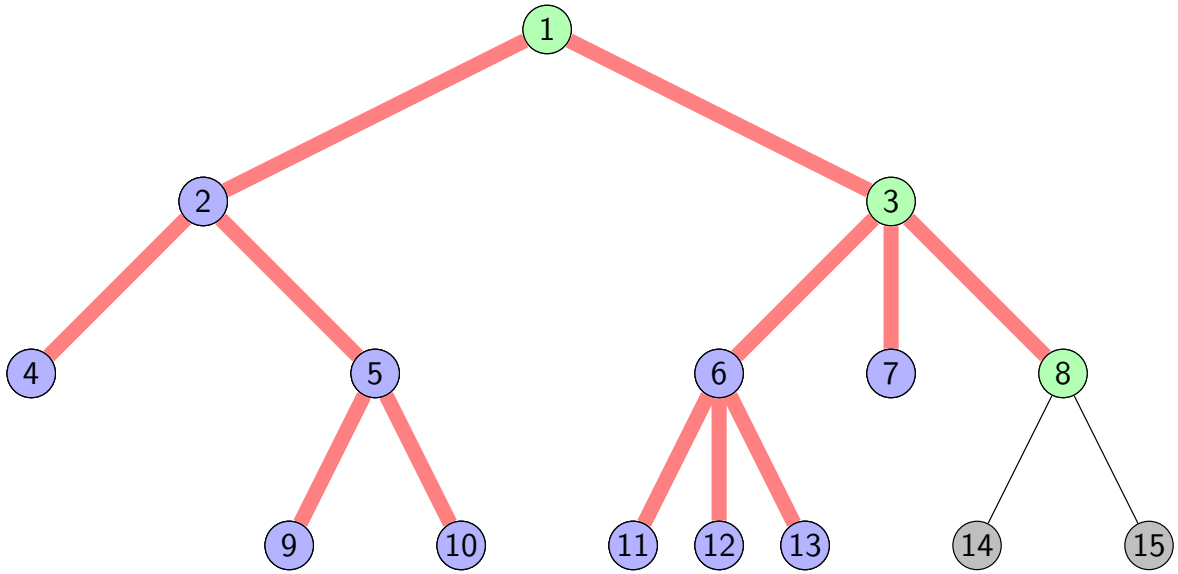
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



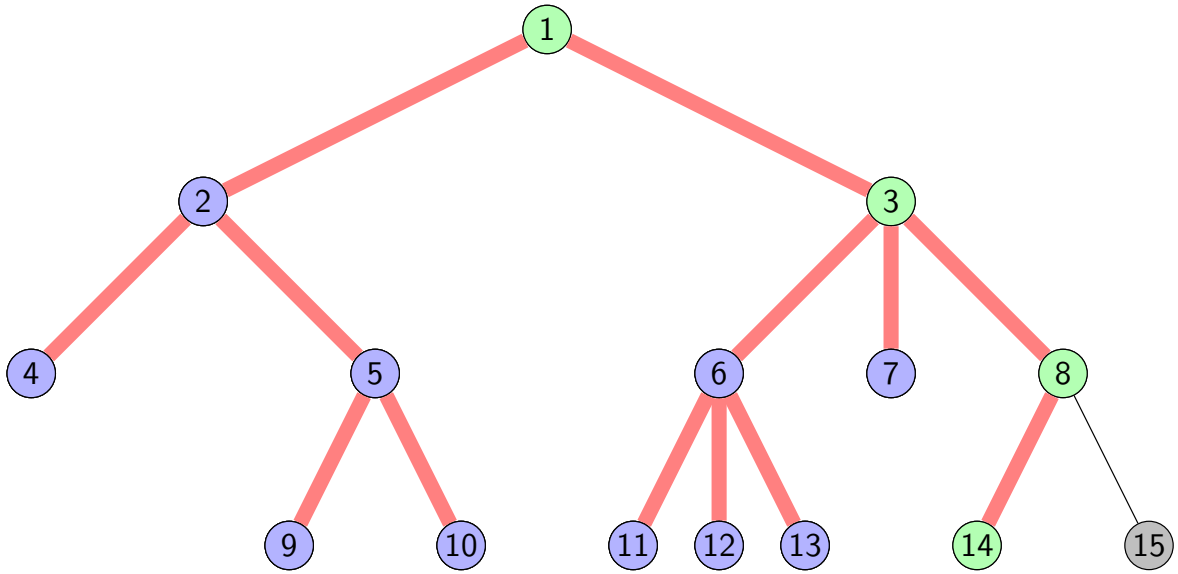
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



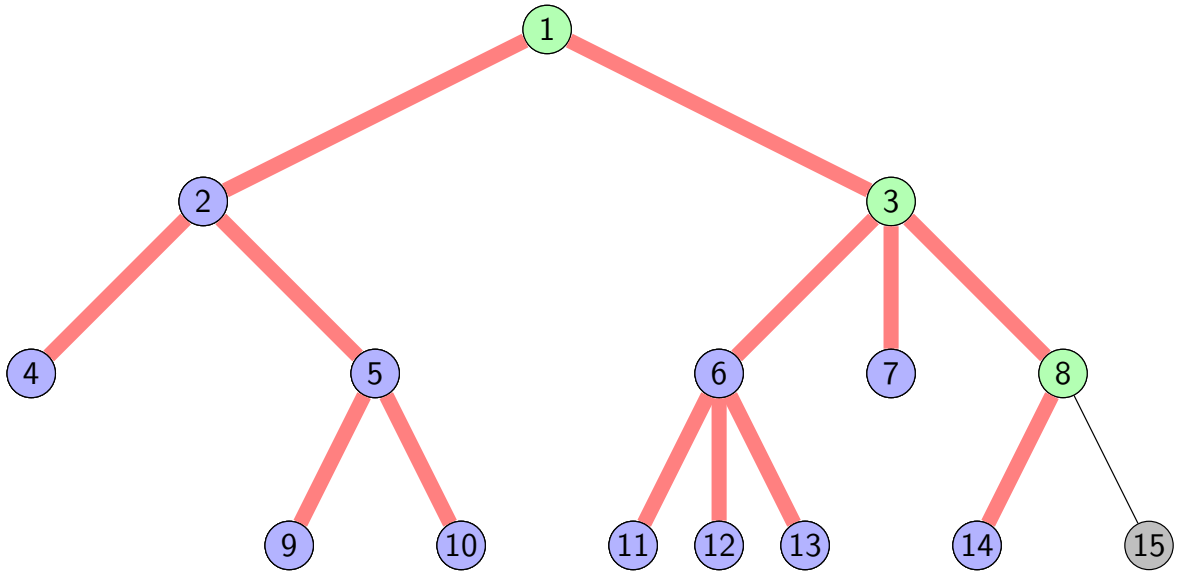
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



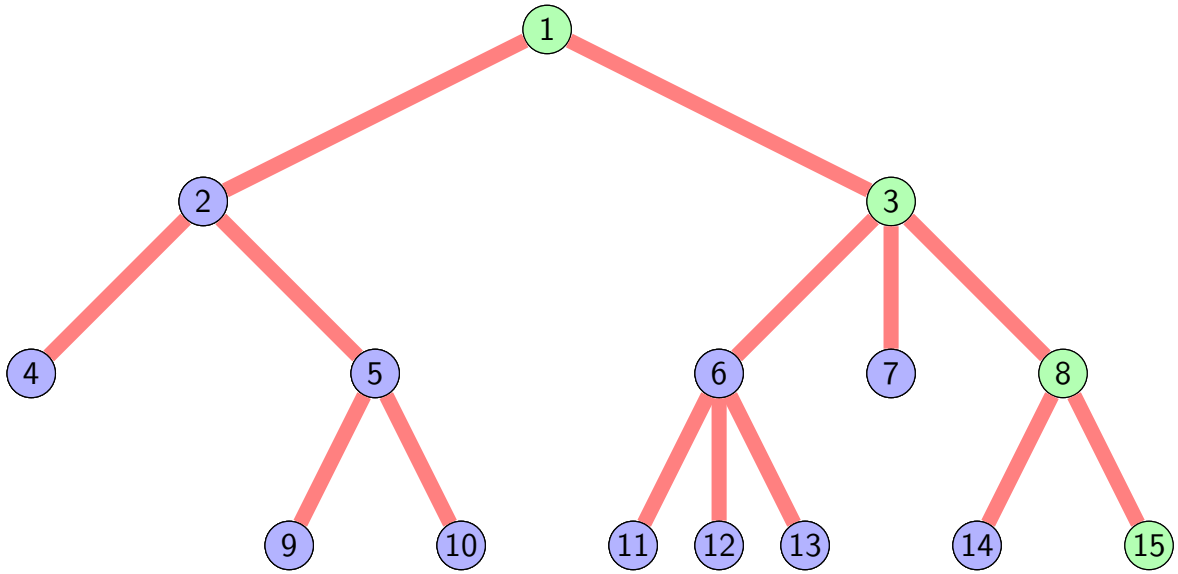
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



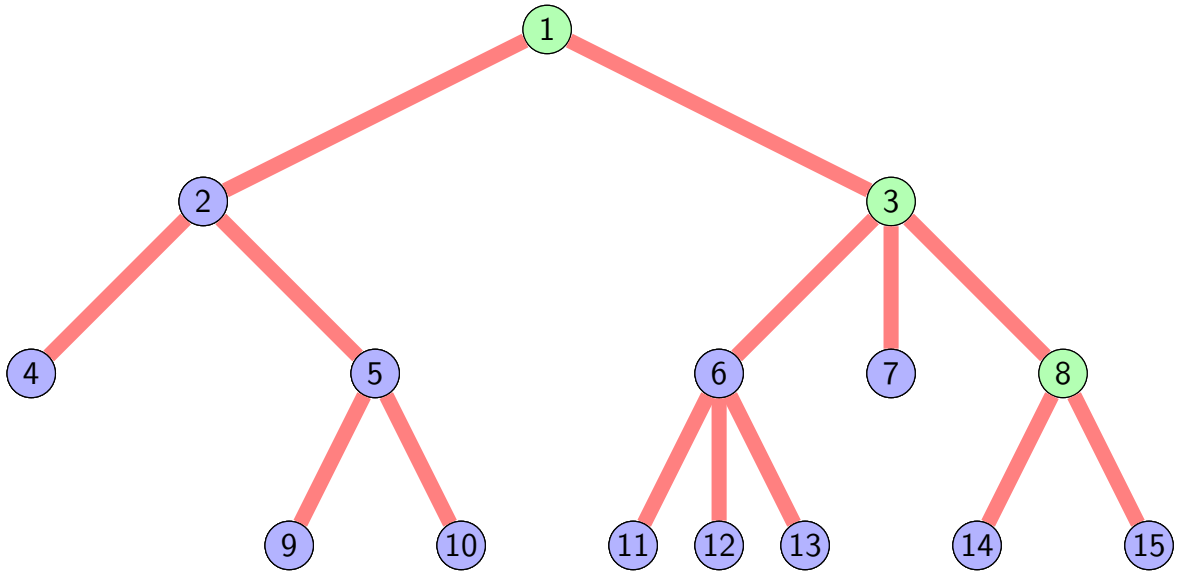
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



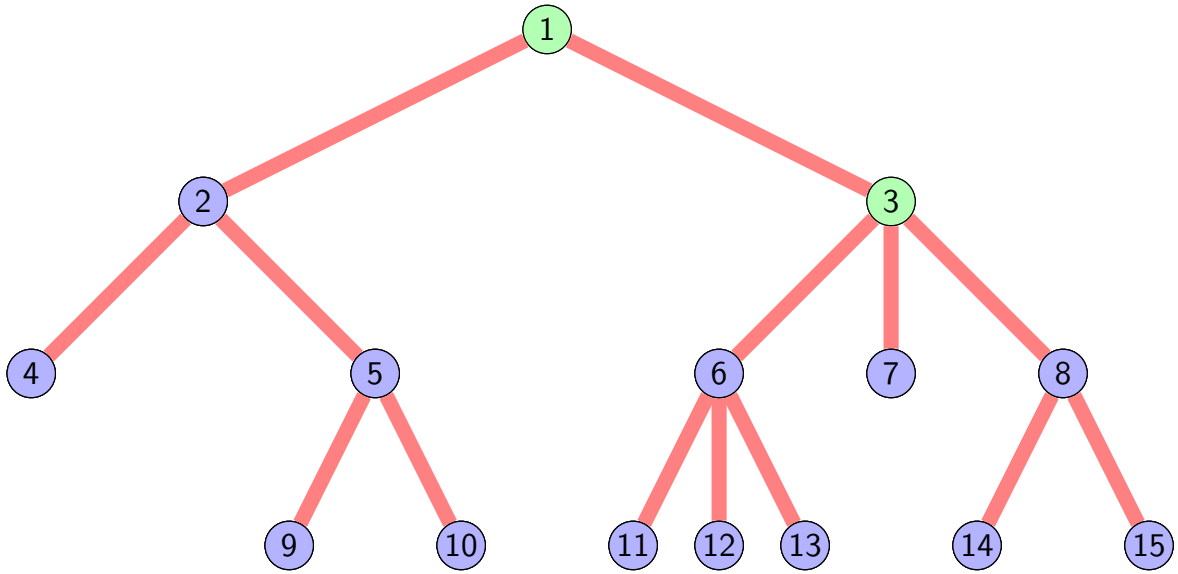
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



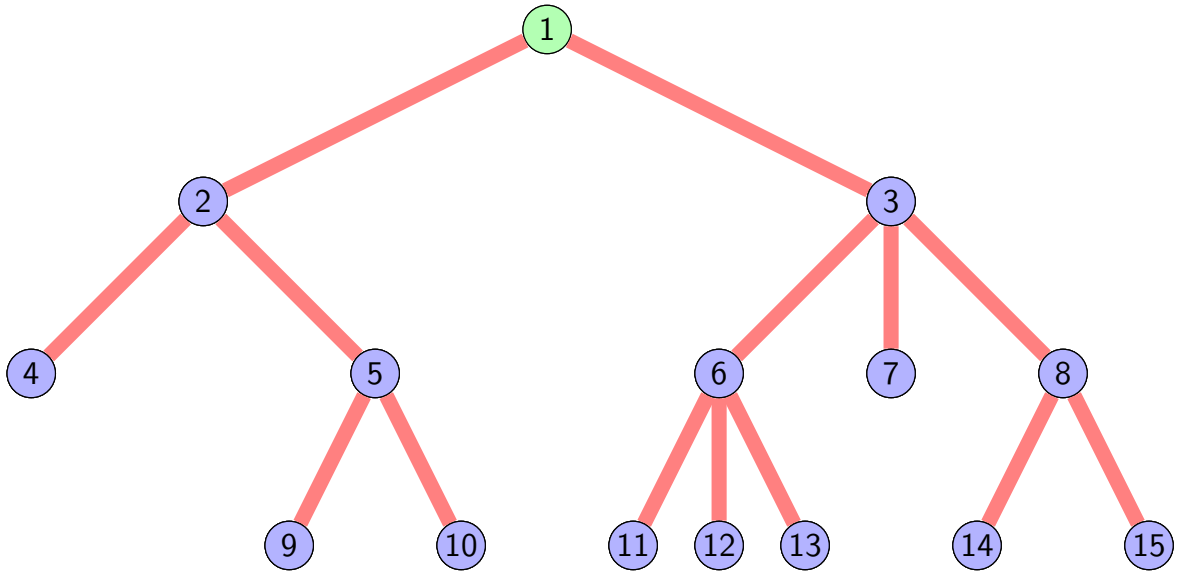
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



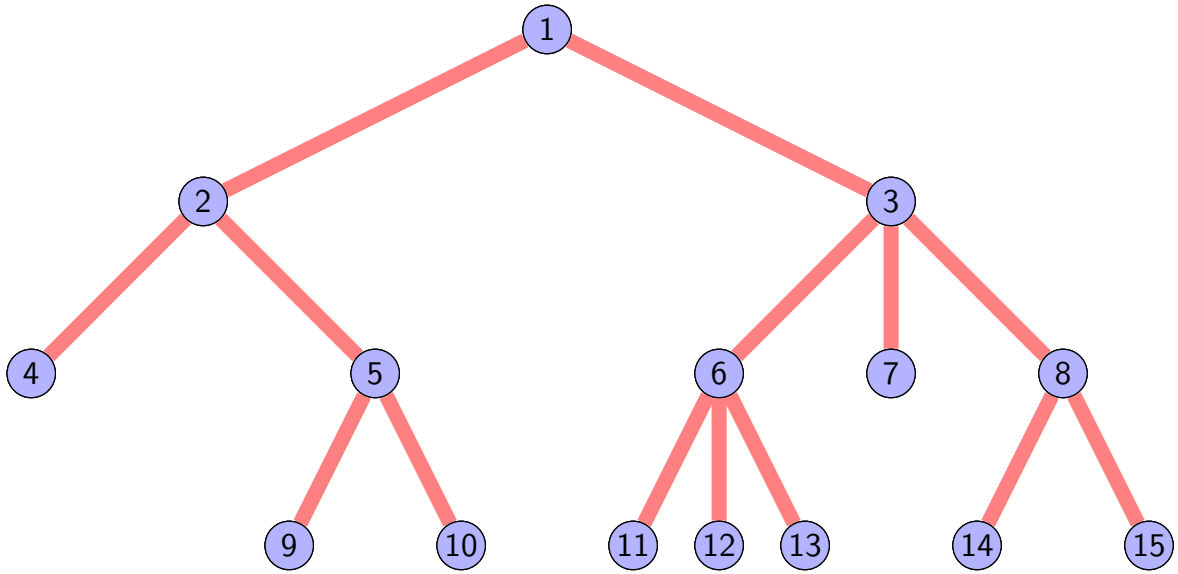
Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



Tiefensuche

Beispiel: Beispiel einer Tiefensuche:



Tiefensuche: Algorithmus

Algorithmus 2: Tiefensuche (DFS)

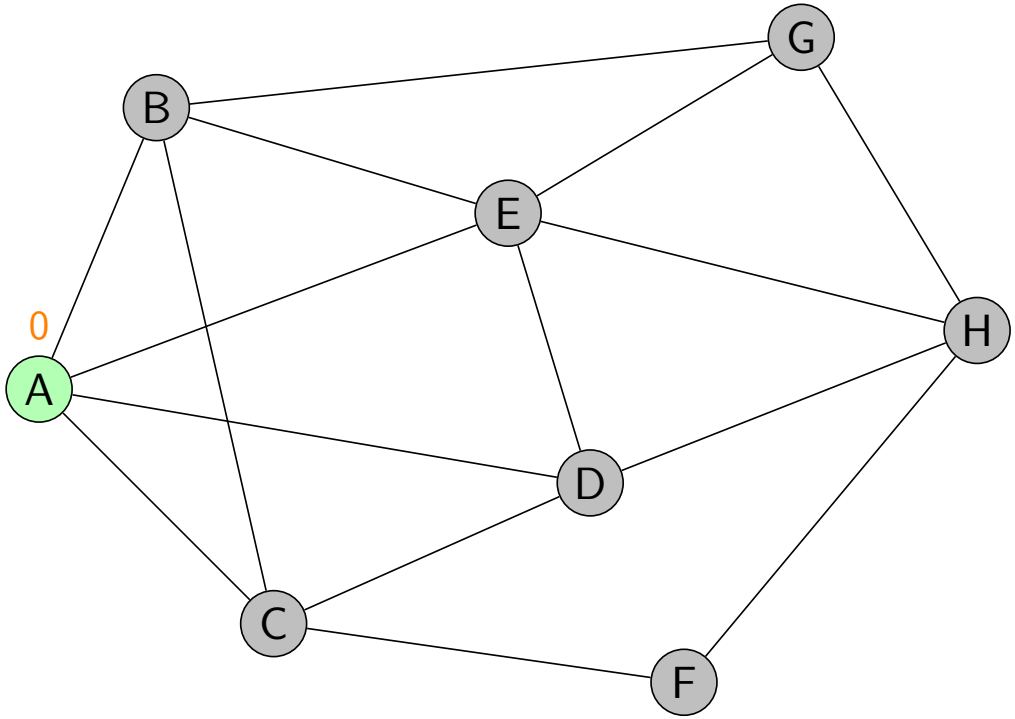
Input: $G = (V, E)$, Startknoten v, Gesuchter Knoten s
Result: vertex $s \in V$, if exists

```
1  Function DFS(G, v, s)
2      Stack S
3      markiere v als besucht
4      S.push(v)
5      while S not empty do
6          u = S.pop()
7          if u = s then
8              return u
9          for alle Nachbarn w von u do
10             if w noch nicht besucht then
11                 markiere w als besucht
12                 S.push(w)
13     return nicht gefunden
```

Anmerkungen

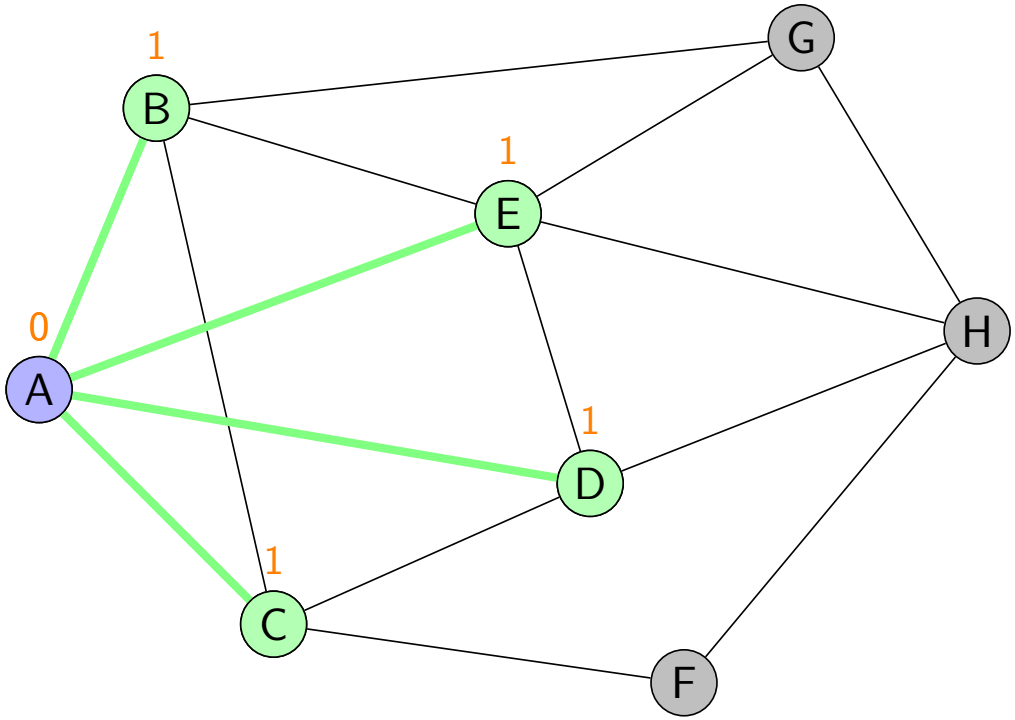
- Die Algorithmen können auch für allgemeine Graphen (und nicht nur Bäume) verwendet werden.
- Dabei werden schon besuchte Knoten *nicht* erneut besucht!
- Anwendungen BFS:
  - 2-färbbarkeit
  - Kürzester Pfad zwischen zwei Knoten
  - Kürzeste-Kreise-Problem
- Anwendungen DFS:
  - Test auf Kreisfreiheit
  - Topologische Sortierung
  - Starke Zusammenhangskomponente

Breitensuche auf allgemeinen Graphen



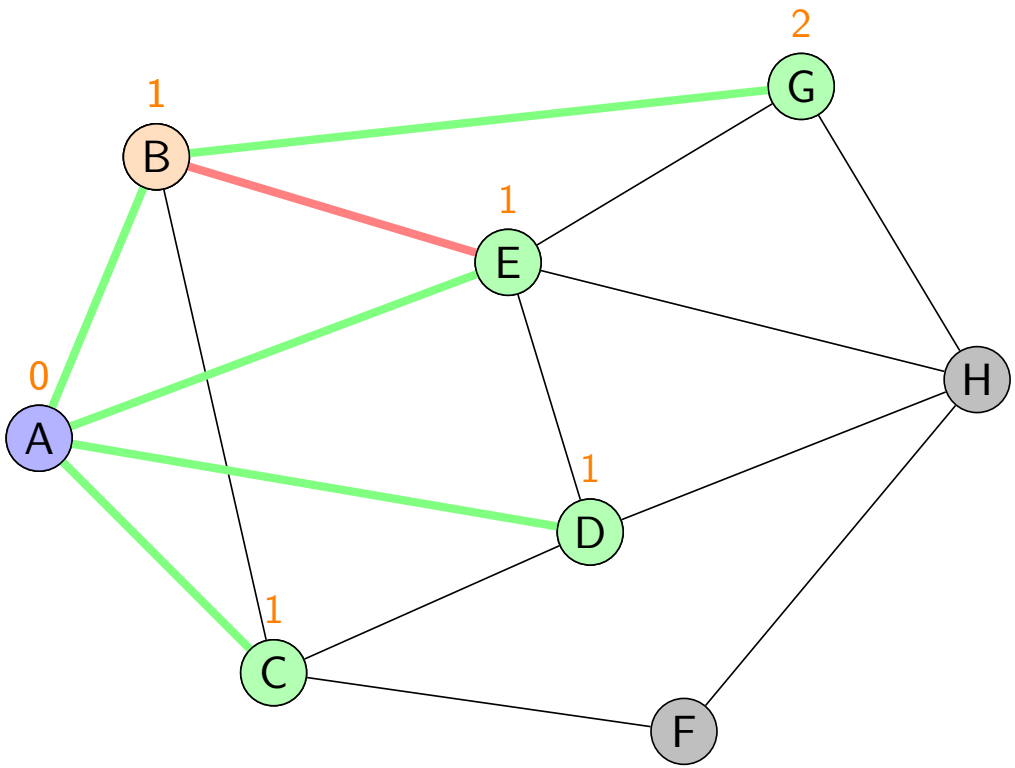
Start mit Knoten A

Breitensuche auf allgemeinen Graphen



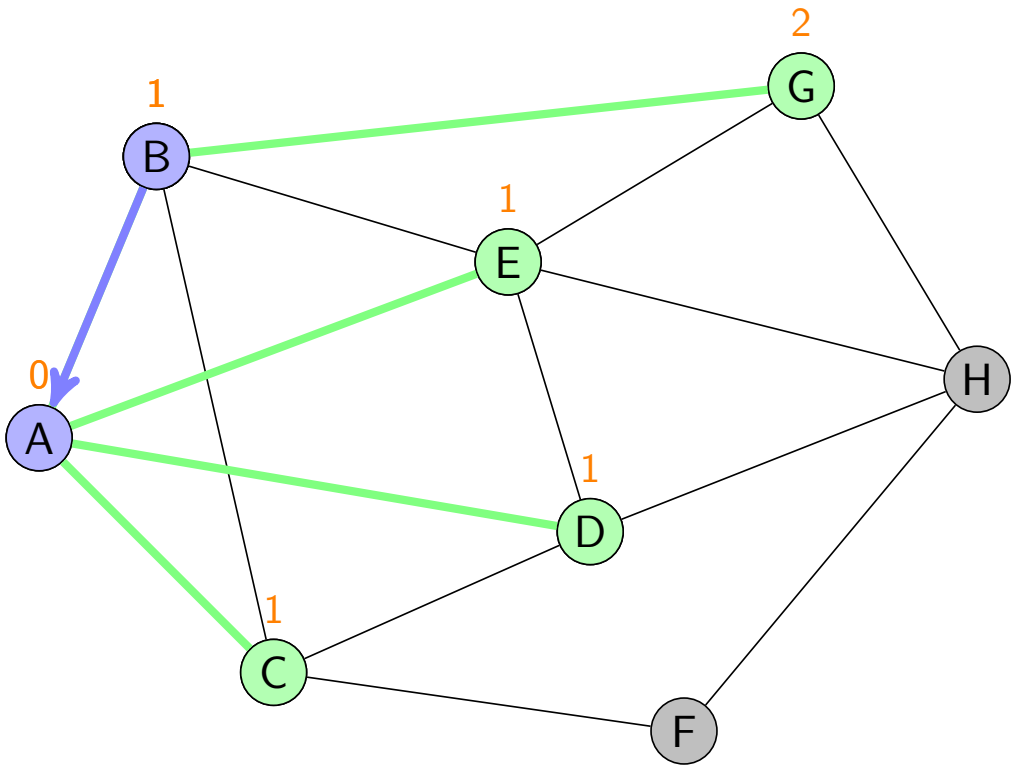
B, C, D, E werden entdeckt (Distanz jeweils 1), A fertiggestellt

Breitensuche auf allgemeinen Graphen



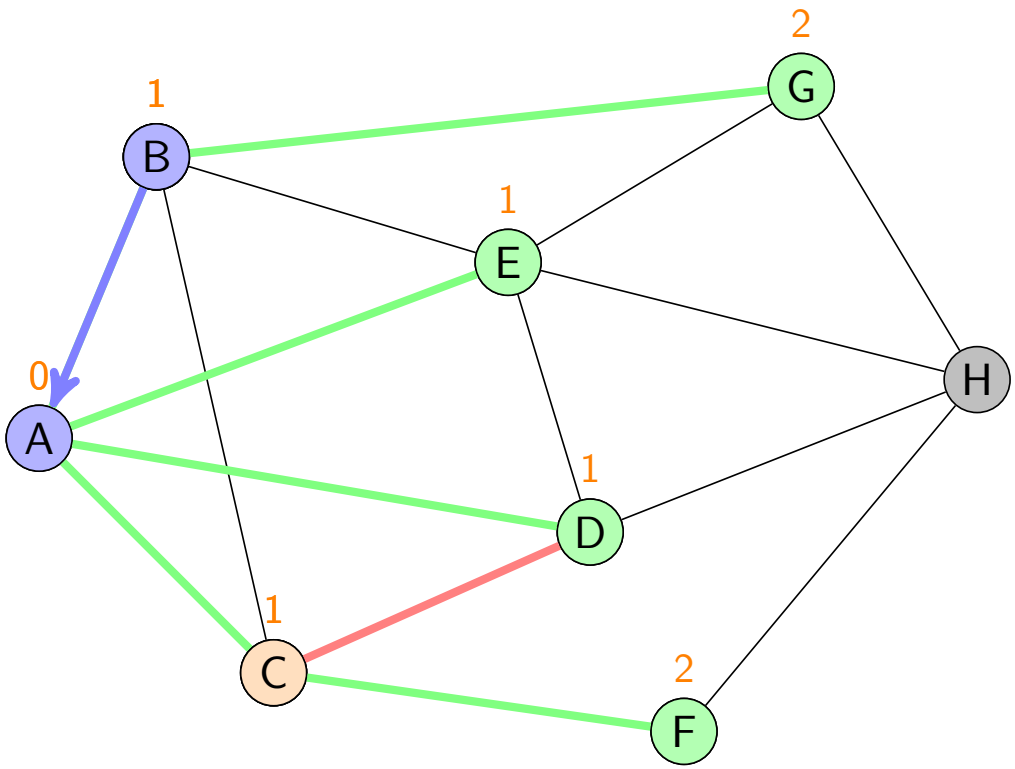
Fortsetzung bei B: [B, E] wird nicht betrachtet, da E schon besucht; G wird entdeckt (Distanz 2)

Breitensuche auf allgemeinen Graphen



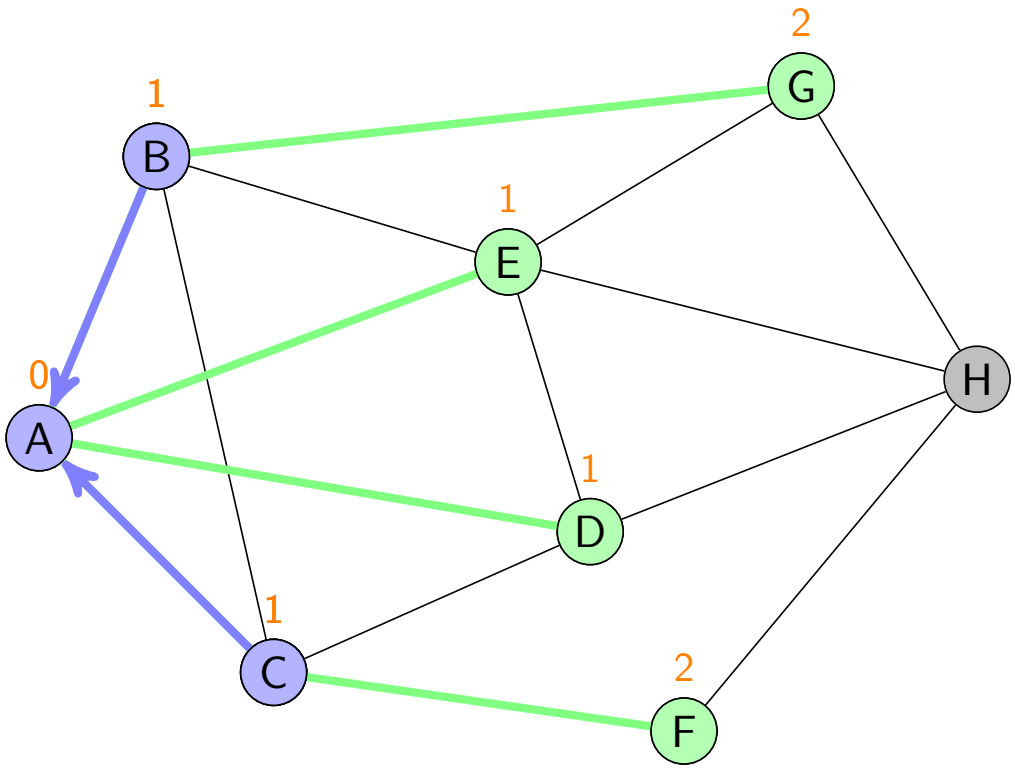
B wird fertiggestellt, und Verweis auf Vorgänger A gespeichert.

