Übersicht

Topologisches Sortieren

Ein reflexiv-transitive Hüllen eines DAG entspricht einer Halbordnung.

Definition

Es sei M eine Menge. Wir sagen $\leq \subseteq M \times M$ ist eine Halbordnung auf M, wenn

1. Für alle $x \in M$ gilt $x \leq x$
2. Für alle $x, y \in M$ gilt $x \leq y \wedge y \leq x \Rightarrow x = y$
3. Für alle $x, y, z \in M$ gilt $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Topologisches Sortieren:

Bette eine Halbordnung in eine Ordnung ein.

Ordne die Knoten eines DAG so, daß keine Kante von einem größeren zu einem kleineren Knoten führt.

Topologisches Sortieren – Anwendungen

Topologisches Sortieren kann viele Probleme lösen.

- In welcher Reihenfolge sollen Vorlesungen gehört werden?
- Wie baut man ein kompliziertes Gerät?
- Entwurf von Klassenhierarchien.
- Task scheduling.
- Tabellenkalkulation.
- Optimierung beim Compilieren.
- ...

Frage: Kann jede Halbordnung topologisch sortiert werden?

Topologisches Sortieren

Theorem

Jede endliche Halbordnung kann in eine (totale) Ordnung eingebettet werden.

Beweis.

Sei $\leq \subseteq M \times M$ eine endliche Halbordnung auf M.

Da M endlich ist, dann gibt es ein $x \in M$, so daß es kein $y \in M$ mit $y \neq x$ und $y \leq x$ gibt.

Wir können die Ordnung mit x als kleinstem Element beginnen und dann eine Ordnung von $M \setminus \{x\}$ anfügen.

Durch Induktion sieht man, daß dies eine (totale) Ordnung auf M ist. $\square$

Frage: Was ist mit unendlichen Mengen?

Topologisches Sortieren

Theorem

Jede endliche Halbordnung kann in eine (totale) Ordnung eingebettet werden.

Beweis.

Sei $\leq \subseteq M \times M$ eine endliche Halbordnung auf M.
Da M endlich ist, dann gibt es ein $x \in M$, so daß es kein $y \in M$ mit $y \neq x$ und $y \leq x$ gibt.
Wir können die Ordnung mit x als kleinstem Element beginnen und dann eine Ordnung von $M \setminus \{x\}$ anfügen.
Durch Induktion sieht man, daß dies eine (totale) Ordnung auf M ist. $\square$

Frage: Was ist mit unendlichen Mengen?

Topologisches Sortieren

Theorem

G sei ein DAG. Wenn wir die Knoten von G nach den finish times einer DFS umgekehrt anordnen, sind sie topologisch sortiert.

Beweis.

Es seien u und v Knoten von G mit $f(u) < f(v)$
Daher kommt u nach v in der konstruierten Ordnung.

Wir zeigen, daß es keine Kante von u nach v gibt:

Falls $d(v) < d(u)$ müßte es eine Rückwärtskante sein, die es nicht gibt (DAG).

Also $d(v) > f(u)$ und v blieb weiß bis zur finish time von u.

Nur möglich, wenn keine Kante von u nach v. □

Topologisches Sortieren

Theorem

G sei ein DAG. Wenn wir die Knoten von G nach den finish times einer DFS umgekehrt anordnen, sind sie topologisch sortiert.

Beweis.

Es seien u und v Knoten von G mit $f(u) < f(v)$
Daher kommt u nach v in der konstruierten Ordnung.

Wir zeigen, daß es keine Kante von u nach v gibt:

Falls $d(v) < d(u)$ müßte es eine Rückwärtskante sein, die es nicht gibt (DAG).

Also $d(v) > f(u)$ und v blieb weiß bis zur finish time von u.

Nur möglich, wenn keine Kante von u nach v. □

Topologisches Sortieren

Theorem

G sei ein DAG. Wenn wir die Knoten von G nach den finish times einer DFS umgekehrt anordnen, sind sie topologisch sortiert.

Beweis.

Es seien u und v Knoten von G mit $f(u) < f(v)$
Daher kommt u nach v in der konstruierten Ordnung.

