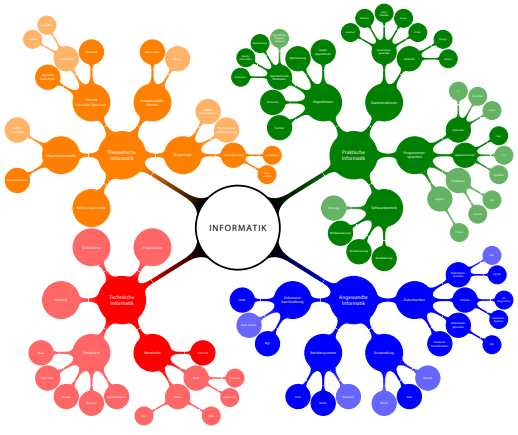
UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

Kapitel 32

Kürzeste und längste Wege

Wer sucht, der findet

Vorlesung Einführung in die Informatik 2 vom 22. April 2014 von Till Tantau



Lernziele von Kapitel 32

1. Konzept der Tiefen- und Breitensuche verstehen
2. Dijkstra-Algorithmus für das Kürzeste-Wege-Problem kennen
3. Backtracking-Algorithmus für das Längste-Wege-Problem kennen

Gliederung von Kapitel 32

Worum es in diesem Kapitel geht.

- Ganz allgemein geht es bei *Wege-Finde-Problemen* darum, in einem Graphen einen Weg zwischen zwei gegebenen Knoten zu finden.
- Die einfachste Variante ist das *Erreichbarkeitsproblem*, bei dem es nur darum geht, ob überhaupt ein Weg existiert.
- Etwas schwieriger ist das *Kürzeste-Wege-Problem*, bei dem man einen kürzesten Weg finden möchte.
- Noch schwieriger ist das *Längste-Wege-Problem*, bei dem man einen möglichst langen Weg finden möchte (wobei man aber keinen Knoten zweimal besuchen darf).

In diesem Kapitel sollen *Algorithmen zur Lösung dieser Probleme vorgestellt werden*.

Warum sind die Problem interessant?

Das *Finden von Wegen in Graphen* kommt in sehr vielen Anwendungen vor:

Beispiele

- Navigationsgerät
- Routenberechner der Bahn
- Analyse von Gen-Pathways

Zur Diskussion

Nennen Sie weitere Anwendungen, bei denen man sich für kürzeste Wege interessiert. Fallen Ihn auch Anwendungen ein, bei welchen *längste* Wege wichtiger sind?

Das Erreichbarkeitsproblem formal.

Definition (Weg in einem Graph)

Sei G ein Graph. Ein *Weg (auch Pfad genannt) in G* ist eine Folge von *unterschiedlichen* Knoten $k_1, \ldots, k_n$, so dass es jeweils eine Kante von k_i zu k_{i+1} in G gibt.

Definition (Erreichbarkeitsproblem)

Das *Erreichbarkeitsproblem* ist folgendes Entscheidungsproblem:

Eingabe (Gerichteter) Graph sowie zwei Knoten s und t in dem Graphen (»source« und »target«).

Frage *Gibt* es einen Weg von s nach t in G?

Erste Lösungsidee: Alle erreichbaren Knoten besuchen.

- Um herauszufinden, ob t erreichbar ist, *besucht* man alle von s aus erreichbare Knoten – ist t dabei, so ist t erreichbar, sonst eben nicht.
- Den Vorgang, alle von einem Knoten aus erreichbaren Knoten einmal zu besuchen, nennt man *traversieren*.
- Wie bei Bäumen gibt es mehrere mögliche *Reihenfolgen,* nach denen man die Knoten besuchen kann. Wir betrachten die zwei wichtigsten:
  1. Breitensuche
  2. Tiefensuche

Die Idee der Breitensuche.

Ziel ist es, alle von s aus erreichbaren Knoten zu besuchen.

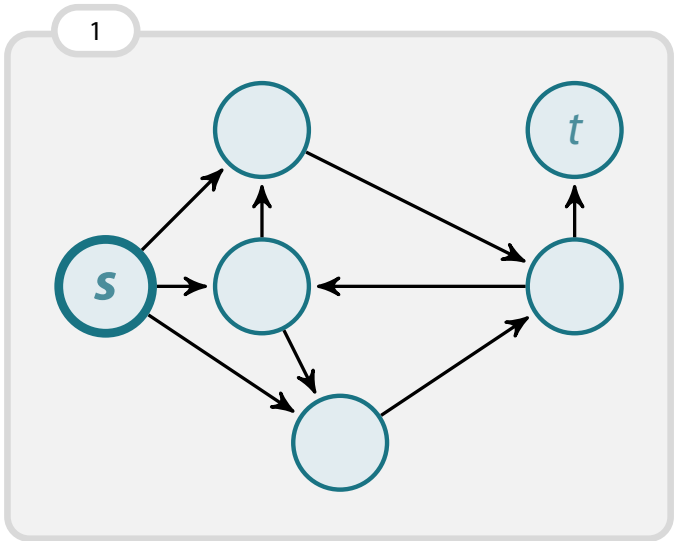
- Wir beginnen beim Startknoten s.
  (Tiefe 0)
- Dann besuchen wir alle seine Nachbarn.
  (Tiefe 1)
- Dann besuchen wir die Nachbarn der Knoten auf Tiefe 1, aber keinen Knoten, an dem wir schon waren.
  (Tiefe 2)
- Dann besuchen wir die Nachbarn der Knoten auf Tiefe 2, aber keinen Knoten, an dem wir schon waren.
  (Tiefe 3)
- Und so weiter.

Motto der Breitensuche

Fraternité! (Geschwister zuerst!)

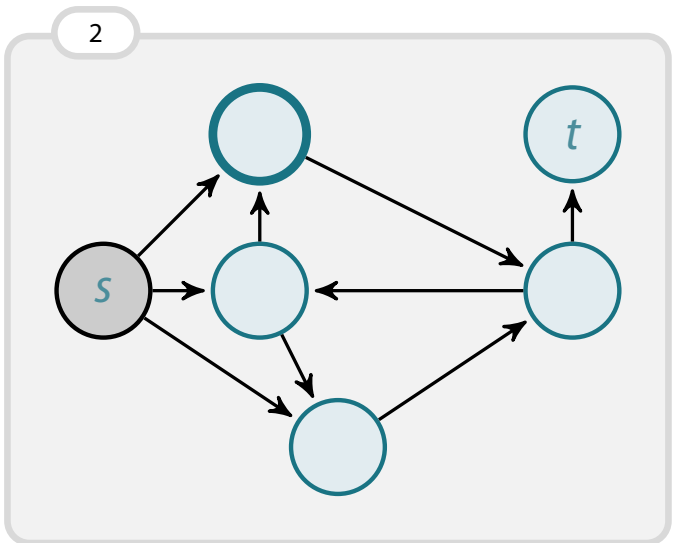
Beispiel einer Breitensuche.

- Der aktuell besuchte Knoten ist **fett**.
- Schon besuchte Knoten sind dunkel.



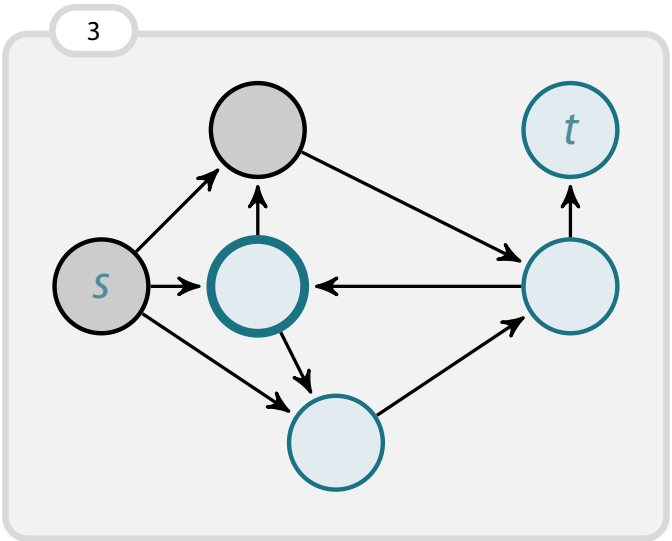
Beispiel einer Breitensuche.

- Der aktuell besuchte Knoten ist **fett**.
- Schon besuchte Knoten sind dunkel.



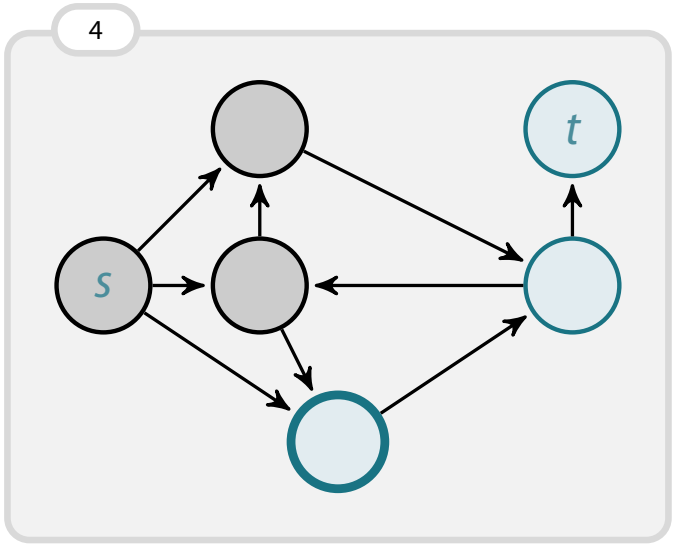
Beispiel einer Breitensuche.

- Der aktuell besuchte Knoten ist **fett**.
- Schon besuchte Knoten sind dunkel.



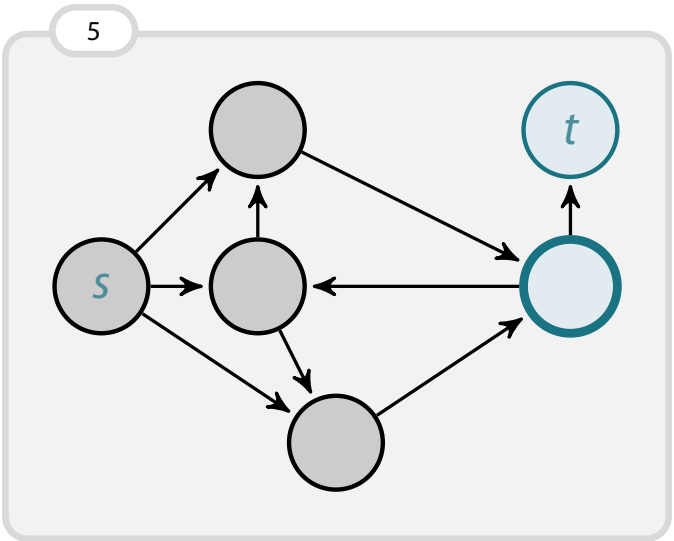
Beispiel einer Breitensuche.

- Der aktuell besuchte Knoten ist **fett**.
- Schon besuchte Knoten sind dunkel.



Beispiel einer Breitensuche.

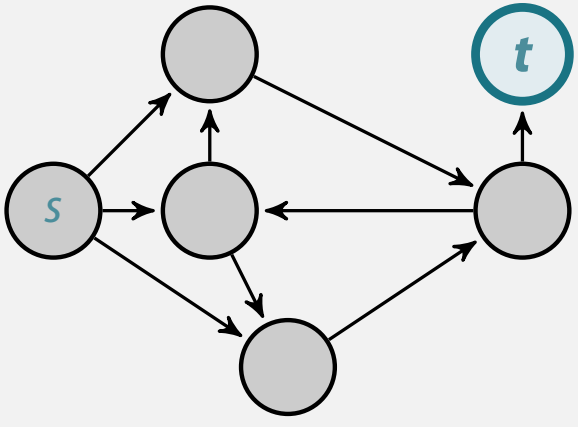
- Der aktuell besuchte Knoten ist **fett**.
- Schon besuchte Knoten sind dunkel.



Beispiel einer Breitensuche.

- Der aktuell besuchte Knoten ist **fett**.
- Schon besuchte Knoten sind dunkel.

6



Zur Implementation einer Breitensuche.

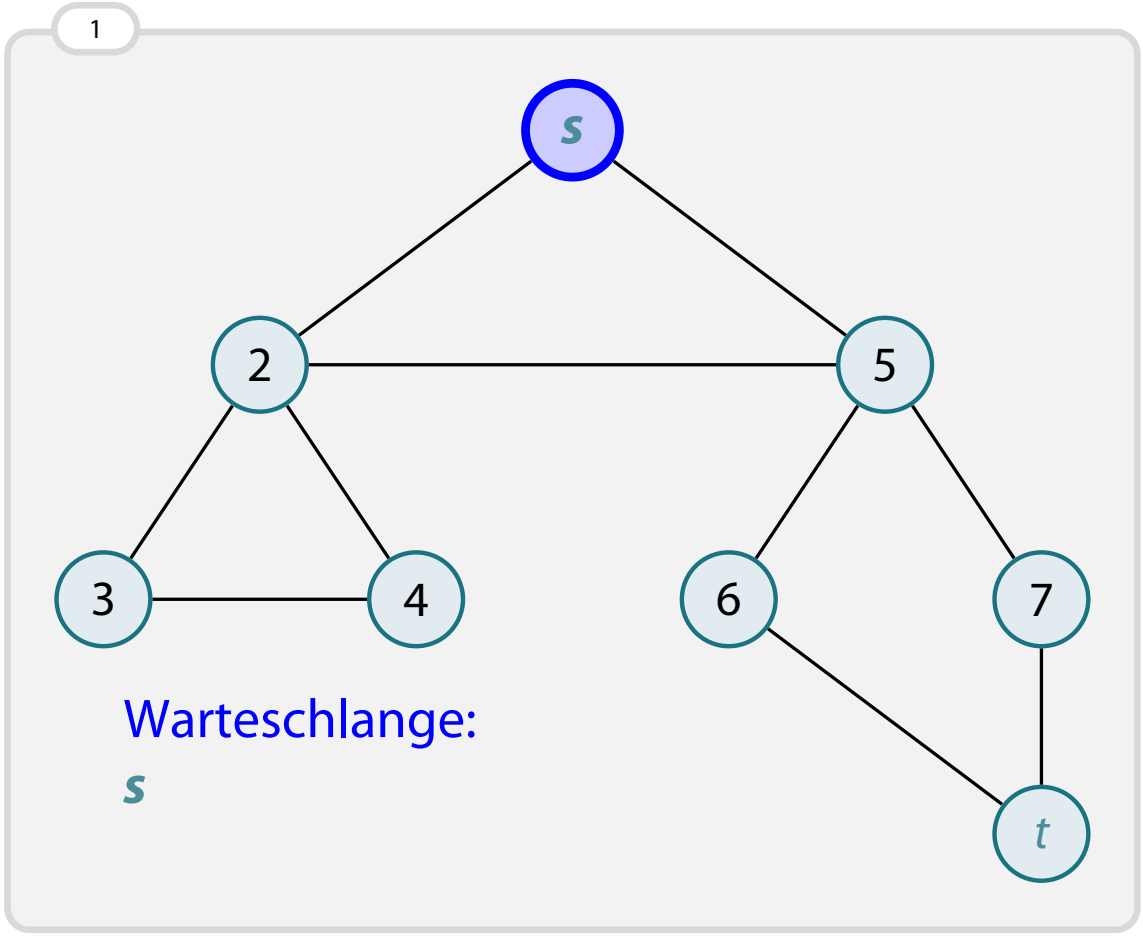
Bei der Implementation einer Breitensuche stellen sich folgende Probleme:

1. Wie merkt man sich, welche Knoten schon besucht wurden?
2. Woher weiß man, welcher Knoten als nächstes kommt?

Lösungen:

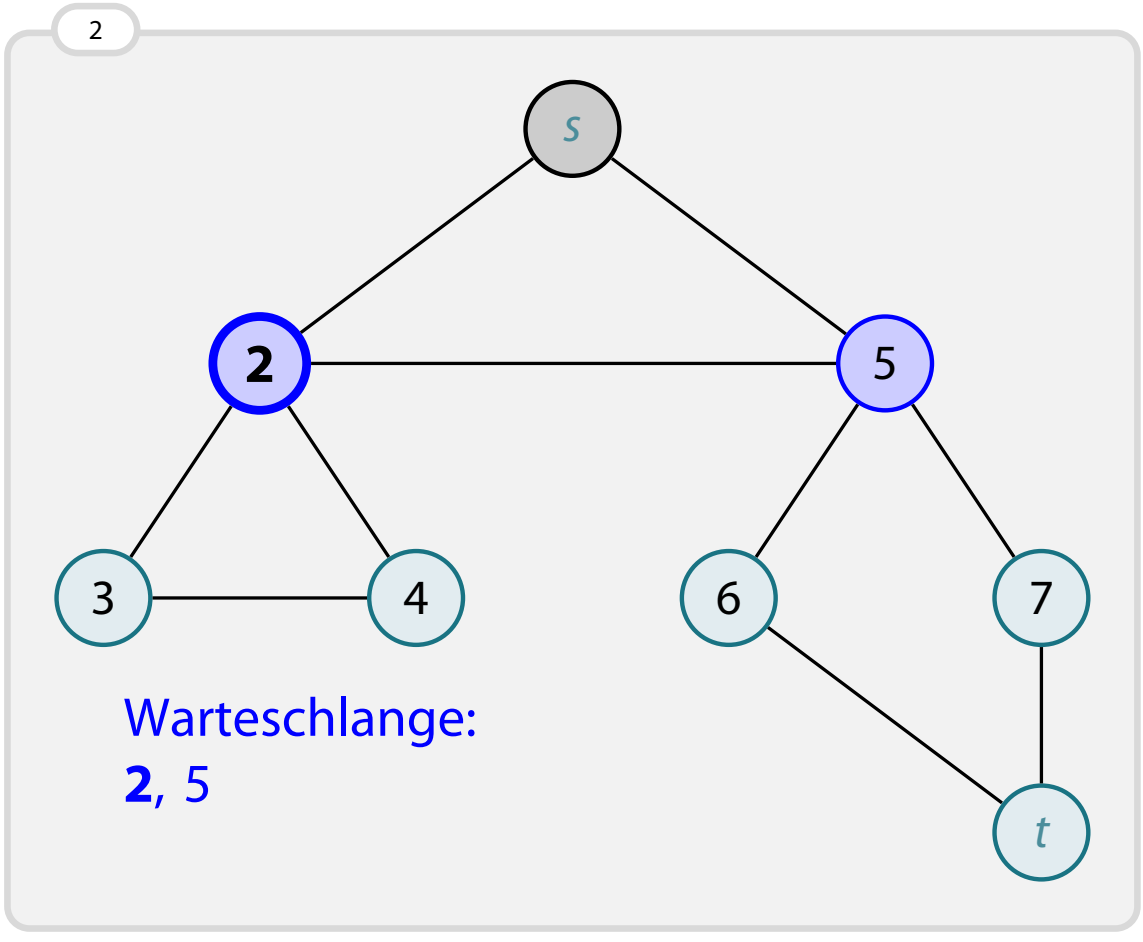
1. Man führt einen Array boolescher Werte mit, der für jeden Knoten angibt, ob man dort schon war.
2. Man überlegt sich, dass Breitensuche folgender Regel entspricht:
   »Besuche den ersten Knoten in einer Liste noch zu besuchender Knoten. Füge dann die noch nicht besuchten Nachbarn dieses ersten Knotens ans Ende der Liste an.«

Beispiel der Warteschlange in einer Breitensuche.



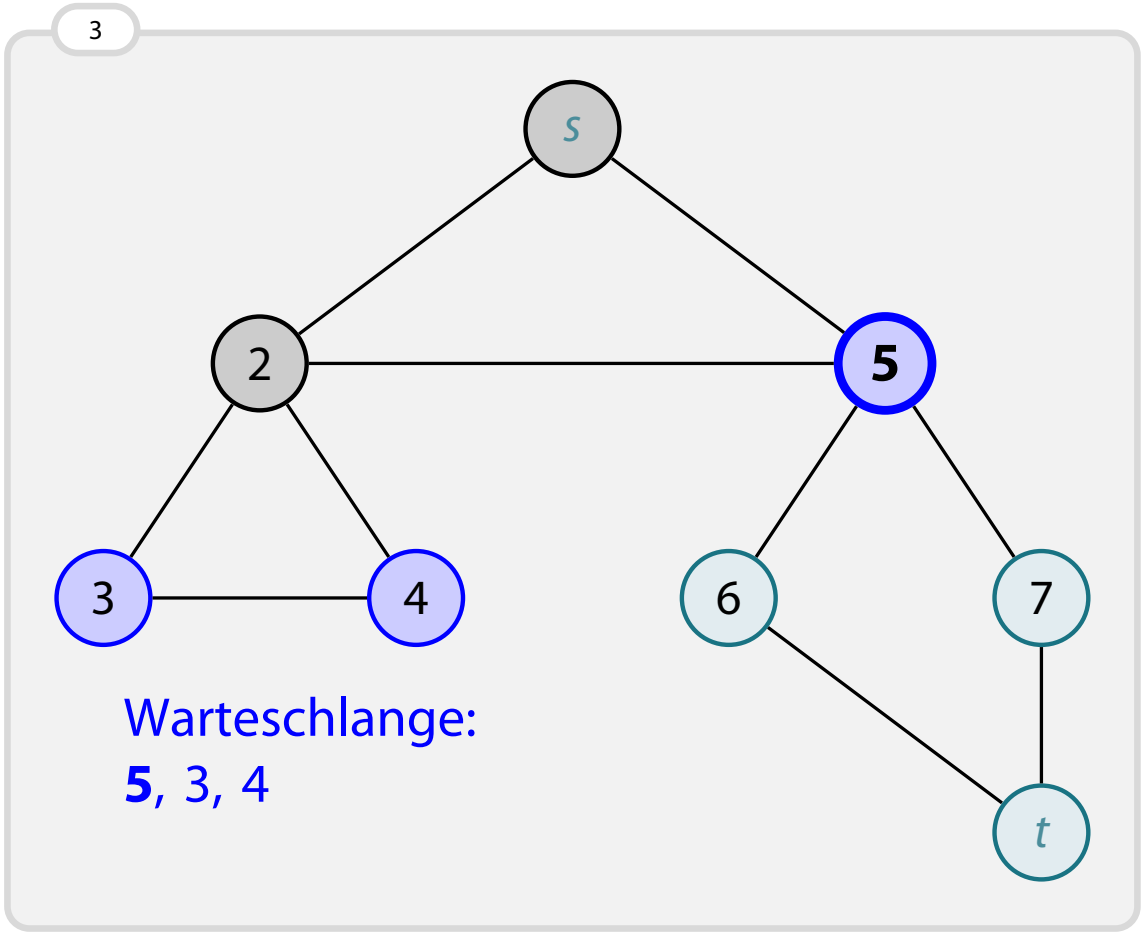
Grünblau = noch nicht besucht, dunkel = schon besucht,
blau = in der Warteschlange, **fett** = gerade besucht

Beispiel der Warteschlange in einer Breitensuche.