Breitensuche auf allgemeinen Graphen



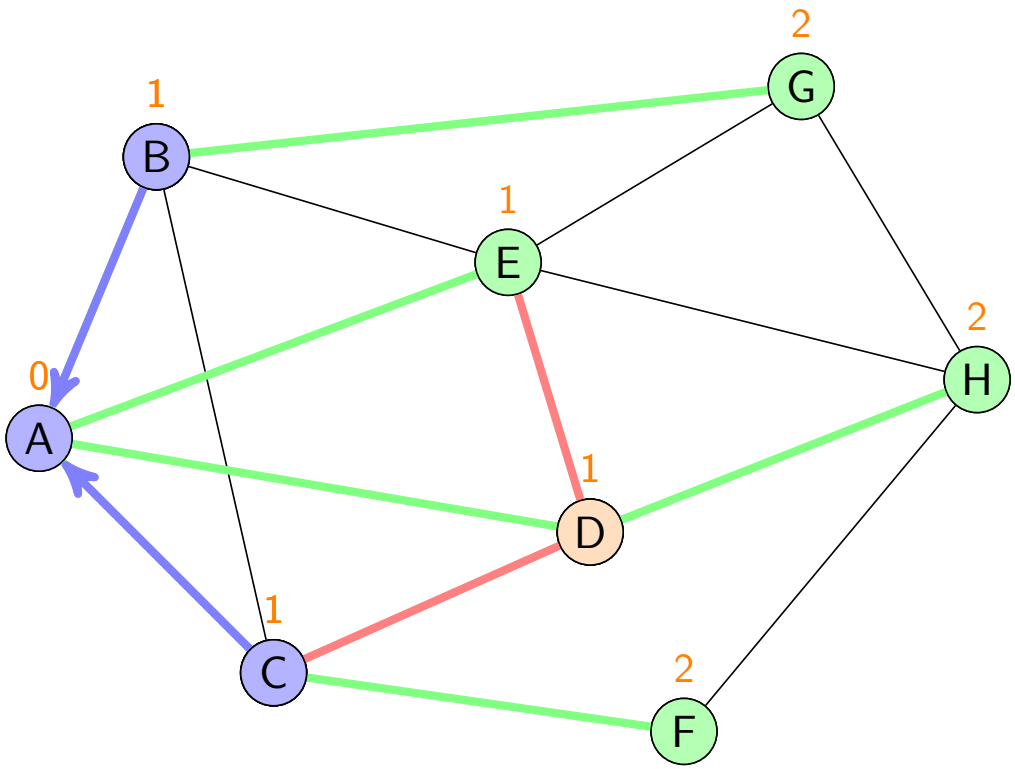
Fortsetzung bei C: [C, D] wird nicht betrachtet, da D schon besucht. F wird entdeckt (Distanz 2)

Breitensuche auf allgemeinen Graphen



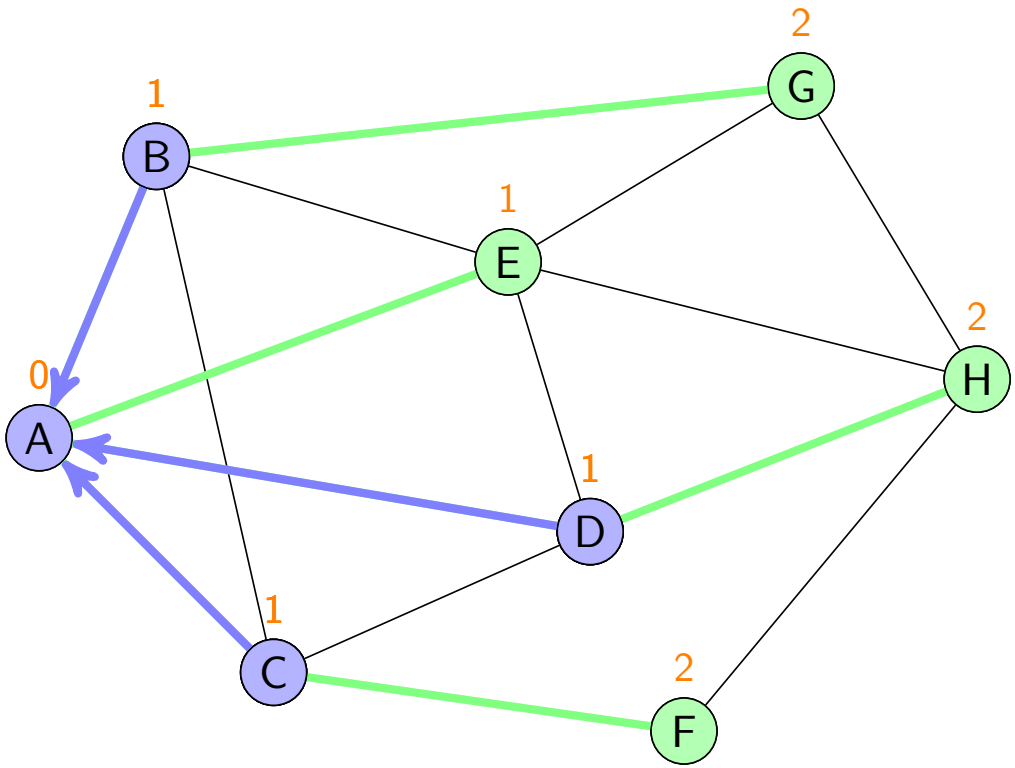
C wird fertiggestellt, Verweis auf Vorgänger A gespeichert.

Breitensuche auf allgemeinen Graphen



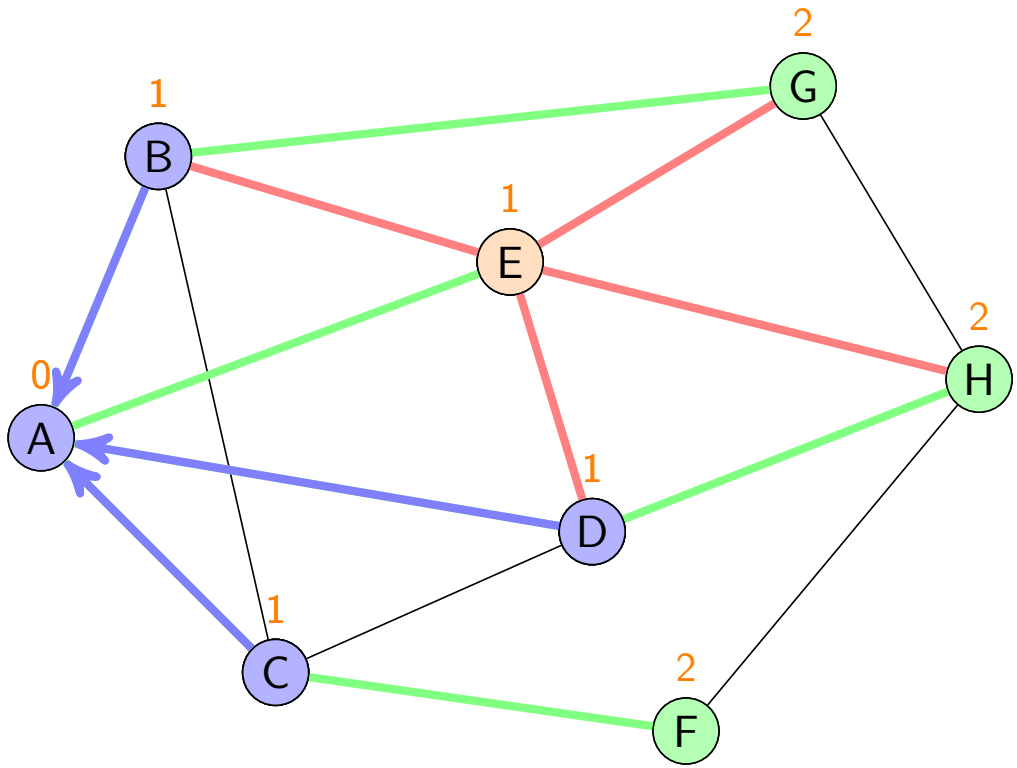
Fortsetzung bei D. Kanten zu C und E werden nicht betrachtet (da jeweils schon besucht). H wird entdeckt (Distanz 2)

Breitensuche auf allgemeinen Graphen



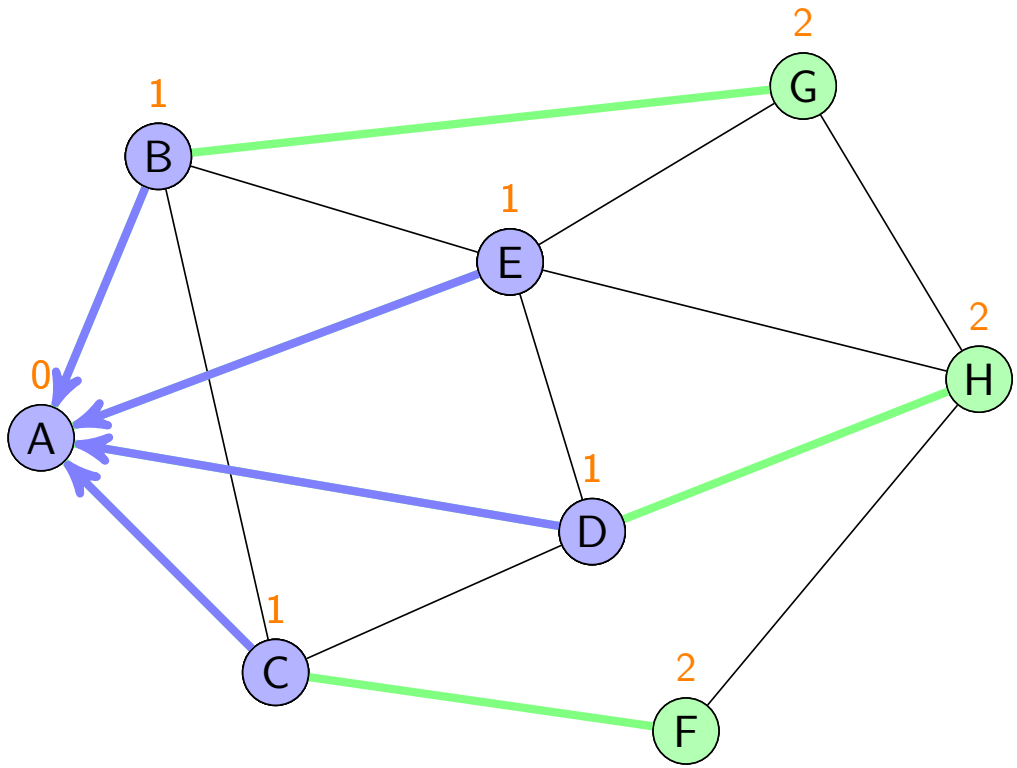
Fertigstellung von D, Verweis auf Vorgänger A

Breitensuche auf allgemeinen Graphen



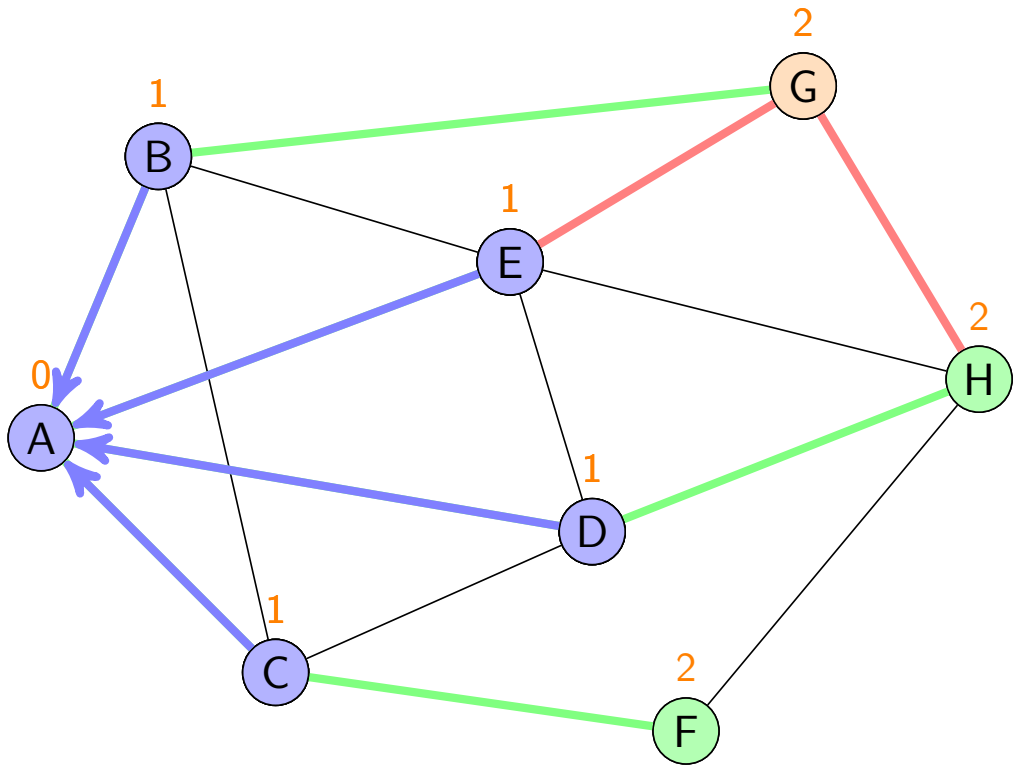
Fortsetzung mit E, jedoch werden keine neuen Knoten entdeckt.

Breitensuche auf allgemeinen Graphen



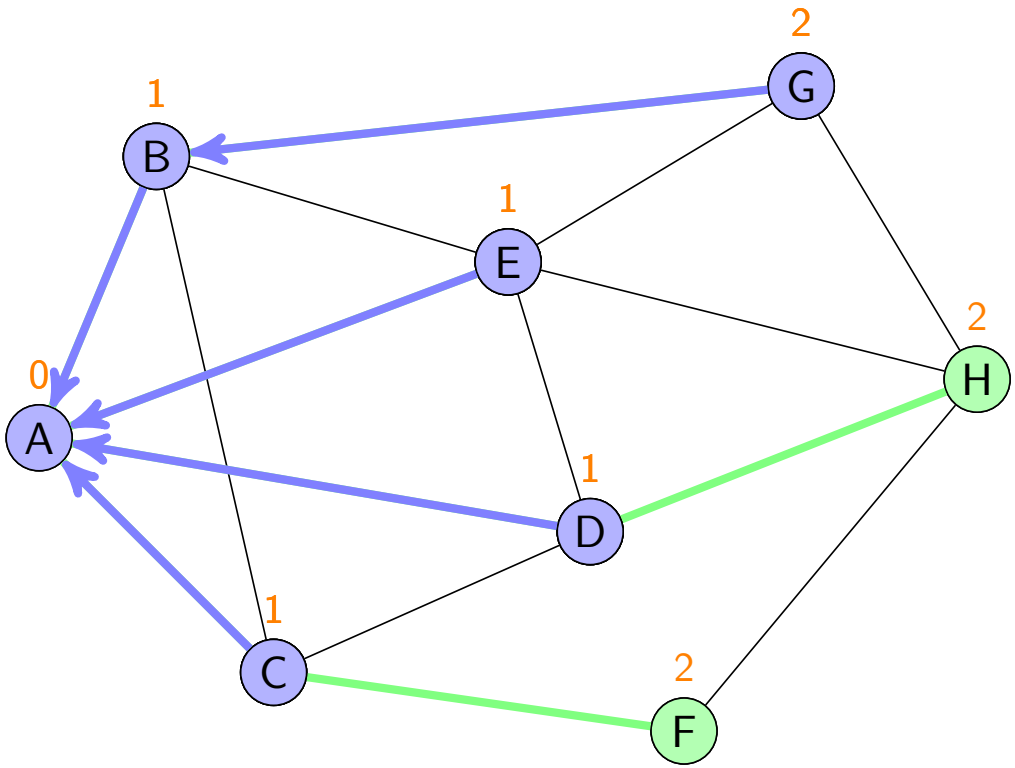
Fertigstellung von E

Breitensuche auf allgemeinen Graphen



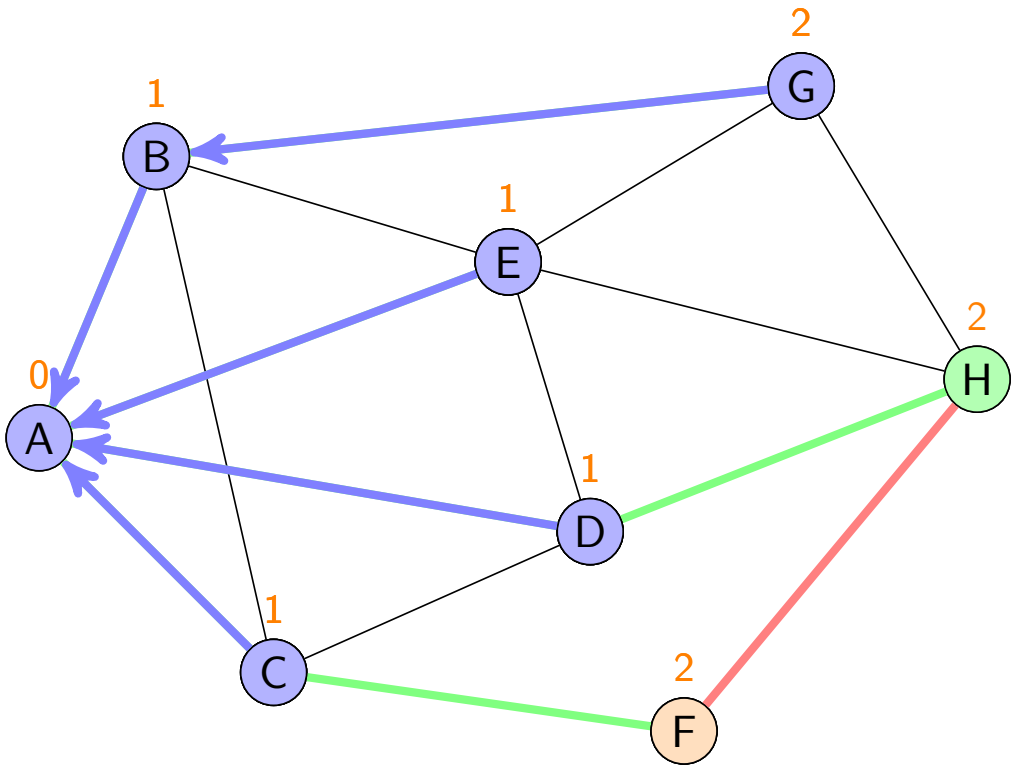
Fortsetzung mit G, jedoch werden auch hier keine neuen Knoten entdeckt.

Breitensuche auf allgemeinen Graphen



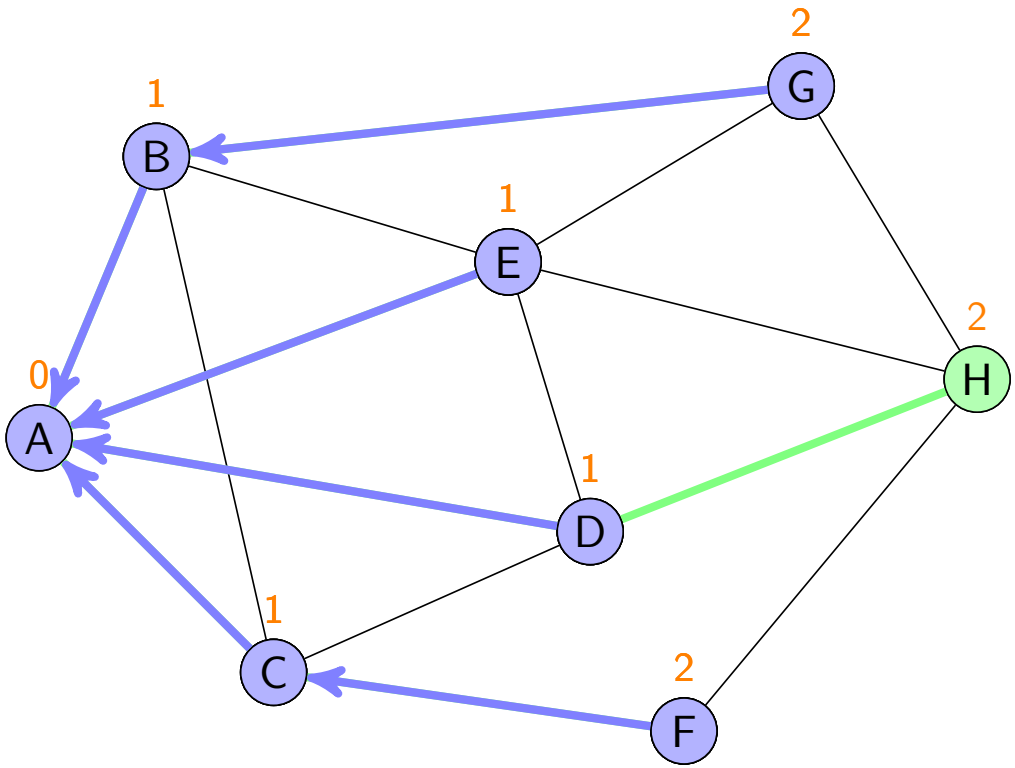
Fertigstellung von G

Breitensuche auf allgemeinen Graphen



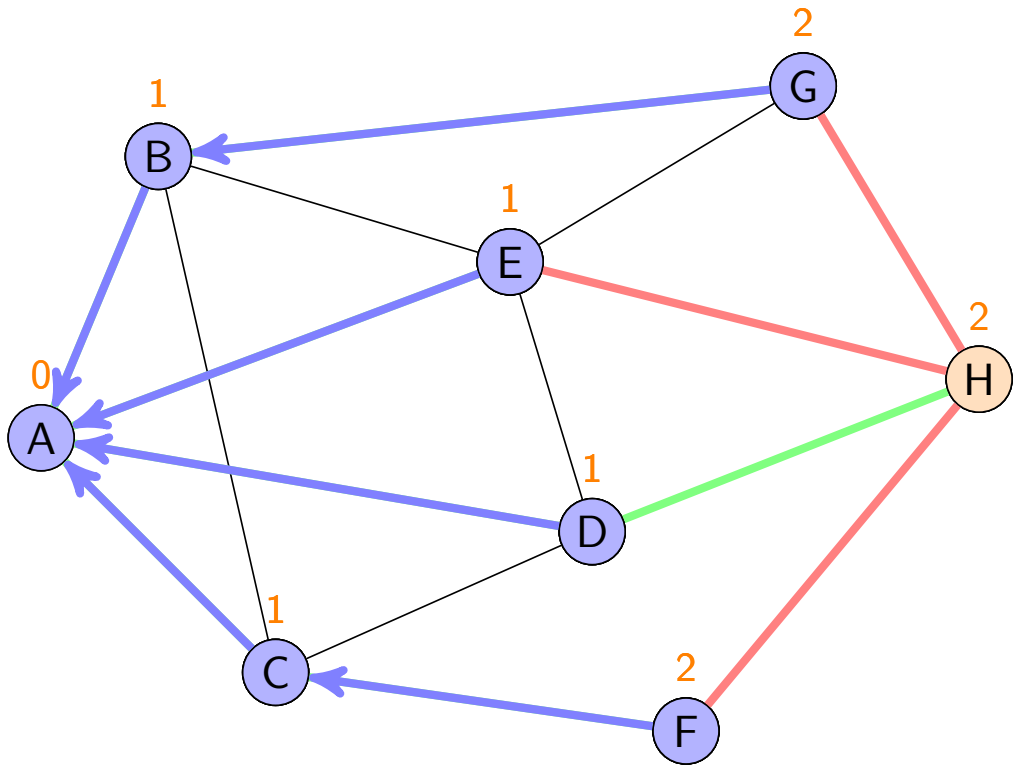
Fortsetzung mit F

Breitensuche auf allgemeinen Graphen



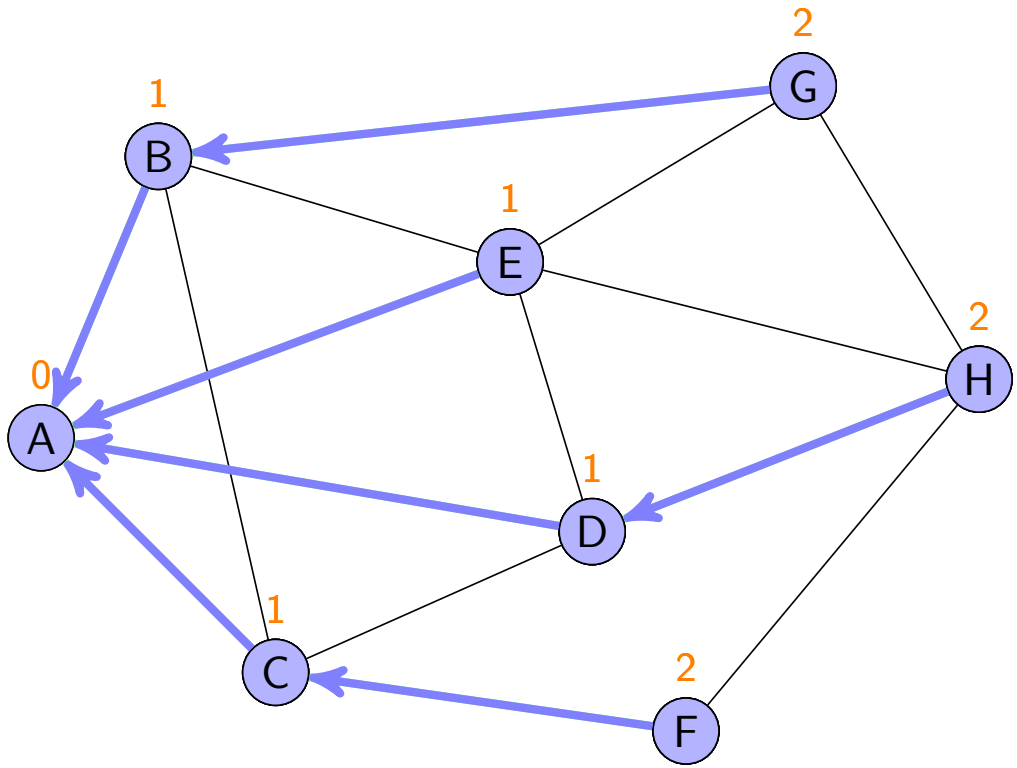
F wird fertiggestellt.

Breitensuche auf allgemeinen Graphen



Fortsetzung mit H

Breitensuche auf allgemeinen Graphen



H wird fertiggestellt.

Breitensuche auf allgemeinen Graphen

- Es entsteht ein (Baum) mit Startknoten als Wurzel
- Bei nicht erreichbaren Knoten im ersten Aufruf wird die Breitensuche erneut aufgerufen
- Dadurch entsteht ein Wald (=mehrere Bäume)
- Distanzberechnung durch Breitensuche möglich, da jeder Knoten um 1 höhere Distanz als sein Vorgänger hat.

Kürzeste Wege in Graphen

Wir betrachten nun das Problem, kürzeste Wege in gewichteten Graphen zu berechnen. Häufig betrachtete Varianten:

- *Single-Pair Shortest Path (SPSP)*: kürzester Weg zwischen gegebenen s und t.
- *Single-Source Shortest Paths (SSSP)*: alle Ziele ab Start s (liefert auch SPSP als Spezialfall).
- *All-Pairs Shortest Paths (APSP)*: Distanzen für alle Paare (u, v). Kann durch wiederholte SSSP oder spezieller über Matrix-/Dynamik-Verfahren gelöst werden.