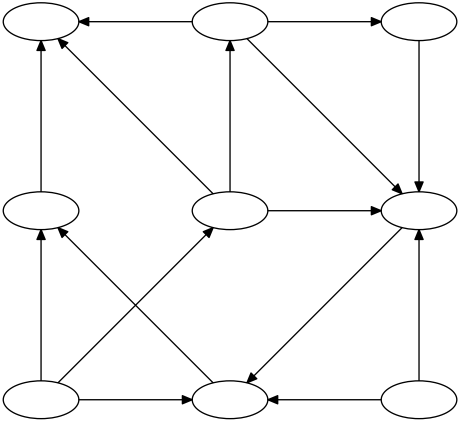
Wir zeigen, daß es keine Kante von u nach v gibt:

Falls $d(v) < d(u)$ müßte es eine Rückwärtskante sein, die es nicht gibt (DAG).

Also $d(v) > f(u)$ und v blieb weiß bis zur finish time von u.

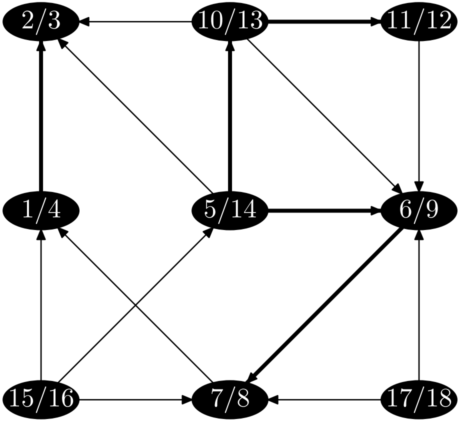
Nur möglich, wenn keine Kante von u nach v. □

Topologisches Sortieren – Beispiel



Topologisches Sortieren in $O(|V| + |E|)$ Schritten.

Topologisches Sortieren – Beispiel



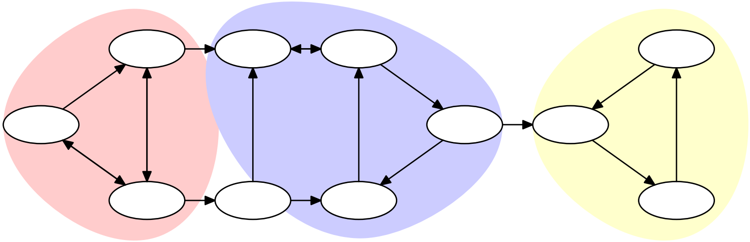
Topologisches Sortieren in $O(|V| + |E|)$ Schritten.

Übersicht

s-t Connectivity

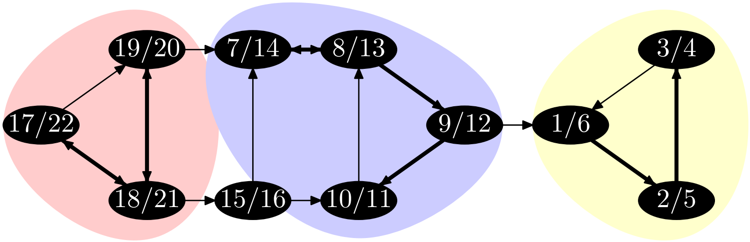
Gegeben: Knoten s und t in gerichtetem Graph

Frage: Ist t von s erreichbar (durch einen gerichteten Pfad)?



s-t Connectivity

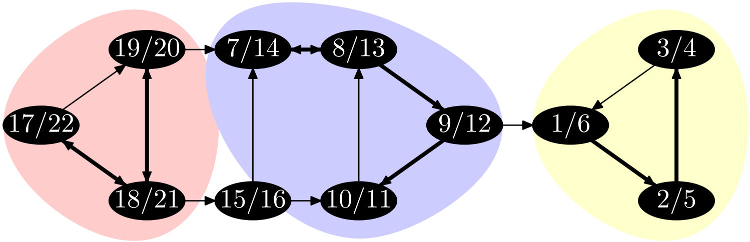
Führe eine DFS aus, starte bei s.



s-t Connectivity

Führe eine DFS aus, starte bei s.