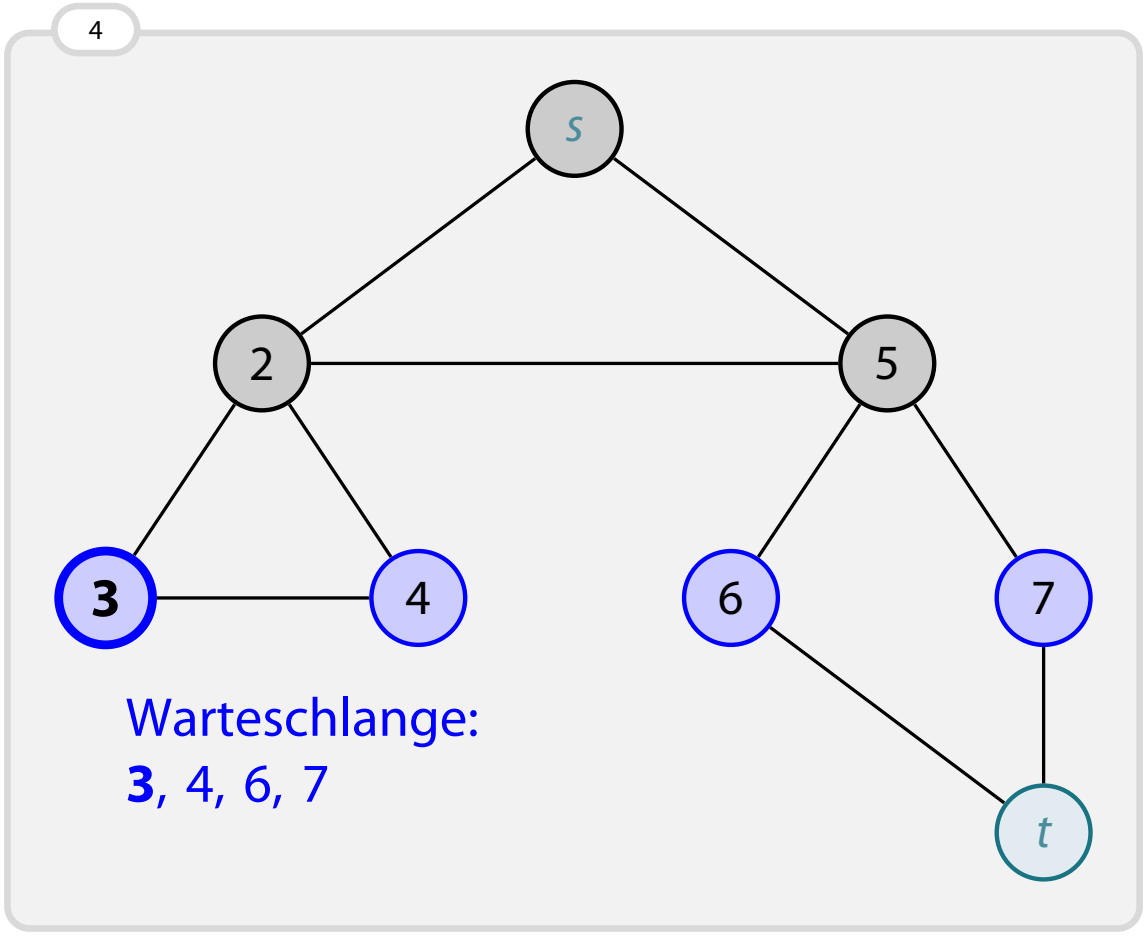
Grünblau = noch nicht besucht, dunkel = schon besucht,
blau = in der Warteschlange, **fett** = gerade besucht

Beispiel der Warteschlange in einer Breitensuche.



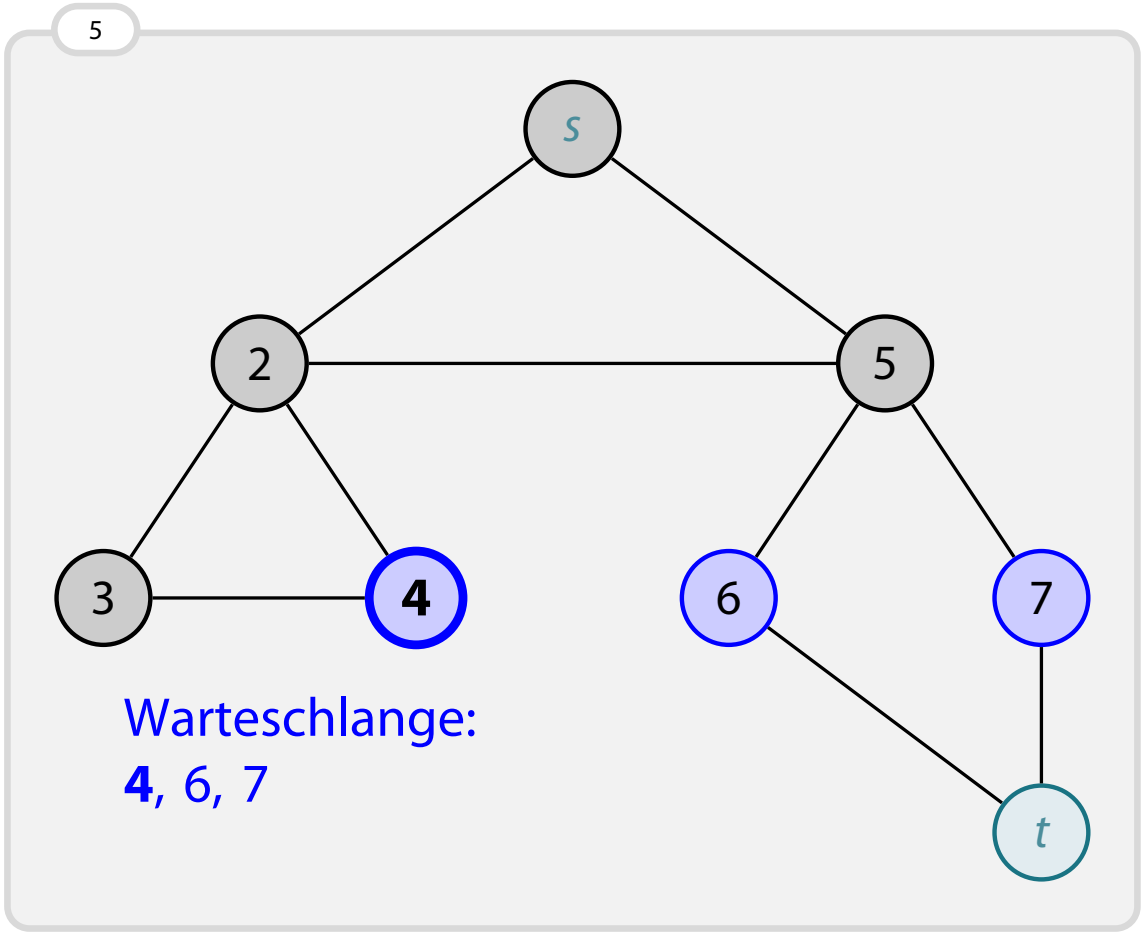
Grünblau = noch nicht besucht, dunkel = schon besucht,
blau = in der Warteschlange, **fett** = gerade besucht

Beispiel der Warteschlange in einer Breitensuche.



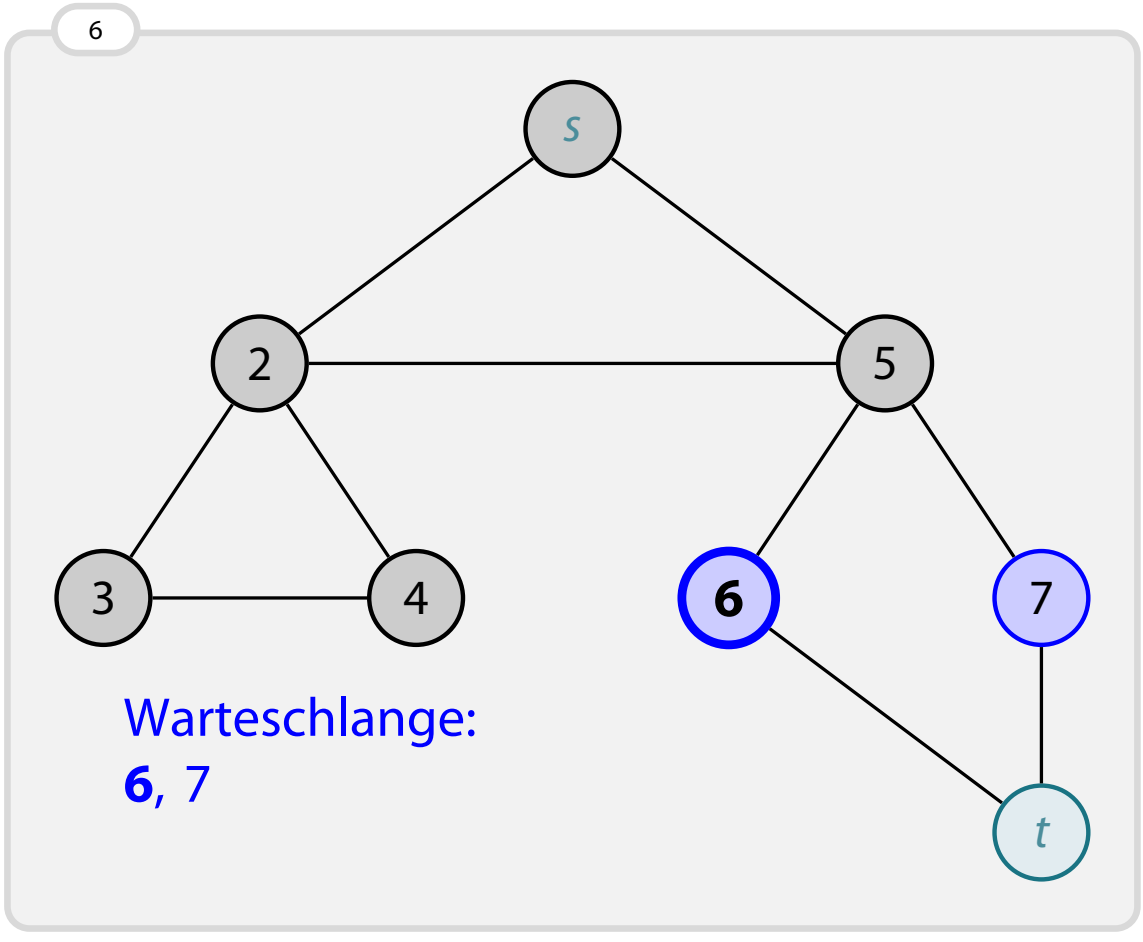
Grünblau = noch nicht besucht, dunkel = schon besucht,
blau = in der Warteschlange, **fett** = gerade besucht

Beispiel der Warteschlange in einer Breitensuche.



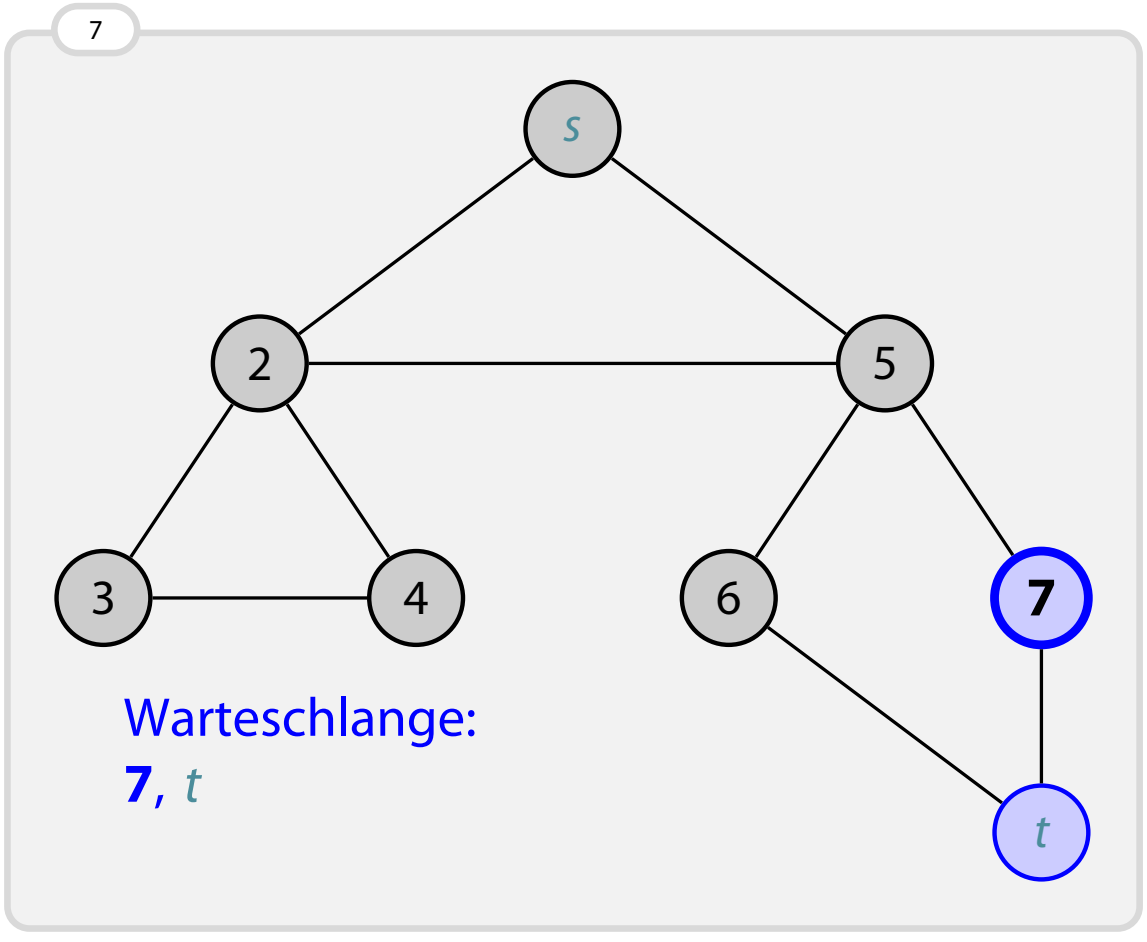
Grünblau = noch nicht besucht, dunkel = schon besucht,
blau = in der Warteschlange, **fett** = gerade besucht

Beispiel der Warteschlange in einer Breitensuche.



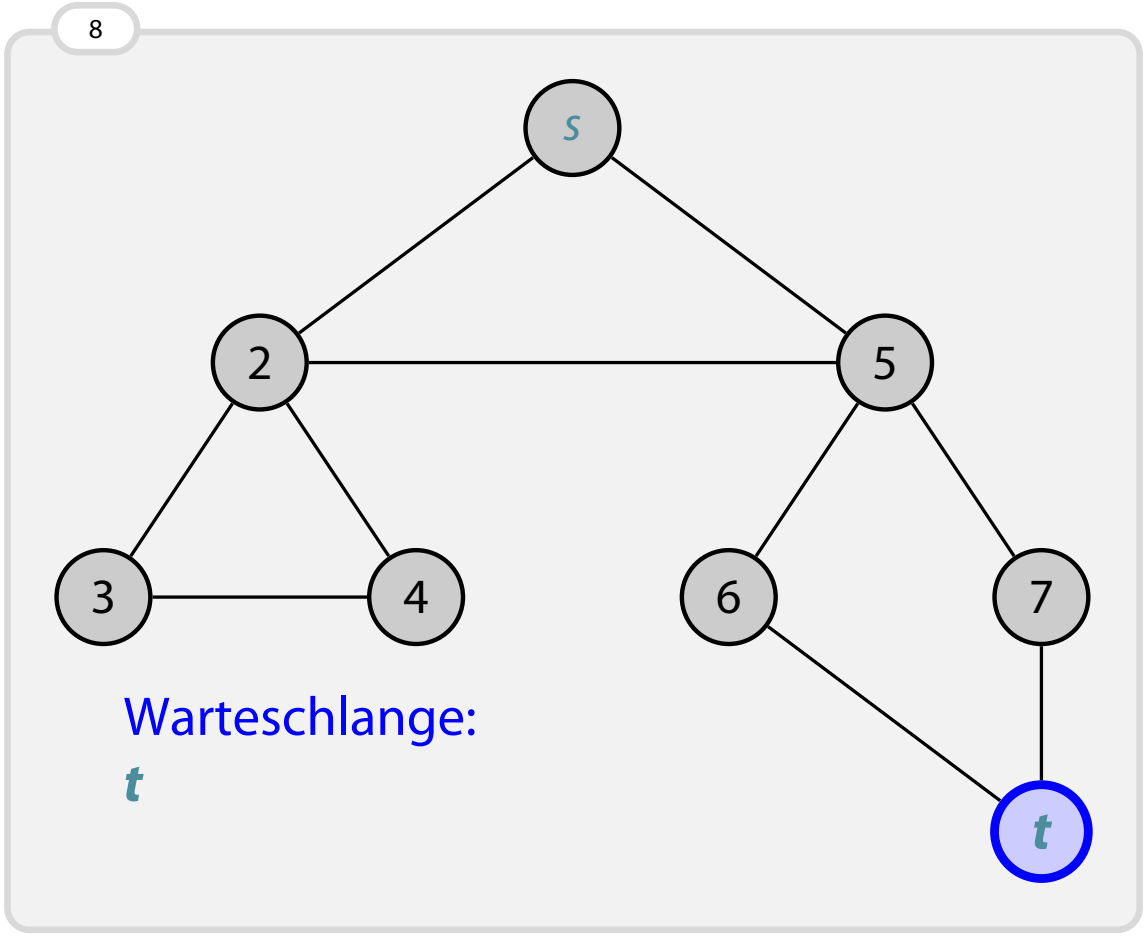
Grünblau = noch nicht besucht, dunkel = schon besucht,
blau = in der Warteschlange, **fett** = gerade besucht

Beispiel der Warteschlange in einer Breitensuche.



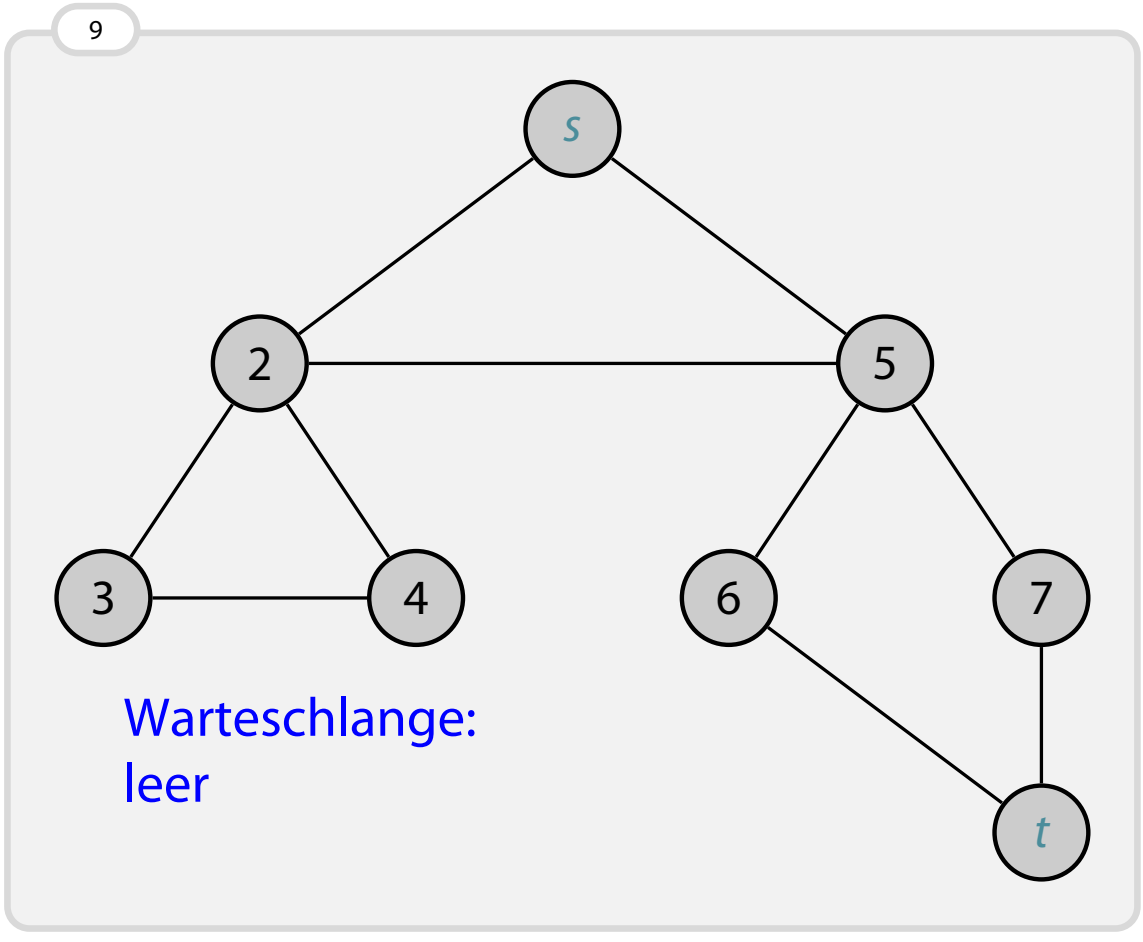
Grünblau = noch nicht besucht, dunkel = schon besucht,
blau = in der Warteschlange, **fett** = gerade besucht

Beispiel der Warteschlange in einer Breitensuche.