Bellman-Gleichungen

Die optimalen Distanzen $d(v)$ (für SSSP ohne negative Zyklen) erfüllen das Optimierungsprinzip

$$d(s) = 0, \qquad d(v) = \min_{(u,v)\in E} d(u) + w(u,v) \quad \text{mit } v \neq s,$$

also eine Fixpunkt-Gleichung, die durch sukzessive Relaxation (BFS, Dijkstra, Bellman–Ford) angenähert und schließlich erreicht wird.

Überblick über zentrale Algorithmen

- **BFS** (SSSP in $O(n+m)$ für ungewichtete / einheitlich gewichtete Graphen).
- **Dijkstra** (SSSP für nichtnegative Gewichte; mit geeigneten Prioritätsstrukturen $O(m \log n)$).
- **Bellman–Ford** (SSSP mit negativen Gewichten, erkennt negative Zyklen; $O(nm)$).
- **Floyd–Warshall** (APSP mittels dynamischer Programmierung; $O(n^3)$; findet negative Zyklen über Diagonale).
- **Matrixbasierte APSP** (Wiederholtes "Min-Plus"-Matrix-Produkt / exponentiation by squaring; theoretisch durch schnelle Matrixmultiplikation asymptotisch verbesserbar, praktisch selten relevant).

Algorithmus von Dijkstra

Der Algorithmus von Dijkstra berechnet die kürzesten Wege in einem gewichteten Graphen mit $w_{ij} \geq 0$, für alle $[i, j] \in E$.

Grundidee:

- Ähnlichkeit zu BFS, jedoch andere Regeln für die Auswahl des nächsten Knoten.
- Ein Aufruf von Dijkstra berechnet die kürzesten Wege von einem Startknoten zu allen anderen Knoten des Graphen.
- In jedem Schritt werden **Zwischenergebnisse** $\delta_k, k \in V$ berechnet, bzw. aktualisiert.
- Diese Zwischenergebnisse sind die Länge des kürzesten *bisher gefundenen* Weges bis zu diesem Knoten.
- Wiederhole (bis alle Knoten abgeschlossen):
  1. Wähle Knoten k mit minimalem δ_k und schließe diesen ab.
  2. Speichere Verweis auf direkten Vorgänger.
  3. Aktualisiere die Werte δ_k für noch nicht abgeschlossene Nachbarknoten von k.

Algorithmus von Dijkstra

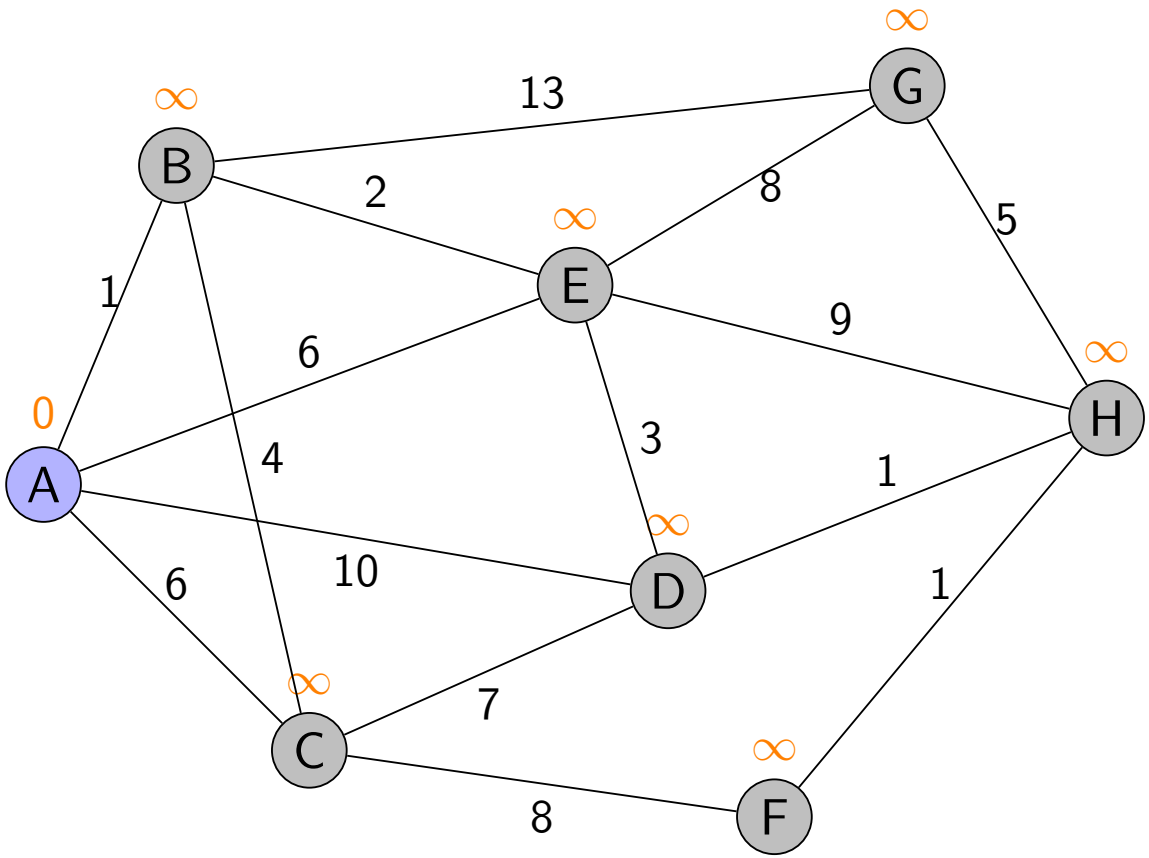
Algorithmus 3: DIJKSTRA

Data: Graph G mit $w_{ij} \geq 0$ für alle $(i,j) \in E(G)$
Data: Startknoten s

```
1  ∀v ∈ V : δ_v ← ∞;
2  δ_s ← 0;
3  Prioritätswarteschlange Q befüllt mit allen Knoten ;
4  while Q ≠ ∅ do
5      u ← Q.getMin();  // entnimmt Knoten u mit kleinstem δ_u
6      Fertigstellung von Knoten u;
7      Speichere Verweis auf direkten Vorgänger von u;
8      for all (u, v) ∈ E, v noch nicht fertiggestellt do
9          if δ_v > δ_u + w_uv then
10             δ_v ← δ_u + w_uv;
```

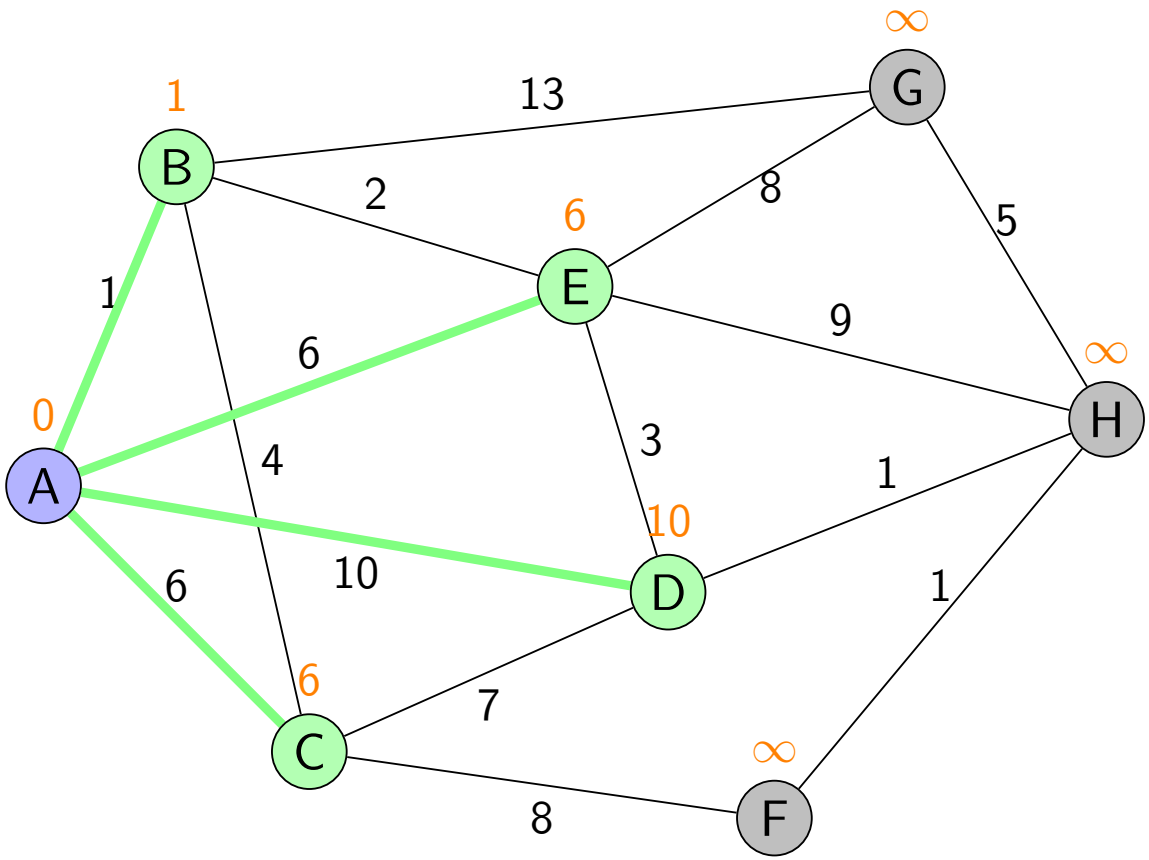
Anmerkung: Die Prioritätswarteschlange Q kann durch ein einfaches Array umgesetzt werden. Um u mit kleinstem δ_u zu finden, muss es zur Gänze durchlaufen werden. Dies ist nachteilig für die Performance des Algorithmus, weshalb die Prioritätswarteschlange meist anhand eines *Heaps* umgesetzt wird.

Algorithmus von Dijkstra: Beispiel



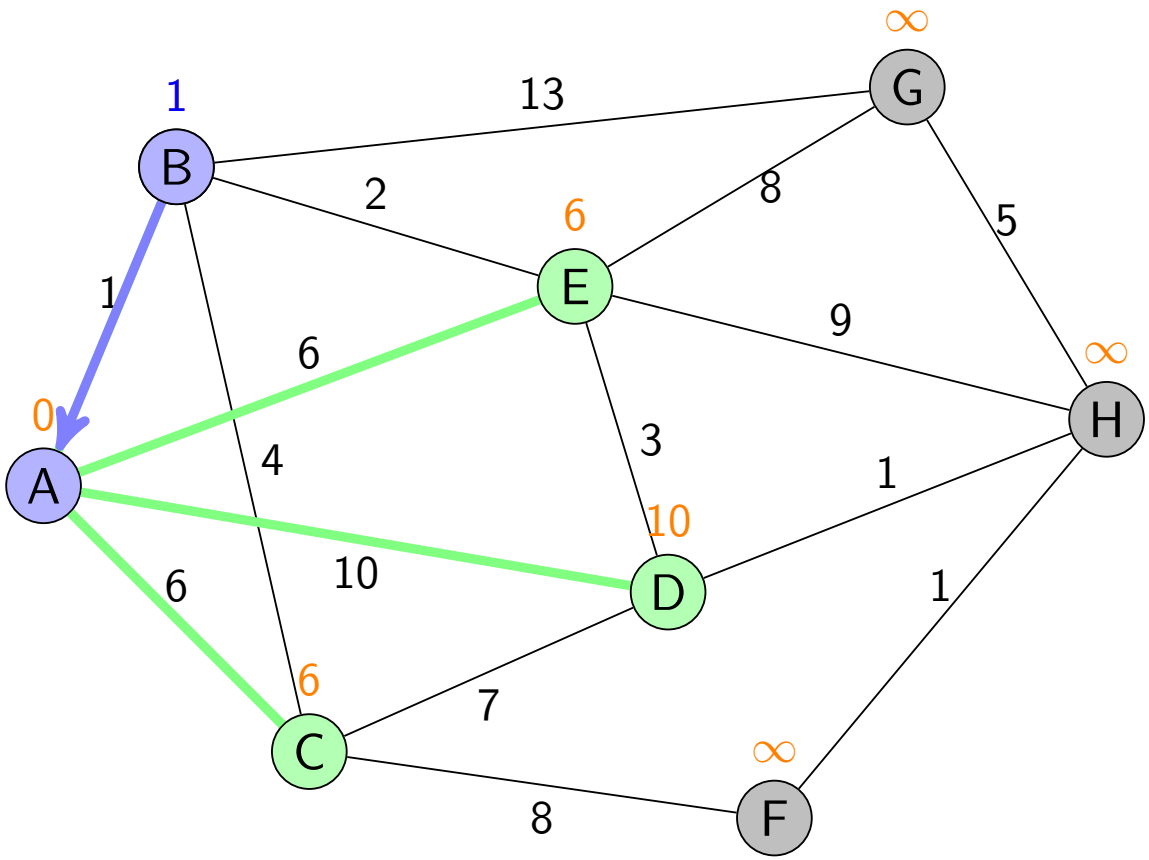
Wir suchen den kürstesten Weg vom Knoten A zum Knoten H. Im ersten Schritt wird der Startknoten "fertiggestellt".

Algorithmus von Dijkstra: Beispiel



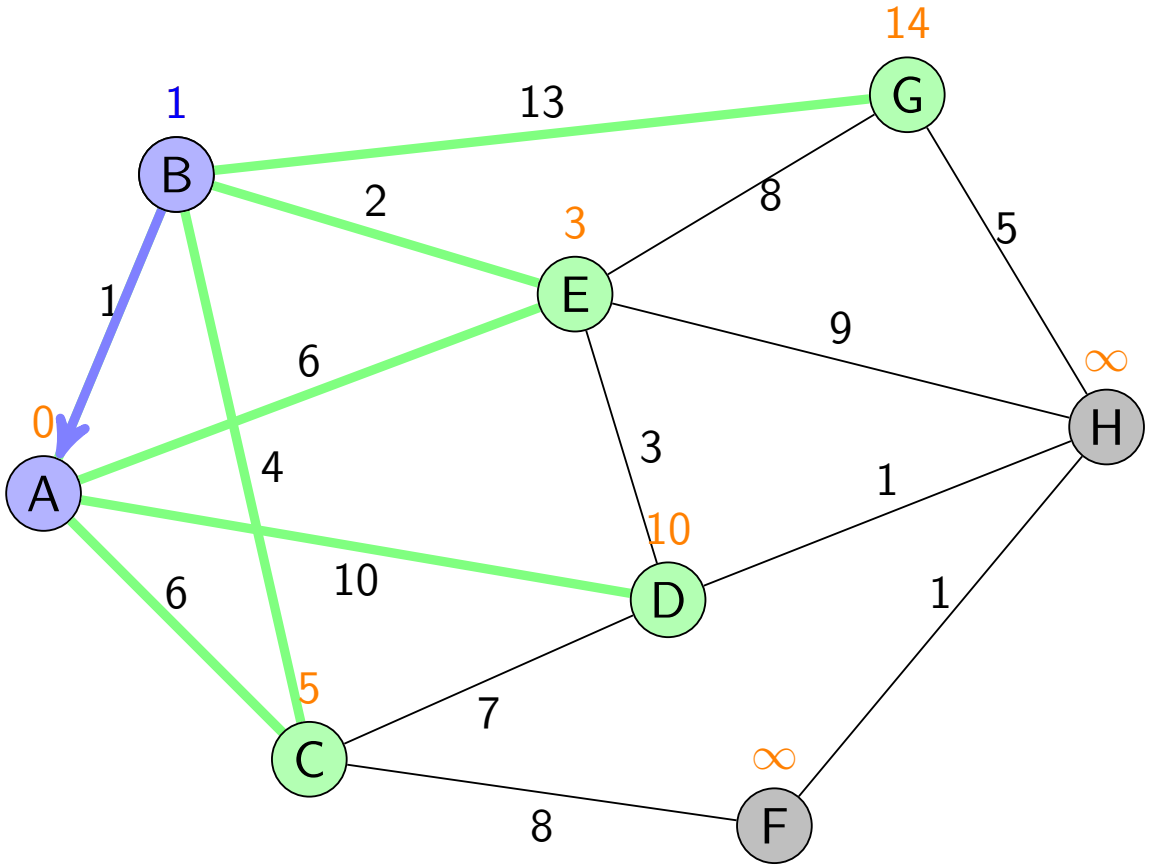
Im nächsten Schritt werden die Nachbarknoten B, C, D und E entdeckt. Die Zwischenwerte werden wie folgt berechnet:
$\delta_A = 0, \delta_B = \delta_A + 1 = 1, \delta E = \delta_A + 6 = 6, \delta_D = \delta_A + 10 = 10, \delta_C = \delta_A + 6 = 6.$

Algorithmus von Dijkstra: Beispiel



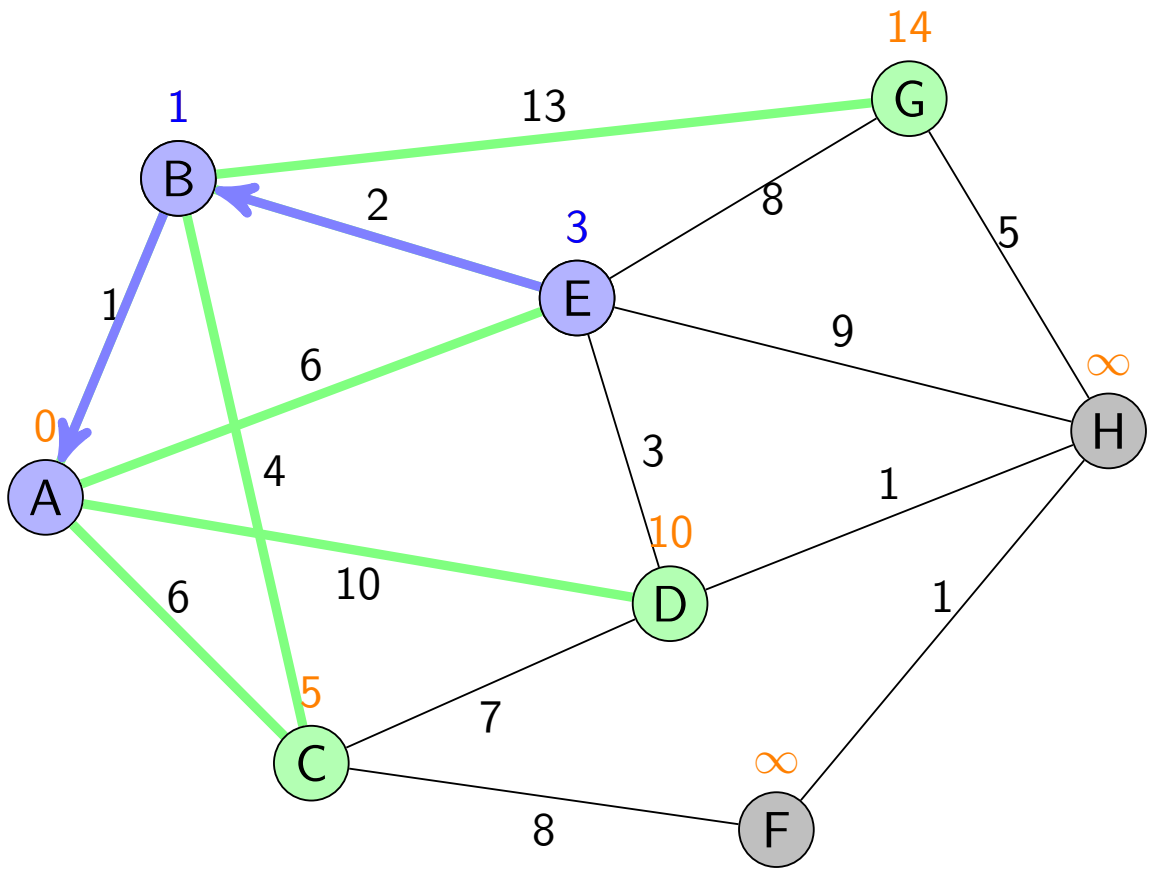
Nun wird der Knoten v mit kleinstem δ_v fertiggestellt. Im konkreten Beispiel ist dies Knoten B.

Algorithmus von Dijkstra: Beispiel



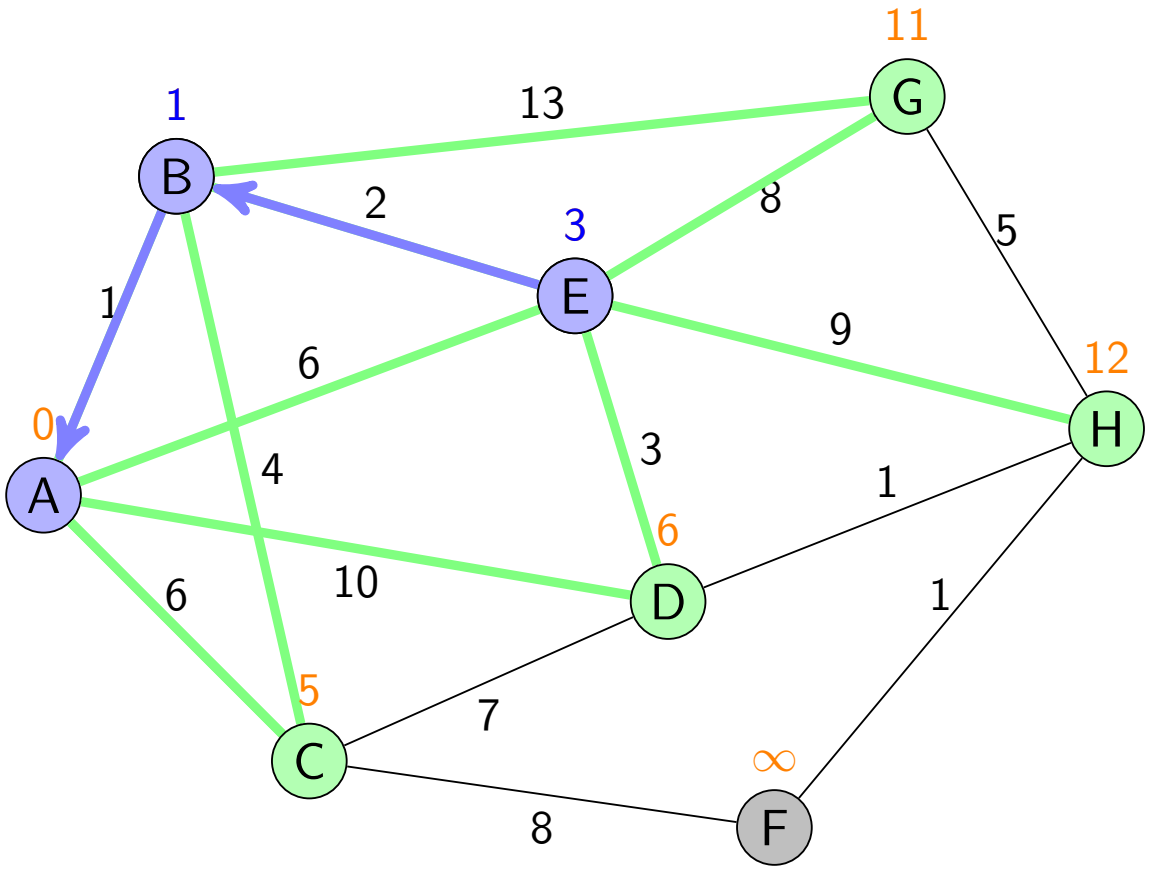
Ausgehend vom letzten fertiggestellten Knoten (B) werden nun neue Zwischenergebnisse für C, E und G berechnet. Wir erhalten $\delta_G = 1 + 13 = 14, \delta_E = 1 + 2 = 3, \delta_C = 1 + 4 = 5$. Für die Knoten C und E erhalten wir kleinere Werte als die bisher gefundenen.

Algorithmus von Dijkstra: Beispiel



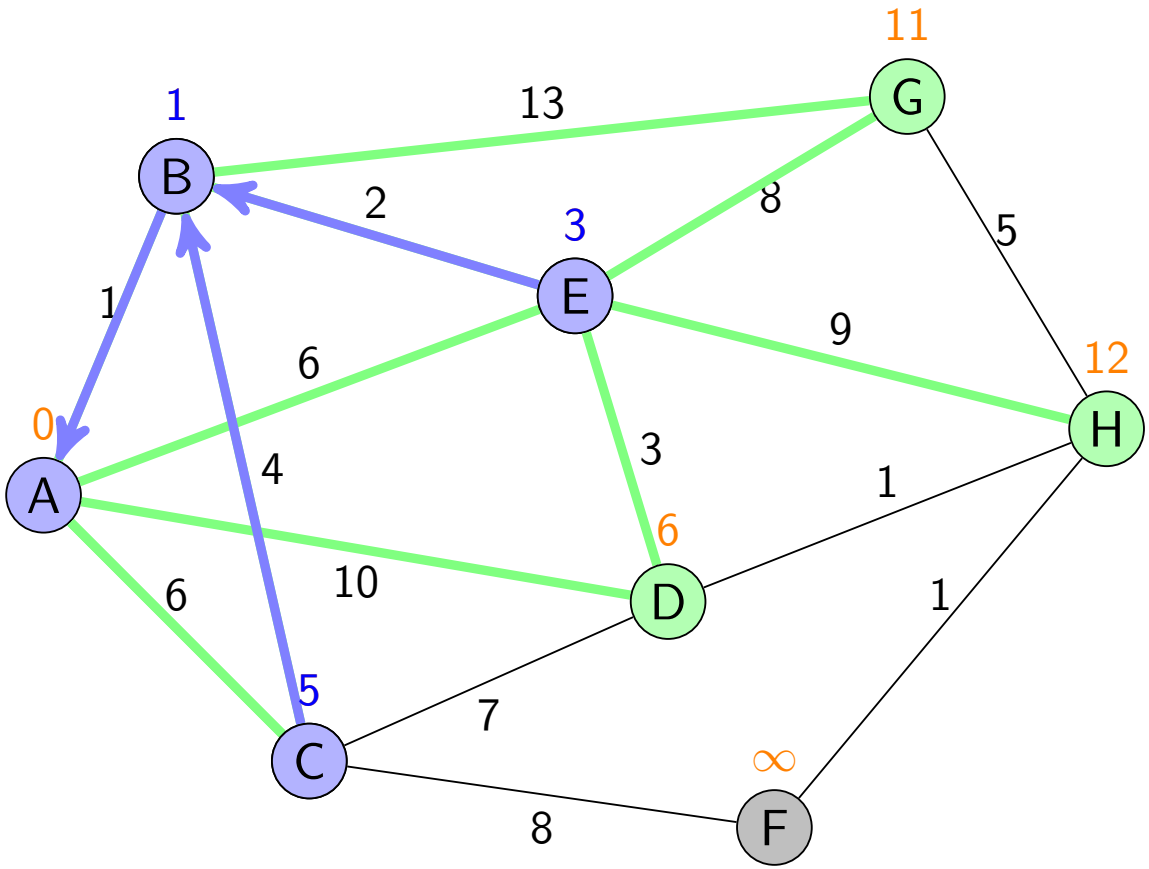
Knoten E wird fertiggestellt. Bei fertiggestellten Knoten merkt man sich wo man hergekommen ist (daher die blaue Kante).

Algorithmus von Dijkstra: Beispiel



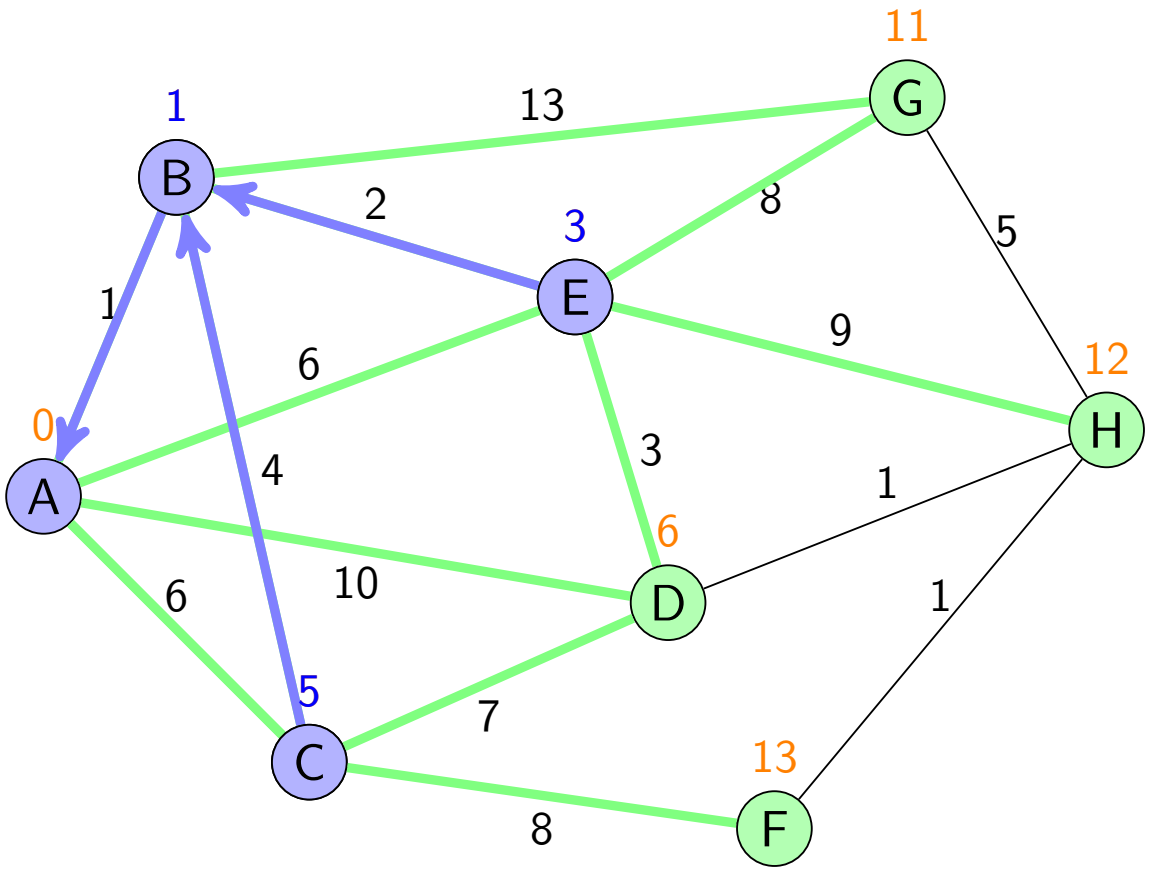
Berechnung neuer Zwischenergebnisse für D, G und H.