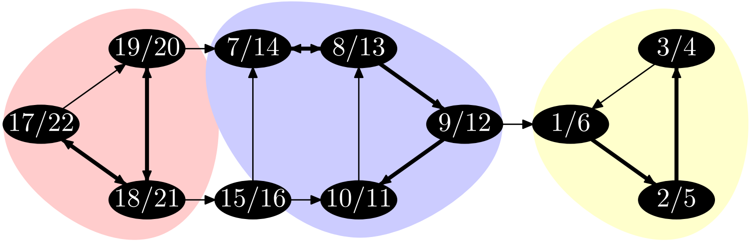
Pfad von s nach t $\iff$ $f(t) < f(s)$.



s-t Connectivity

Führe eine DFS aus, starte bei s.

Pfad von s nach t $\iff$ $f(t) < f(s)$.

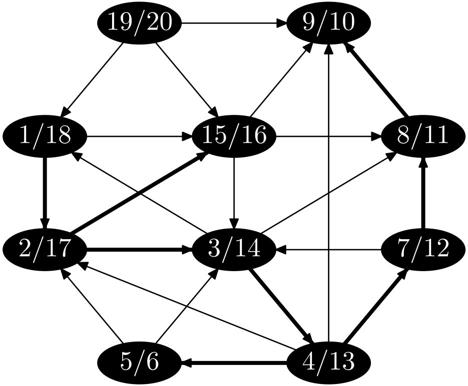


Beweis: Wenn s schwarz wird, sind alle von s erreichbaren Knoten schwarz.

s-t Connectivity

Theorem

Gegeben sei ein gerichteter Graph $G = (V, E)$ und $s \in V$.
Wir können in linearer Zeit alle von s erreichbaren Knoten finden.



Single Source Shortest Paths

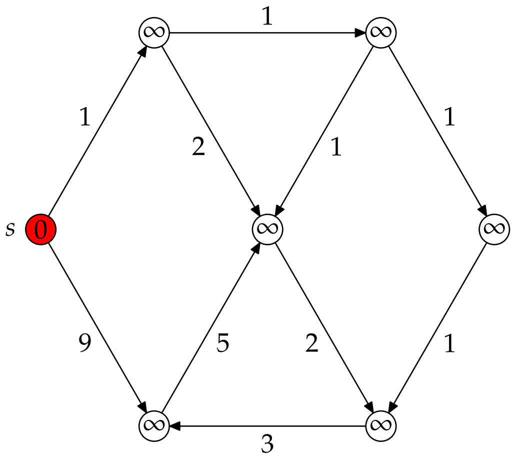
Gegeben ein gerichteter Graph $G = (V, E)$ mit nicht-negativen Kantengewichten $length : E \to \mathbf{Q}$ und ein Knoten $s \in V$, finde die kürzesten Wege von s zu allen Knoten.

- Wir lösen das Problem mit dynamischer Programmierung.
- Menge F von Knoten, deren Abstand bekannt ist.
- Anfangs ist $F = \{s\}$.
- F wird in jeder Iteration größer.
- Invariante: Kein Knoten $v \notin F$ hat kleineren Abstand zu s als jeder Knoten in F.

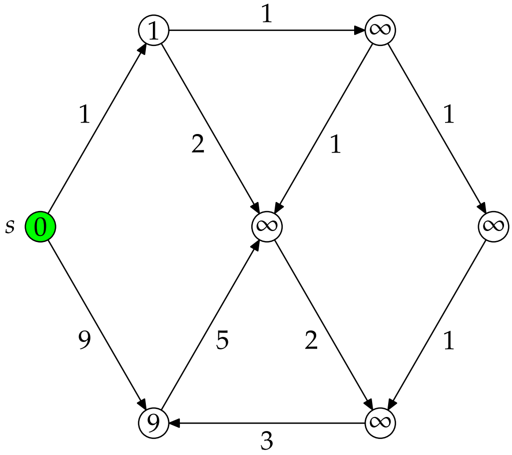
Single Source Shortest Paths

Gegeben ein gerichteter Graph $G = (V, E)$ mit nicht-negativen Kantengewichten $\mathit{length} : E \to \mathbf{Q}$ und ein Knoten $s \in V$, finde die kürzesten Wege von s zu allen Knoten.