Grünblau = noch nicht besucht, dunkel = schon besucht,
blau = in der Warteschlange, **fett** = gerade besucht

Beispiel der Warteschlange in einer Breitensuche.



Grünblau = noch nicht besucht, dunkel = schon besucht,
blau = in der Warteschlange, **fett** = gerade besucht

Die Idee hinter der Tiefensuche.

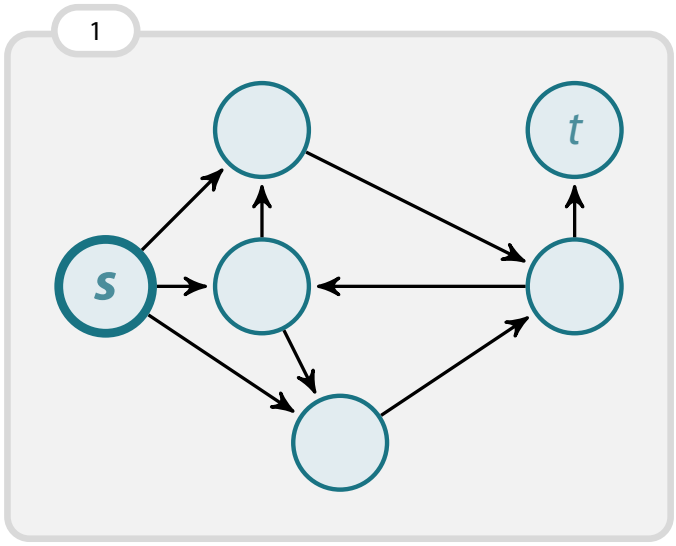
- Ziel ist es wieder, alle von s erreichbare Knoten zu besuchen.
- Diesmal wird der Graph aber nicht »schichtenweise« durchsucht.
- Man versucht vielmehr, möglichst schnell »in die Tiefe zu kommen«.

Motto der Tiefensuche

Vive les enfants! (Kinder zuerst.)

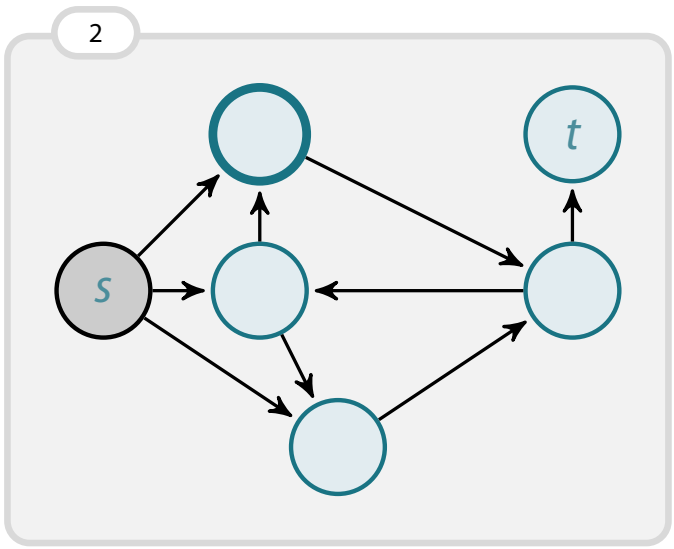
Beispiel einer Tiefensuche.

- Der aktuell besuchte Knoten ist **fett**.
- Schon besuchte Knoten sind dunkel.



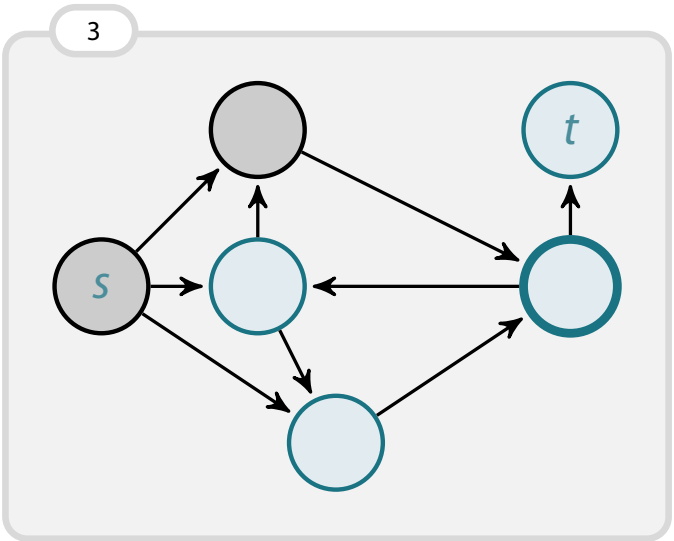
Beispiel einer Tiefensuche.

- Der aktuell besuchte Knoten ist **fett**.
- Schon besuchte Knoten sind dunkel.



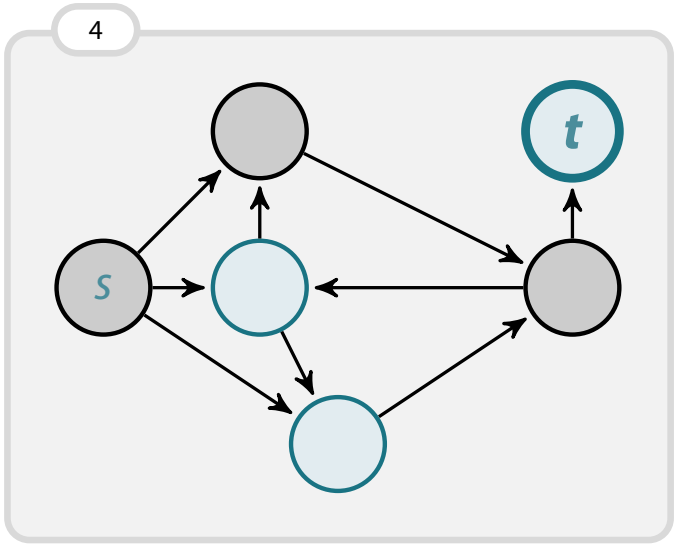
Beispiel einer Tiefensuche.

- Der aktuell besuchte Knoten ist **fett**.
- Schon besuchte Knoten sind dunkel.



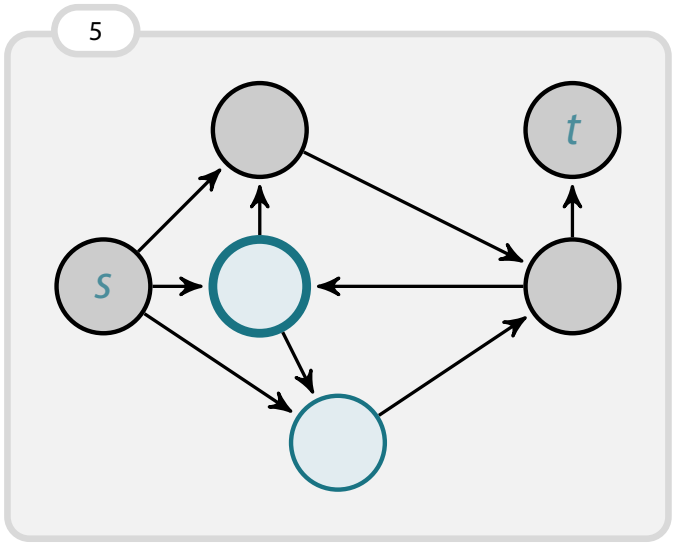
Beispiel einer Tiefensuche.

- Der aktuell besuchte Knoten ist **fett**.
- Schon besuchte Knoten sind dunkel.



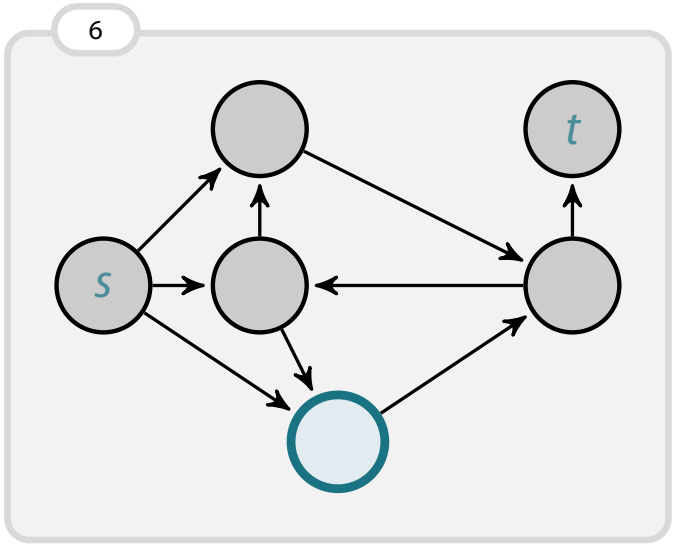
Beispiel einer Tiefensuche.

- Der aktuell besuchte Knoten ist **fett**.
- Schon besuchte Knoten sind dunkel.



Beispiel einer Tiefensuche.

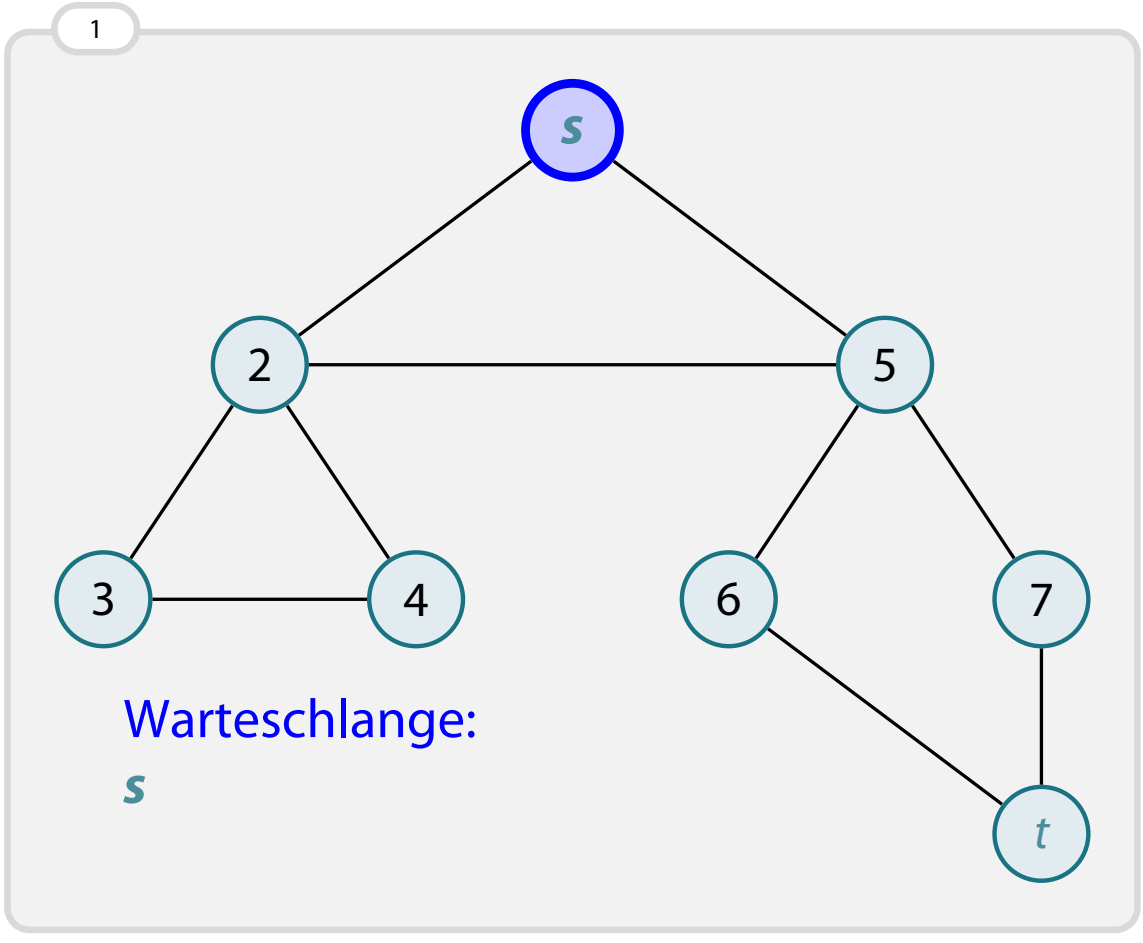
- Der aktuell besuchte Knoten ist **fett**.
- Schon besuchte Knoten sind dunkel.



Zur Implementation einer Tiefensuche.

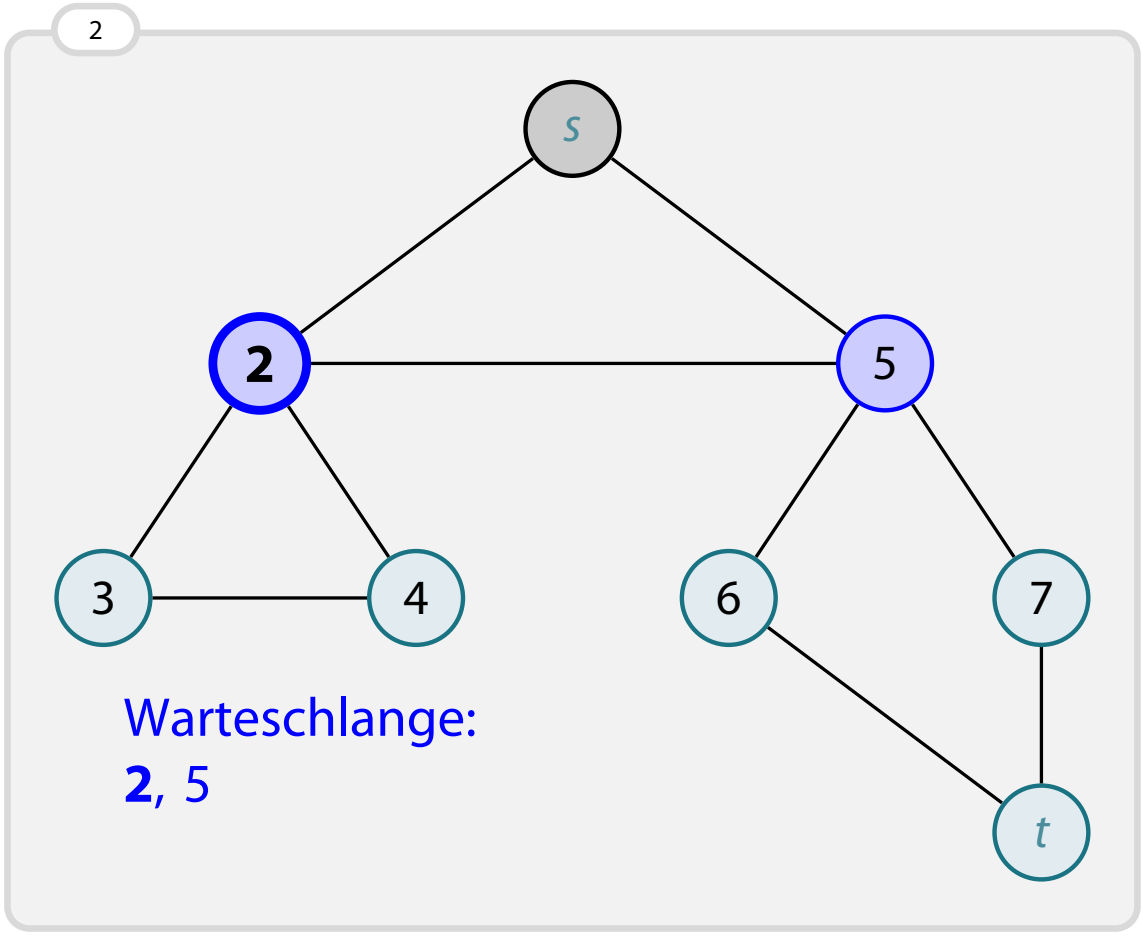
Die Regel ist fast gleich:
»Besuche den ersten Knoten in einer Liste noch zu besuchender Knoten. Füge dann die noch nicht besuchten Nachbarn dieses ersten Knotens *an den Anfang* der Liste an.«

Beispiel der Warteschlange in einer Tiefensuche.



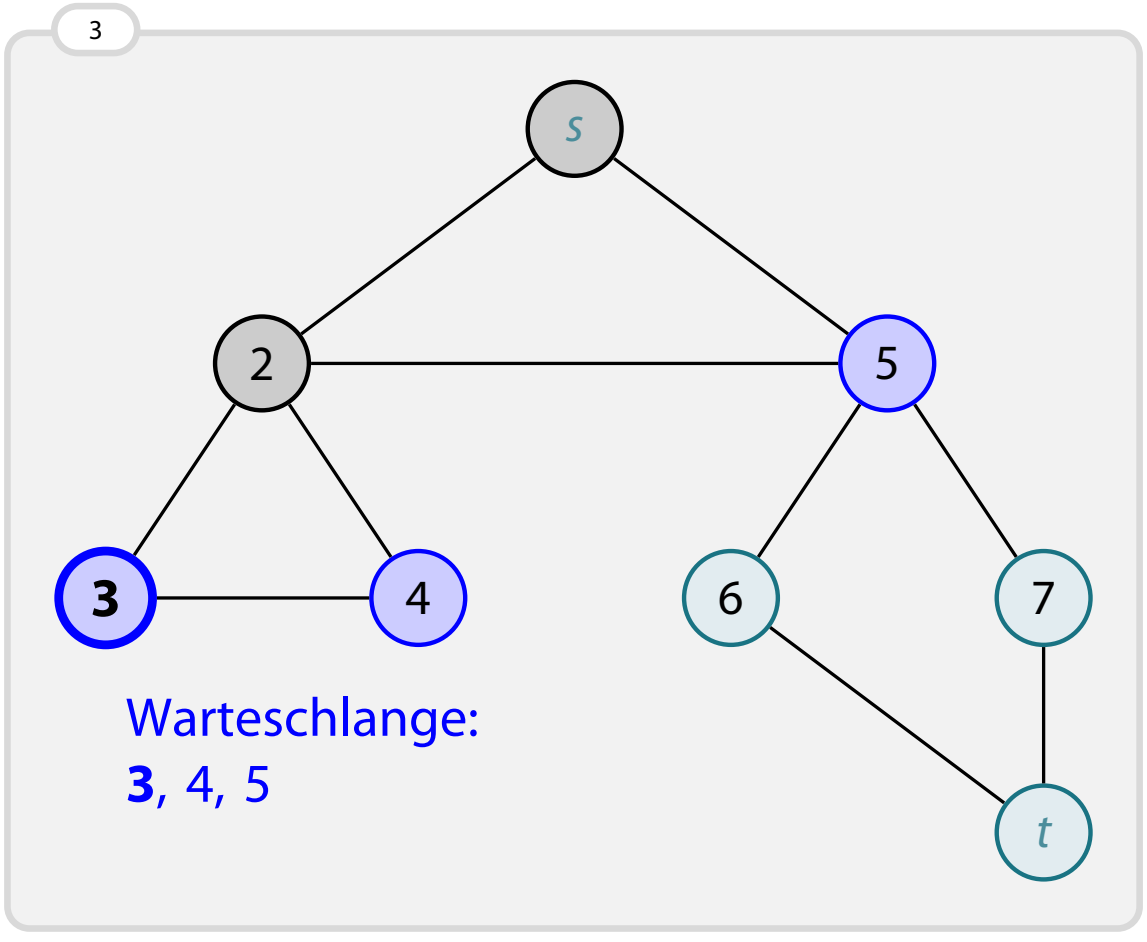
Grünblau = noch nicht besucht, dunkel = schon besucht,
blau = in der Warteschlange, **fett** = gerade besucht

Beispiel der Warteschlange in einer Tiefensuche.



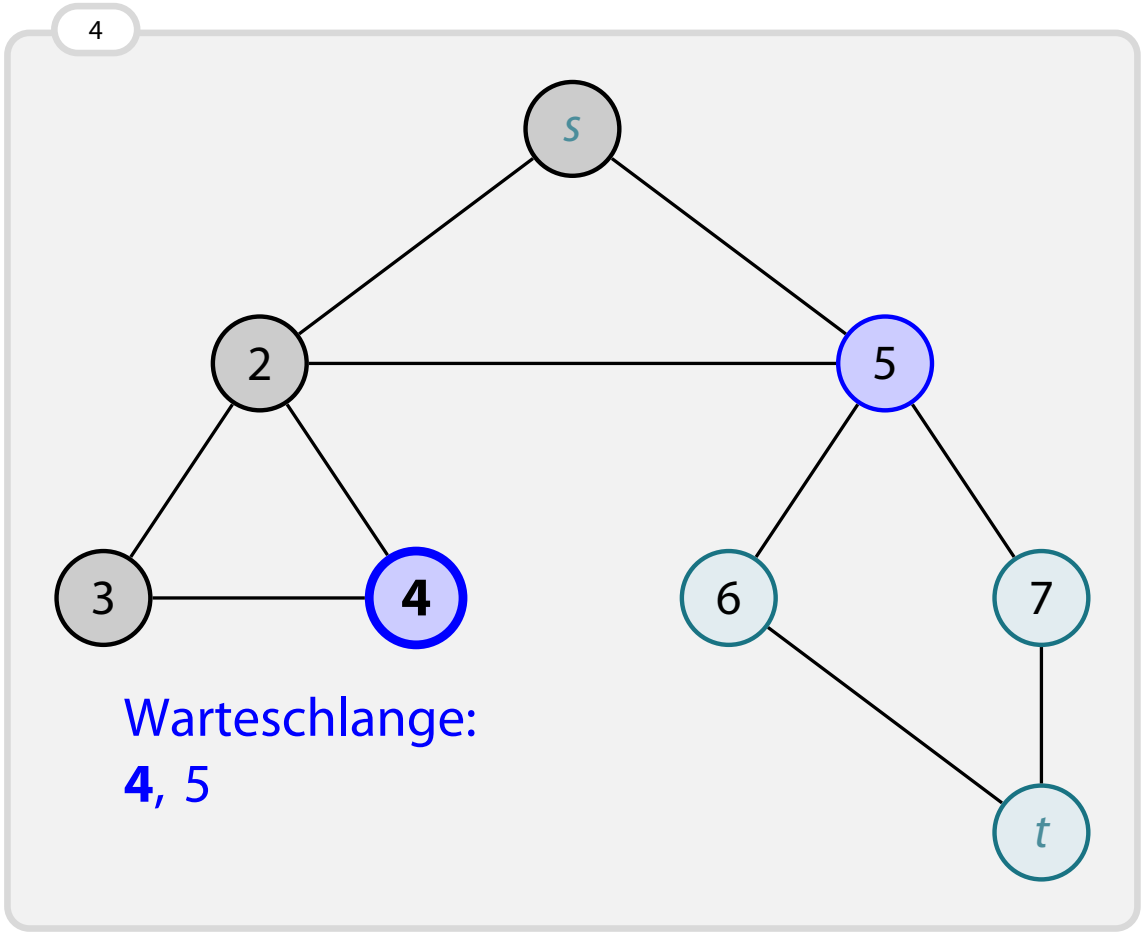
Grünblau = noch nicht besucht, dunkel = schon besucht,
blau = in der Warteschlange, **fett** = gerade besucht

Beispiel der Warteschlange in einer Tiefensuche.