Algorithmus von Dijkstra: Beispiel



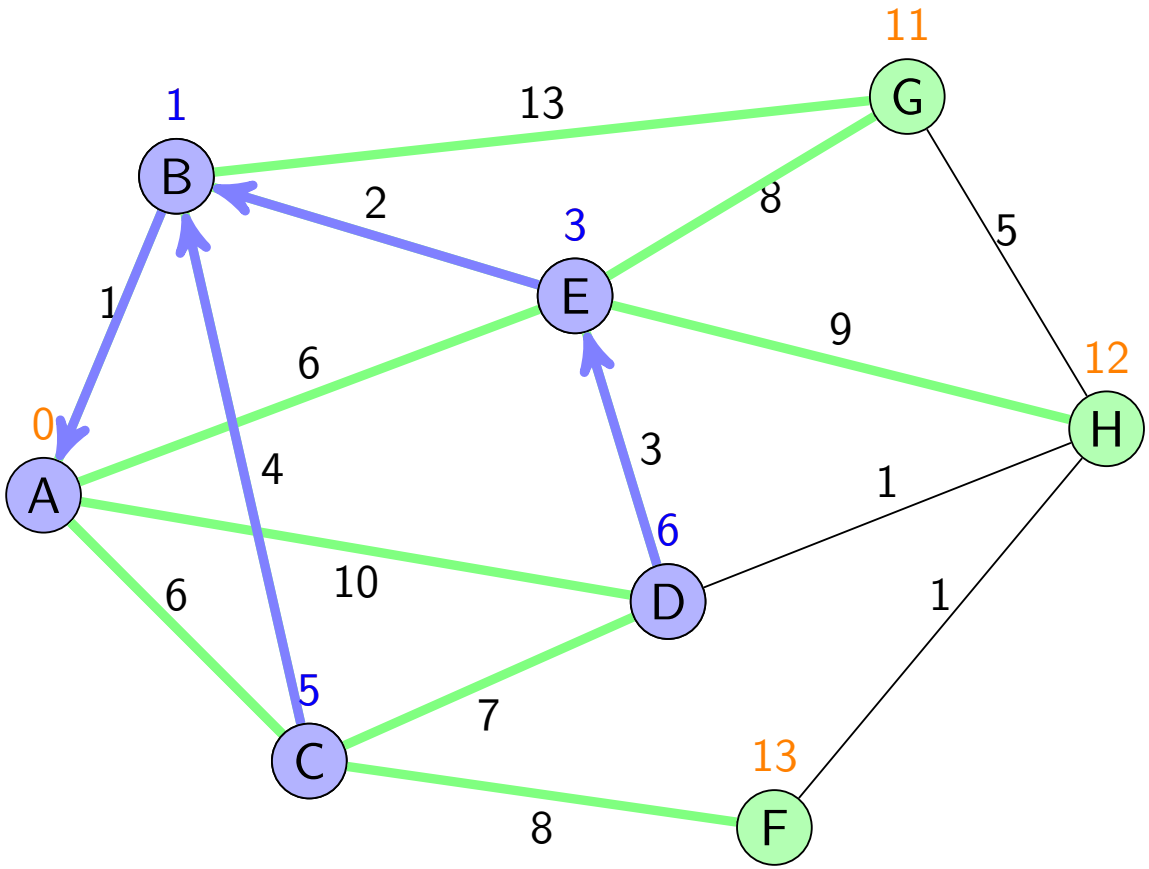
Knoten C wird fertiggestellt, da er nun den kleinsten Zwischenwert δ hat.

Algorithmus von Dijkstra: Beispiel



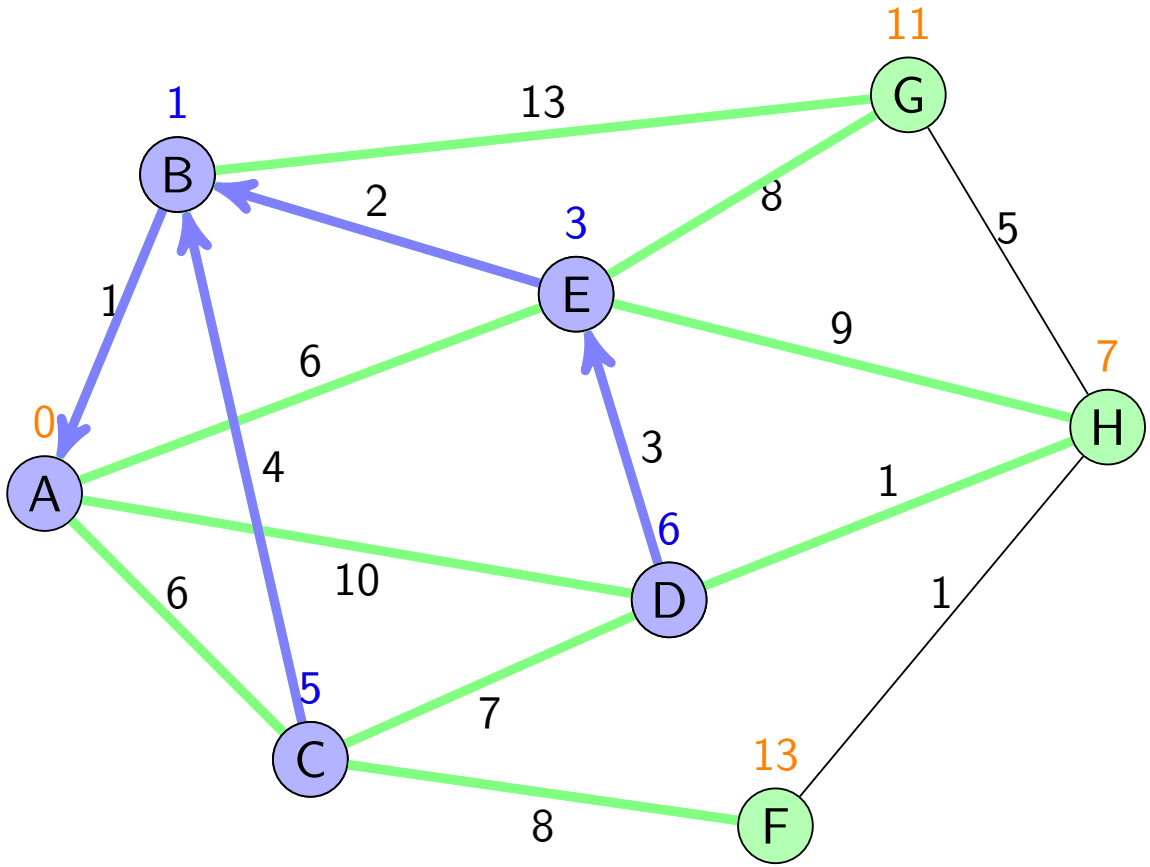
Ausgehend von C werden die Werte δ_D und δ_F berechnet. Da $5 + 7 > 6$ kommt es bei δ_D zu keiner Änderung.

Algorithmus von Dijkstra: Beispiel



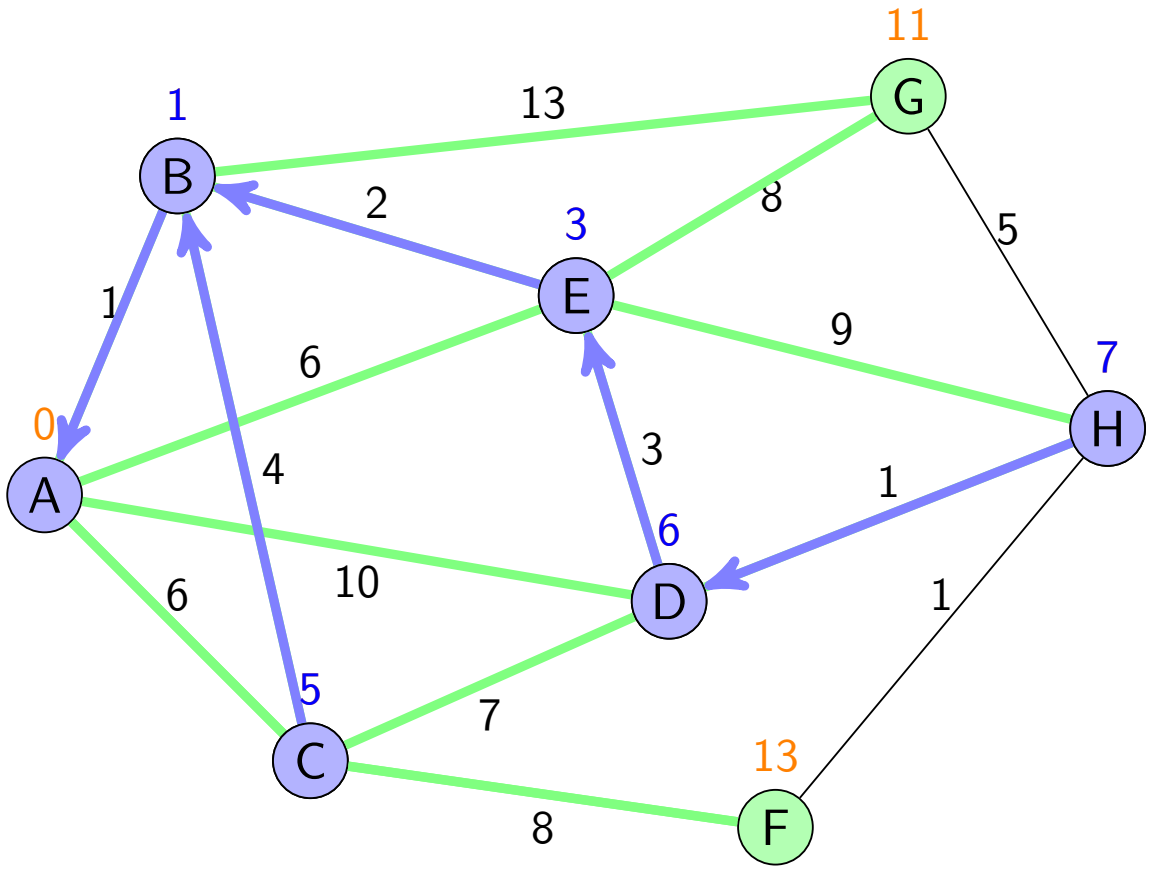
Fortgefahren wird mit Knoten D, da kleinstes δ.

Algorithmus von Dijkstra: Beispiel



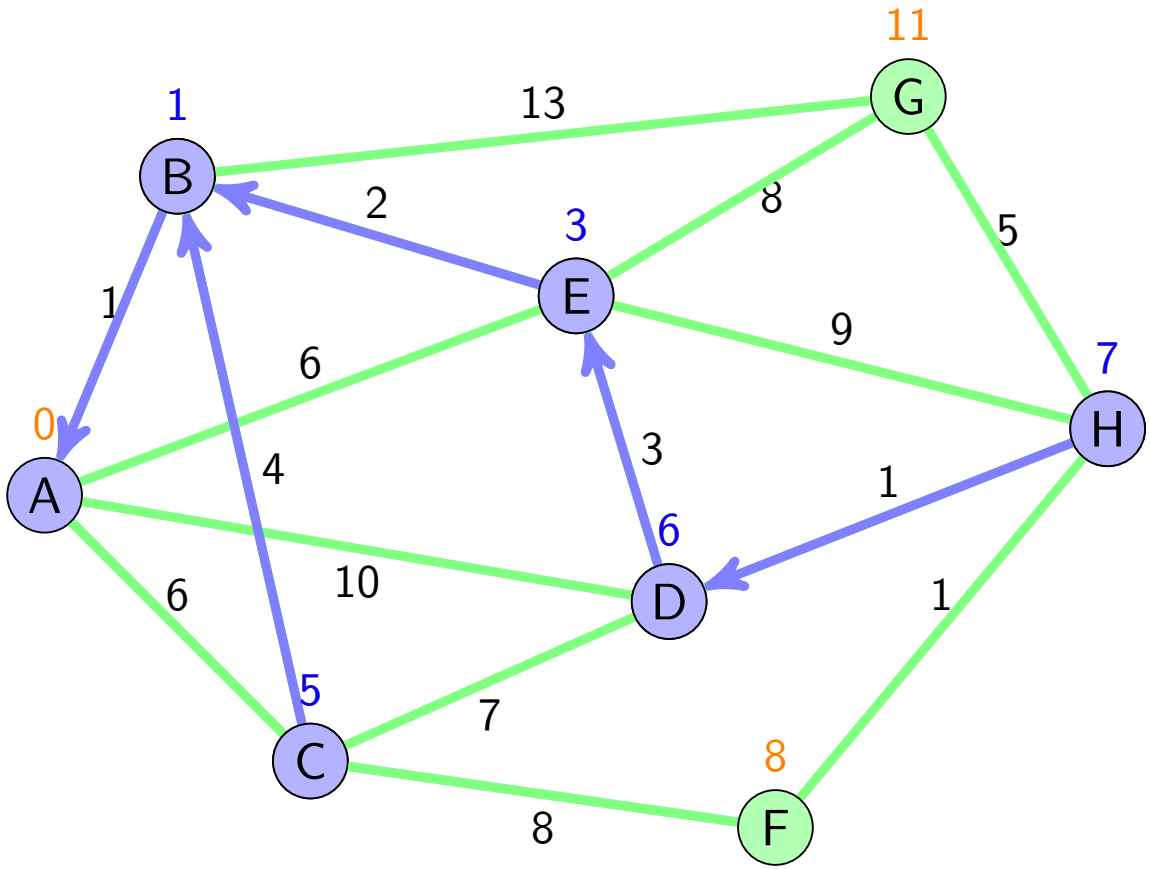
δ_H wird aktualisiert.

Algorithmus von Dijkstra: Beispiel



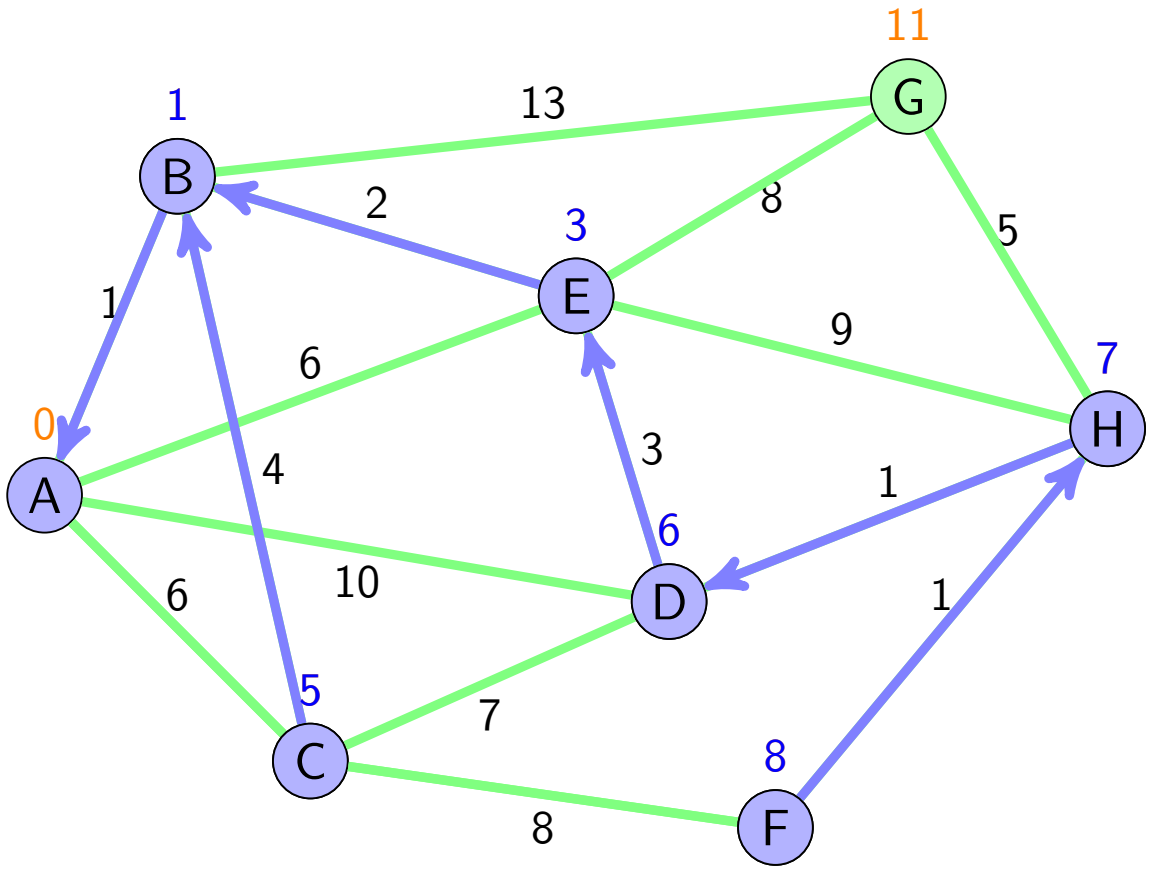
H wird fertiggestellt.

Algorithmus von Dijkstra: Beispiel



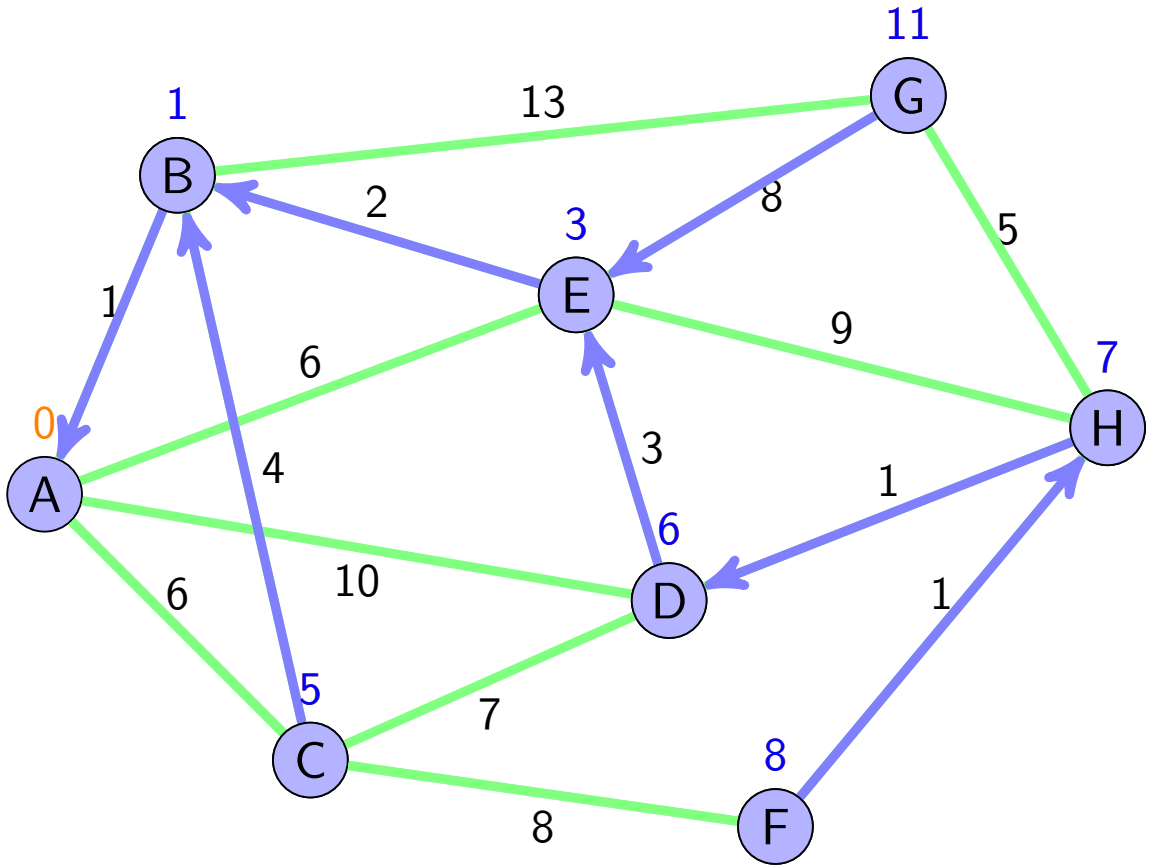
Update von δ_F und δ_G, wobei nur δ_F tatsächlich geändert wird.

Algorithmus von Dijkstra: Beispiel



F wird fertiggestellt.

Algorithmus von Dijkstra: Beispiel



G wird fertiggestellt (Vorgänger E).

Algorithmus von Dijkstra

- Die `blauen Kanten` bilden einen Wurzelbaum[1], der die kürzesten Wege vom Startknoten zu jedem Knoten enthält.
- Die Berechnung des kürzesten Weges von einem Startknoten zu einem Zielknoten beinhaltet also die Berechnung der kürzesten Wege vom Startknoten zu *allen* anderen Knoten.
- Ist nur der kürzeste Weg zu einem Zielknoten von Interesse, kann die Berechnung abgebrochen werden sobald dieser fertiggestellt wird.

[1] streng genommen bilden die umgedrehten blauen Kanten einen Wurzelbaum

Algorithmus von Dijkstra – Laufzeitanalyse

Mit $n = |V|$ und $m = |E|$ können wir die Laufzeiteigenschaften angeben. Diese hängen von der konkreten Umsetzung der Prioritätswarteschlange Q ab.

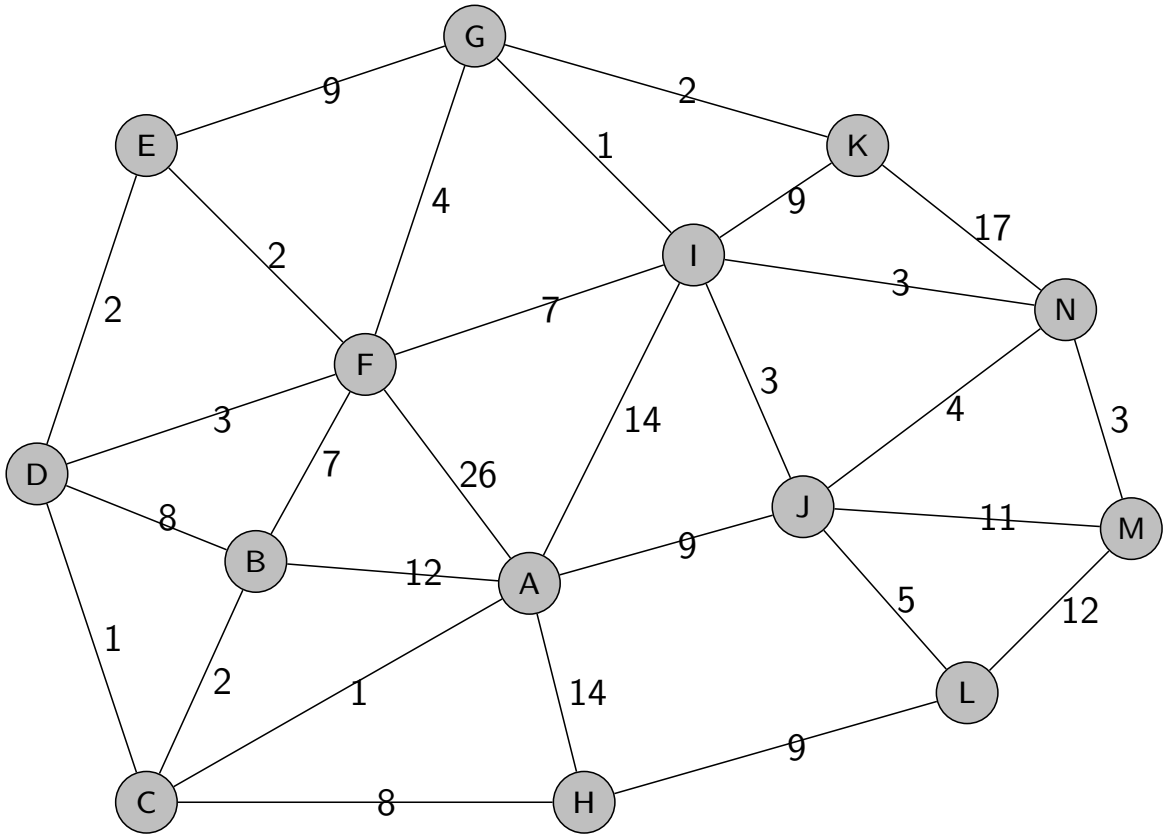
Operation		Queue Implementierung		
Name	Anzahl	Liste	Heap	Fibonacci Heap[2]
decreaseKey [10]	m	$O(1)$	$O(\log n)$	$O(1)$
getMin [5]	n	$O(n)$	$O(\log n)$	$O(\log n)$
create [3]	1	$O(n)$	$O(n)$	$O(n)$
Gesamt		$O(n^2 + m)$ $= O(n^2)$	$O((n+m)\log n)$	$O(n \log n + m)$

Anmerkung: In der Spalte ganz links ist in eckigen Klammern auf die Zeile der jeweiligen Operation im Pseudocode verwiesen!

[2] amortisierte Laufzeit

Beispiel

Berechnen Sie mit dem Algorithmus von Dijkstra für den gegebenen Graphen den kürzesten Weg vom Knoten B zum Knoten N. Der Algorithmus kann beendet werden, sobald der Zielknoten fertiggestellt wird.

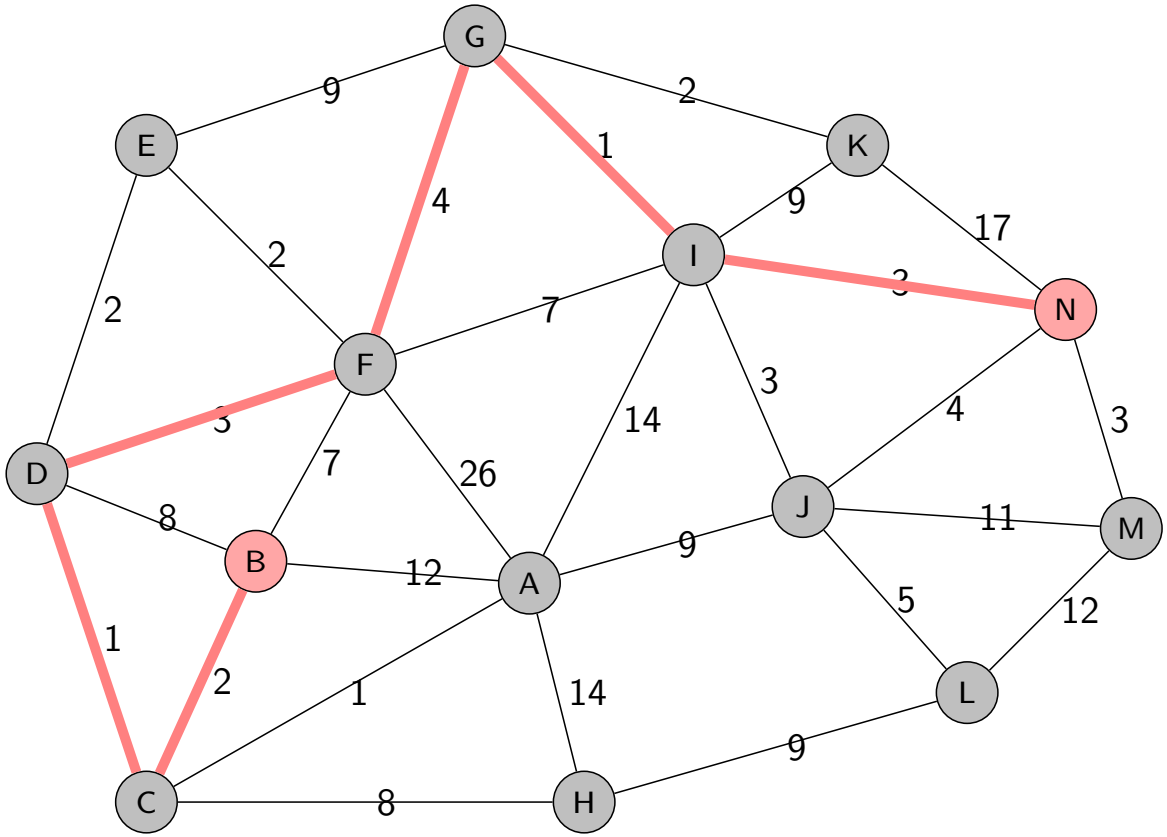


Beispiel

Abgeschlossen	Aktuelle δ-Werte (∞ = unbekannt)													
	A	B	C	D	E	F	G	H	I	J	K	L	M	N
B (0)	12	0	2	8	∞	7	∞	∞	∞	∞	∞	∞	∞	∞
C (2)	3	0	2	3	∞	7	∞	10	∞	∞	∞	∞	∞	∞
A (3)	3	0	2	3	∞	7	∞	10	17	12	∞	∞	∞	∞
D (3)	3	0	2	3	5	6	∞	10	17	12	∞	∞	∞	∞
E (5)	3	0	2	3	5	6	14	10	17	12	∞	∞	∞	∞
F (6)	3	0	2	3	5	6	10	10	13	12	∞	∞	∞	∞
G (10)	3	0	2	3	5	6	10	10	11	12	12	∞	∞	∞
H (10)	3	0	2	3	5	6	10	10	11	12	12	19	∞	∞
I (11)	3	0	2	3	5	6	10	10	11	12	12	19	∞	14
J (12)	3	0	2	3	5	6	10	10	11	12	12	17	23	14
K (12)	3	0	2	3	5	6	10	10	11	12	12	17	23	14
N (14) (Stop)	3	0	2	3	5	6	10	10	11	12	12	17	23	14

Beispiel

Kürzester Pfad (Länge 14) im Graphen G_1:



Minimale Spannbäume

Definition 1 (Spannbaum)

Ein *Spannbaum* T zu einem Graphen $G = (V, E)$ ist ein (auf-)spannender Teilgraph von G der ein Baum ist.

Wir betrachten nun ungerichtete, schlichte und zusammenhängende Graphen $G = (V, E)$ mit einer Gewichtsfunktion $w : E \to \mathbb{R}^+$ auf den Kanten $e \in E(G)$:

Definition 2 (Minimaler Spannbaum)

Ein *Minimaler Spannbaum* T von $G = (V, E)$ mit $w : E \to \mathbb{R}^+$ für alle $e \in E(G)$ ist ein zusammenhängender Teilgraph mit $|E(T)| = |V(G)| - 1$ der alle Knoten enthält, und für den gilt:

$$\sum_{e \in E(T)} w(e) = \min$$

Ein Minimaler Spannbaum (Minimum Spanning Tree (MST)) ist also ein Baum mit minimalen Kantengewichten (=Kantenkosten).