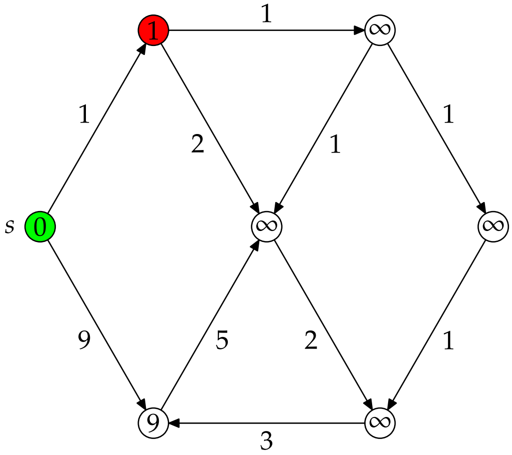
- Wir lösen das Problem mit dynamischer Programmierung.
- Menge F von Knoten, deren Abstand bekannt ist.
- Anfangs ist $F = \{s\}$.
- F wird in jeder Iteration größer.
- Invariante: Kein Knoten $v \notin F$ hat kleineren Abstand zu s als jeder Knoten in F.



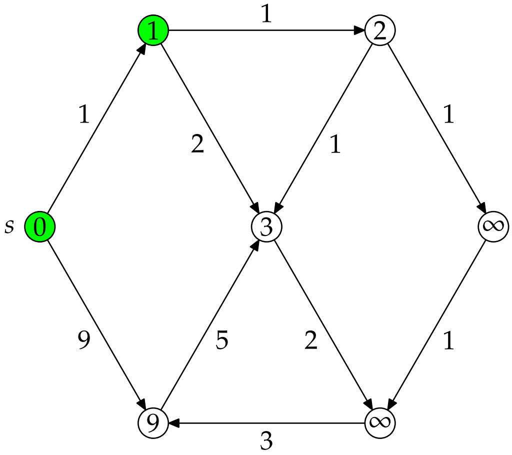
- Grüne Knoten: F
- Roter Knoten aktiv, relaxiert seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



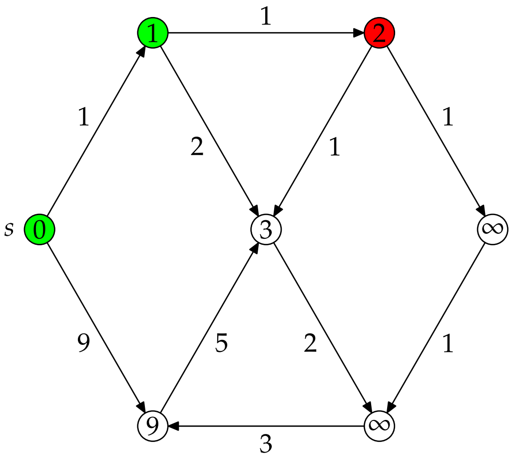
- Grüne Knoten: F
- Roter Knoten aktiv, relaxiert seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



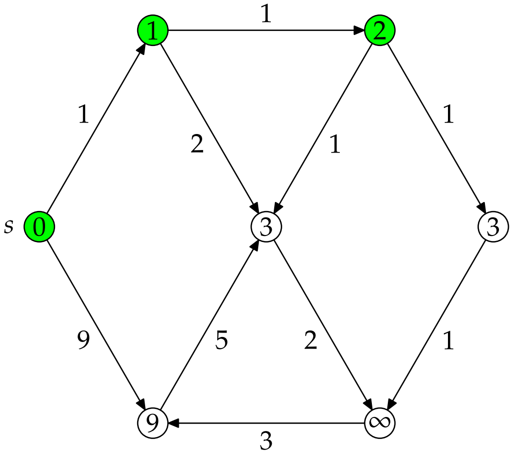
- Grüne Knoten: F
- Roter Knoten aktiv, relaxiert seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