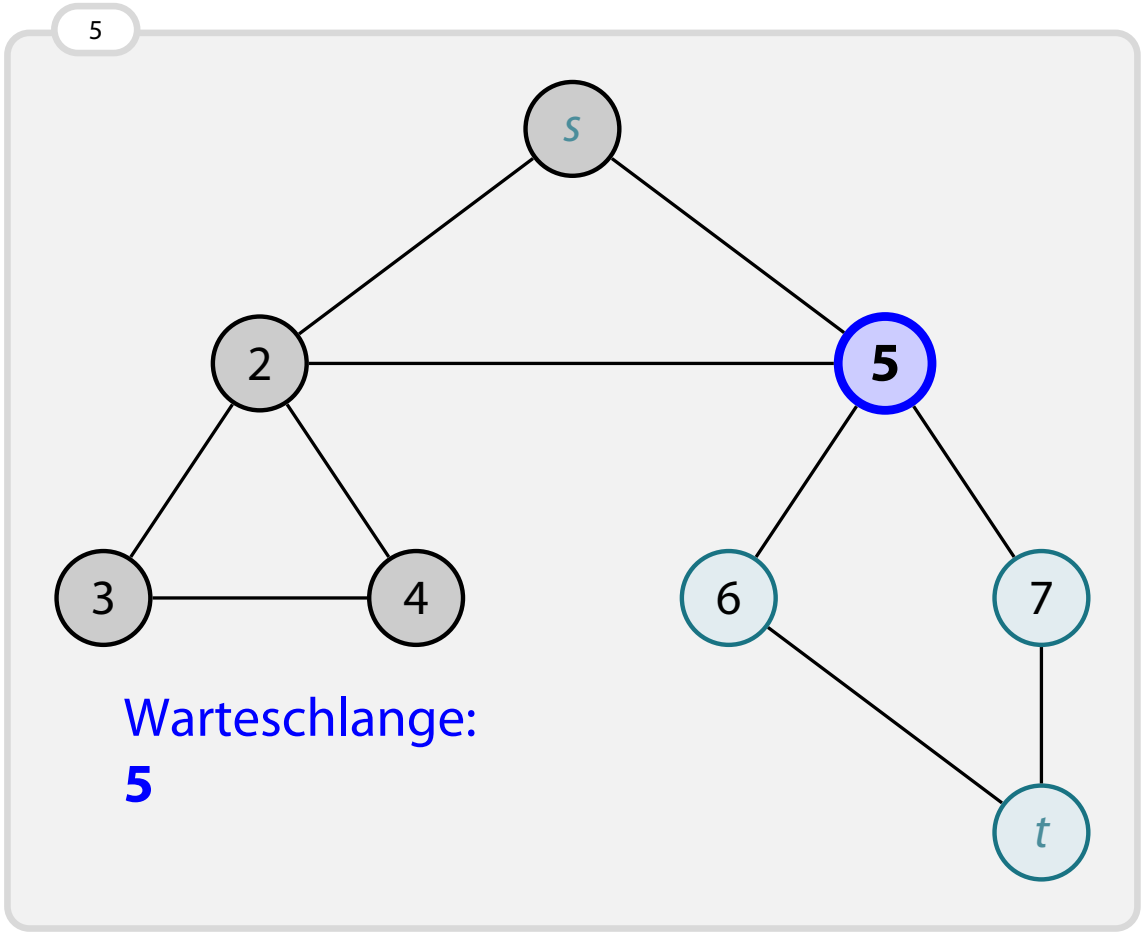
Grünblau = noch nicht besucht, dunkel = schon besucht,
blau = in der Warteschlange, **fett** = gerade besucht

Beispiel der Warteschlange in einer Tiefensuche.



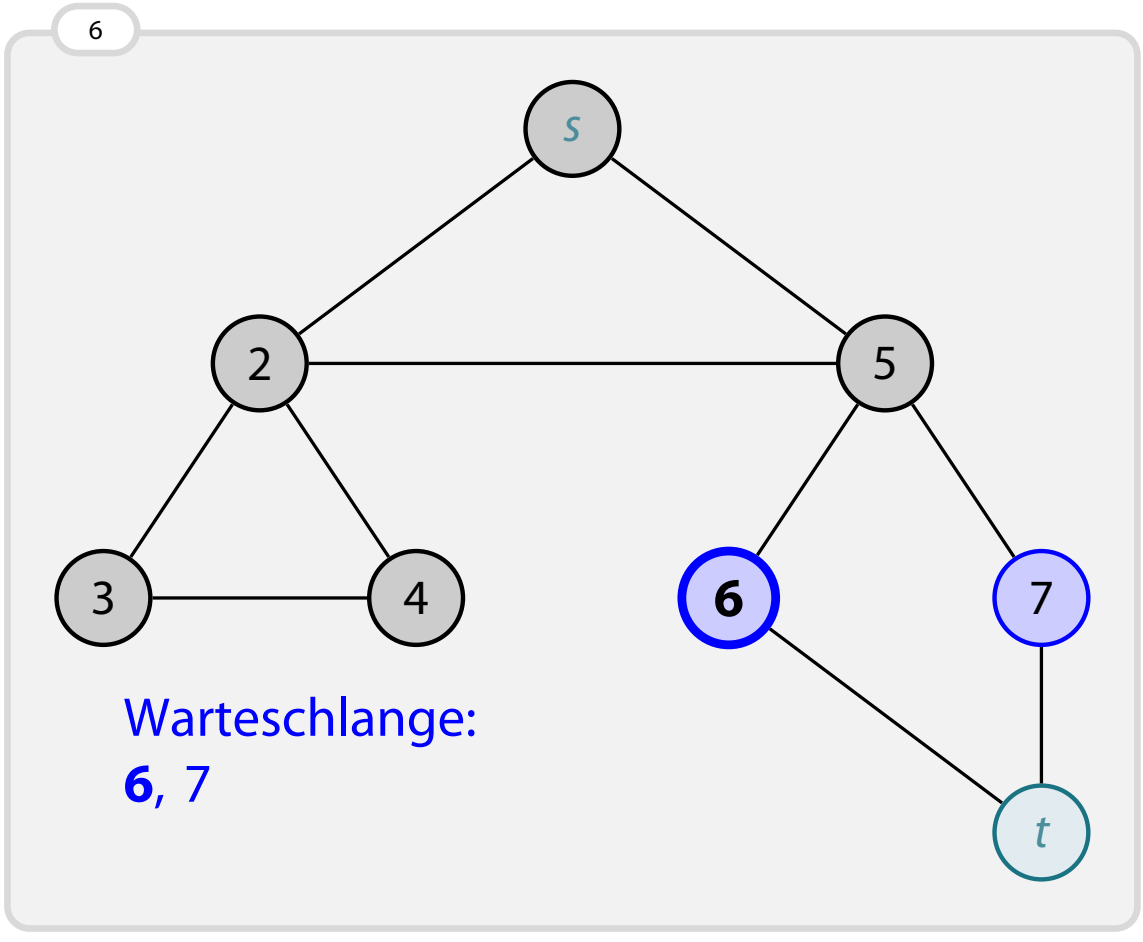
Grünblau = noch nicht besucht, dunkel = schon besucht,
blau = in der Warteschlange, **fett** = gerade besucht

Beispiel der Warteschlange in einer Tiefensuche.



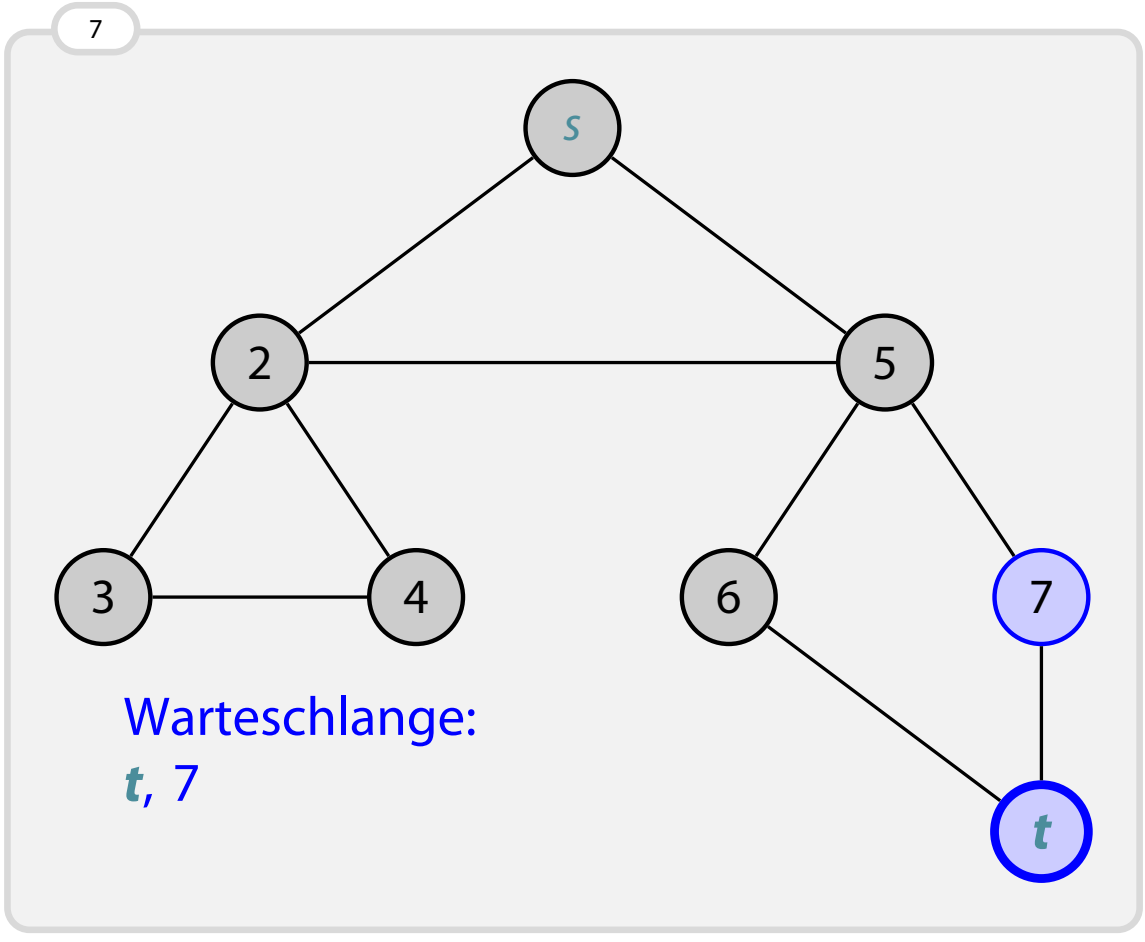
Grünblau = noch nicht besucht, dunkel = schon besucht,
blau = in der Warteschlange, **fett** = gerade besucht

Beispiel der Warteschlange in einer Tiefensuche.



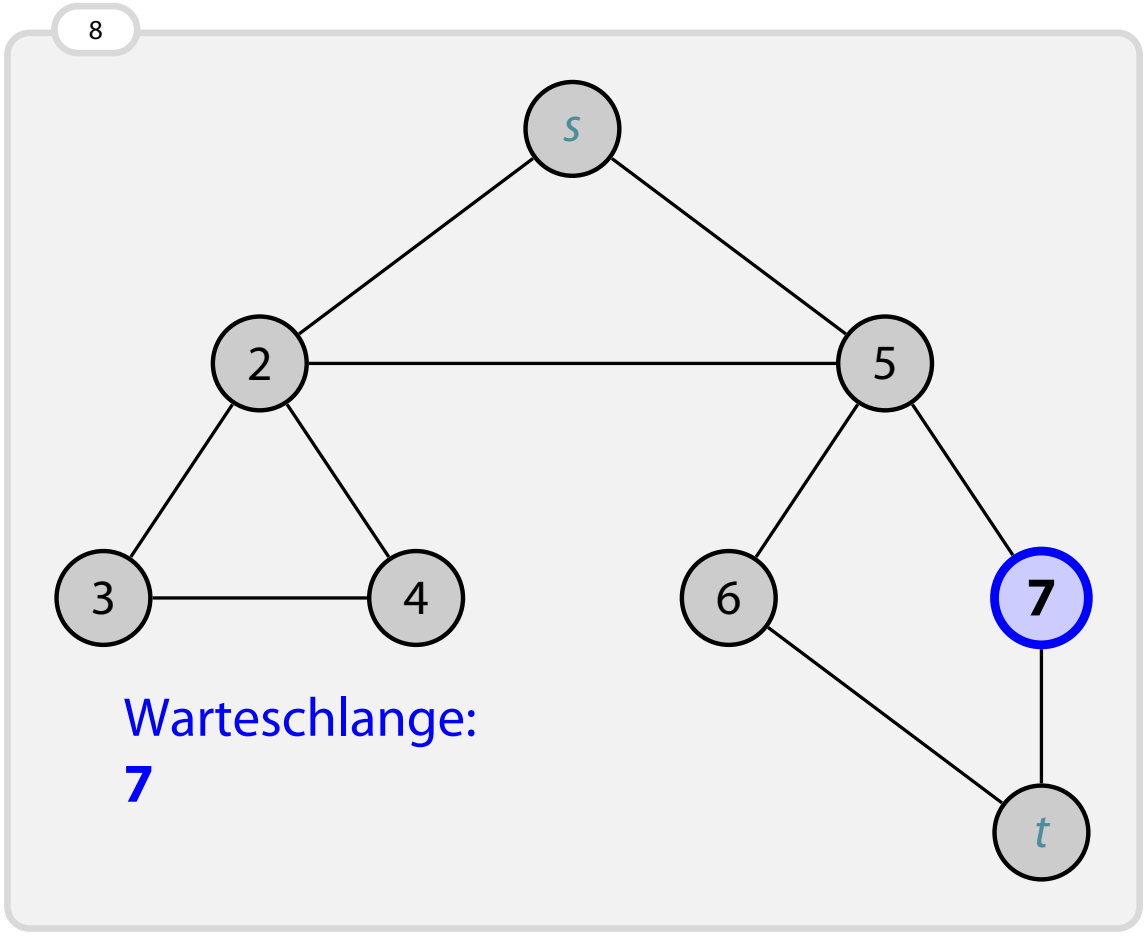
Grünblau = noch nicht besucht, dunkel = schon besucht,
blau = in der Warteschlange, **fett** = gerade besucht

Beispiel der Warteschlange in einer Tiefensuche.



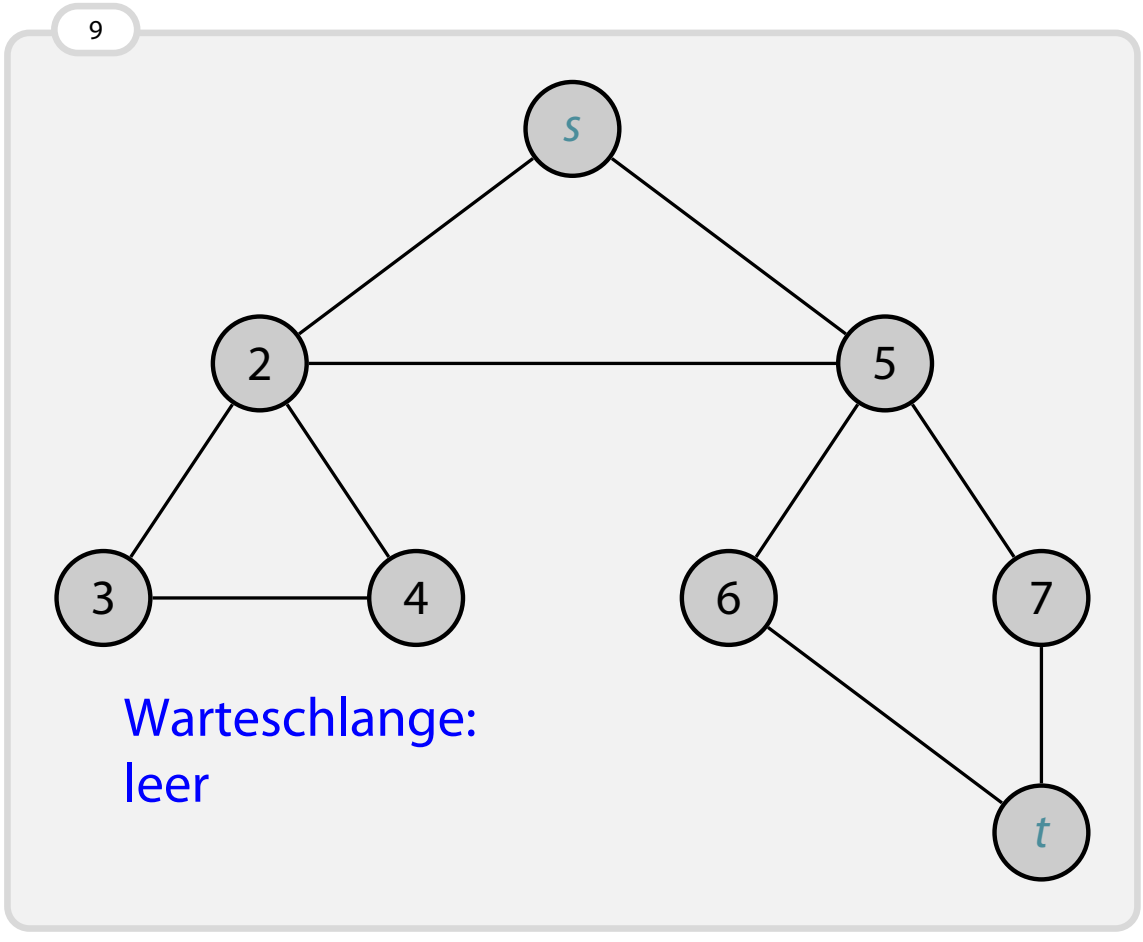
Grünblau = noch nicht besucht, dunkel = schon besucht,
blau = in der Warteschlange, **fett** = gerade besucht

Beispiel der Warteschlange in einer Tiefensuche.



Grünblau = noch nicht besucht, dunkel = schon besucht,
blau = in der Warteschlange, **fett** = gerade besucht

Beispiel der Warteschlange in einer Tiefensuche.

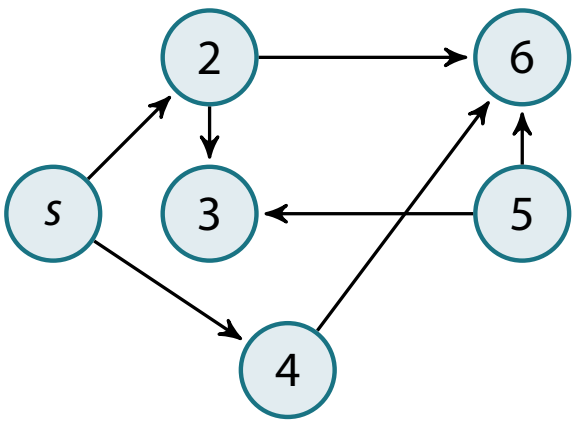


Grünblau = noch nicht besucht, dunkel = schon besucht,
blau = in der Warteschlange, **fett** = gerade besucht

5-Minuten-Aufgabe

Zur Übung

In welcher Reihenfolge werden die Knoten des folgenden Graphen bei einer Tiefensuche und in welcher bei einer Breitensuche, jeweils beginnend bei Knoten *s*, durchlaufen?



Vergleich Breiten- und Tiefensuche

- Breiten- und Tiefensuche werden ähnlich implementiert.
- Der einzige Unterschied liegt darin, wo die noch nicht besuchten Nachbarn in die Warteschlage eingefügt werden.
- Bei einer *Breitensuche* werden sie *hinten* angefügt.
  (Man spricht von »fifo« – first in, first out.)
- Bei einer *Tiefensuche* werden sie *vorne* angefügt.
  (Man spricht von »lifo« – last in, first out.)

Das Kürzeste-Wege-Problem

Beispiel (Kürzeste-Wege-Problem)

Eingabe Ein gerichteter oder ungerichteter Graph eventuell mit *positiven* Kantengewichten.
Zusätzlich zwei Knoten s (source) und t (target).

Ausgabe Ein kürzester Weg von s nach t (die Summe der Gewichte soll minimal sein).

Dieses Problem muss gelöst werden

- von einem Navigationssystem,
- von einer Telefongesellschaft oder auch
- von einem Internet-Anbieter.

Breitensuche löst das Kürzeste-Wege-Problem in Graphen ohne Kantengewichte.