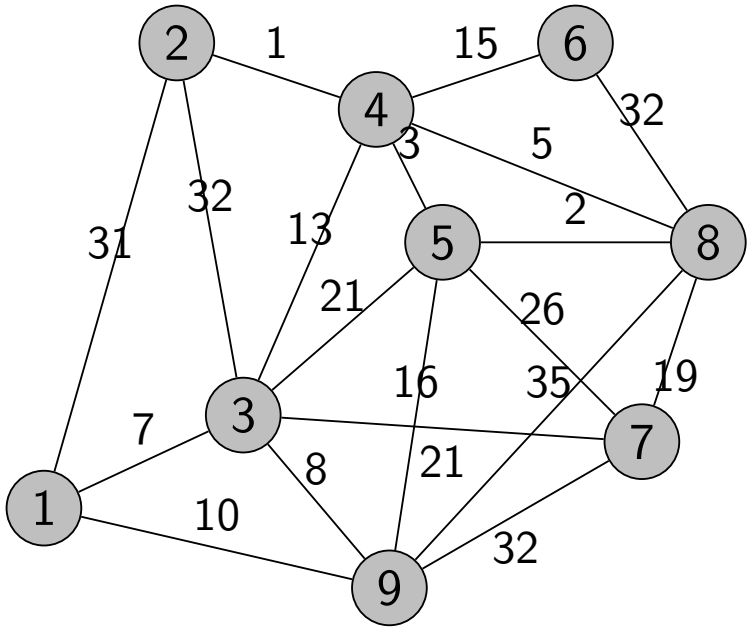
Algorithmus von Kruskal

Algorithmus 4: Algorithmus von Kruskal

1 **Function** *Kruskal(Graph* $G = (V, E)$*)*
  Result: Minimum Spanning Tree T
2 Ordne alle Kanten $e \in E(G)$ aufsteigend nach $w(e)$;
3 $\Rightarrow L = (e_1, e_2, \ldots, e_n)$ mit $w(e_1) \leq w(e_2) \leq \ldots \leq w(e_n)$
4 $V(T) = V(G)$;
5 $E(T) = \emptyset$;
6 **for** $e \in L$ **do**
7 **if** $T = (V, E(T) \cup \{e\})$ *ist kreisfrei* **then**
8 $E(T) = E(T) \cup \{e\}$;

- Die Kreisfreiheit kann mit DFS berechnet werden.
- Nach Hinzufügen von $n - 1$ Kanten kann der Algorithmus beendet werden.

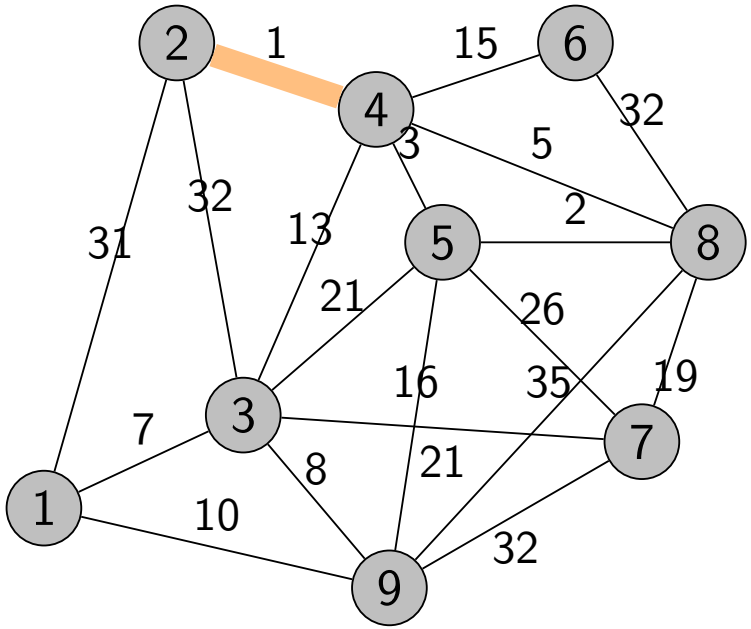
Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$
$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$
$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$
$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

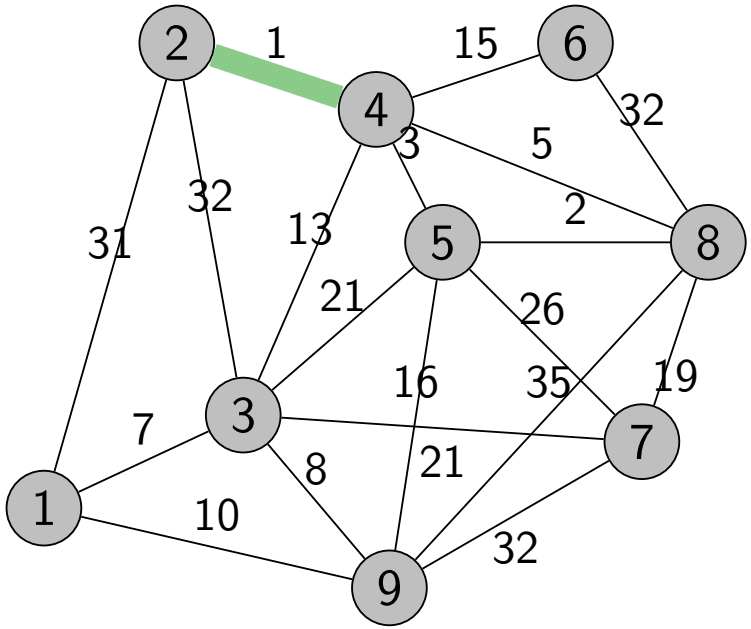
Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$
$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$
$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$
$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

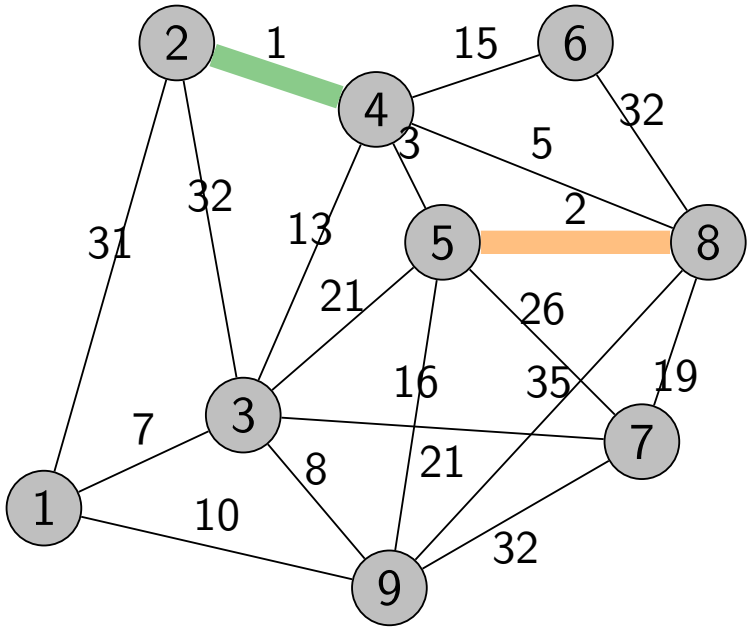
Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$
$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$
$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$
$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

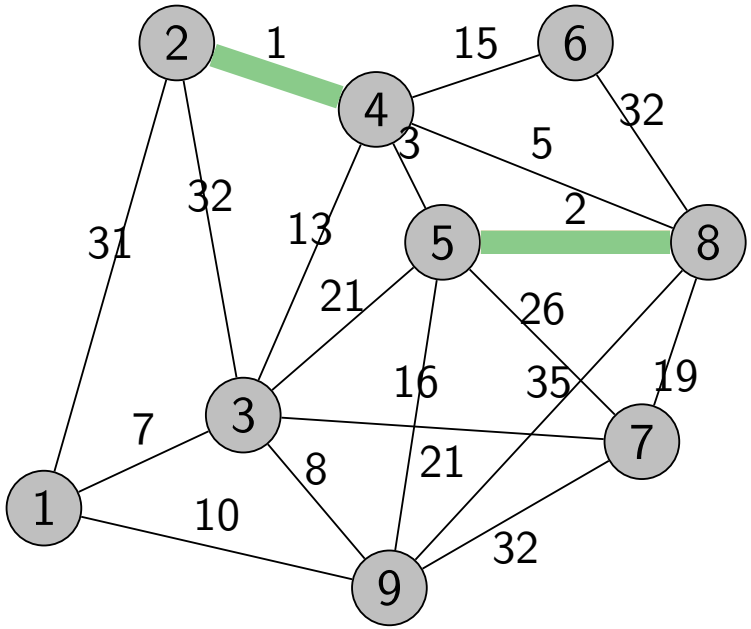
Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$
$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$
$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$
$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

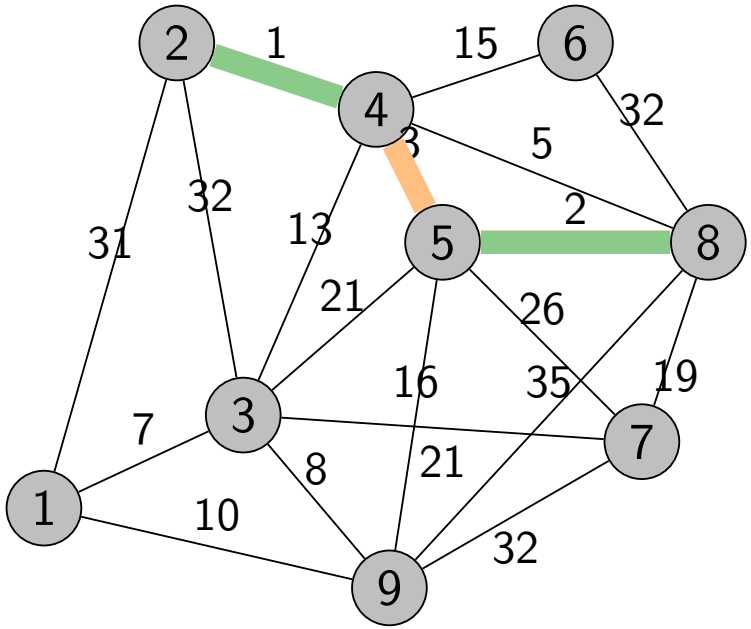
$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$

$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$

$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$

$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

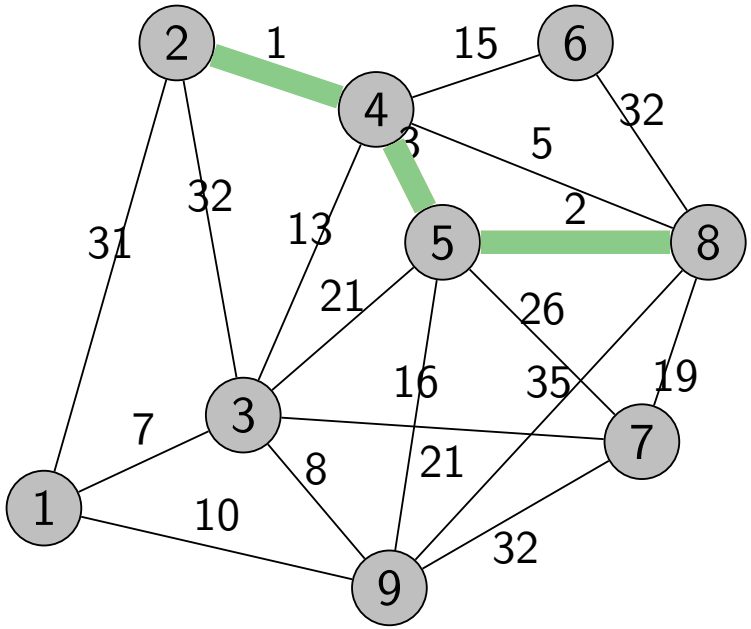
Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$
$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$
$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$
$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

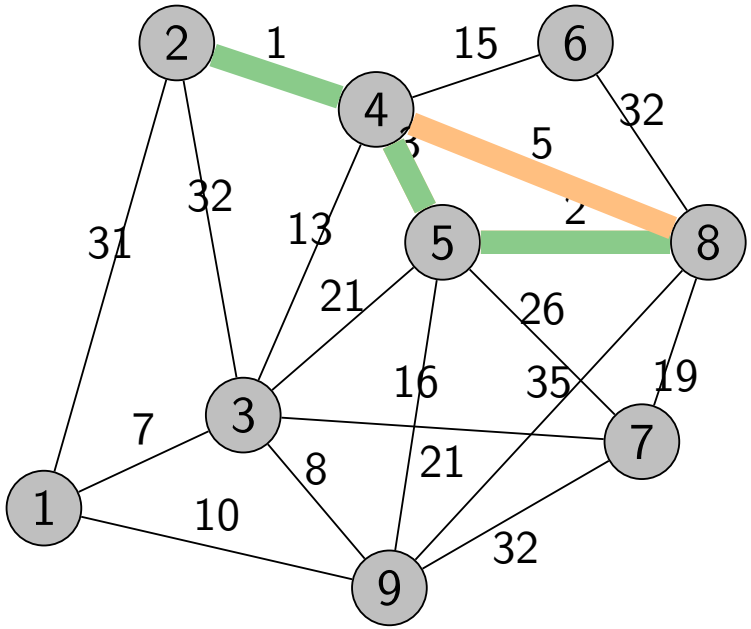
$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$

$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$

$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$

$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

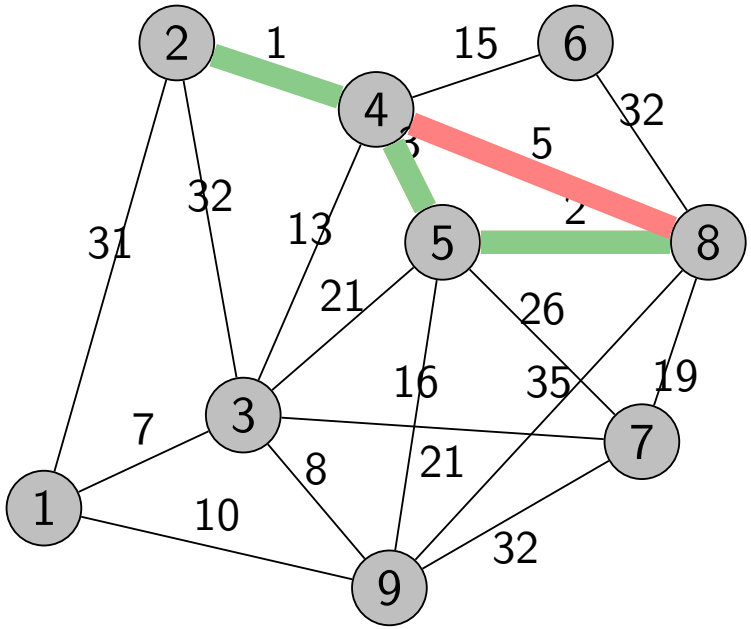
Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$
$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$
$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$
$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

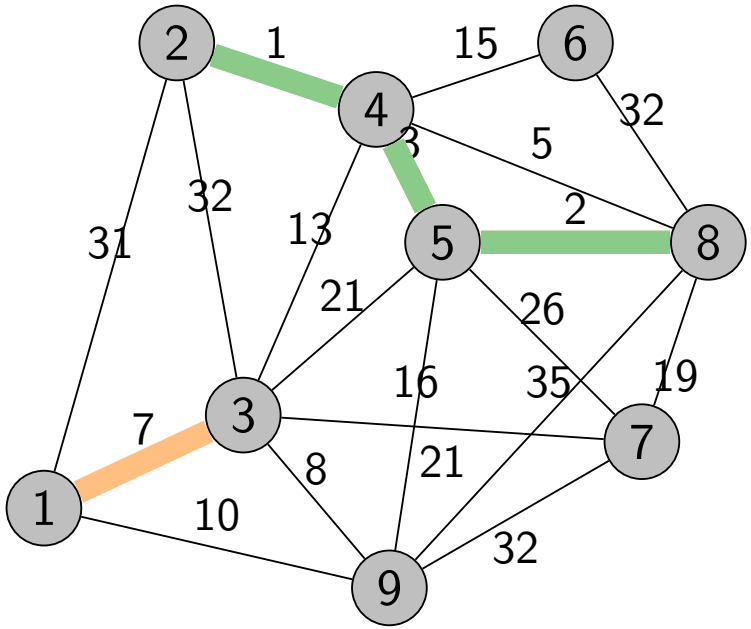
Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$
$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$
$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$
$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

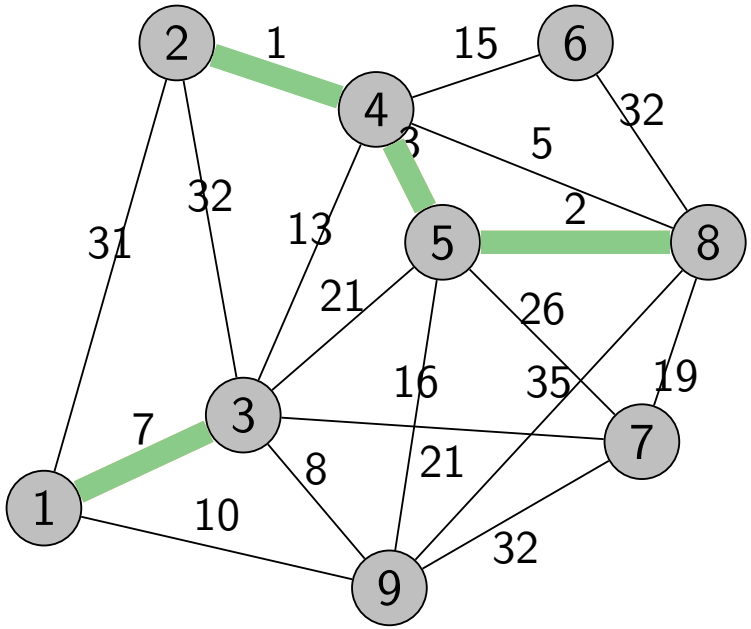
Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$
$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$
$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$
$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

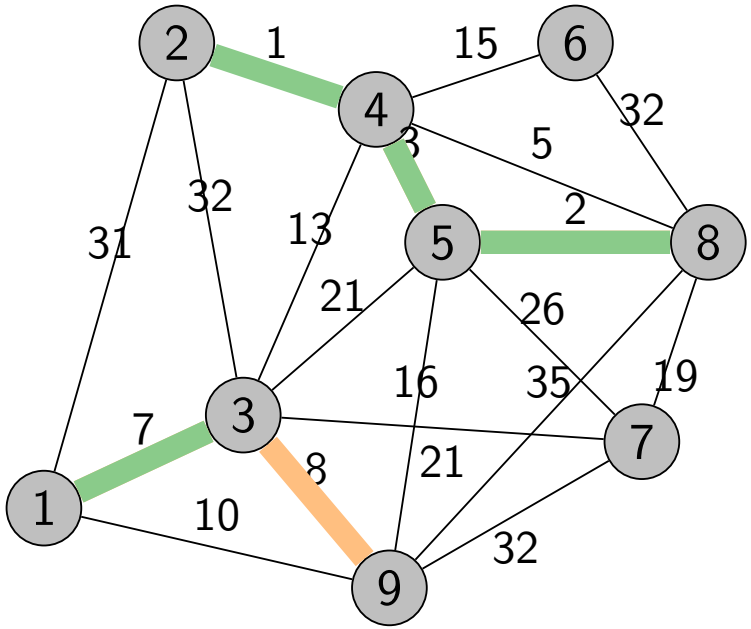
Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$
$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$
$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$
$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

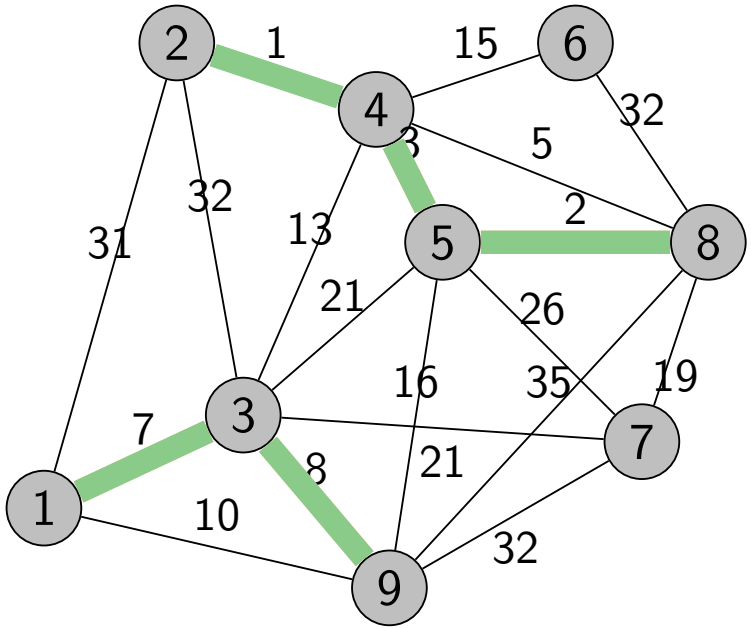
Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$
$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$
$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$
$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

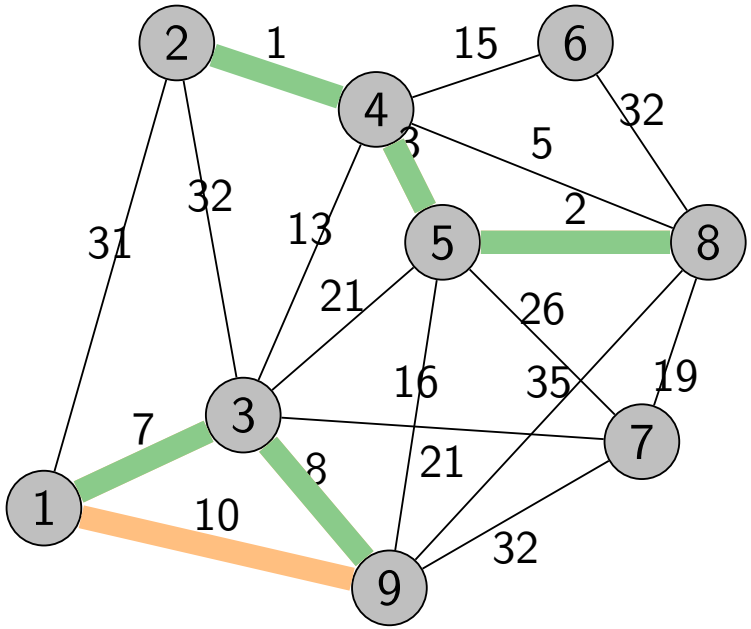
Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$
$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$
$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$
$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

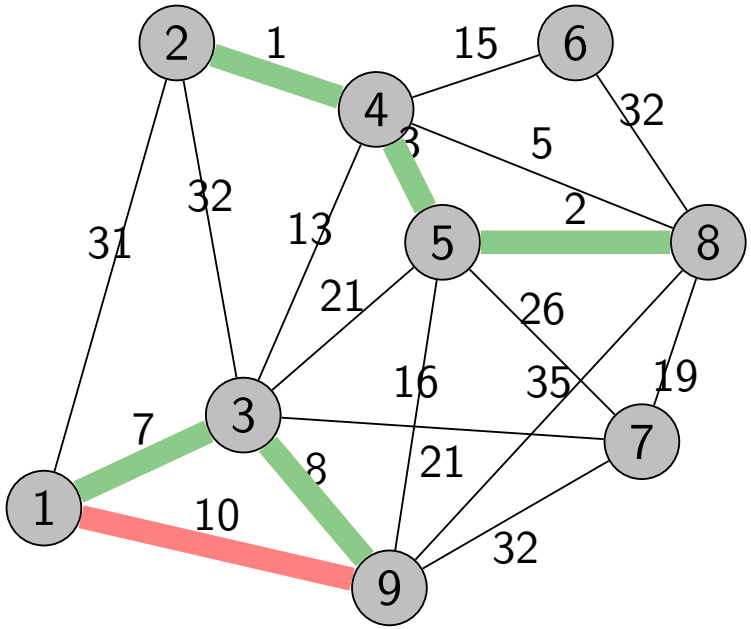
Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$
$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$
$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$
$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

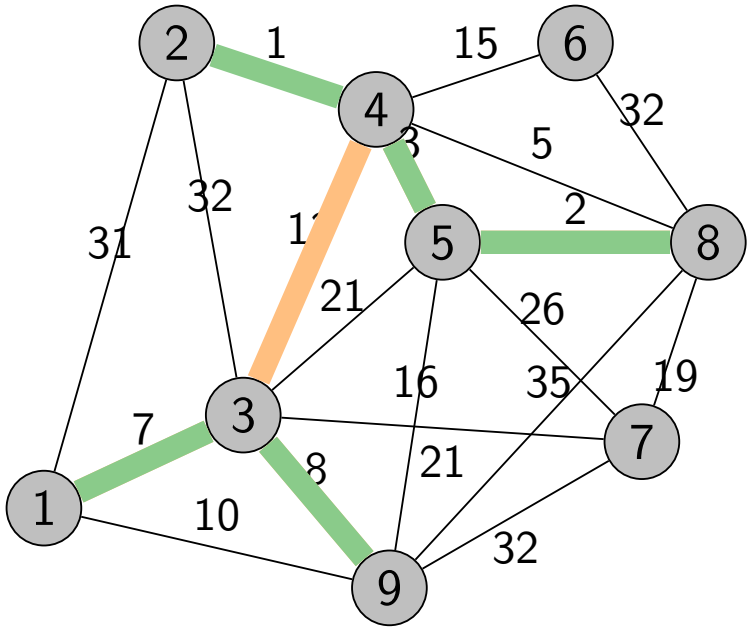
Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$
$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$
$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$
$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

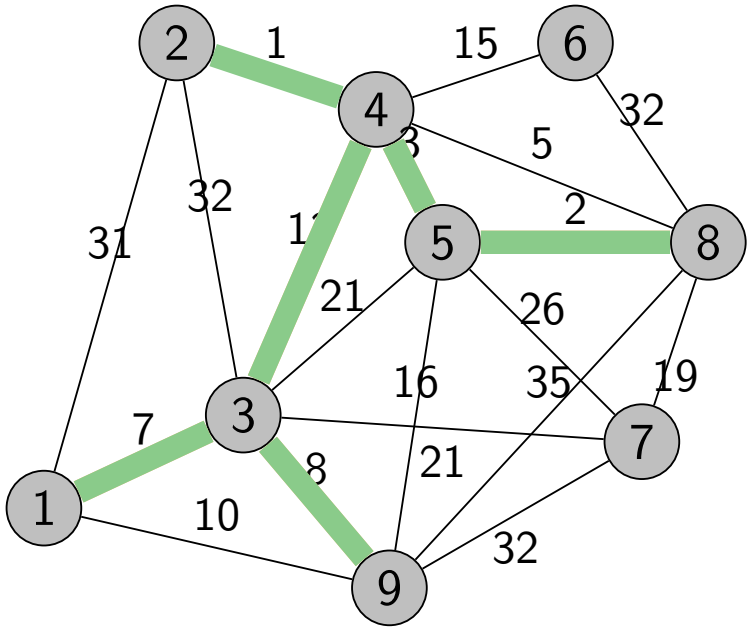
Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$
$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$
$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$
$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

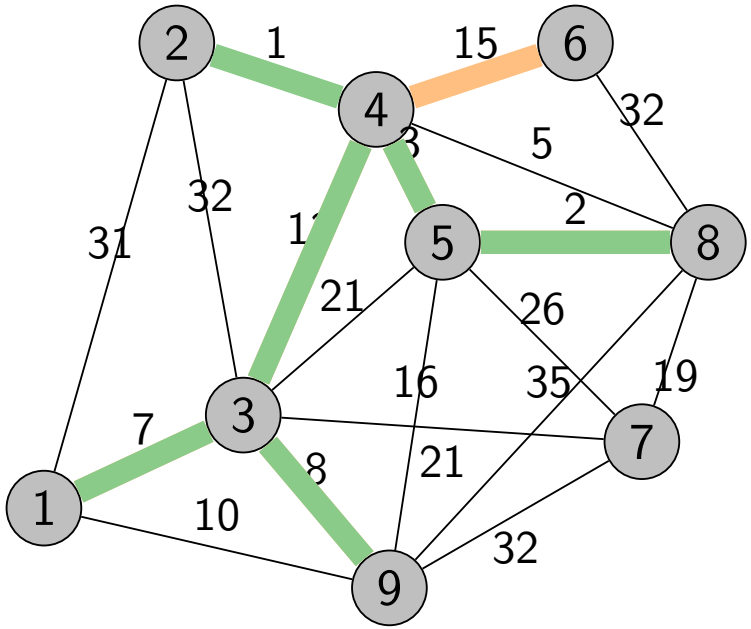
$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$

$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$

$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$

$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

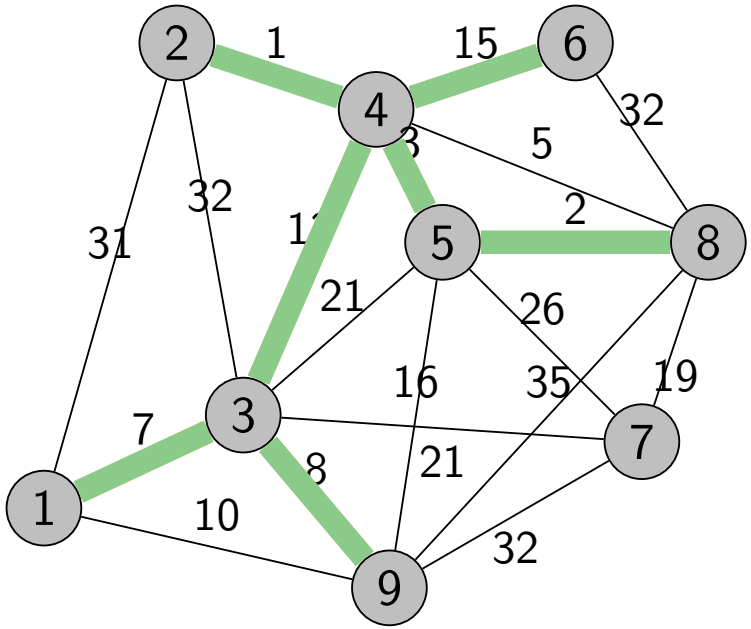
$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$

$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$

$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$

$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

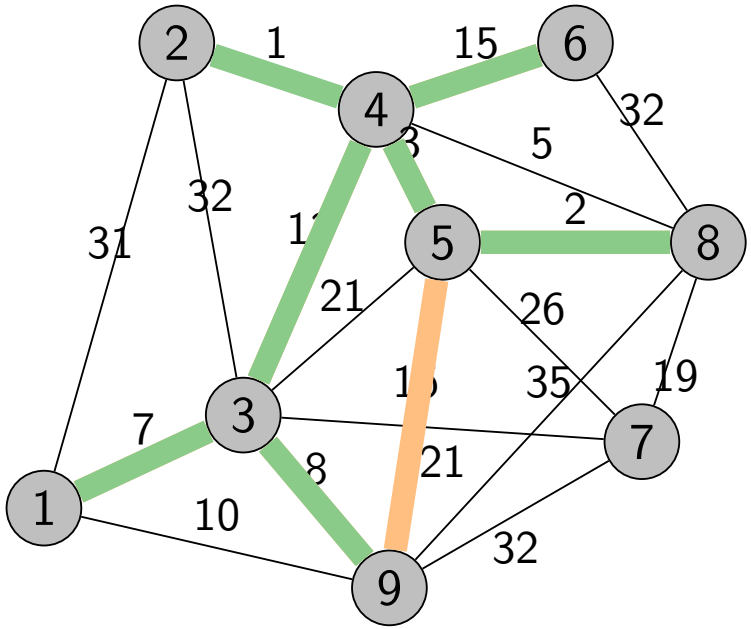
$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$