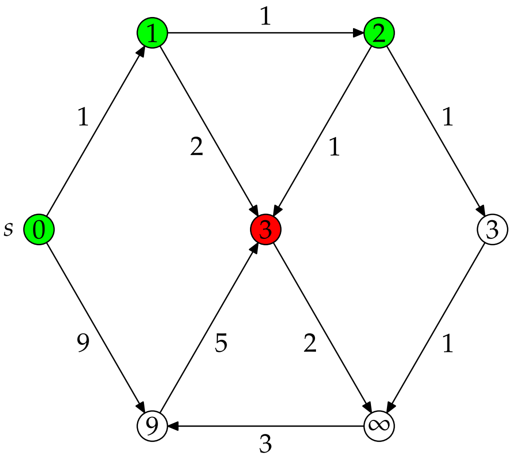
- Grüne Knoten: F
- Roter Knoten aktiv, relaxiert seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



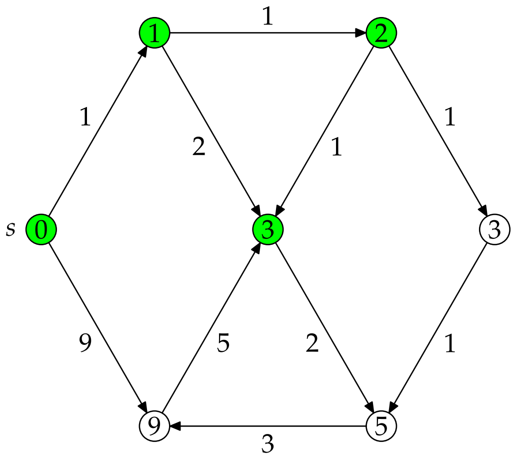
- Grüne Knoten: F
- Roter Knoten aktiv, relaxiert seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



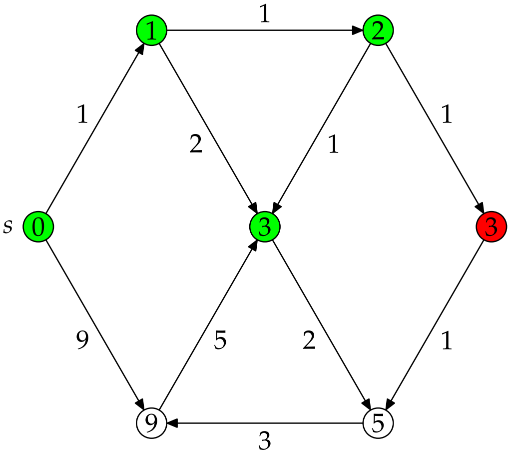
- Grüne Knoten: F
- Roter Knoten aktiv, relaxiert seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



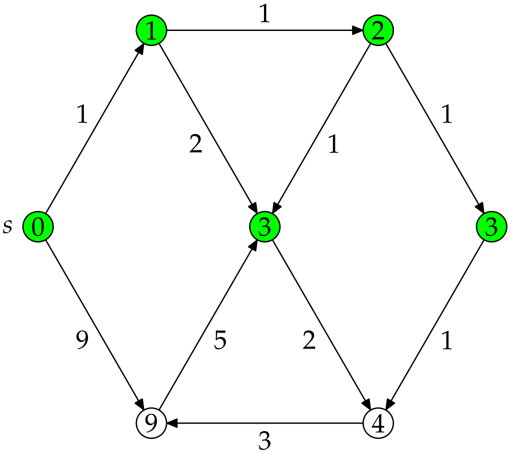
- Grüne Knoten: F
- Roter Knoten aktiv, relaxiert seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



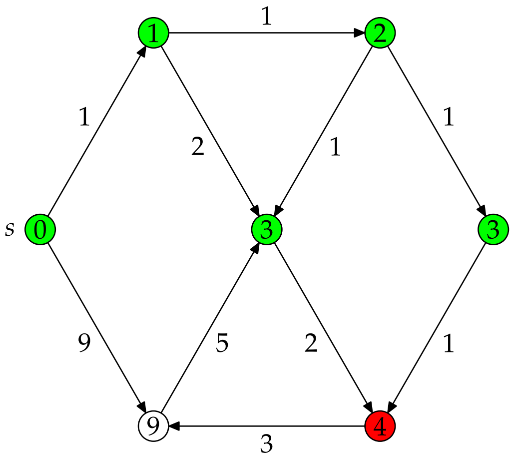
- Grüne Knoten: F
- Roter Knoten aktiv, relaxiert seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