Beobachtung

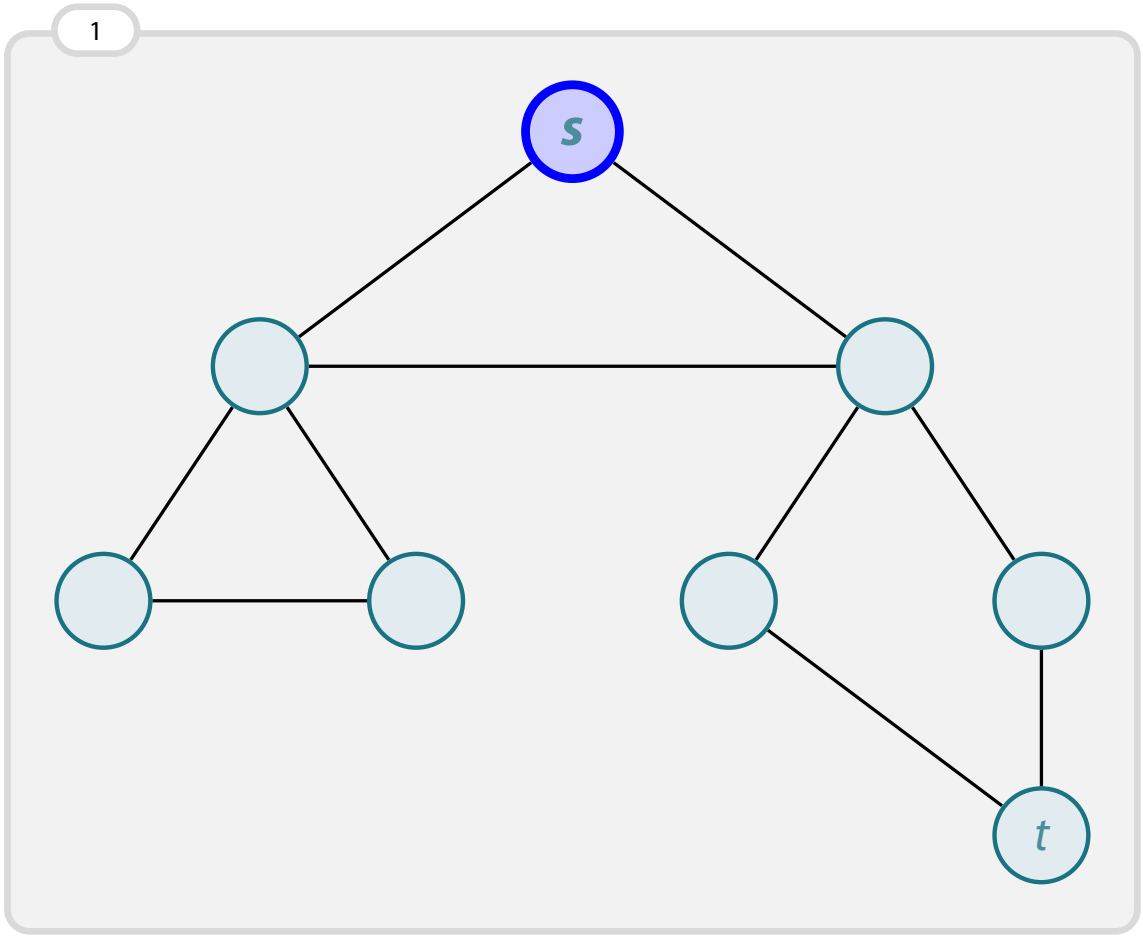
Die Breitensuche besucht

- zuerst die Knoten mit Entfernung 1 von s,
- dann die Knoten mit Entfernung 2 von s,
- dann die Knoten mit Entfernung 3 von s
- und so weiter.

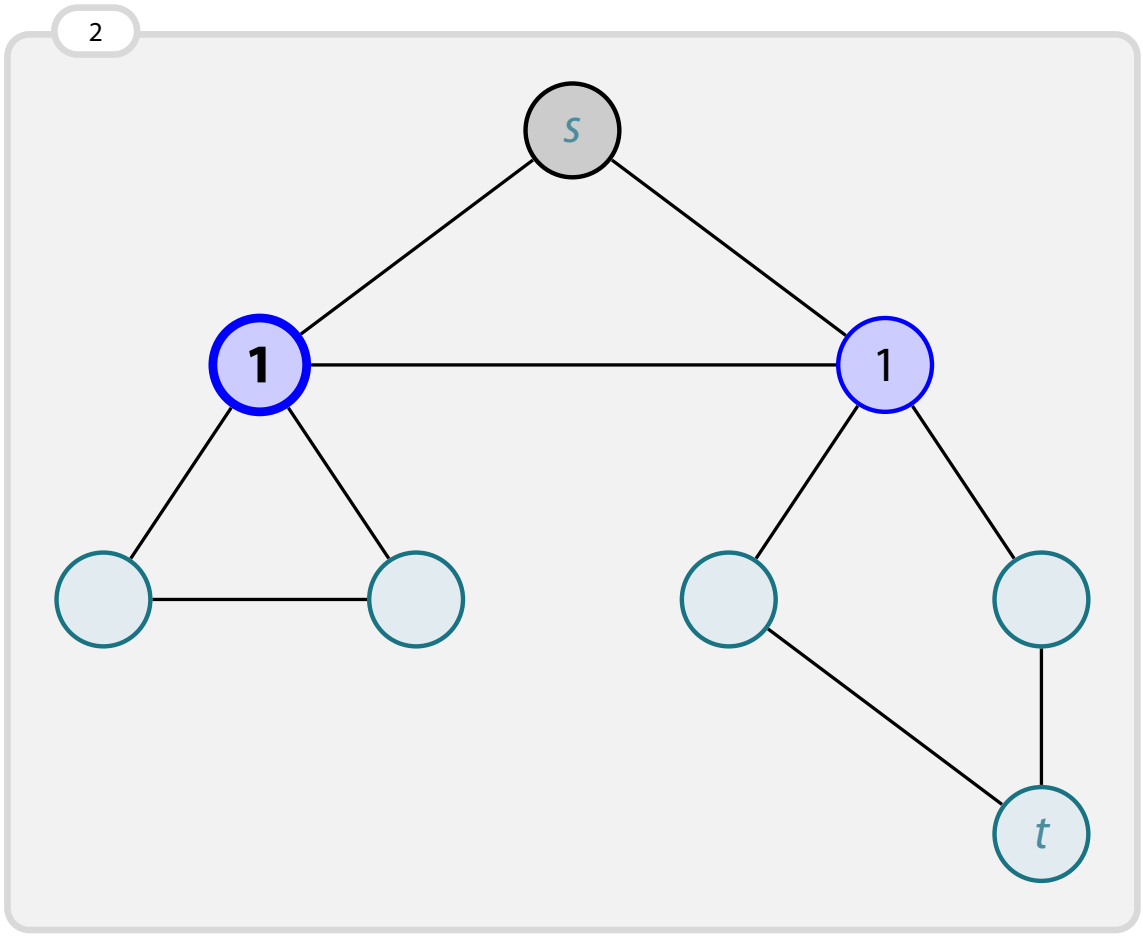
Algorithmus für kürzeste Wege in ungewichteten Graphen

1. Führe eine Breitensuche ausgehend von s durch.
2. Speichere dabei in jedem Knoten nicht nur, ob er besucht wurde, sondern auch seine Entfernung von s.
3. Der gesuchte Weg von s nach t ergibt sich am Ende, indem man von t immer zum Nachbarknoten mit der um 1 kleineren Entfernung geht.

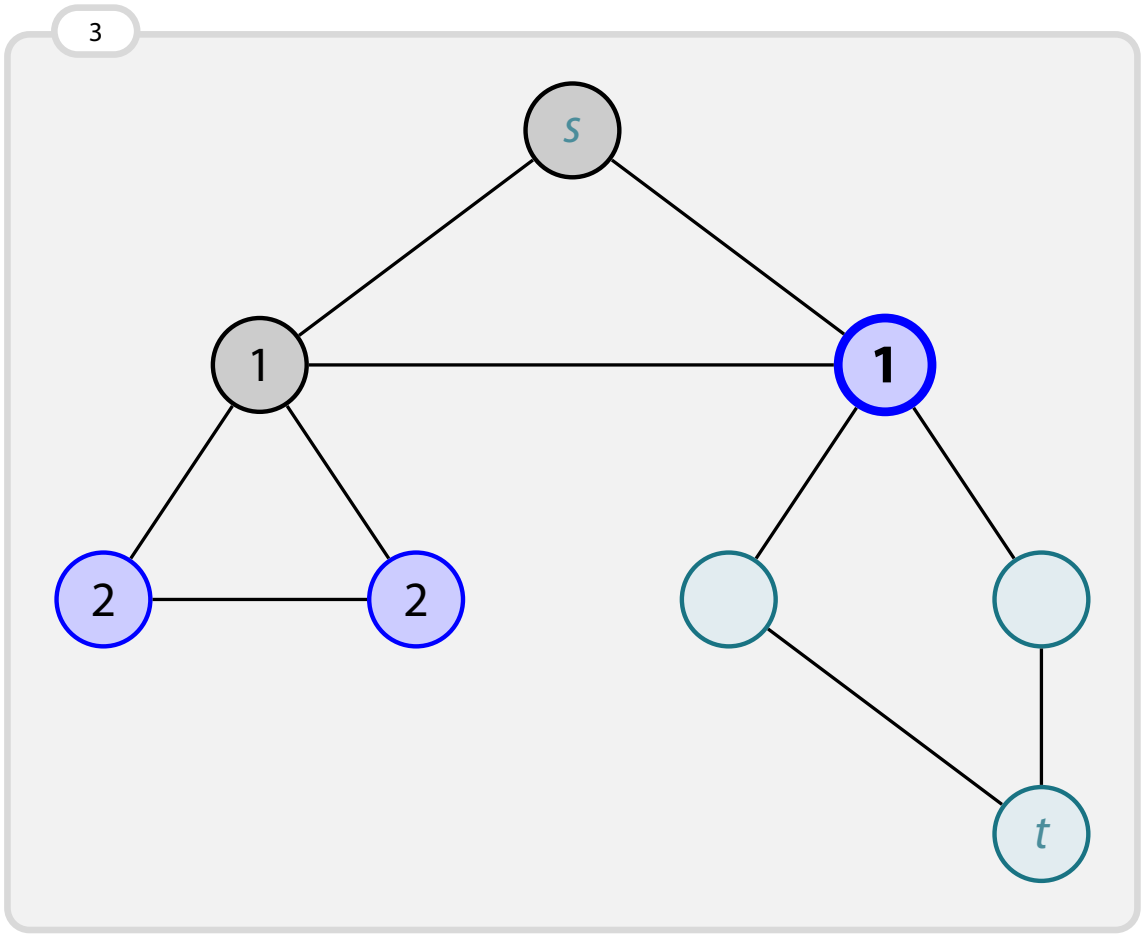
Beispielablauf des Algorithmus.



Beispielablauf des Algorithmus.

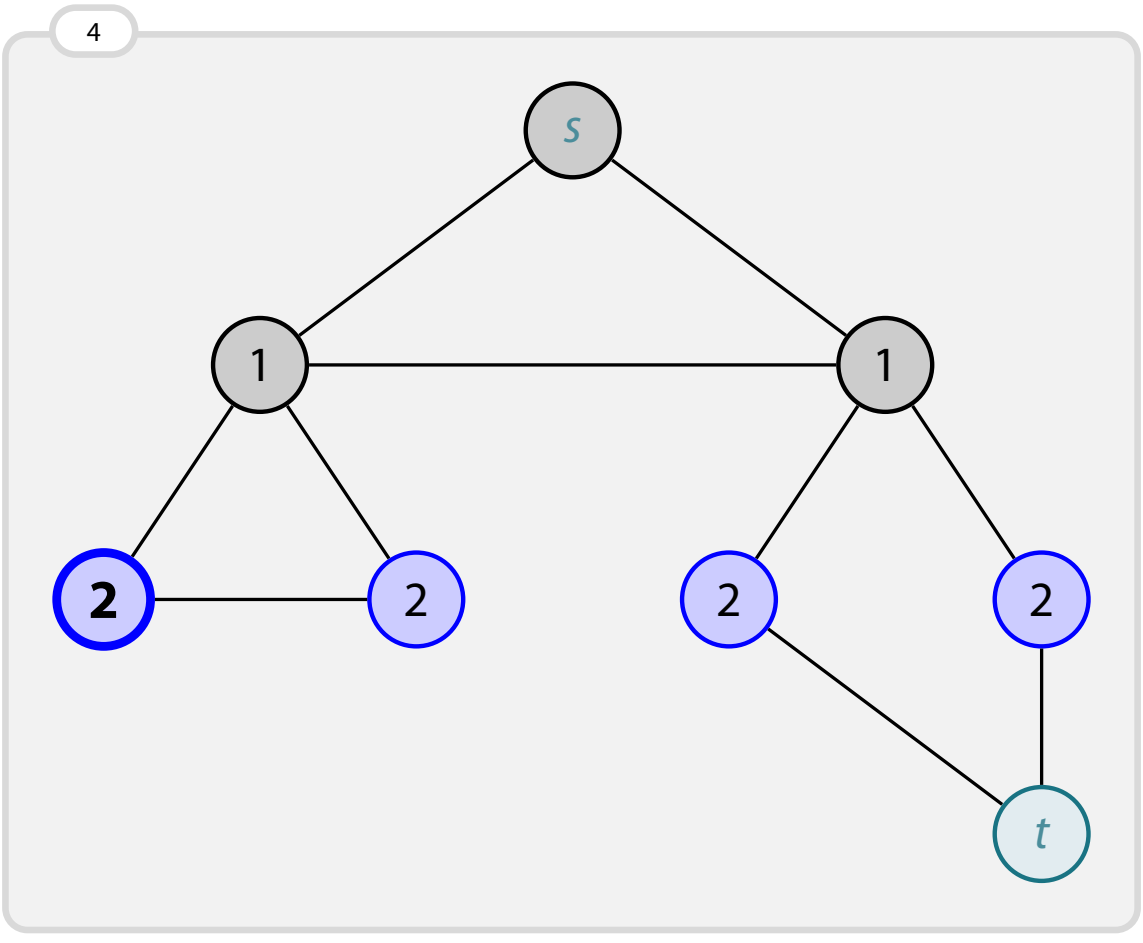


Beispielablauf des Algorithmus.

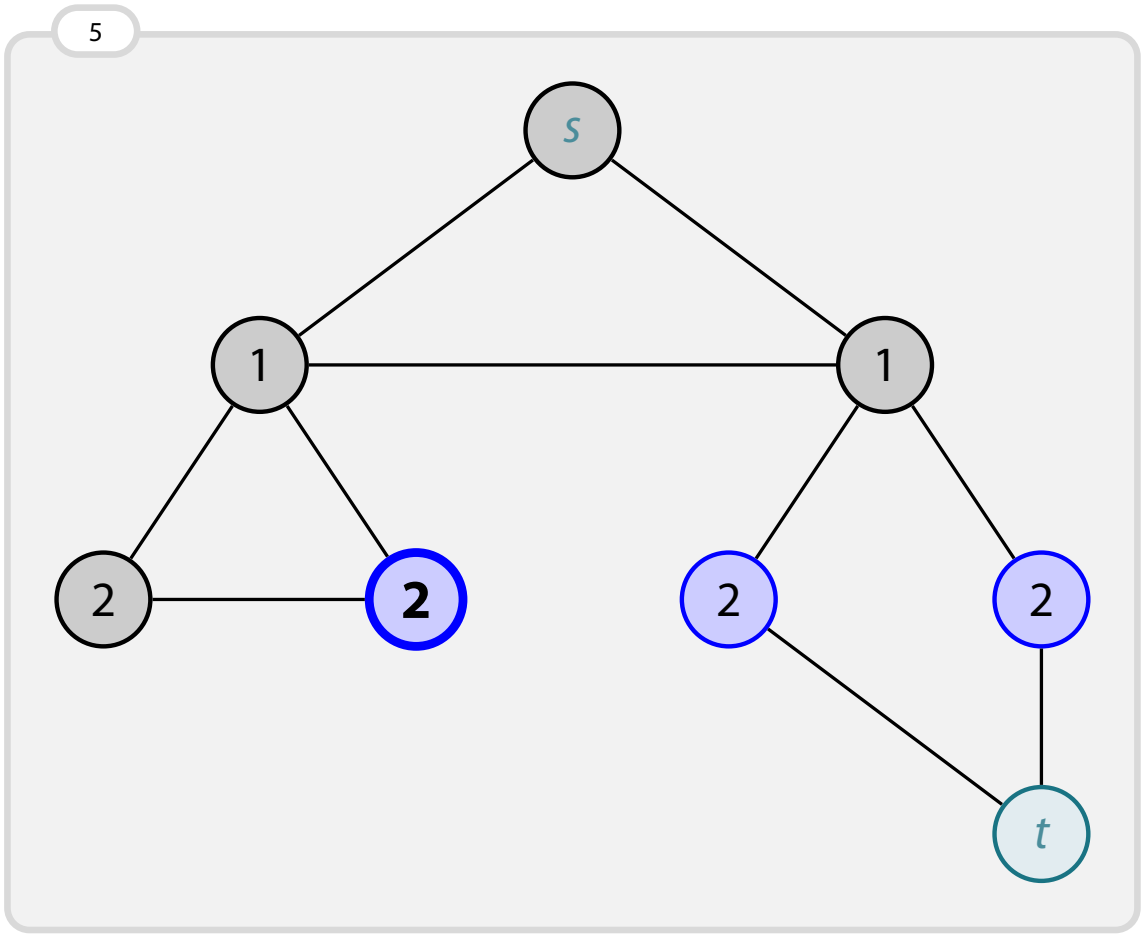


Beispielablauf des Algorithmus.

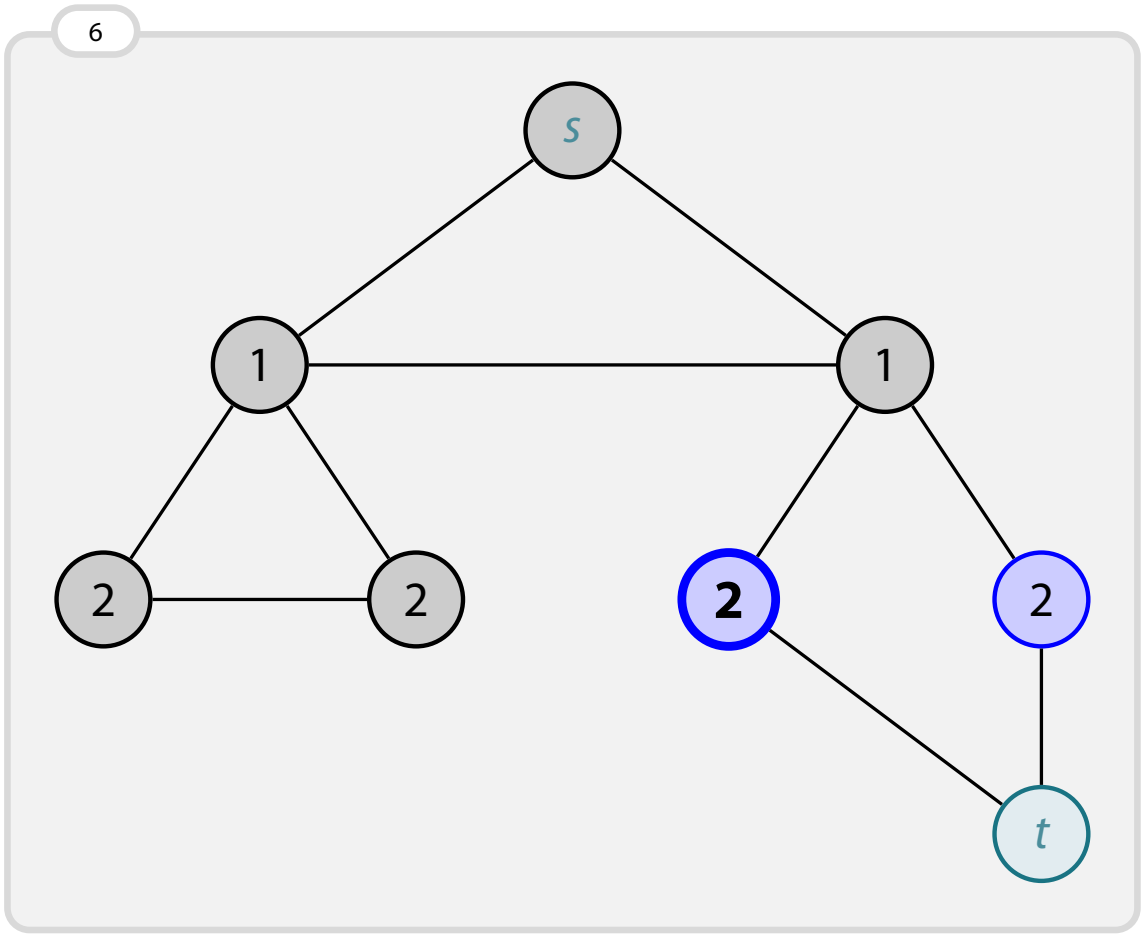
4



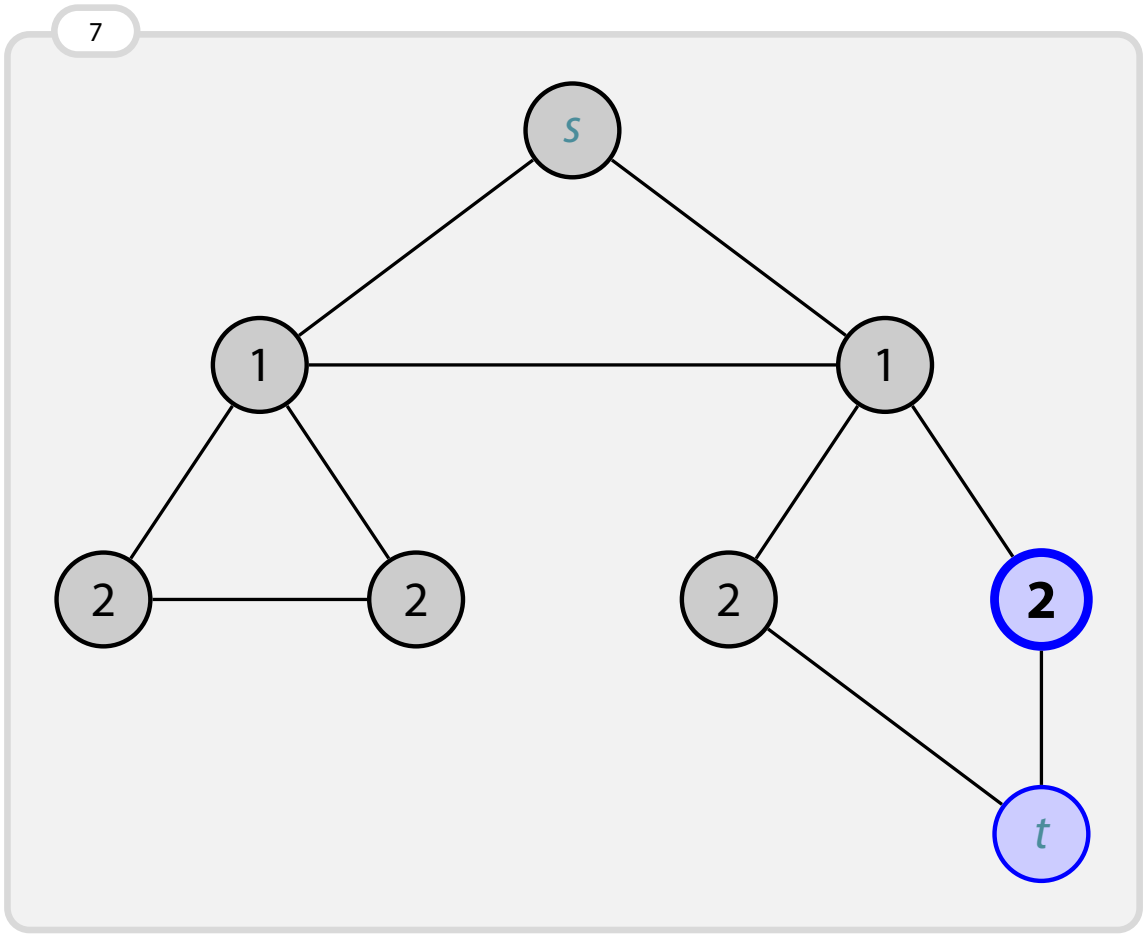
Beispielablauf des Algorithmus.