$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$

$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$

$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

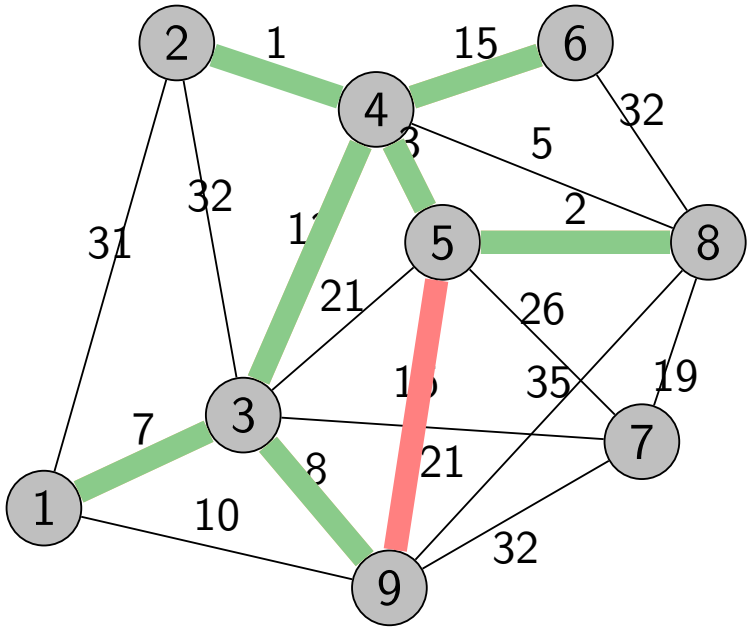
$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$

$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$

$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$

$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

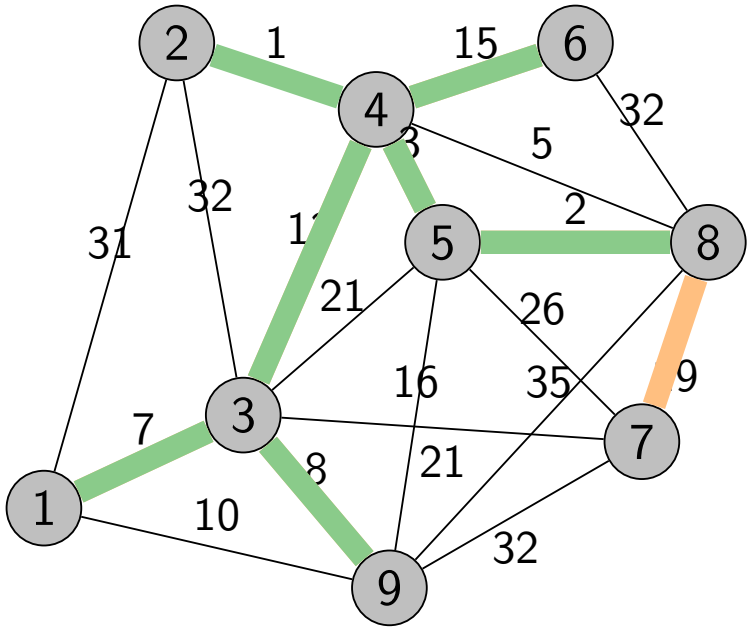
Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$
$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$
$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$
$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

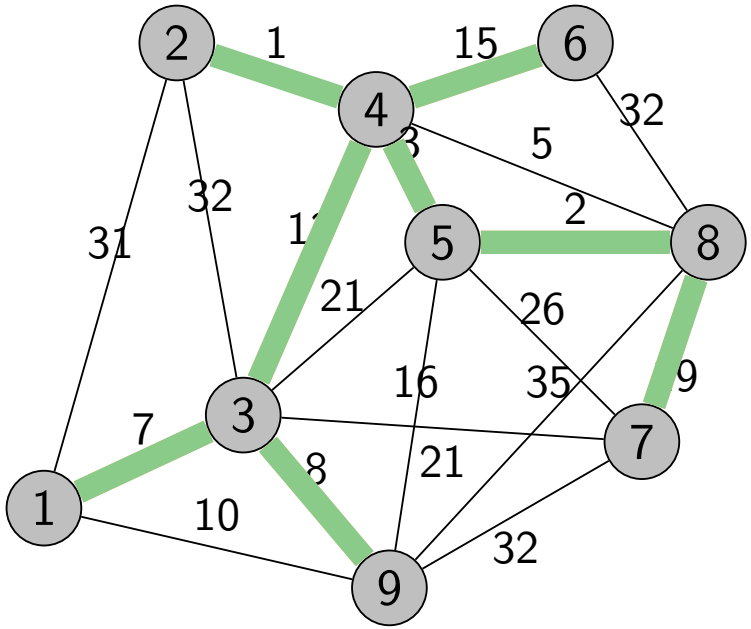
$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$

$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$

$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$

$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

Algorithmus von Kruskal



Aufsteigend sortierte Kanten:

$w(\{2,4\}) = 1, w(\{5,8\}) = 2, w(\{4,5\}) = 3, w(\{4,8\}) = 5$

$w(\{1,3\}) = 7, w(\{3,9\}) = 8, w(\{1,9\}) = 10, w(\{3,4\}) = 13$

$w(\{4,6\}) = 15, w(\{5,9\}) = 16, w(\{7,8\}) = 19, w(\{3,5\}) = 21$

$w(\{3,7\}) = 21, w(\{5,7\}) = 26, \ldots$

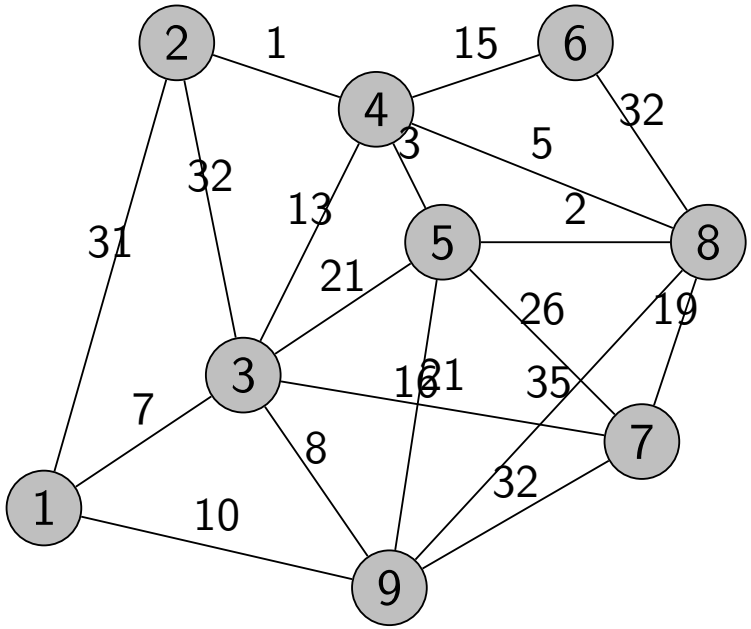
Algorithmus von Prim

Der Algorithmus von Prim konstruiert ebenfalls einen MST.

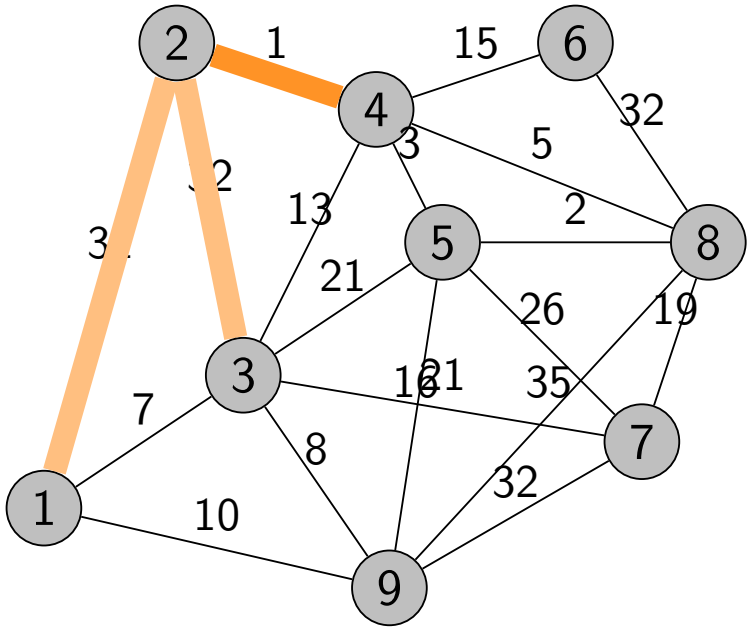
Algorithmus 5: Algorithmus von Prim

```
1  Function Prim(Graph G)
       Result: Minimum Spanning Tree T
2      E(T) = ∅;
3      V(T) = ein zufaellig gewaehlter Startknoten;
4      while V(T) ⊂ V(G) do
5          Sei E' die Menge aller Kanten zwischen Knoten
6          aus V(T) und V(G)\V(T)
7          e' = argmin_{e∈E'} w(e); // Kante mit kleinstem Gewicht
8          E(T) = E(T) ∪ e';
9          Füge neuen Knoten von e' zu V(T) hinzu;
```

Algorithmus von Prim

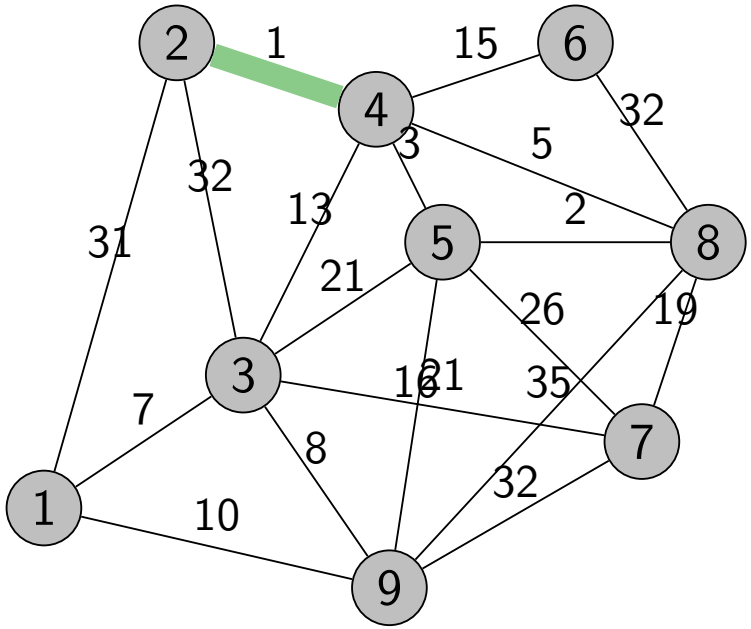


Algorithmus von Prim

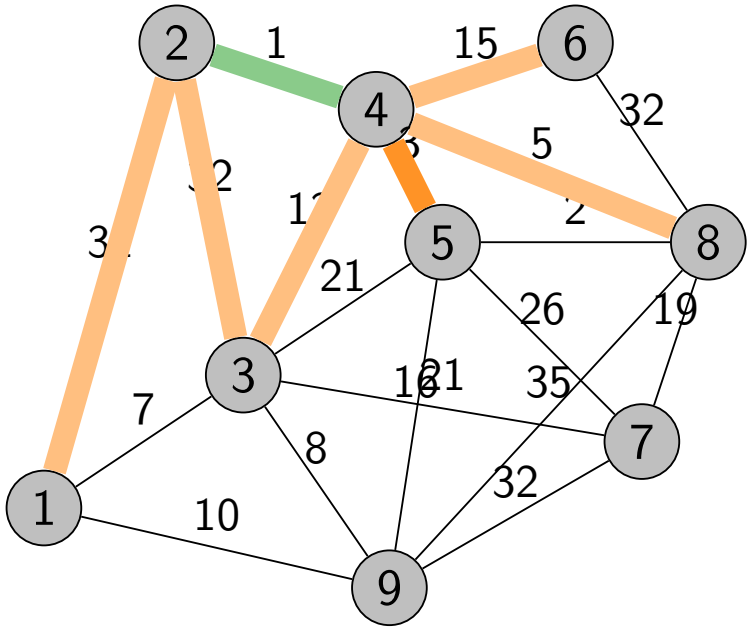


Startknoten: 2

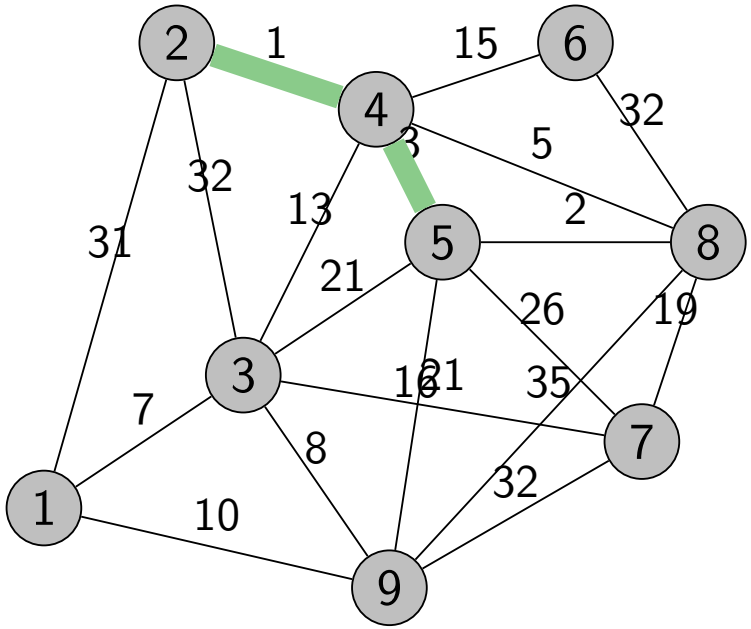
Algorithmus von Prim



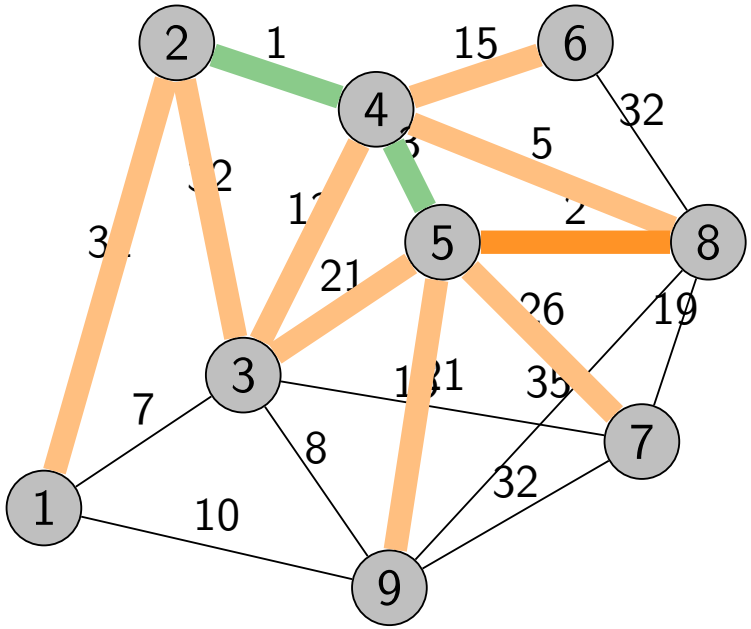
Algorithmus von Prim



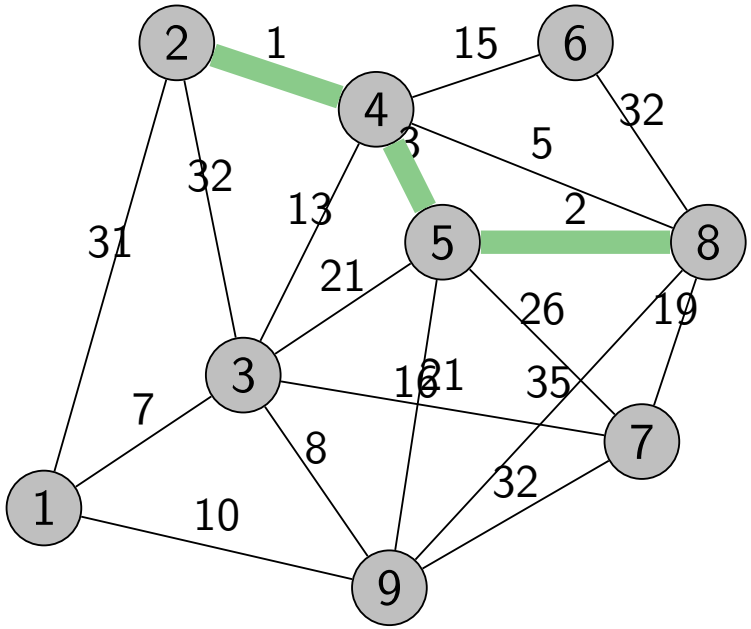
Algorithmus von Prim



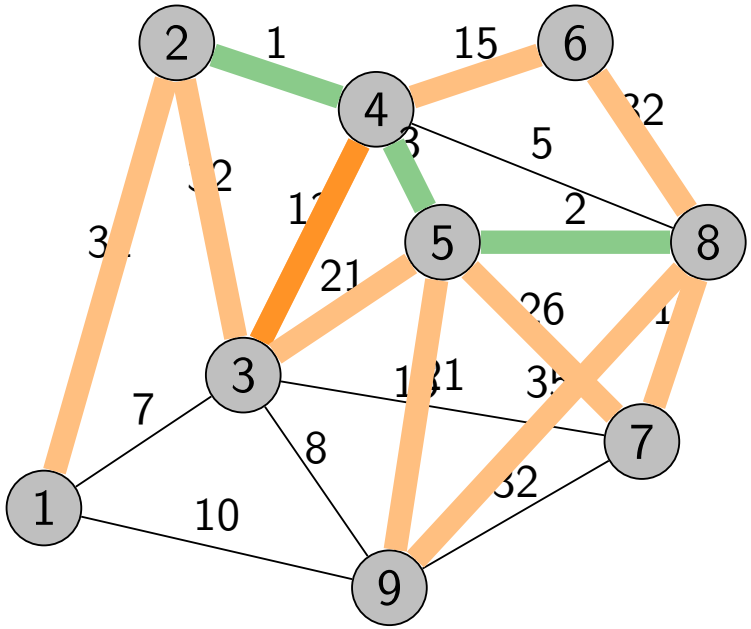
Algorithmus von Prim



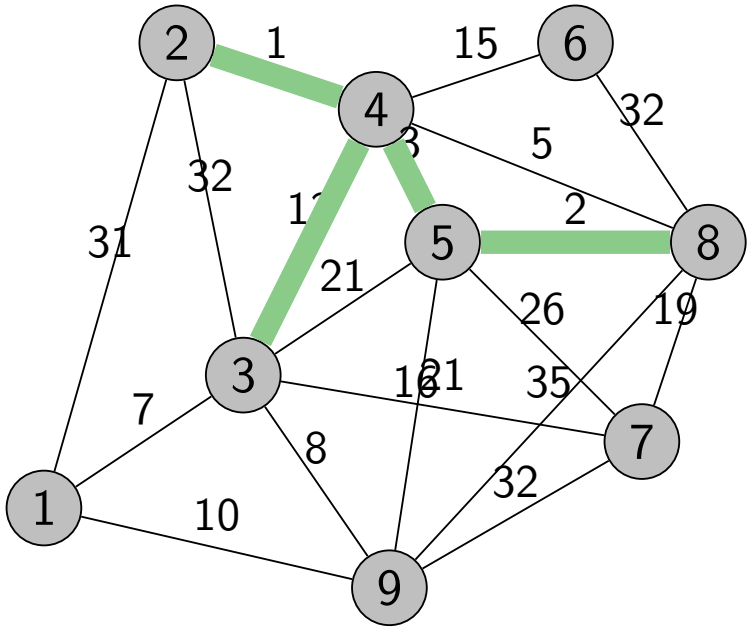
Algorithmus von Prim



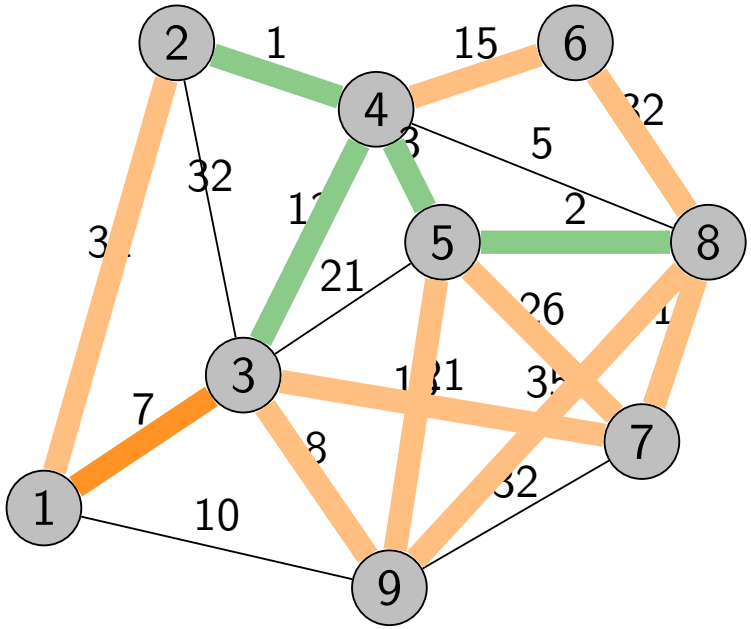
Algorithmus von Prim



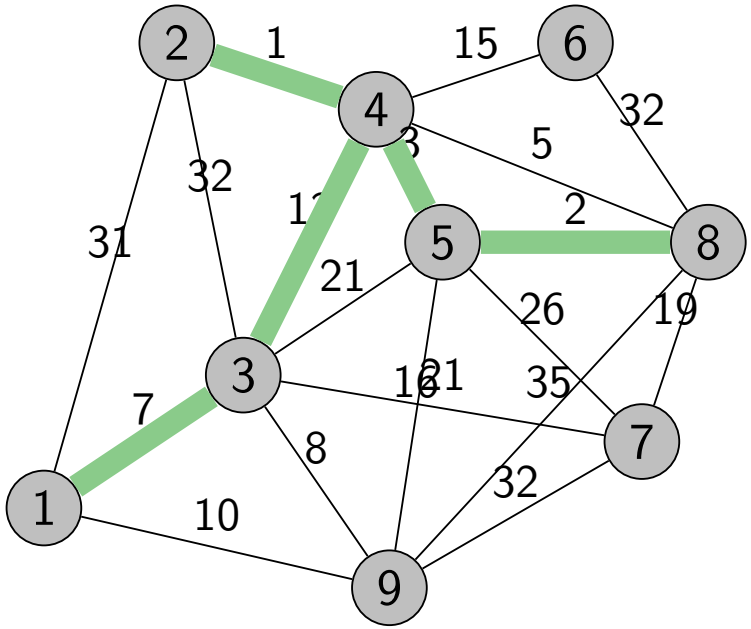
Algorithmus von Prim



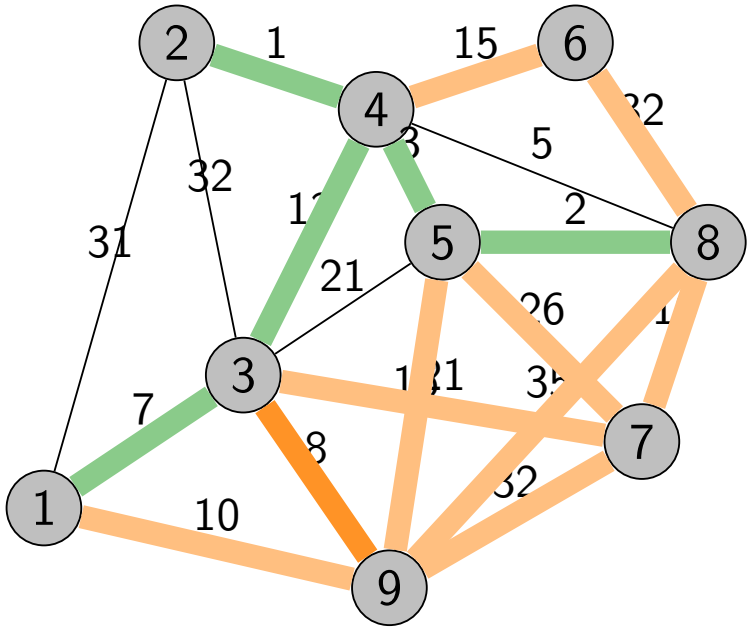
Algorithmus von Prim



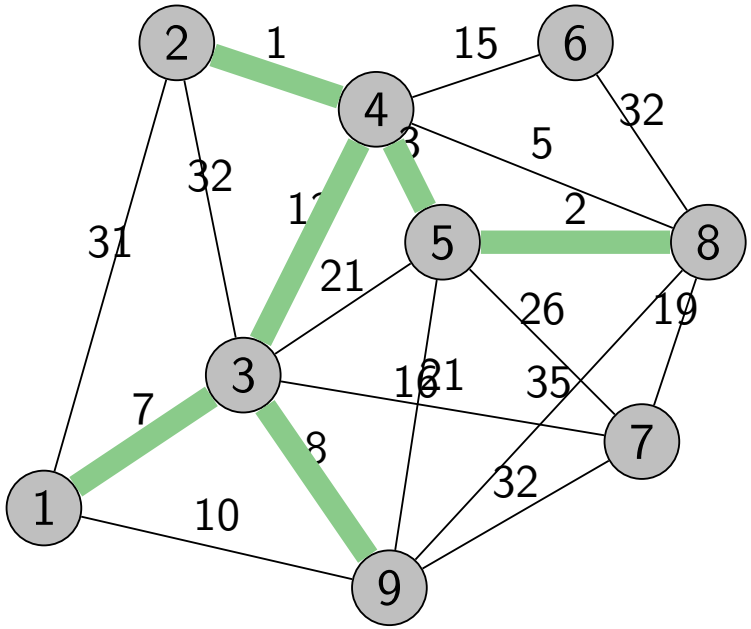
Algorithmus von Prim



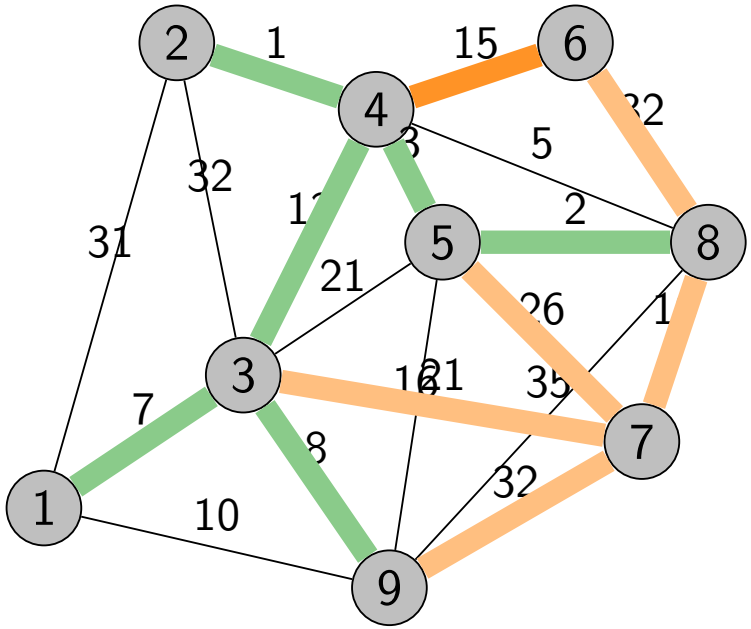
Algorithmus von Prim



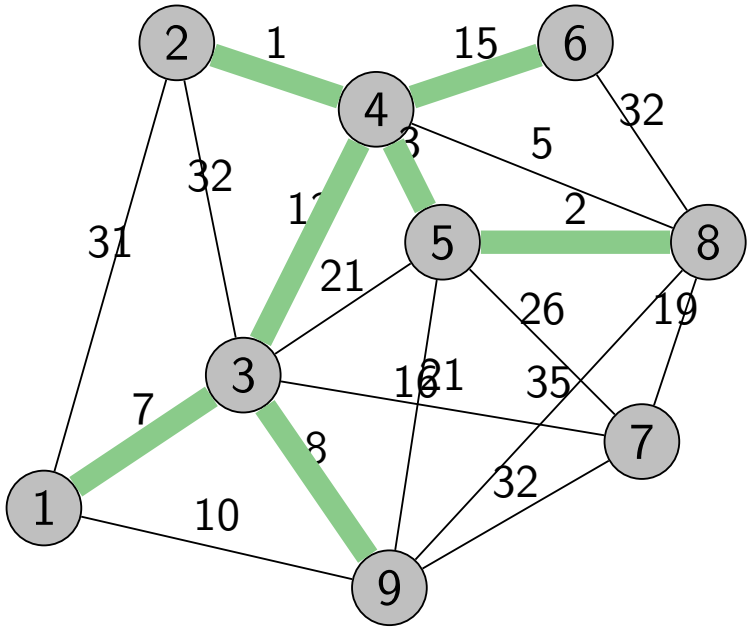
Algorithmus von Prim



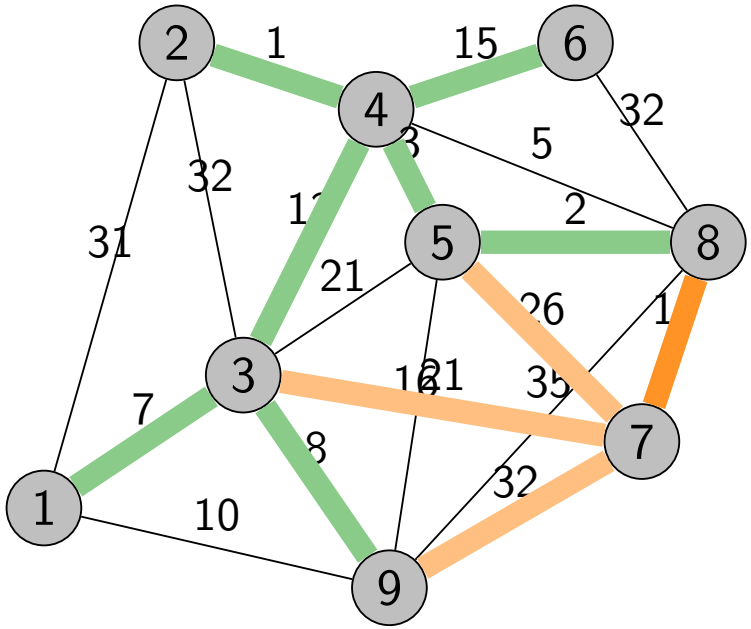
Algorithmus von Prim



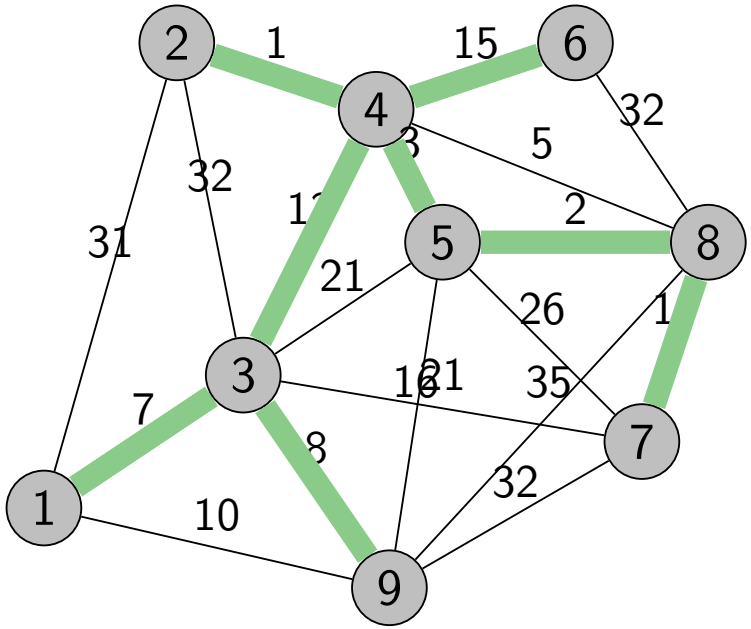
Algorithmus von Prim



Algorithmus von Prim



Algorithmus von Prim



Vergleich: Kruskal vs. Prim

- Beide Algorithmen (Kruskal und Prim) finden den MST in gewichteten Graphen.
- Bei *dicht*[3] besetzten Graphen ist der Algorithmus von Prim jedoch effizienter.
- Bei *dünn*[4] besetzten Graphen ist Kruskal besser.
- Beim Algorithmus von Kurskal müssen die Kanten vorab sortiert werden, was bei vielen Kanten einen erheblichen Aufwand darstellt (sogar den Hauptaufwand des gesamten Verfahrens).
- Bei vergleichsweise wenigen Kanten überwiegen jedoch die Vorteile der einfacheren darauffolgenden Schritte.