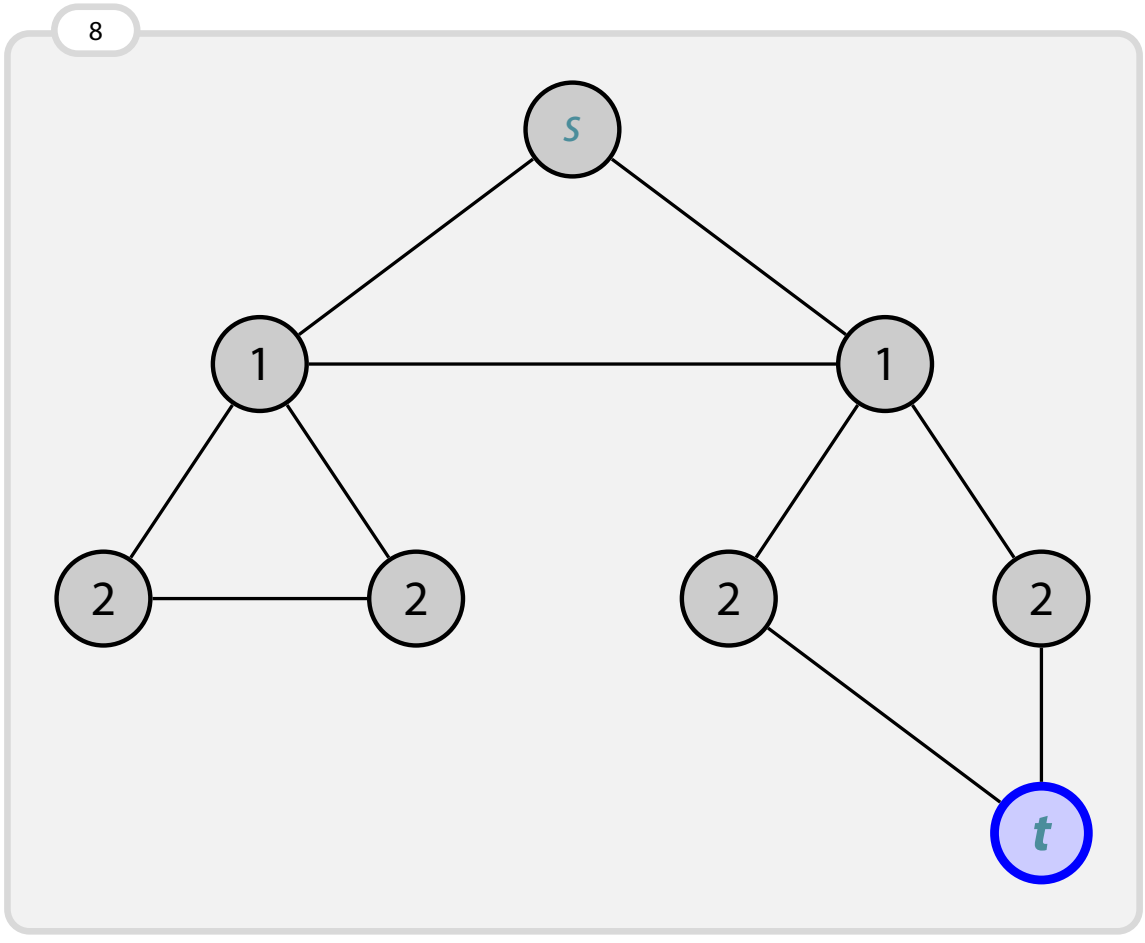
Beispielablauf des Algorithmus.



Beispielablauf des Algorithmus.

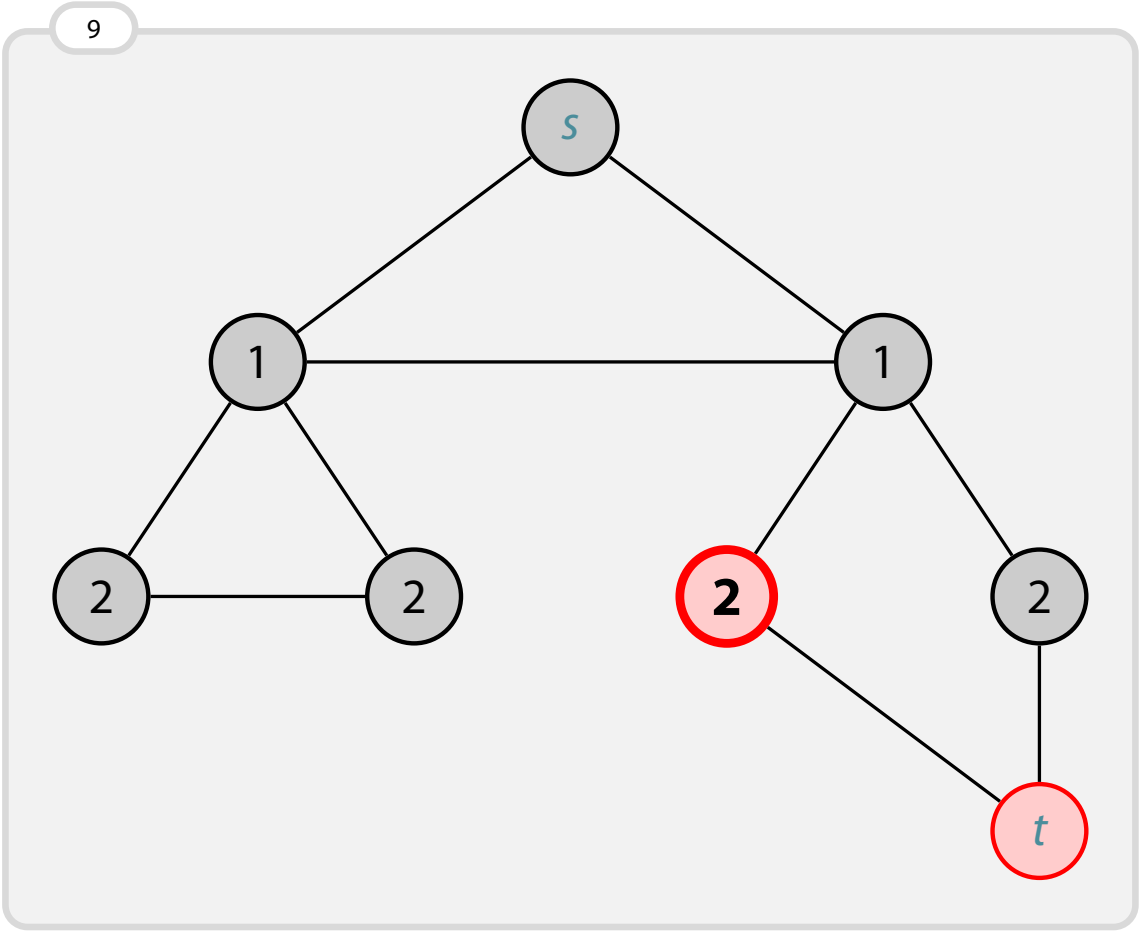


Beispielablauf des Algorithmus.

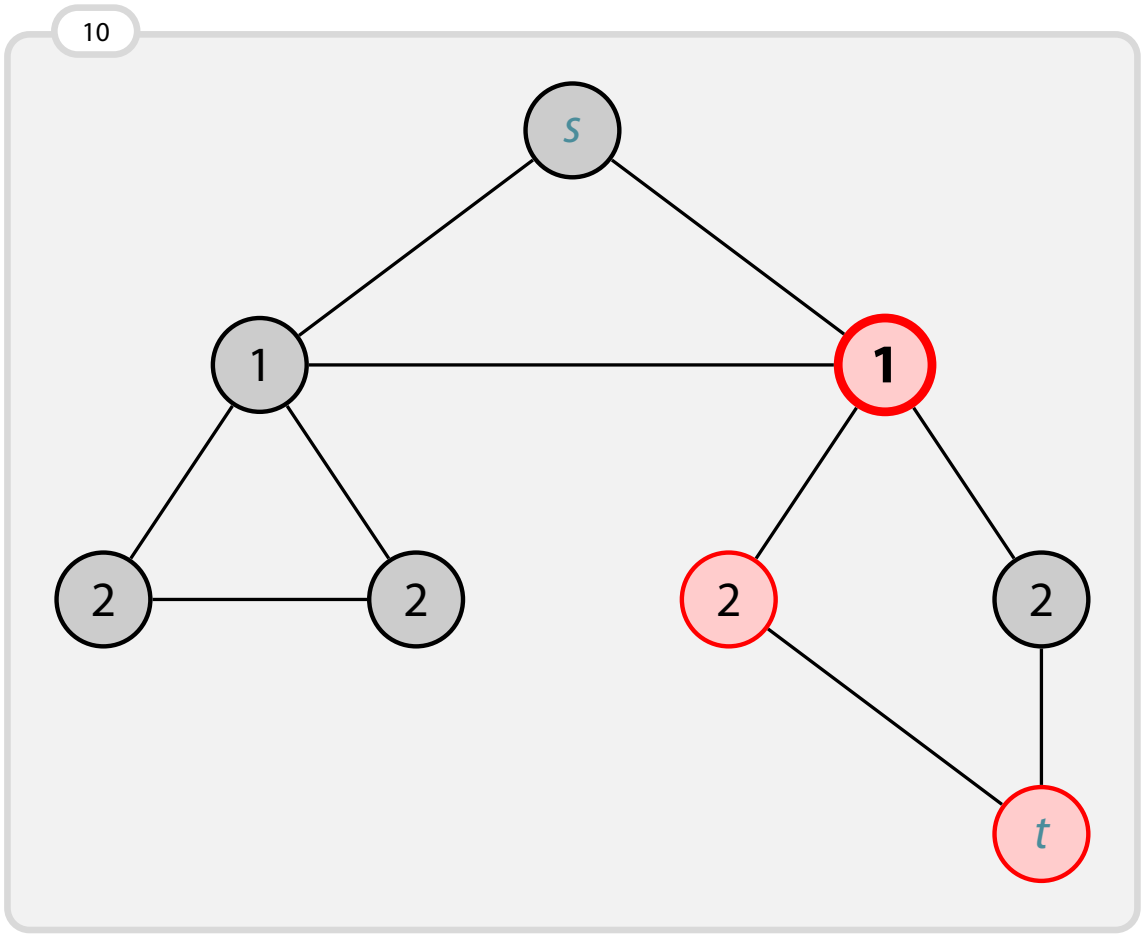


Beispielablauf des Algorithmus.

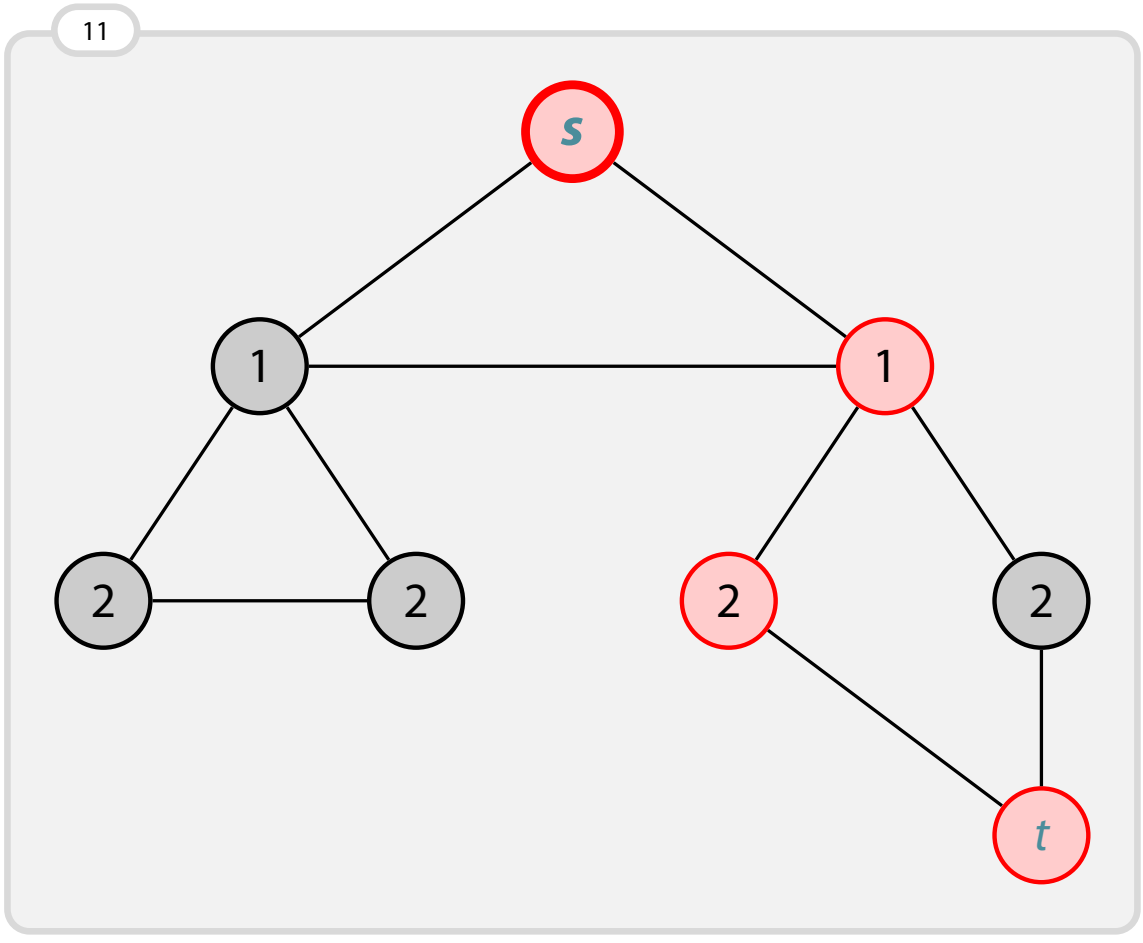
9



Beispielablauf des Algorithmus.



Beispielablauf des Algorithmus.

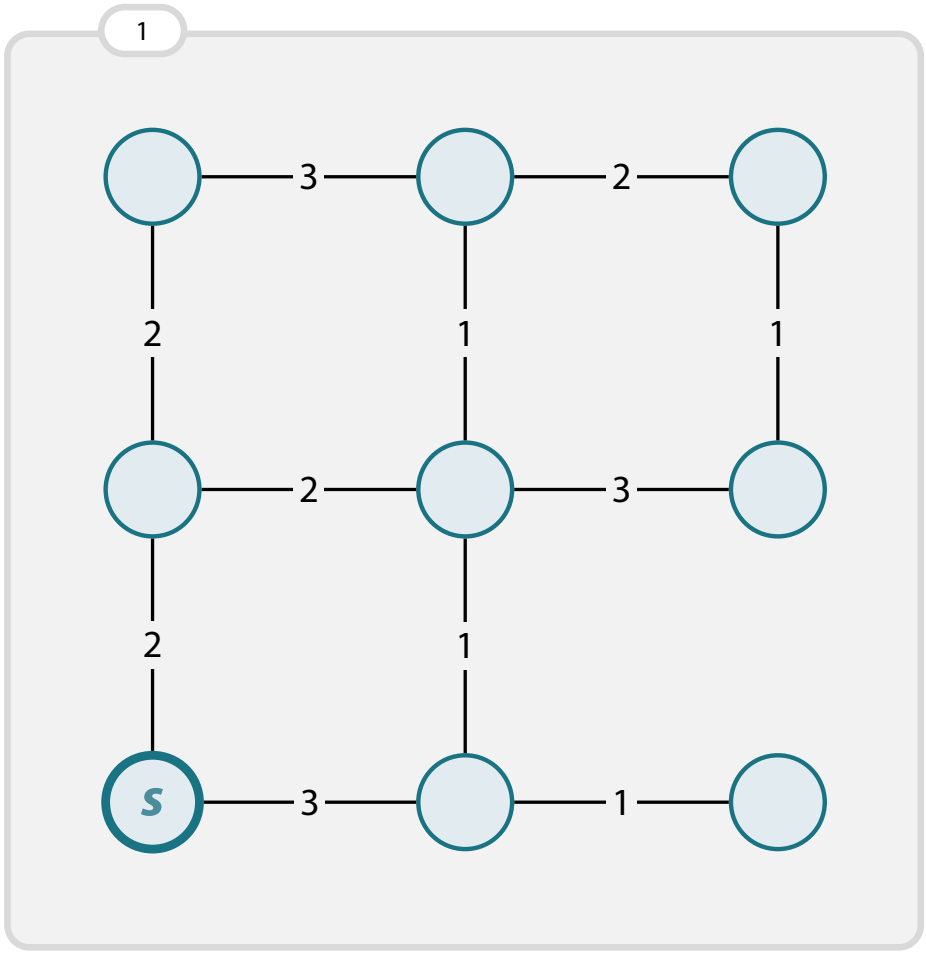


Die Grundidee von Dijkstras Algorithmus.

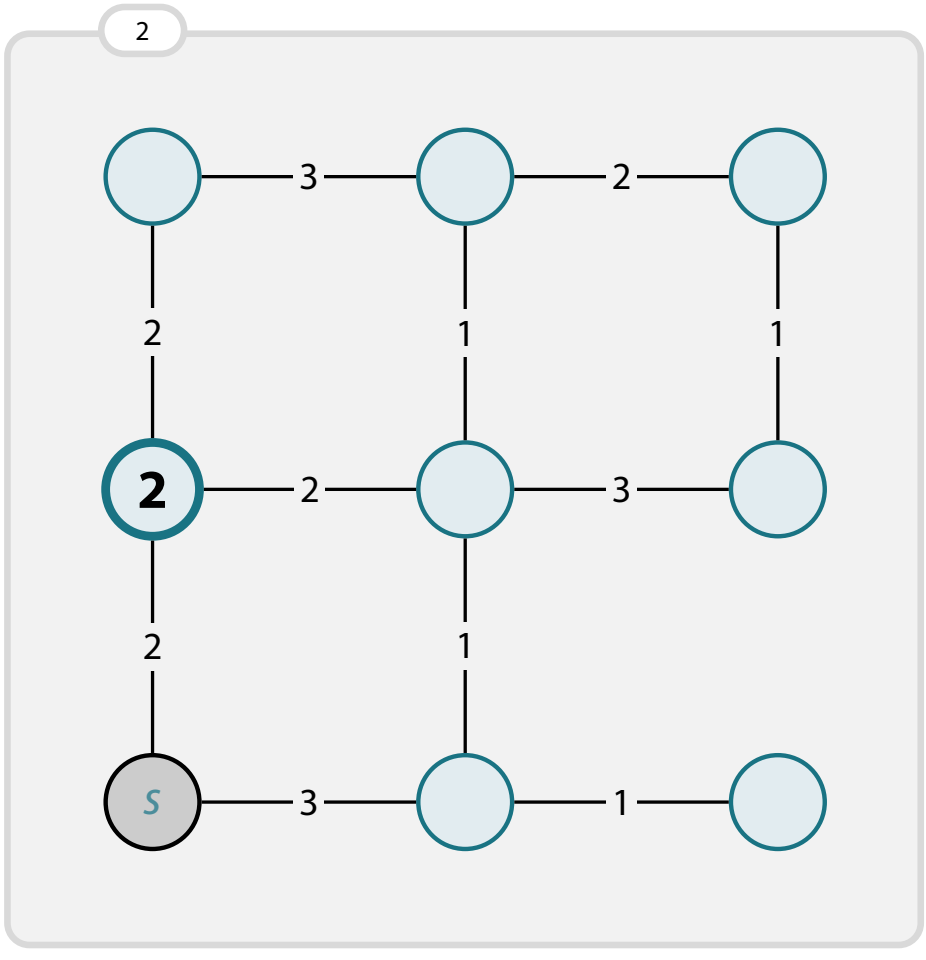
Dijkstras Algorithmus ist eine Art *verallgemeinerte Breitensuche, die auch Kantengewichte zulässt*.

- Der Algorithmus besucht ebenfalls alle von s erreichbare Knoten.
- Er speichert ebenfalls für jeden schon besuchten Knoten seine *Entfernung vom Startknoten*.
- Neu ist: In jedem Schritt besuchen wir als nächstes den Knoten, der durch eine Kante so mit einem schon besuchten Knoten verbunden werden kann, dass die *Entfernung zu s minimal* ist.