---

[3] $|E| \in O(|V|^2)$, d.h. die Anzahl der Kanten liegt in der Größenordnung der Anzahl der Kanten eines vollständigen Graphen.

[4] $|E| \in O(|V|)$, d.h. die Anzahl der Kanten liegt in der Größenordnung der Anzahl der Knoten; der Graph enthält also relativ wenige Kanten.

Zusammenfassung (MST)

- Beide Algorithmen (Kruskal, Prim) sind sogenannte **Greedy-Algorithmen**
- Sie wählen in jedem Schritt ("gierig") die nächst-beste Erweiterung der Teillösung
- Normalerweise erreicht man mit einer derartigen Strategie nur **Näherungslösungen** (d.h. nicht die insgesamt beste Lösung)
- In diesem Fall finden jedoch beide Greedy-Algorithmen das **globale Optimum**, d.h. die insgesamt beste Lösung
- ...der Minimale Spannbaum kann also (wie der Euler-Zyklus) leicht gefunden werden.
- Andere Spannbaum-Probleme (Varianten) sind jedoch wesentlich schwieriger!

Chinese Postman Problem

- (Geschlossene) Eulersche-Linien haben große praktische Bedeutung für
  - Abfallentsorgung
  - Schneeräumung
  - Briefzustellung
- Ein Straßengraph ist jedoch nicht unbedingt *Eulersch*
- Ebenso spielt die Länge der einzelnen Straßen eine Rolle!
- **Chinese Postman Problem:**
  - Benannt nach chinesischem Mathematiker Mei Ko Kwan (*1934)
  - **Gegeben:** *gewichteter* Graph (nicht notwendigerweise Eulersch)
  - **Ziel:** Finde kürzesten Zyklus im Graphen, der jede Kante *mindestens* einmal enthält
  - $\Rightarrow$ möglichst kurze Kanten sollen gegebenenfalls doppelt benutzt werden.

Einschub: Paarung/Matching

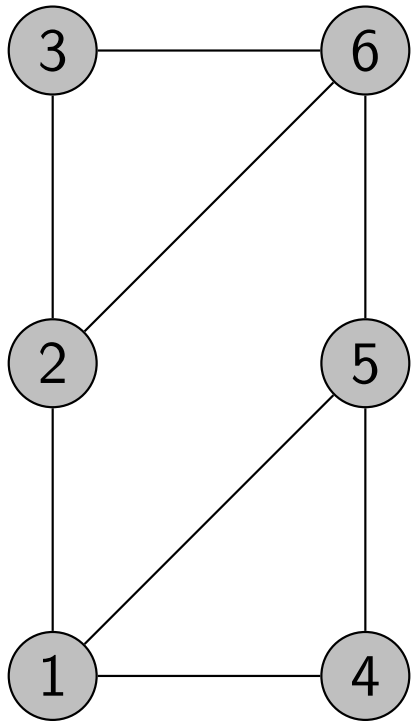
- Bildung von 2-elementigen Teilmengen ("Paaren") von Knoten

Definition 3 (Paarung/Matching)

Gegeben sei ein Graph $G = (V, E)$. Eine Menge $M \subseteq E$ heißt *Matching* (oder Paarung), wenn kein Knoten aus V zu mehr als einer Kanten aus M inzident ist.

- Paarungen heißen **vollständig**, wenn alle Knoten gepaart sind.
- In gewichteten Graphen sind meist Paarungen minimalen oder maximalen Gewichts von Interesse, wobei

$$w(M) = \sum_{e \in M} w(e)$$



Beispiel für ein (vollständiges) Matching in einem ungewichteten Graphen

Einschub: Paarung/Matching

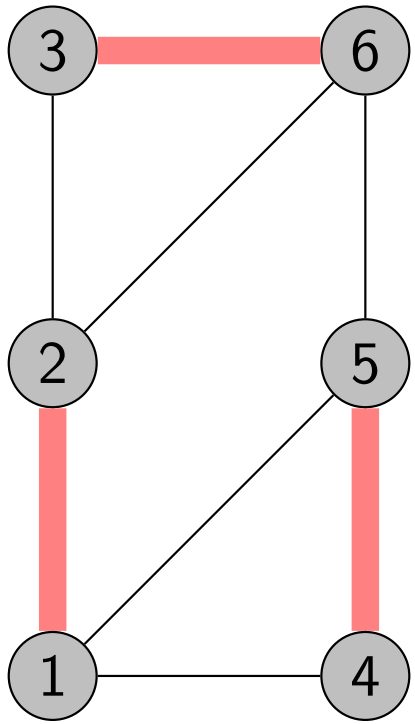
- Bildung von 2-elementigen Teilmengen ("Paaren") von Knoten

Definition 3 (Paarung/Matching)

Gegeben sei ein Graph $G = (V, E)$. Eine Menge $M \subseteq E$ heißt *Matching* (oder Paarung), wenn kein Knoten aus V zu mehr als einer Kanten aus M inzident ist.

- Paarungen heißen **vollständig**, wenn alle Knoten gepaart sind.
- In gewichteten Graphen sind meist Paarungen minimalen oder maximalen Gewichts von Interesse, wobei

$$w(M) = \sum_{e \in M} w(e)$$

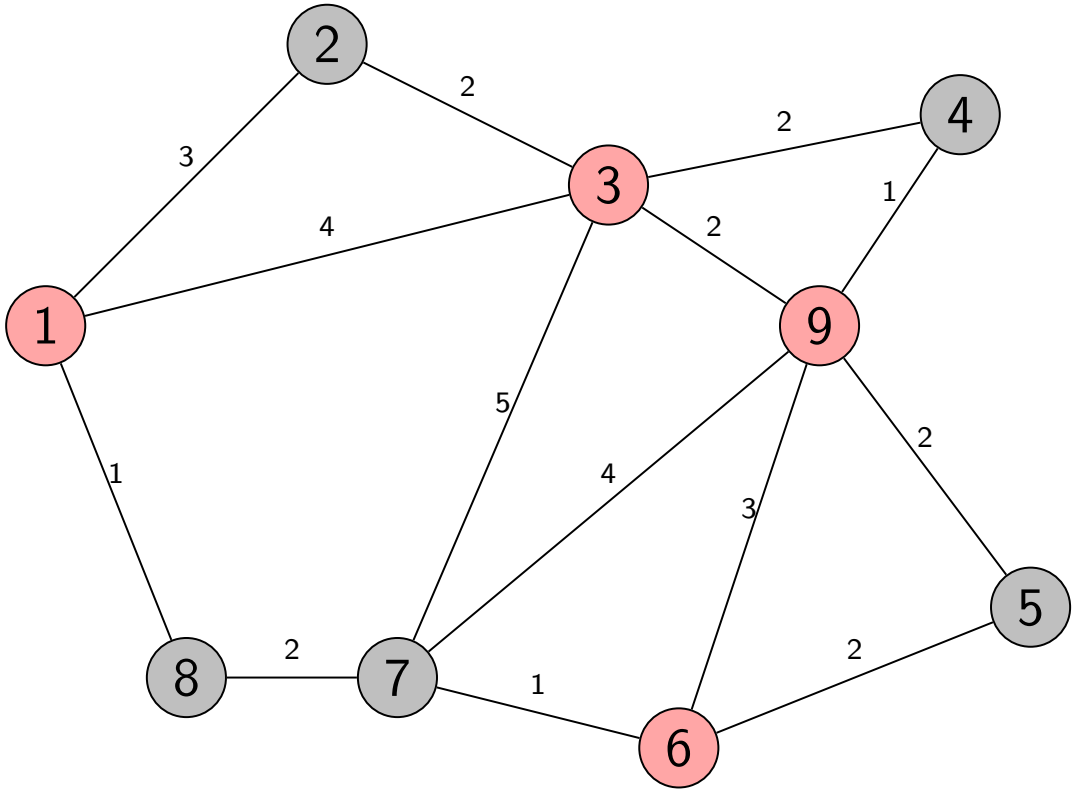


Beispiel für ein (vollständiges) Matching in einem ungewichteten Graphen

Chinese Postman Problem

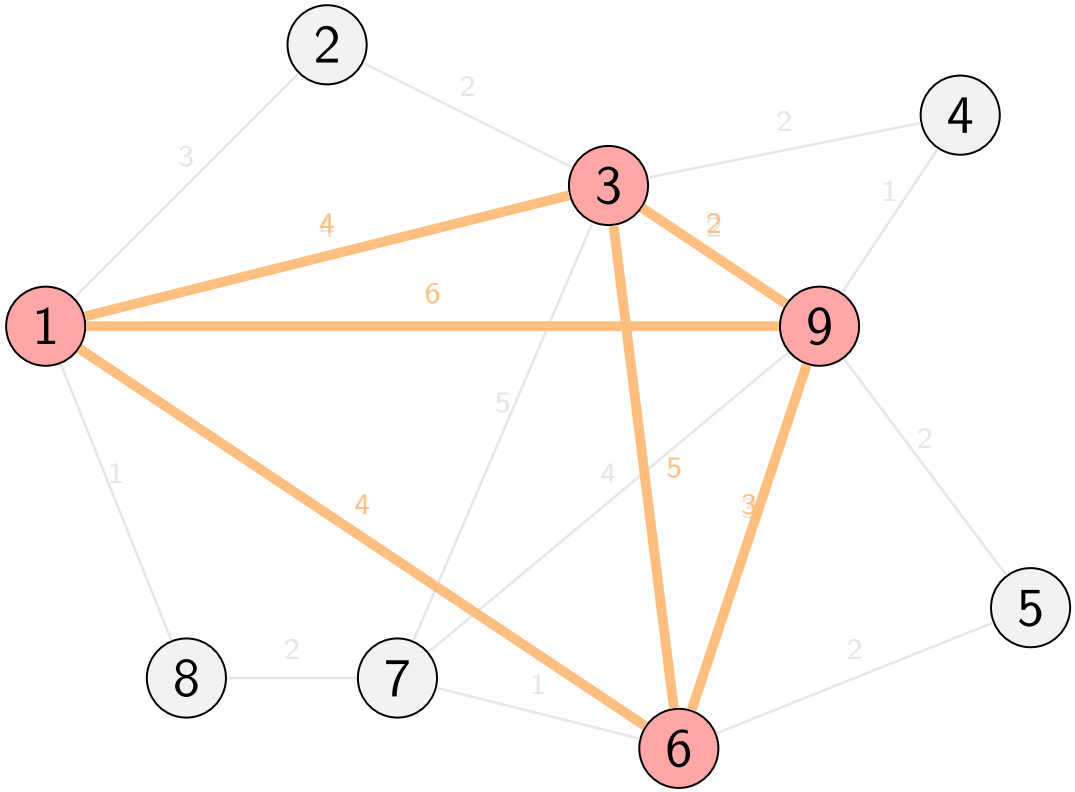
- Wähle alle Knoten mit ungeradem Grad
- Bilde vollständigen Graphen aus diesen Knoten
- Jeder Kante werden Kosten zugeordnet, die der Distanz der Knoten im ursprünglichen Graphen entsprechen
- Bilde kostenminimale *Paarung* von Knoten (ohne Details)
- Dupliziere Kanten entlang der kürzesten Wege der gepaarten Knoten im ursprünglichen Graphen
- Der resultierende Graph ist nun Eulersch, der Euler-Zyklus entspricht der kürzest möglichen geschlossenen Kantenfolge im ursprünglichen Graphen, die alle Kanten mindestens einmal enthält!

Beispiel: Chinese Postman Problem



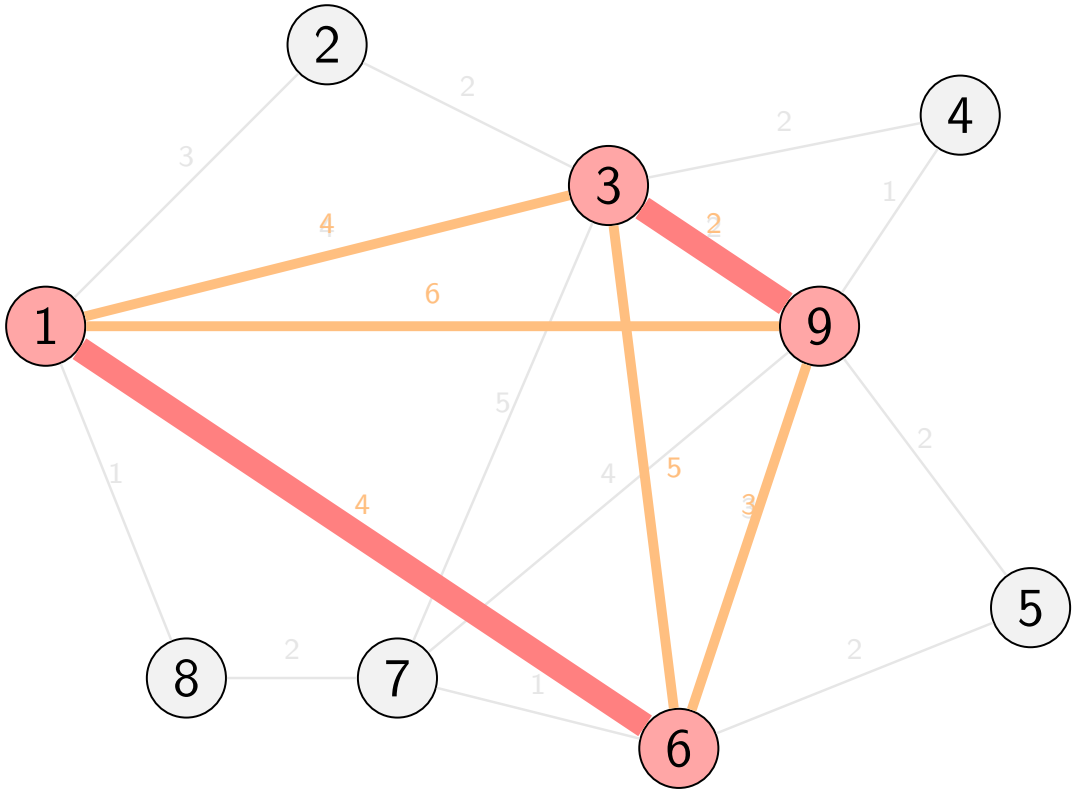
Die markierten Knoten haben ungeraden Grad, wodurch der Graph insgesamt *nicht* Eulersch ist.

Beispiel: Chinese Postman Problem



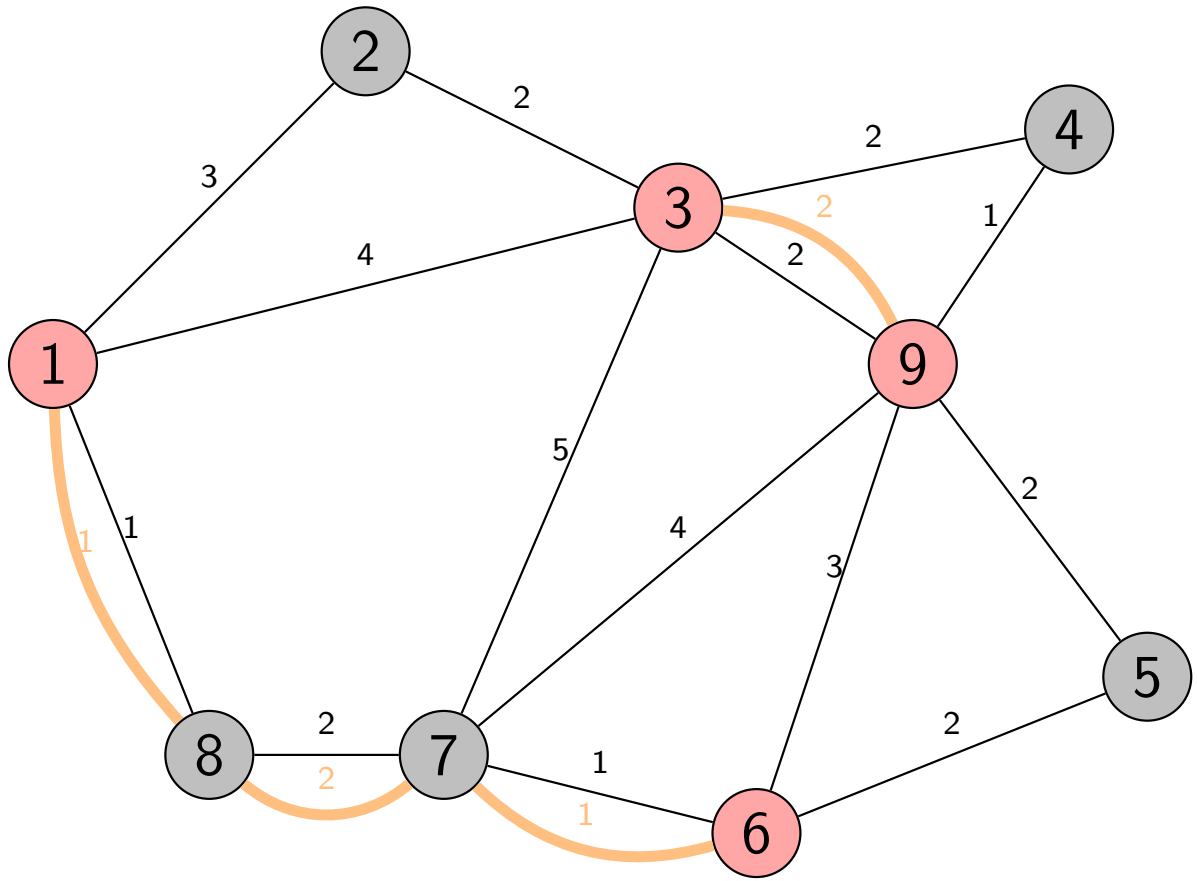
Bezüglich der "ungeraden" Knoten wird ein vollständiger Graph gebildet, dessen Kantengewichte den Distanzen im ursprünglichen Graphen entsprechen. $\Rightarrow$ Bildung von kostenminimalem Matching (mit Edmond's Blossom-Algorithmus in $O(n^3)$).

Beispiel: Chinese Postman Problem



Bezüglich der "ungeraden" Knoten wird ein vollständiger Graph gebildet, dessen Kantengewichte den Distanzen im ursprünglichen Graphen entsprechen. $\Rightarrow$ Bildung von kostenminimalem Matching (mit Edmond's Blossom-Algorithmus in $O(n^3)$).

Beispiel: Chinese Postman Problem



Alle Kanten entlang der kürzesten Pfade zwischen den gepaarten Knoten werden nun im ursprünglichen Graphen verdoppelt, wodurch dieser nun Eulersch wird!

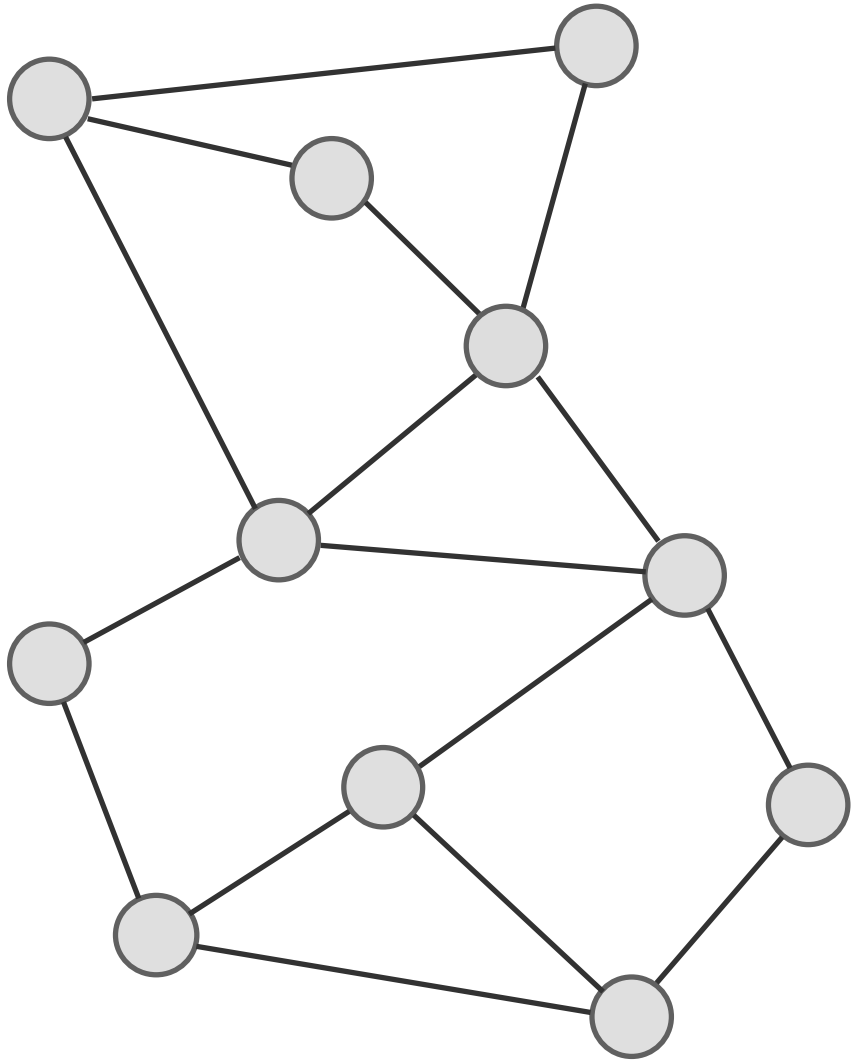
WH: Zusammenhangskomponenten

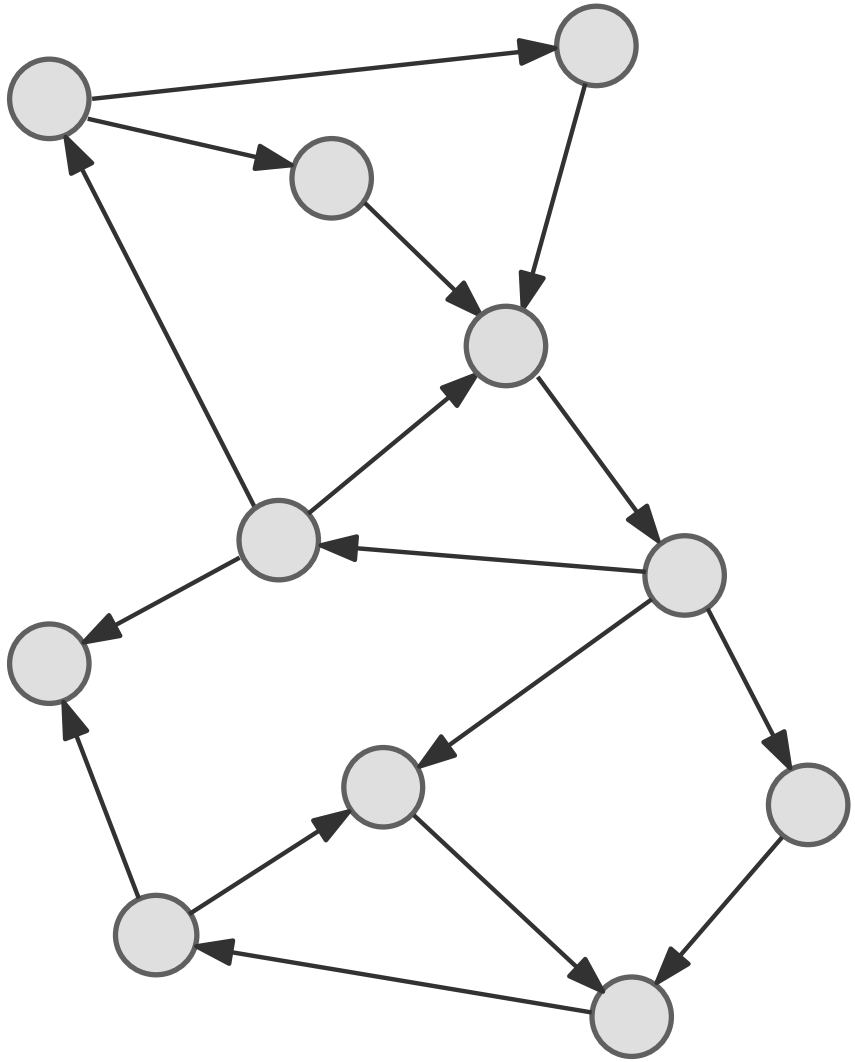
Definition 4 (Zusammenhangskomponenten)

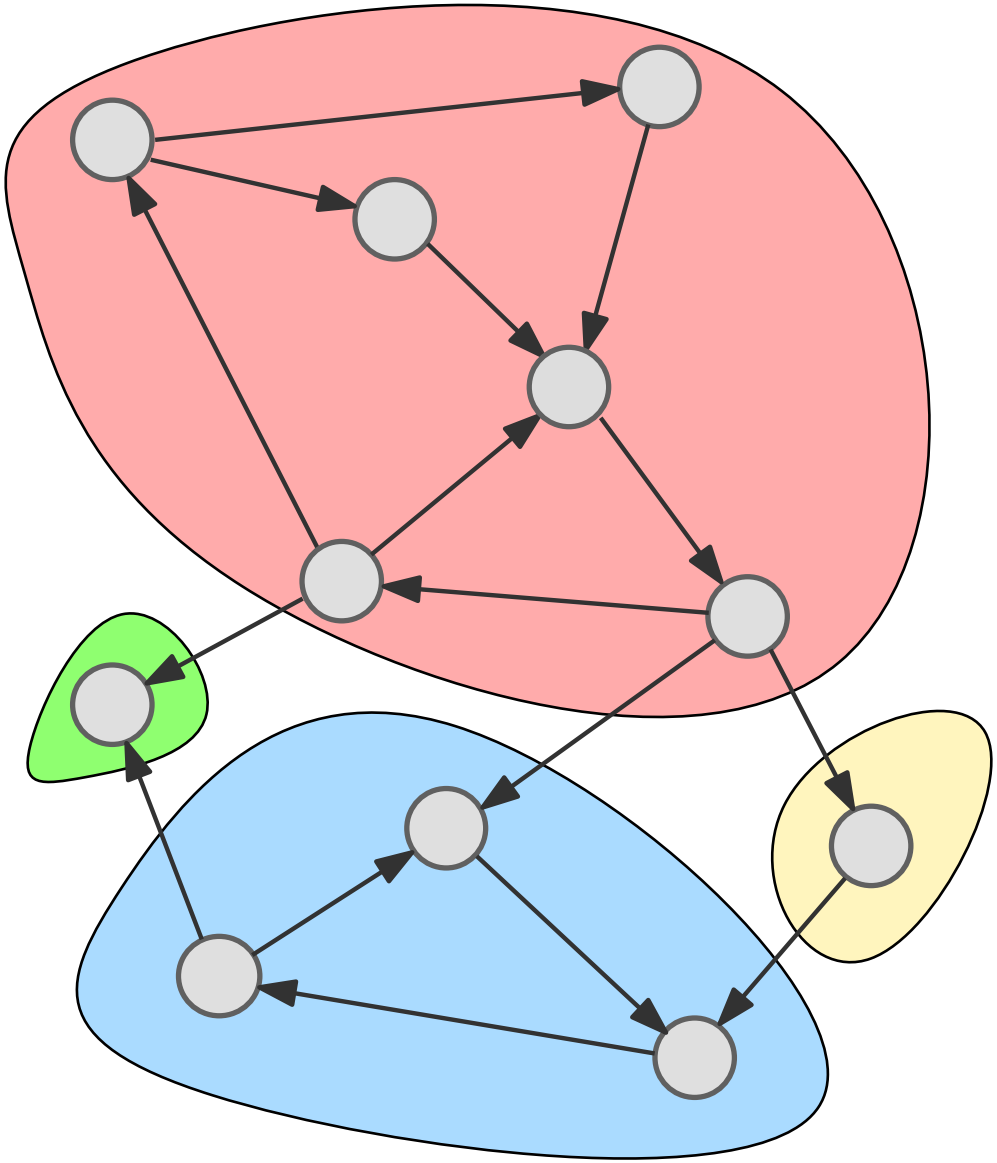
In einem ungerichteten Graphen G ist eine *Zusammenhangskomponente* ein maximaler, zusammenhängender Teilgraph. Zwei Knoten u und v sind in der selben Zusammenhangskomponente, genau dann wenn $u \rightsquigarrow v$. [a]

[a] $u \rightsquigarrow v$ bedeutet, dass v von u aus erreichbar ist. Im ungerichteten Graphen impliziert dies $v \rightsquigarrow u$.

Wie kann der Begriff der *Zusammenhangskomponente* auf gerichtete Graphen übertragen werden?







Zusammenhang in gerichteten Graphen

Definition 5 (Schwacher Zusammenhang)

Man spricht von *schwachem Zusammenhang* in einem gerichteten Graphen, wenn der zugrunde liegende ungerichtete Graph ("Schatten") zusammenhängend ist.

Definition 6 (Starke Zusammenhangskomponente)

Eine *Starke Zusammenhangskomponente* ist eine maximale Teilmenge an Knoten $U \subseteq V$ mit der Eigenschaft, dass für alle Paare an Knoten $u, v \in U$ gilt, dass sowohl $u \rightsquigarrow v$ als auch $v \rightsquigarrow u$.

Definition 7 (Transponierter Graph G^T)

Sei $G^T = (V, A^T)$ der Graph der aus dem gerichteten Graphen $G = (V, A)$ entsteht indem alle gerichteten Kanten "umgedreht" werden, also $A^T = \{(j, i) \mid (i, j) \in A\}$.

Aufruf der Tiefensuche DFS(G):

- Im ersten Aufruf werden nicht unbedingt alle Knoten erreicht
- Die Tiefensuche wird dann so lange für die verbleibenden (unbesuchten) Knoten aufgerufen, bis alle Knoten besucht sind.
- Auch für erneuten Aufruf gilt: besuchte Knoten werden nicht erneut besucht
- Wenn zu jedem Knoten eine gerichtete Kante vom Vorgänger auf diesen Knoten gespeichert wird, entsteht **Tiefensuch-Wald** (mehrere gerichtete Bäume)

Algorithmus zur Berechnung der starken Zusammenhangskomponente

Algorithmus von Kosaraju-Shahir

1. Aufruf von DFS(G) $\Rightarrow \tau_f(v)$
2. Berechne G^T
3. Aufruf von DFS(G^T) für Knoten $v \in V$ in absteigender Reihenfolge bezüglich $\tau_f(v)$ aus Schritt (1).
4. Die Knotenmengen die in einem Aufruf der DFS in Schritt (3) gefunden werden entsprechen den starken Zusammenhangskomponenten.

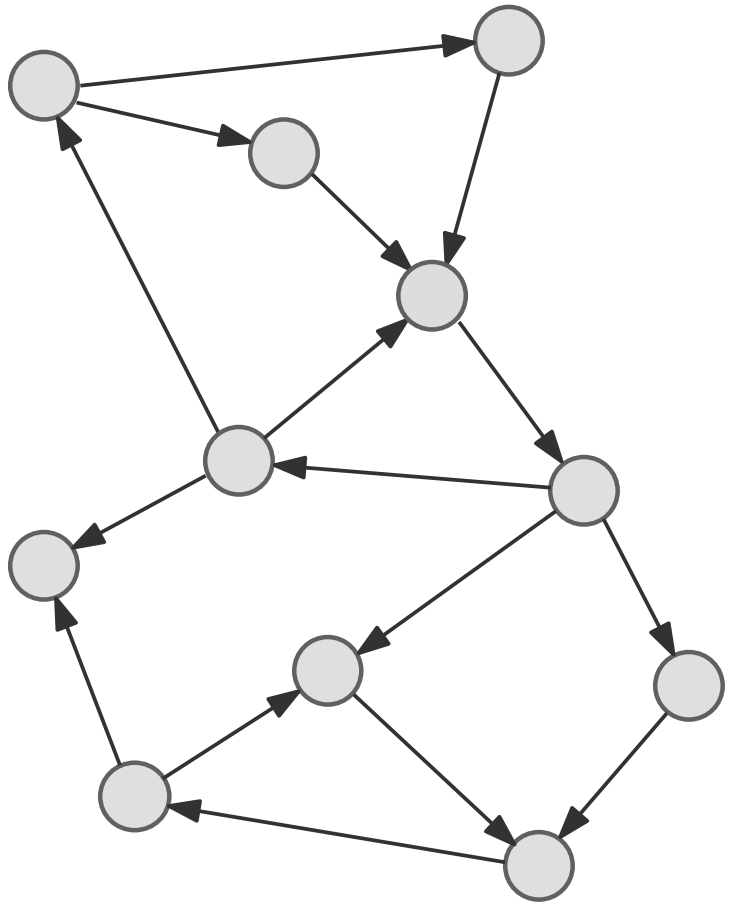


Abbildung: Beispiel: Bestimmung der starken Zusammenhangskomponenten

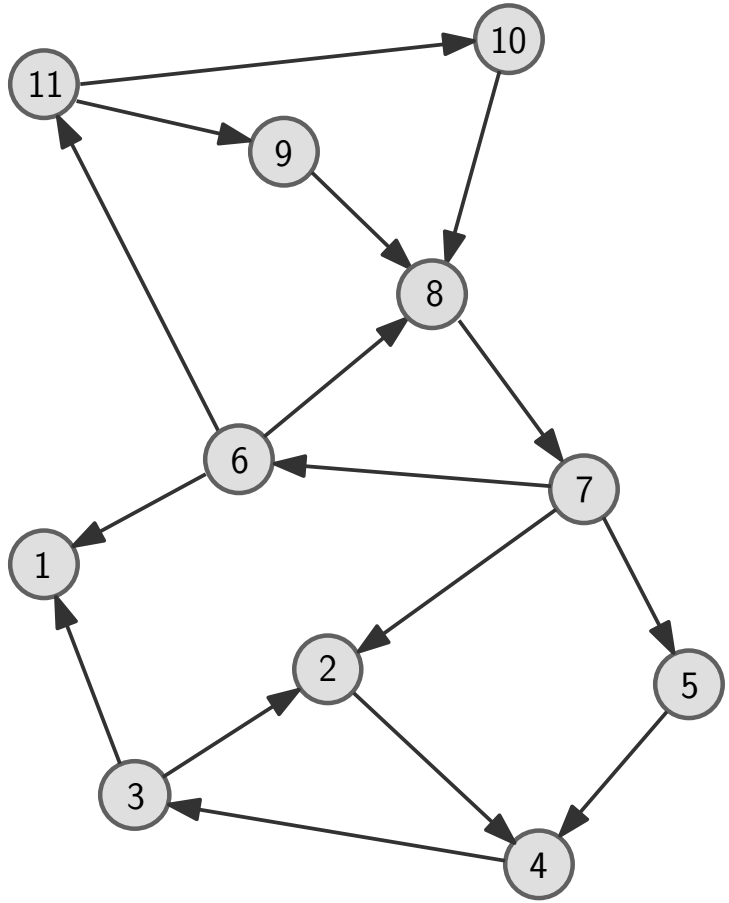


Abbildung: Die Werte in den Knoten entsprechen den Fertigstellungszeiten τ_f (Bem. τ_d's werden hier vernachlässigt)

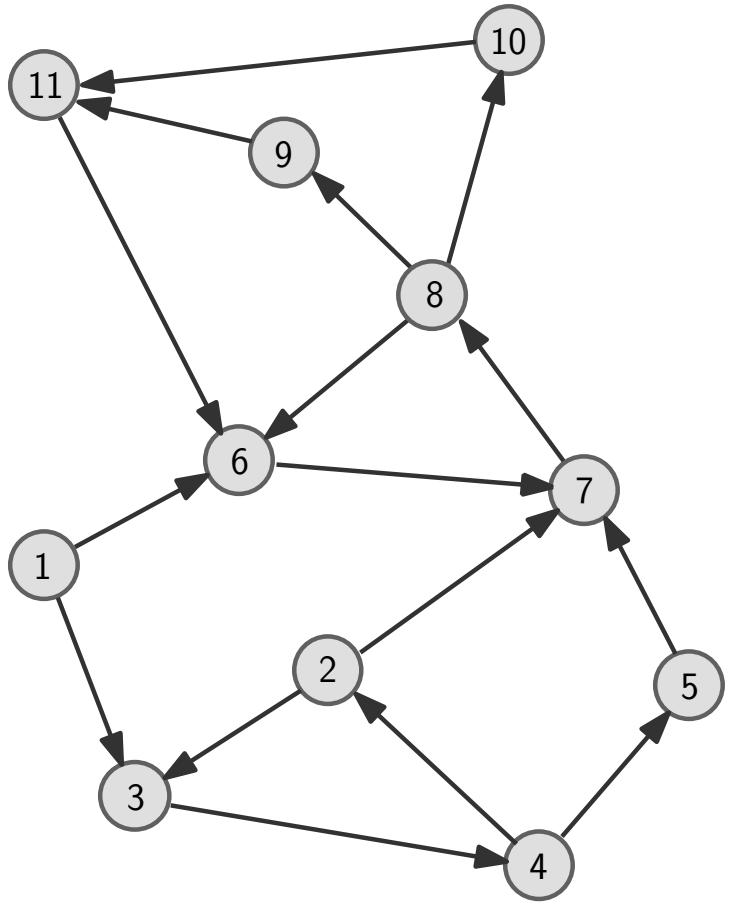


Abbildung: Nach dem ersten Aufruf der DFS wird der transponierte Graph G^T von G berechnet.

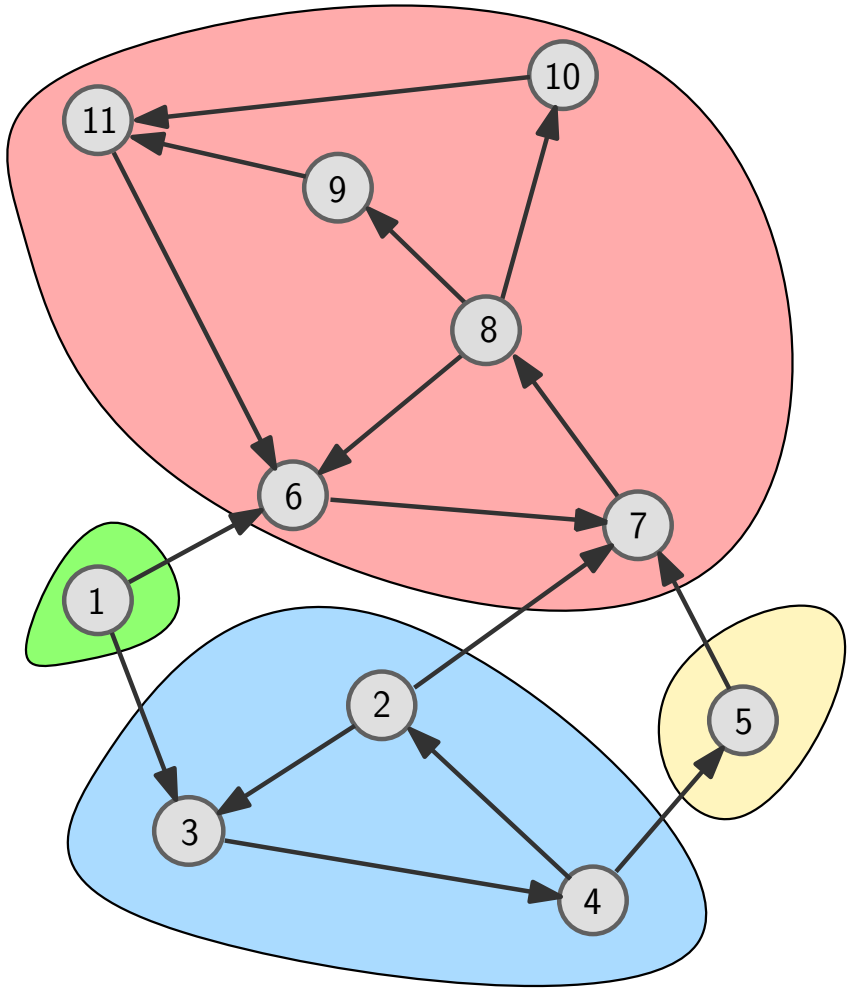


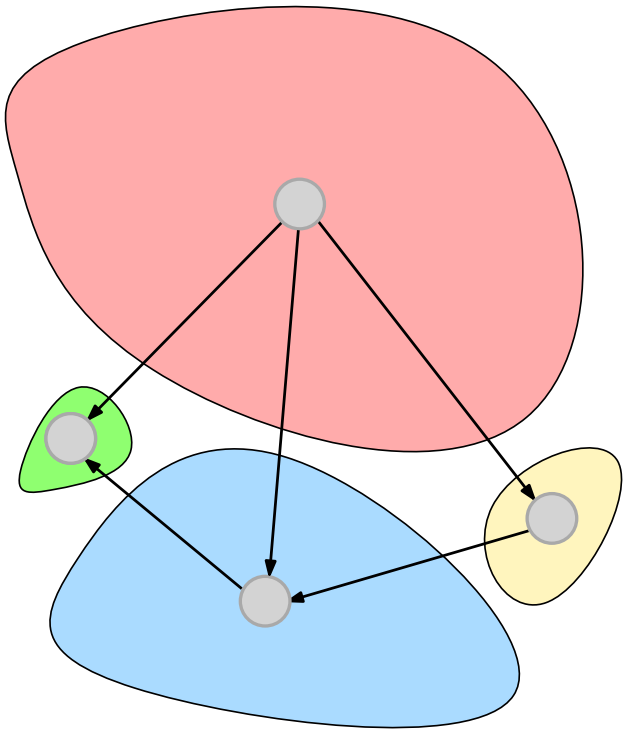
Abbildung: Die DFS Aufrufe in Schritt (3) liefern als Ergebnis die Starken Zusammenhangskomponenten von G^T (und somit von G).

Lemma 8

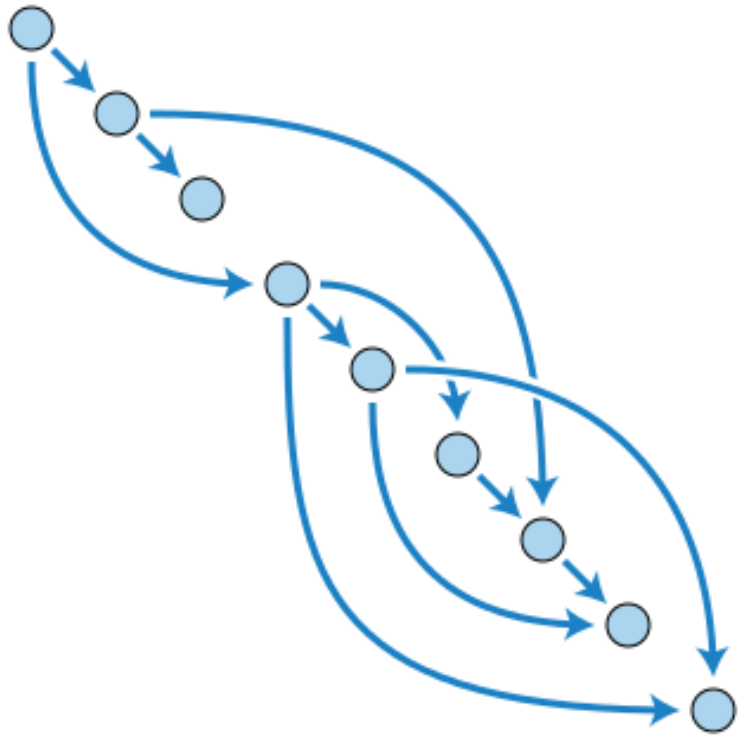
Sei $G' = (V', E')$ der Graph der daraus entsteht indem in G jede Starke-Zusammenhangs-Komponente (SZK) in einen einzelnen Knoten kontrahiert wird. G' ist dann ein gerichteter azyklischer Graph, (engl. directed acyclic graph, **DAG***).*

Beweis:

- Angenommen es gibt einen Kreis $C_1, C_2, \ldots, C_n$ mit $C_i \in V'$
- $\Rightarrow$ Dann existiert eine gerichtete Kante $a = (i, j)$ mit $i \in C_k$ und $j \in C_{k+1}$ (und jeweils $i \in C_n$ und $j \in C_1$) für $1 \leq k \leq n-1$
- Es existiert ein Pfad P von jedem Knoten in C_i zu jedem Knoten in C_j für alle $i, j \in \{1 \ldots n\}$.
- Alle $v \in V(\bigcup_{i \in \text{Kreis}} C_i)$ gehören zur selben SZK. $\square$

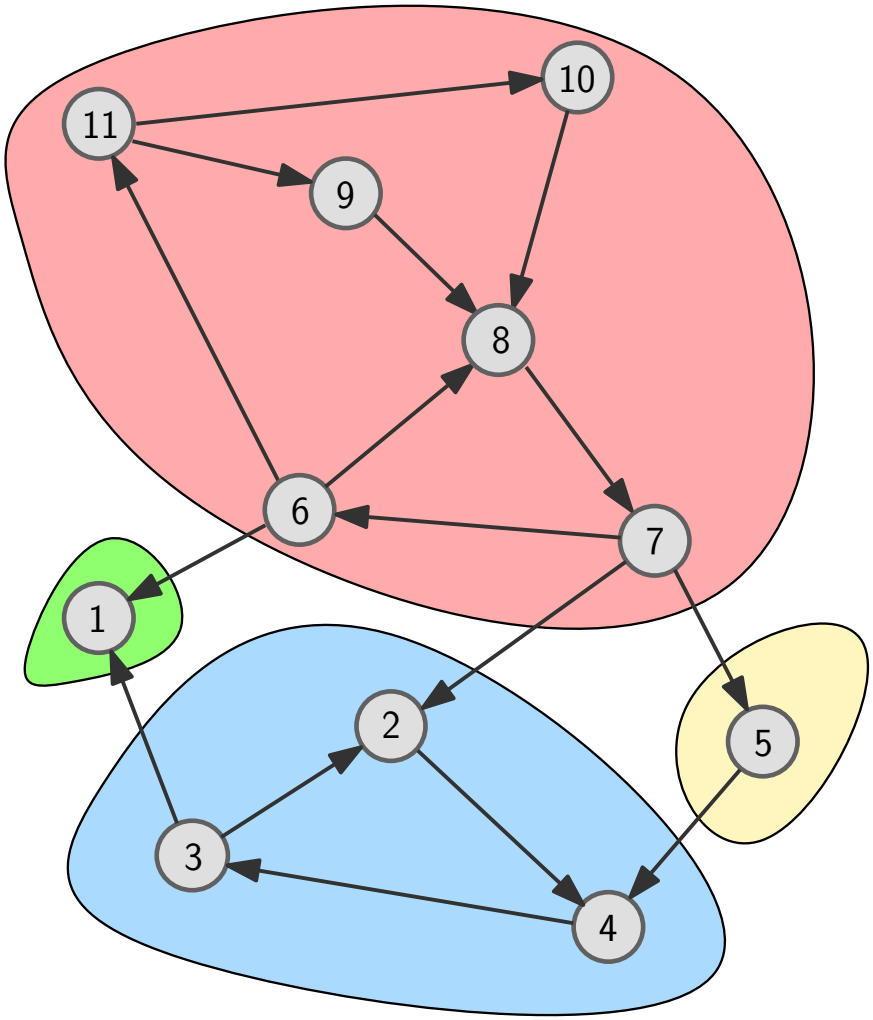


- Ein DAG enthält keinen gerichteten Kreis.
- Die Knoten können so angeordnet werden, dass alle Kanten von links nach rechts verlaufen.
- Diese Anordnung wird auch *topologische Sortierung* genannt.
- $\Rightarrow$ **Jeder DAG enthält mindestens einen Knoten** v **mit** $d^+(v) = 0$**, und mindestens einen Knoten** v **mit** $d^-(v) = 0$.



Lemma 9

Sei C eine SZK von G ohne auslaufenden Kanten. Ein DFS-Aufruf für einen Knoten $v \in C$ besucht **genau alle** *Knoten $u \in C$.*



Beweis:

- Sei v ein beliebiger Knoten in C.
- $\forall u \in C : v \rightsquigarrow u$.
- Da C keine auslaufenden Kanten hat, sind keine weiteren Knoten erreichbar. □

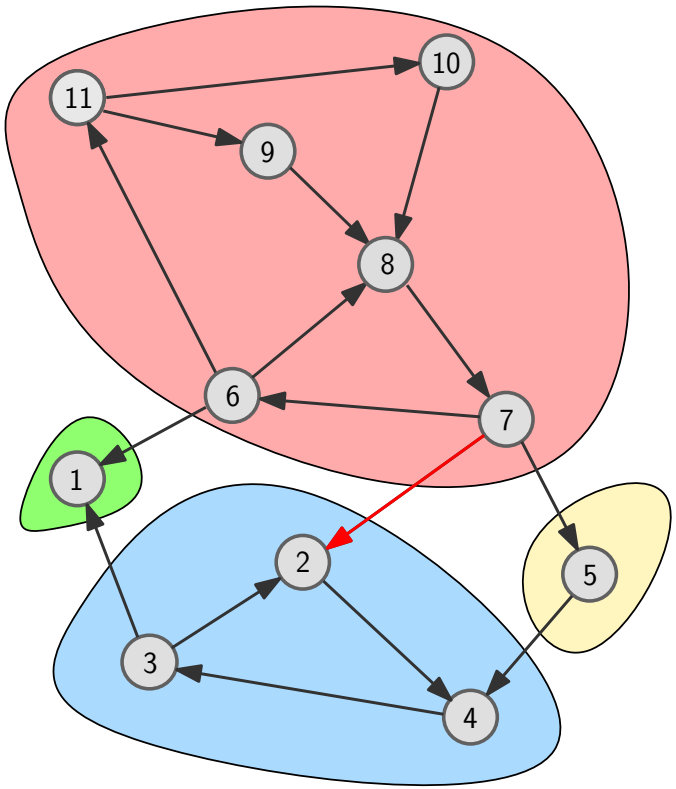
Lemma 10

Seien C_1 und C_2 zwei SZK von G. Weiters sei $a = (i, j)$ eine gerichtete Kante von einer Komponente in die andere, also $i \in C_1, j \in C_2$. Sei v^* der erste Knoten in C_1 der von DFS besucht wird. Dann gilt: $\tau_f(v^*) > \tau_f(v_k)\ \forall v_k \in C_2$.

Beweis:

case 1: DFS wird für einen Knoten $v \in C_1$ aufgerufen bevor sie für irgend einen Knoten in C_2 aufgerufen wird.

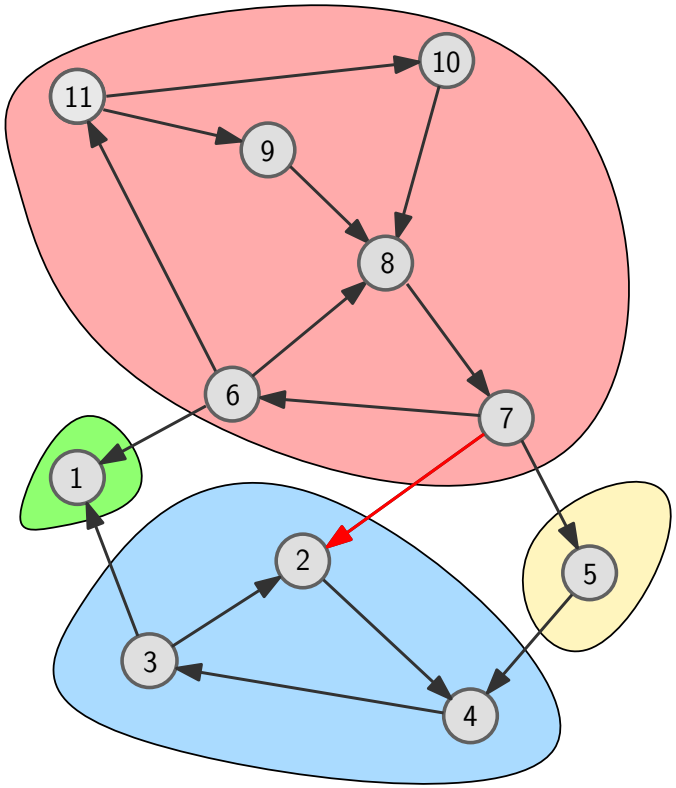
- Aufgrund der Annahme sind alle Knoten in C_1 und C_2 vom Knoten v aus erreichbar.
- DFS(v) ist fertig wenn alle Knoten in C_1 und C_2 abgeschlossen wurden.
- $\tau_f(v^*) > \tau_f(v_k)\ \ \forall v_k \in C_2$.



Proof:

case 2: DFS wird für einen Knoten $v \in C_2$ bevor es für irgend einen anderen Knoten in C_1 aufgerufen wird.

- Es gibt keinen Weg von einem Knoten in C_2 zu einem in C_1.
- DFS ist für alle Knoten in C_2 abgeschlossen wenn es für den ersten Knoten in C_1 aufgerufen wird.
- $\tau_f(v^*) > \tau_f(v_k) \quad \forall v_k \in C_2$.



Lemma 11

Der Knoten v mit $\max(\tau_f(v))$ in G gehört zur SZK ohne einlaufende Kanten.

Beweis: folgt direkt aus Lemma 10 □

Lemma 11

Der Knoten v mit $\max(\tau_f(v))$ in G gehört zur SZK ohne einlaufende Kanten.

Beweis: folgt direkt aus Lemma 10 □

Lemma 12

Die Knotenmengen der SZKn von G entsprechen jenen in G^T.

Beweis: Wir betrachten zwei Knoten $u, v \in G$ in der selben SZK in G.
$u \rightsquigarrow v \wedge v \rightsquigarrow u$
$\Leftrightarrow u \rightsquigarrow v \wedge v \rightsquigarrow u$ in G^T
u und v sind somit in der selben SZK in G^T. □

Die zusammenhängenden Komponenten des resultierenden Tiefensuch-Waldes in G^T (Schritt 3 des Algorithmus) entsprechen den SZK in G.

Die zusammenhängenden Komponenten des resultierenden Tiefensuch-Waldes in G^T (Schritt 3 des Algorithmus) entsprechen den SZK in G.

Beweis:

- *Teil 1:* Erster Aufruf der DFS in G^T:
  Es folgt aus Lemma 11, dass DFS für die SZK ohne auslaufende Kanten (in G^T) aufgerufen wird, $\Rightarrow$ DFS besucht genau alle Knoten dieser SZK.

- *Teil 2:* weitere DFS-Aufrufe in G^T:
  Sei v der Knoten für den DFS aufgerufen wird, und C_v die zugehörige SZK. Alle Knoten in C_v sind erreichbar und werden von diesem DFS-Aufruf besucht. Sei weiters $v \rightsquigarrow w$, $w \notin C_v$ (sondern in C_w). Es wird nun gezeigt, dass w *nicht* in diesem DFS-Aufruf besucht wird.

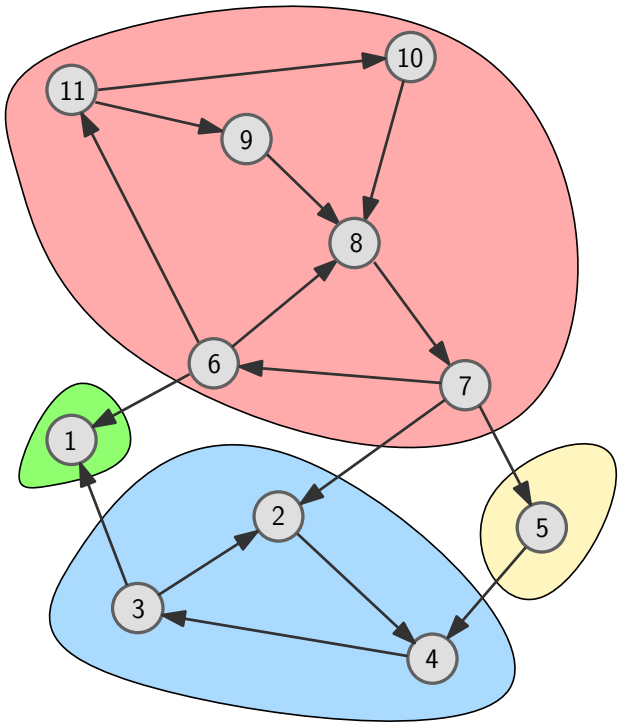
  In G gilt, dass $w \rightsquigarrow v$. Wenn w schon in einem vorangegangenen DFS-Aufruf besucht wurde, wird es nicht erneut besucht. Somit betrachten wir die (kritische) Situation, dass w noch unbesucht ist.

  Es muss somit gelten, dass $\max_{w' \in C_w}(\tau_f(w')) < \tau_f(v)$. (Anderenfalls wären ja alle Knoten von C_w schon zuvor besucht worden).

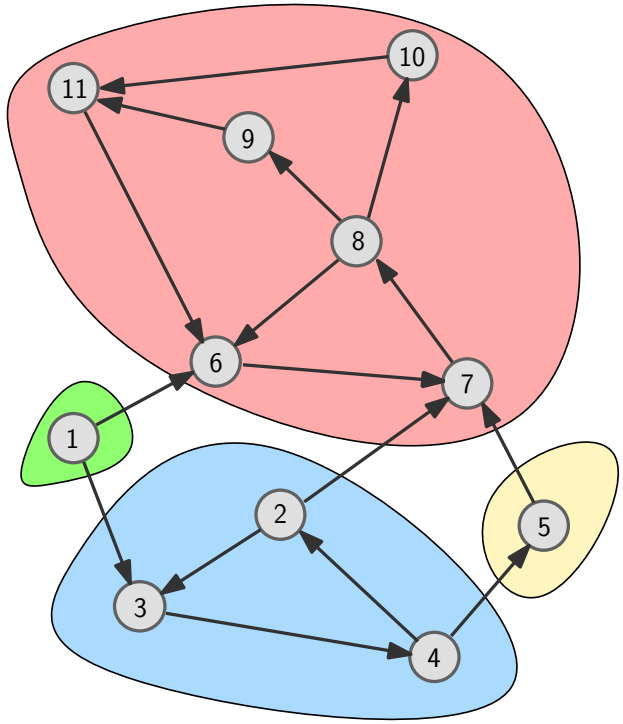
  Es folgt somit, dass der DFS-Aufruf in G schon für alle Knoten $u \in C_w$ abgeschlossen war, bevor v abgeschlossen wurde.

  Da jedoch $u \rightsquigarrow v \; \forall u \in C_w$in G folgt, dass ein Knoten $u \in C_w$ existieren muss, für den $\tau_f(u) > \tau_f(v)$ gilt, was ein Widerspruch zu der vorigen Aussage ist.

G :



G^T:



Anmerkung: Wenn also eine Kante in G^T von C_v nach C_w führt, müssen alle Knoten in C_w *nach* jenen aus C_v fertiggestellt worden sein, da ja in G in diesem Fall eine Kante von C_w nach C_v verlaufen ist. Somit müssen diese Knoten im Schritt 3 schon zuvor fertiggestellt worden sein (da ja die Aufrufe in absteigender Reihenfolge der Fertigstellungszeiten aus Schritt 1 erfolgt.