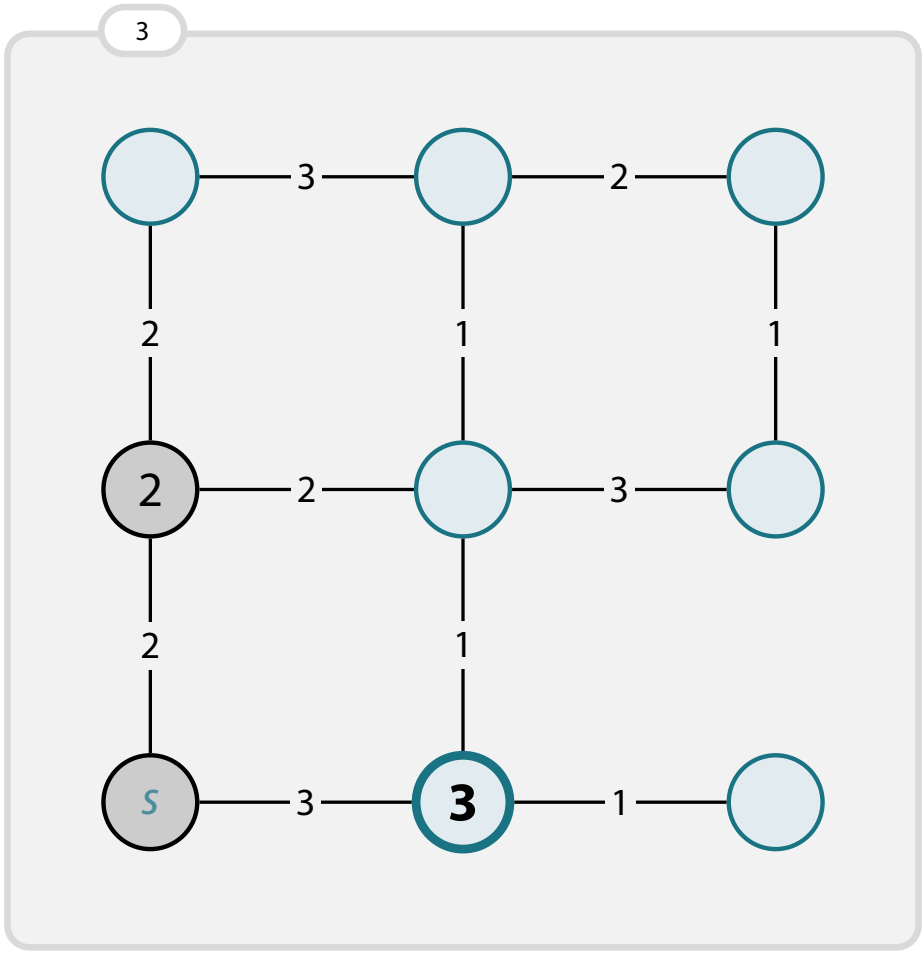
Der Dijkstra-Algorithmus anhand eines Beispiels.



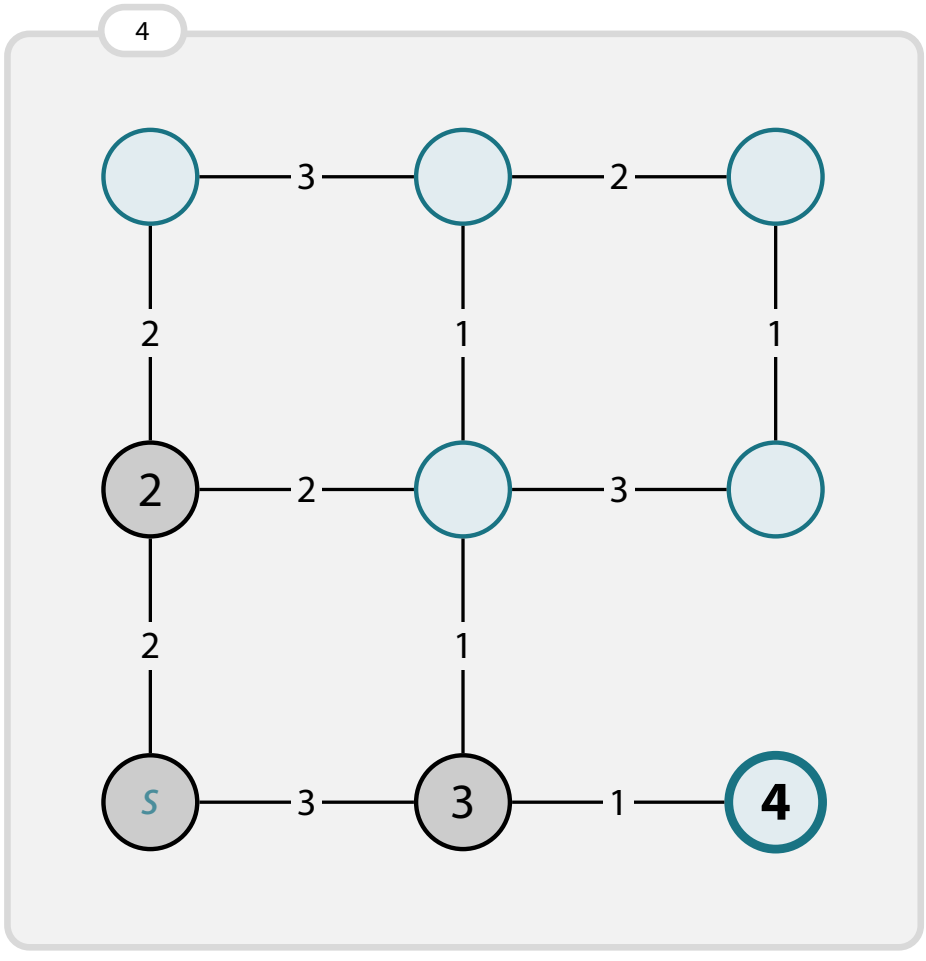
Der Dijkstra-Algorithmus anhand eines Beispiels.



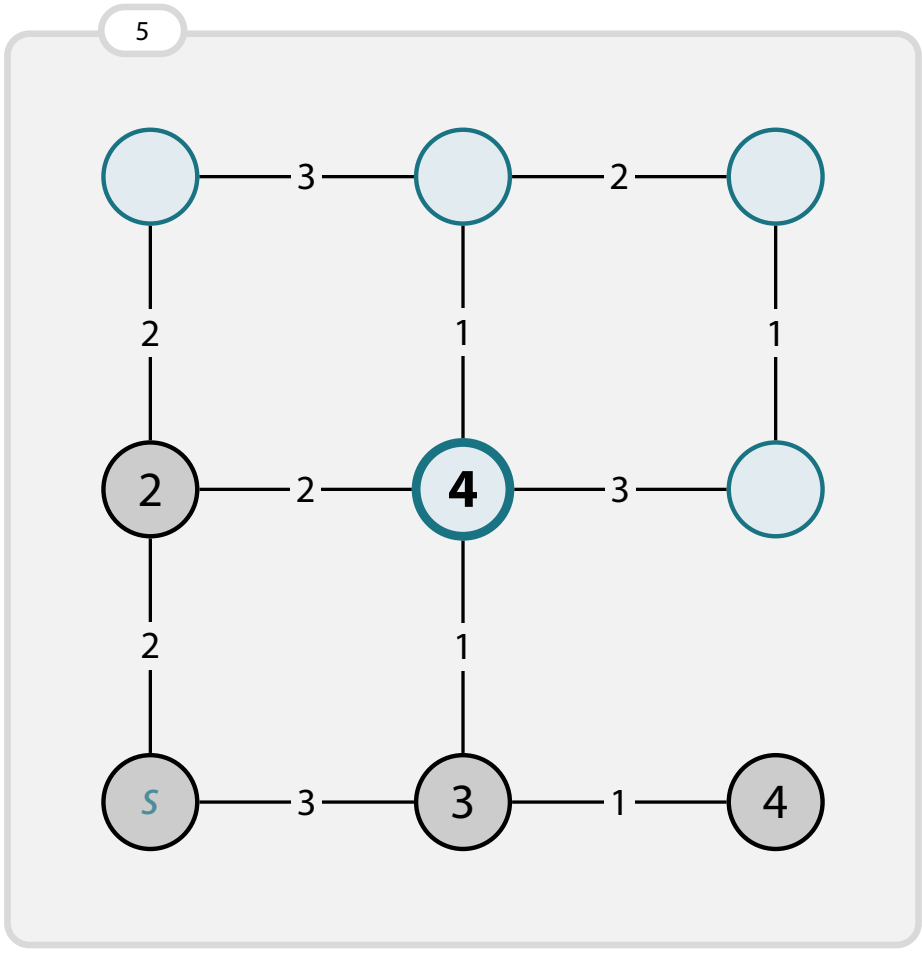
Der Dijkstra-Algorithmus anhand eines Beispiels.



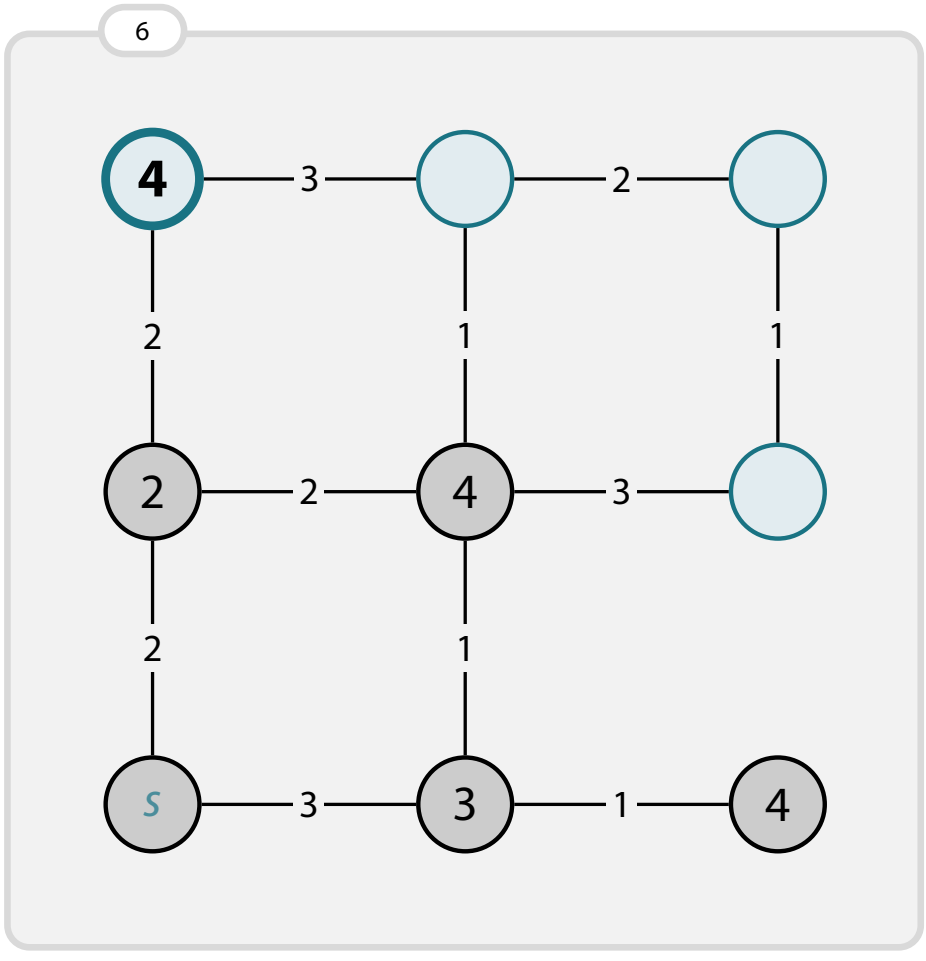
Der Dijkstra-Algorithmus anhand eines Beispiels.



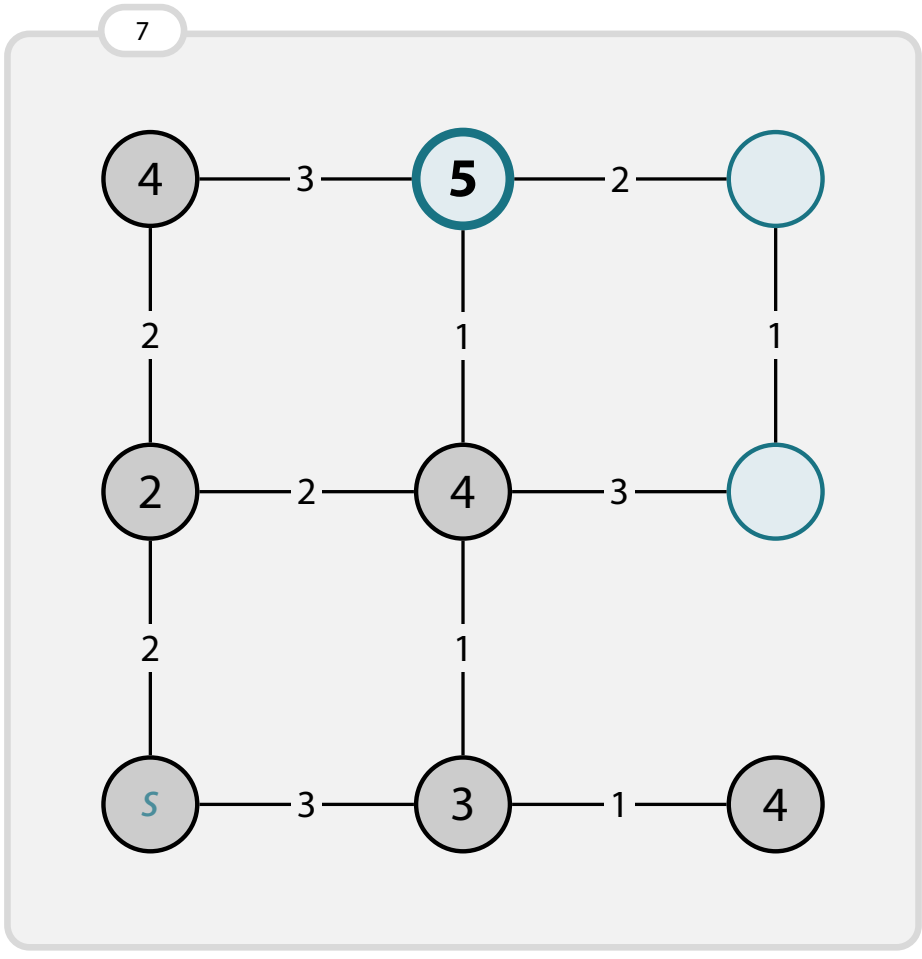
Der Dijkstra-Algorithmus anhand eines Beispiels.



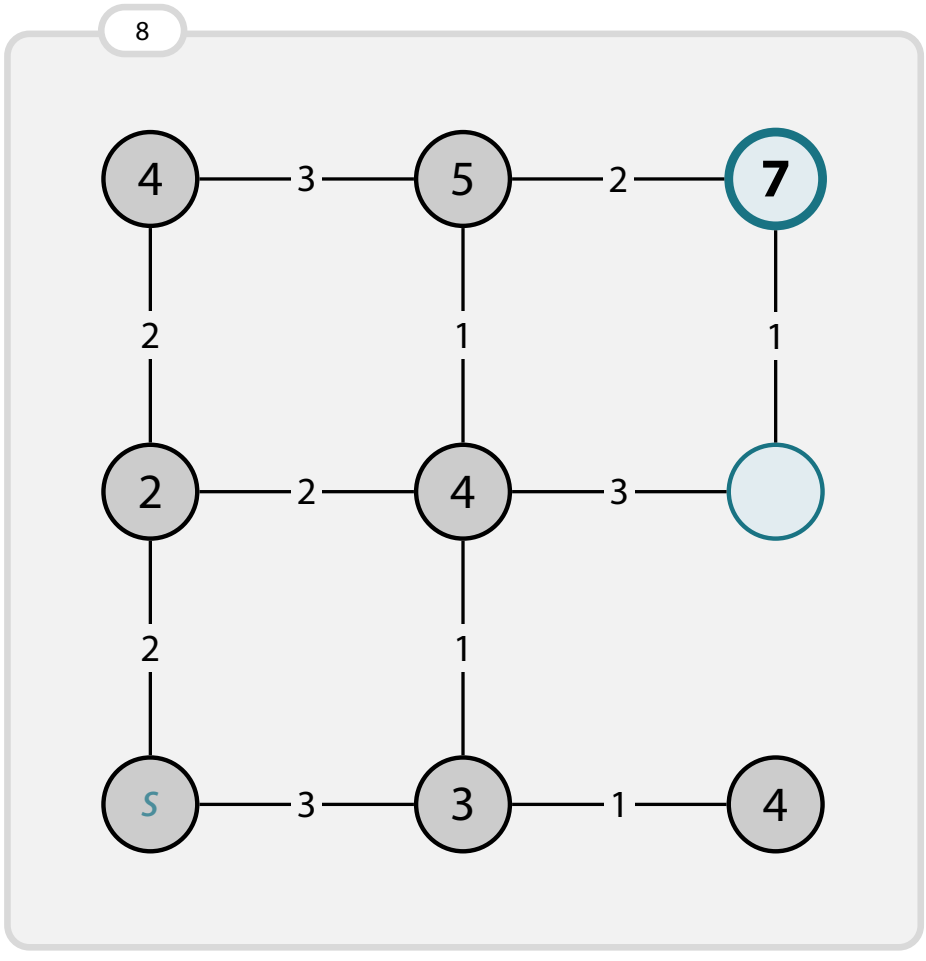
Der Dijkstra-Algorithmus anhand eines Beispiels.



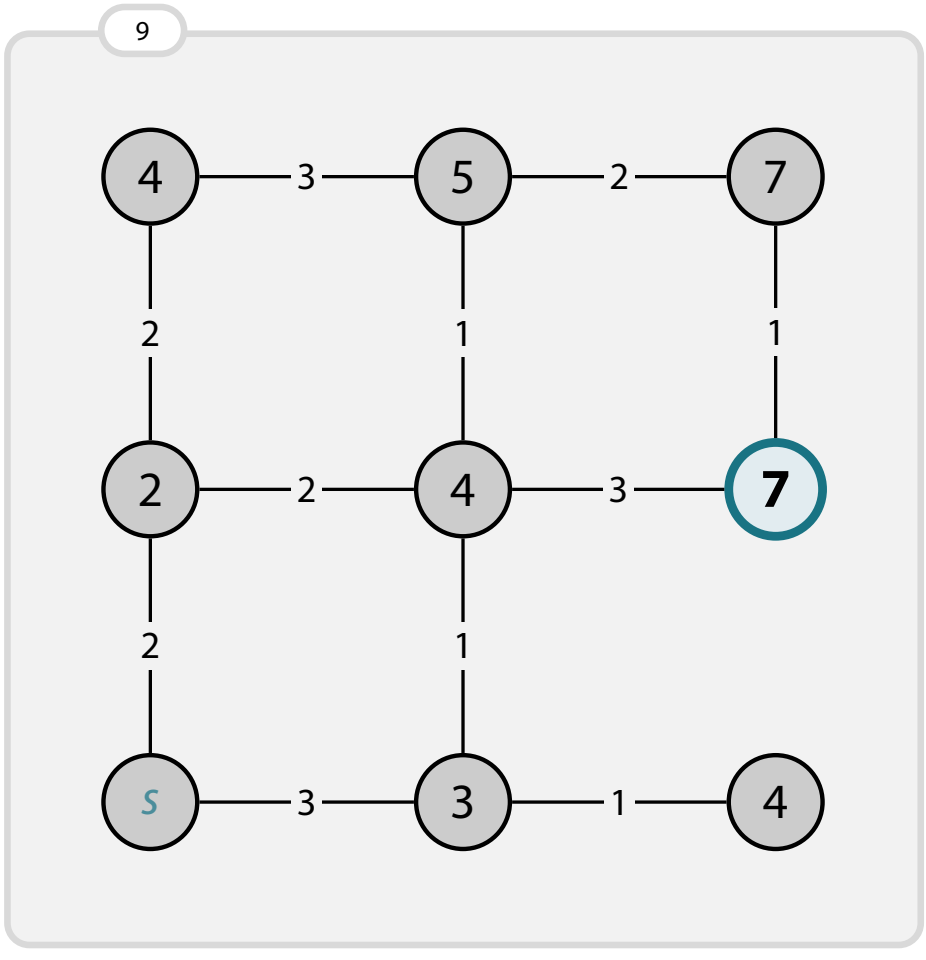
Der Dijkstra-Algorithmus anhand eines Beispiels.



Der Dijkstra-Algorithmus anhand eines Beispiels.



Der Dijkstra-Algorithmus anhand eines Beispiels.



Zur Laufzeit des Dijkstra-Algorithmus.

Sei n die Anzahl der Knoten und m die Anzahl der Kanten.

- Naive Implementation:
  Man muss n mal den nächsten Knoten finden, was jeweils n^2 lang dauert, insgesamt also $O(n^3)$.
- Nicht ganz so naive Implementation:
  Man merkt sich für jeden noch nicht besuchten Knoten den Abstand von s, wenn man ihn durch eine Kante zu den bereits besuchten verbinden würde. Dann kann man immer das Minimum dieser Liste nehmen.
  Man bekommt eine Laufzeit von $O(n^2)$ hin.
- Clevere Implementation:
  Man benutzt eine neue Datenstruktur, genannt Heap. Dies liefert eine Laufzeit von $O\big((n+m)\log n\big)$.
- Ganz, ganz clevere Implementation:
  Man benutzt Fibbonacci-Heaps (eine ziemlich komplexe Datenstruktur). Dies liefert eine Laufzeit von $O(n\log n + m)$.

Das Längste-Wege-Problem

Beispiel (Längste-Wege-Problem)

Eingabe Ein gerichteter oder ungerichteter Graph. Zusätzlich zwei Knoten s (source) und t (target).

Ausgabe Ein längster Weg von s nach t (die Summe der Gewichte soll maximal sein).

Dieses Problem muss gelöst werden

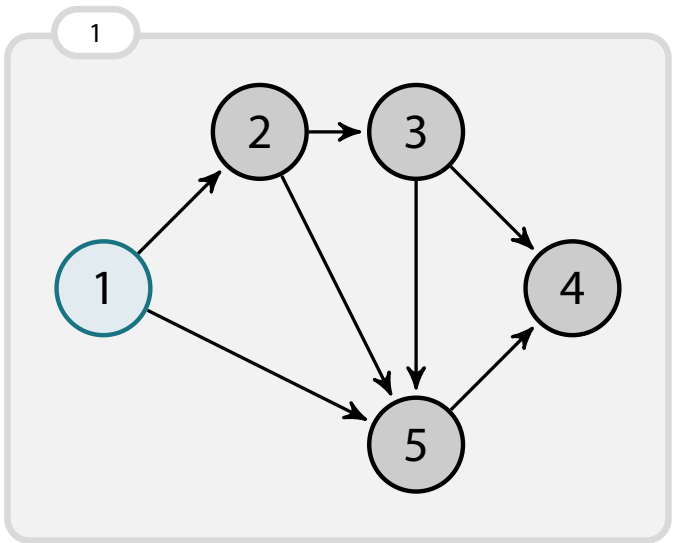
- bei der Berechnung *kritischer Pfade*,
- als Vorstufe des allgemeinen Handlungsreisendenproblems.

Lösung durch Backtracking

Algorithmus

Wiederhole Folgendes, bis es nicht mehr geht:

1. Erweitere den Pfad beliebig, bis es nicht mehr geht.
2. Hat der gefundene Pfad eine gewünscht *Mindestlänge*, höre auf.
3. Sonst mache so viele der letzten Schritte rückgängig, bis der Pfad anders fortgesetzt werden kann.

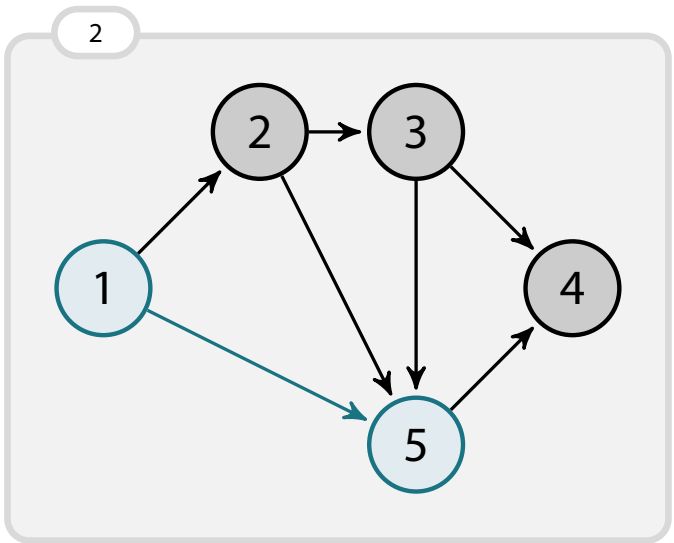


Lösung durch Backtracking

Algorithmus

Wiederhole Folgendes, bis es nicht mehr geht:

1. Erweitere den Pfad beliebig, bis es nicht mehr geht.
2. Hat der gefundene Pfad eine gewünscht *Mindestlänge*, höre auf.
3. Sonst mache so viele der letzten Schritte rückgängig, bis der Pfad anders fortgesetzt werden kann.

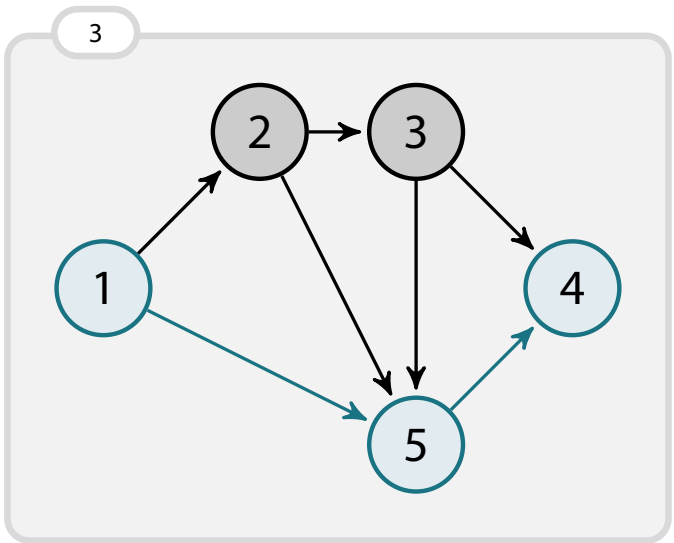


Lösung durch Backtracking

Algorithmus

Wiederhole Folgendes, bis es nicht mehr geht:

1. Erweitere den Pfad beliebig, bis es nicht mehr geht.
2. Hat der gefundene Pfad eine gewünscht *Mindestlänge*, höre auf.
3. Sonst mache so viele der letzten Schritte rückgängig, bis der Pfad anders fortgesetzt werden kann.

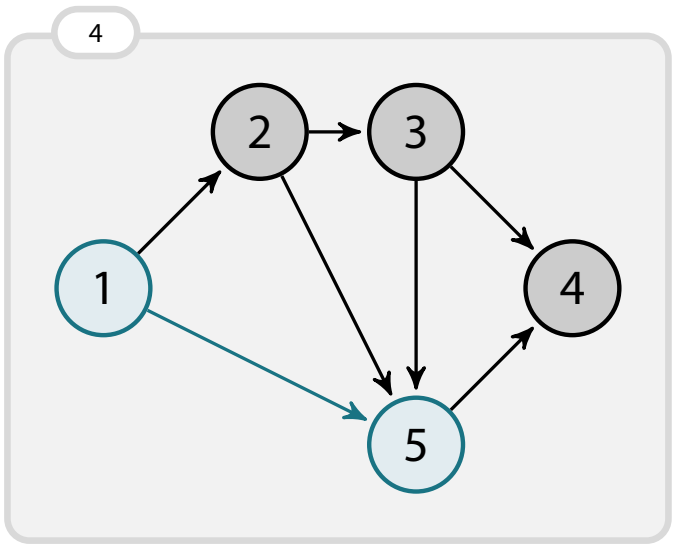


Lösung durch Backtracking

Algorithmus

Wiederhole Folgendes, bis es nicht mehr geht:

1. Erweitere den Pfad beliebig, bis es nicht mehr geht.
2. Hat der gefundene Pfad eine gewünscht *Mindestlänge*, höre auf.
3. Sonst mache so viele der letzten Schritte rückgängig, bis der Pfad anders fortgesetzt werden kann.

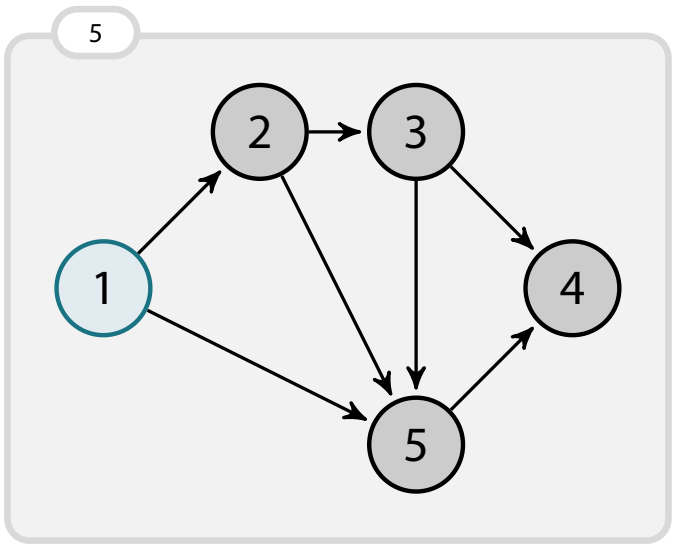


Lösung durch Backtracking

Algorithmus

Wiederhole Folgendes, bis es nicht mehr geht:

1. Erweitere den Pfad beliebig, bis es nicht mehr geht.
2. Hat der gefundene Pfad eine gewünscht *Mindestlänge*, höre auf.
3. Sonst mache so viele der letzten Schritte rückgängig, bis der Pfad anders fortgesetzt werden kann.

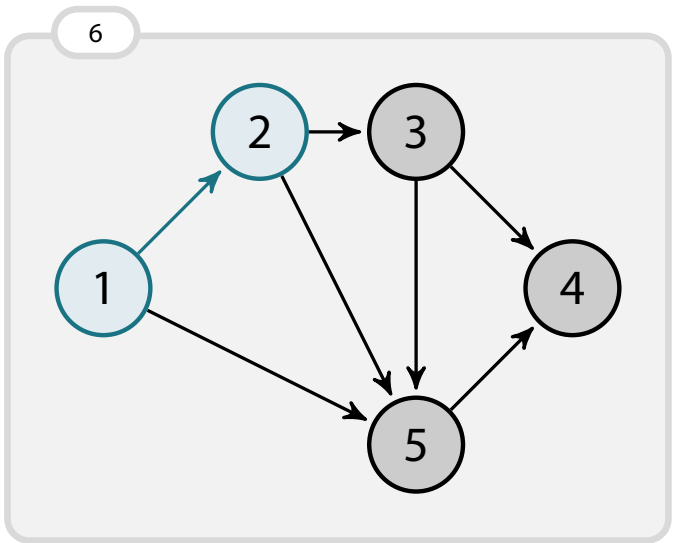


Lösung durch Backtracking

Algorithmus

Wiederhole Folgendes, bis es nicht mehr geht:

1. Erweitere den Pfad beliebig, bis es nicht mehr geht.
2. Hat der gefundene Pfad eine gewünscht *Mindestlänge*, höre auf.
3. Sonst mache so viele der letzten Schritte rückgängig, bis der Pfad anders fortgesetzt werden kann.

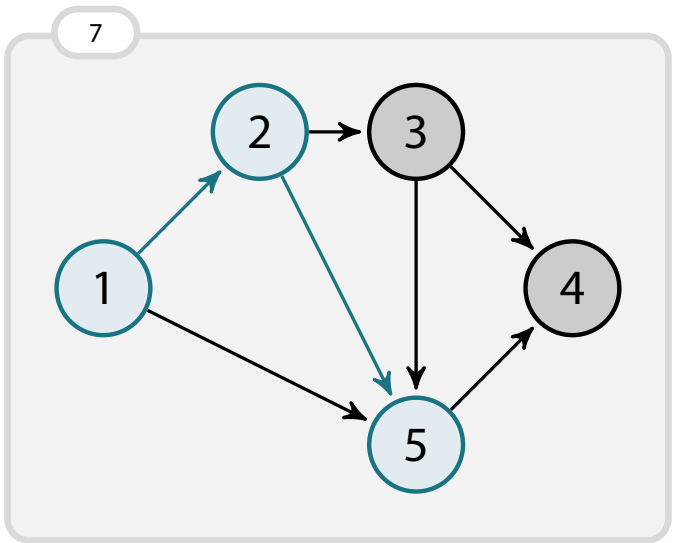


Lösung durch Backtracking

Algorithmus

Wiederhole Folgendes, bis es nicht mehr geht:

1. Erweitere den Pfad beliebig, bis es nicht mehr geht.
2. Hat der gefundene Pfad eine gewünscht *Mindestlänge*, höre auf.
3. Sonst mache so viele der letzten Schritte rückgängig, bis der Pfad anders fortgesetzt werden kann.

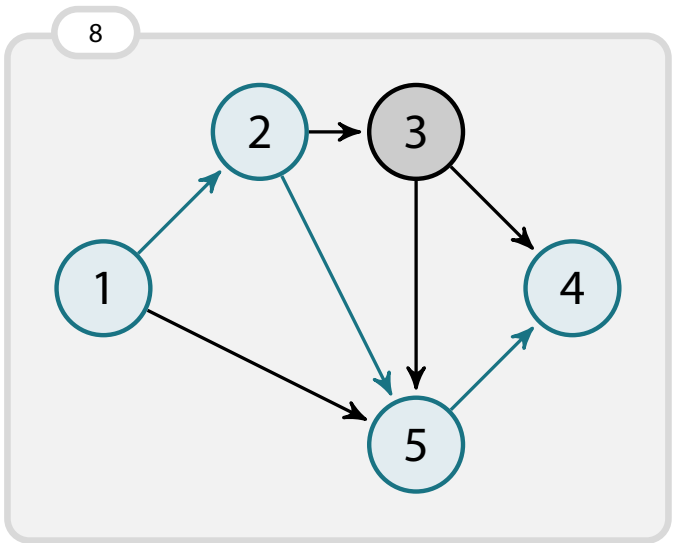


Lösung durch Backtracking

Algorithmus

Wiederhole Folgendes, bis es nicht mehr geht:

1. Erweitere den Pfad beliebig, bis es nicht mehr geht.
2. Hat der gefundene Pfad eine gewünscht *Mindestlänge*, höre auf.
3. Sonst mache so viele der letzten Schritte rückgängig, bis der Pfad anders fortgesetzt werden kann.

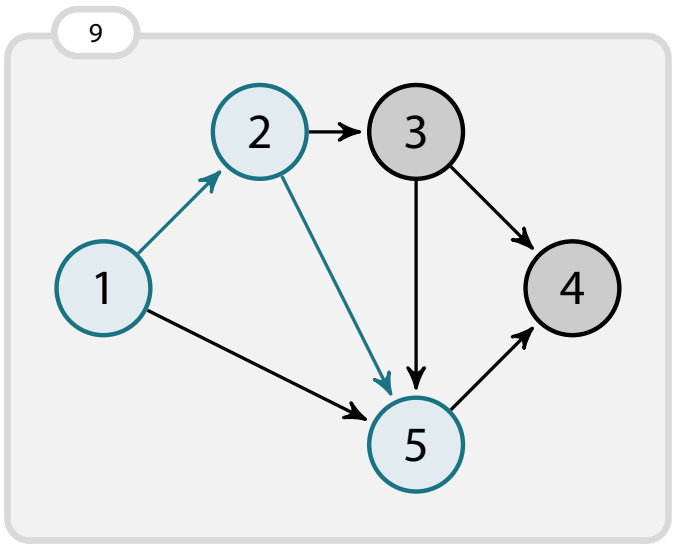


Lösung durch Backtracking

Algorithmus

Wiederhole Folgendes, bis es nicht mehr geht:

1. Erweitere den Pfad beliebig, bis es nicht mehr geht.
2. Hat der gefundene Pfad eine gewünscht *Mindestlänge*, höre auf.
3. Sonst mache so viele der letzten Schritte rückgängig, bis der Pfad anders fortgesetzt werden kann.

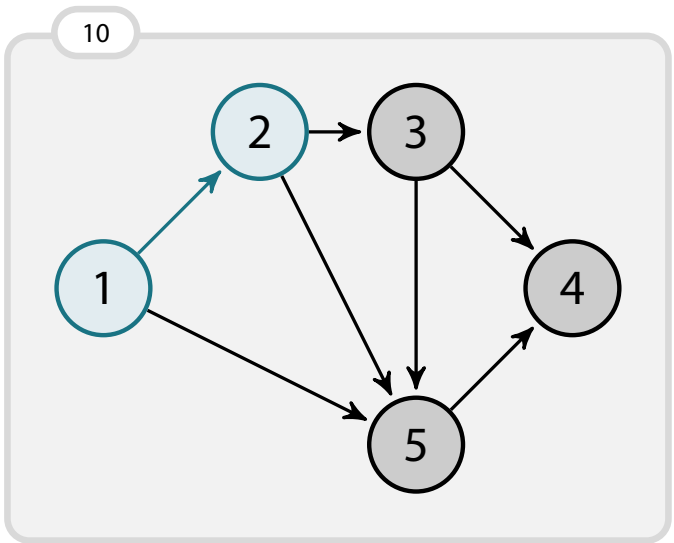


Lösung durch Backtracking

Algorithmus

Wiederhole Folgendes, bis es nicht mehr geht:

1. Erweitere den Pfad beliebig, bis es nicht mehr geht.
2. Hat der gefundene Pfad eine gewünscht *Mindestlänge*, höre auf.
3. Sonst mache so viele der letzten Schritte rückgängig, bis der Pfad anders fortgesetzt werden kann.

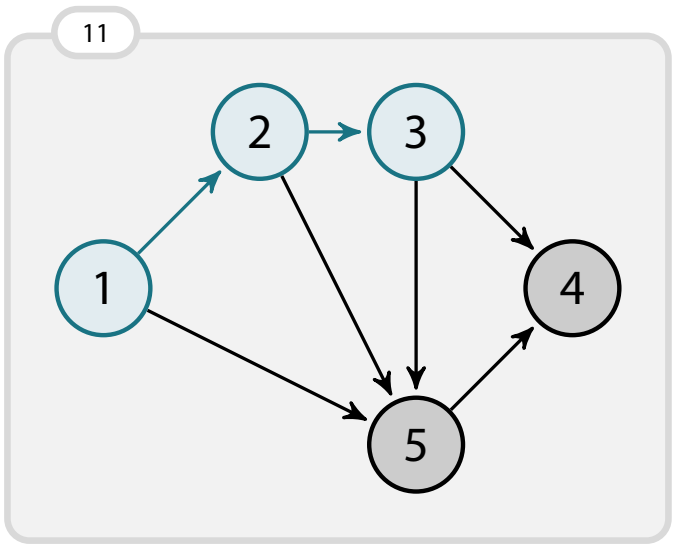


Lösung durch Backtracking

Algorithmus

Wiederhole Folgendes, bis es nicht mehr geht:

1. Erweitere den Pfad beliebig, bis es nicht mehr geht.
2. Hat der gefundene Pfad eine gewünscht *Mindestlänge*, höre auf.
3. Sonst mache so viele der letzten Schritte rückgängig, bis der Pfad anders fortgesetzt werden kann.

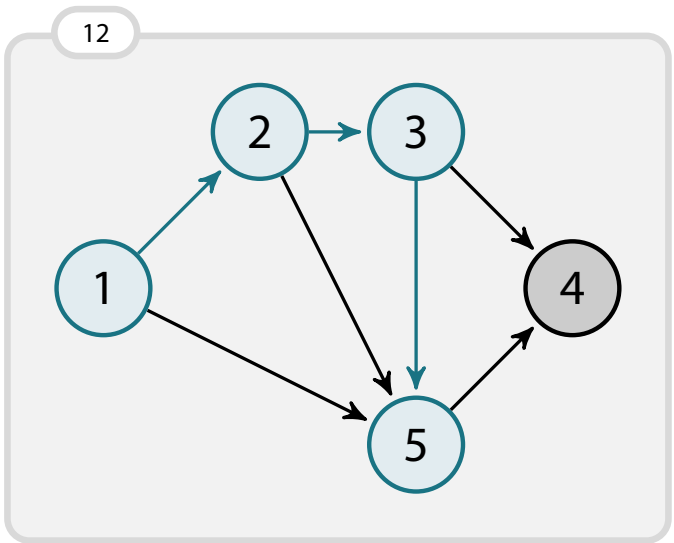


Lösung durch Backtracking

Algorithmus

Wiederhole Folgendes, bis es nicht mehr geht:

1. Erweitere den Pfad beliebig, bis es nicht mehr geht.
2. Hat der gefundene Pfad eine gewünscht *Mindestlänge*, höre auf.
3. Sonst mache so viele der letzten Schritte rückgängig, bis der Pfad anders fortgesetzt werden kann.

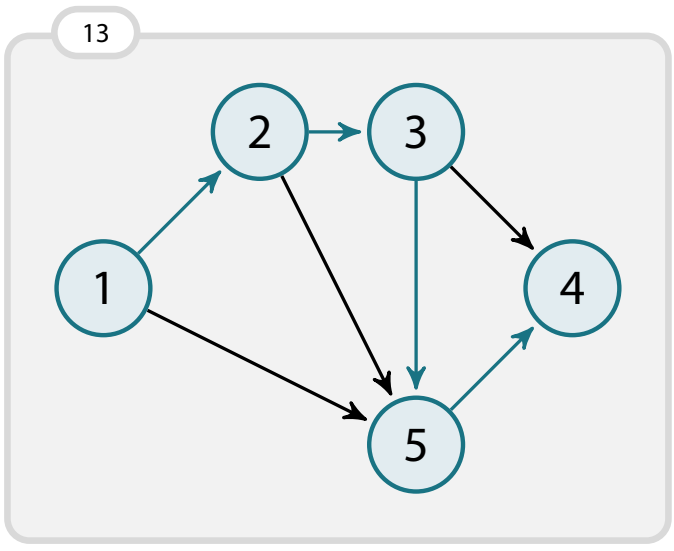


Lösung durch Backtracking

Algorithmus

Wiederhole Folgendes, bis es nicht mehr geht:

1. Erweitere den Pfad beliebig, bis es nicht mehr geht.
2. Hat der gefundene Pfad eine gewünscht *Mindestlänge*, höre auf.
3. Sonst mache so viele der letzten Schritte rückgängig, bis der Pfad anders fortgesetzt werden kann.



Zusammenfassung

Wege-Finden-Probleme

- Bei *Wege-Finden-Probleme* hat man einen Graphen und zwei Knoten gegeben.
- Das Ziel ist dann, einen möglichst kurzen oder möglichst langen Weg zwischen den Knoten zu finden.
- Bei gewichteten Graphen ist die »Länge« des Weges die Summe der Kantengewichte.

Erreichbarkeit

Beim *Erreichbarkeitsproblem* überprüft man, ob *überhaupt ein Weg zwischen zwei Knoten existiert*. Man löst es mittels

- Breitensuchen: besuche alle Knoten in aufsteigender Reihenfolge ihrer Entfernung; oder
- Tiefensuche: besuche immer möglichst schnell Knoten »in der Tiefe« des Graphen.

Beide Verfahren werden sehr ähnlich implementiert.

Zusammenfassung

Dijkstra-Algorithmus zum Finden kürzester Wege

Speichere für jeden schon besuchten Knoten k die »Entfernung« $d(s,k)$ von s bis k. Solange noch nicht alle Knoten besucht sind, wiederhole:

1. Finde die Kante e, die einen schon besuchten Knoten k mit einem noch nicht besuchten v verbindet, so dass $d(s,k)+w(e)$ minimal ist, wobei $w(e)$ das Gewicht der Kante ist.
2. Markiere u als besucht und speichere dort $d(s,k)+w(e)$.

Längste-Wege-Finden

Längste Wege finden ist *schwierig* und muss mittels *Backtracking* gelöst werden.