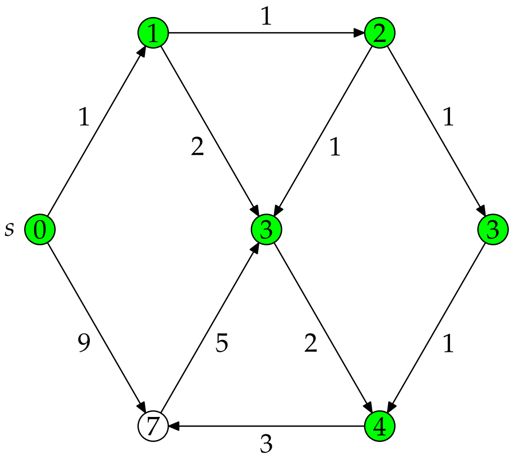
- Grüne Knoten: F
- Roter Knoten aktiv, relaxiert seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



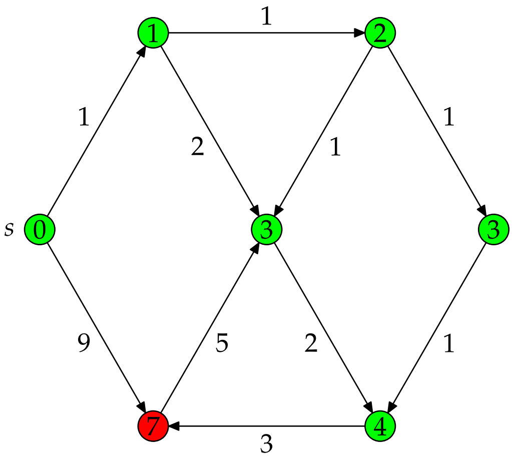
- Grüne Knoten: F
- Roter Knoten aktiv, relaxiert seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



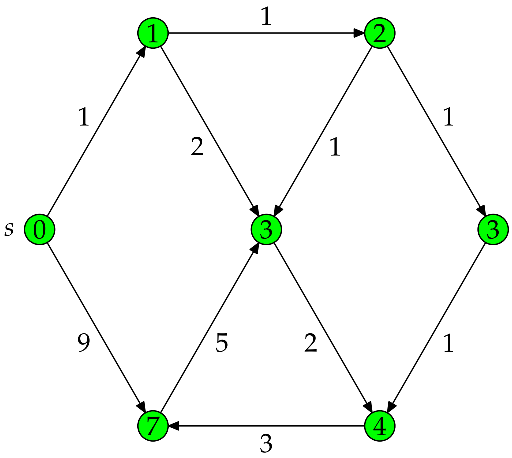
- Grüne Knoten: F
- Roter Knoten aktiv, relaxiert seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



- Grüne Knoten: F
- Roter Knoten aktiv, relaxiert seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



- Grüne Knoten: F
- Roter Knoten aktiv, relaxiert seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.



- Grüne Knoten: F
- Roter Knoten aktiv, relaxiert seine Nachbarn
- Weiße Knoten enthalten Abstand zu s über grüne Knoten.

Korrektheit

Lemma

- *Jeder grüne Knoten enthält den Abstand von s.*
- *Jeder weiße Knoten enthält Abstand von s über grüne Knoten.*

Beweis.

Sei v einer weißer Knoten, dessen Beschriftung minimal ist.
Betrachte einen kürzesten Pfad von s nach v und den ersten weißen Knoten u auf diesem Pfad.
Es gilt $u = v$, da der Abstand von s zu u mindestens so groß ist wie der Abstand von s zu v.
Wenn ein Knoten grün wird, garantiert die Relaxation, daß die zweite Bedingung weiter gilt. □

Korrektheit

Lemma

- *Jeder grüne Knoten enthält den Abstand von s.*
- *Jeder weiße Knoten enthält Abstand von s über grüne Knoten.*

Beweis.

Sei v einer weißer Knoten, dessen Beschriftung minimal ist.
Betrachte einen kürzesten Pfad von s nach v und den ersten weißen Knoten u auf diesem Pfad.
Es gilt $u = v$, da der Abstand von s zu u mindestens so groß ist wie der Abstand von s zu v.
Wenn ein Knoten grün wird, garantiert die Relaxation, daß die zweite Bedingung weiter gilt. $\square$

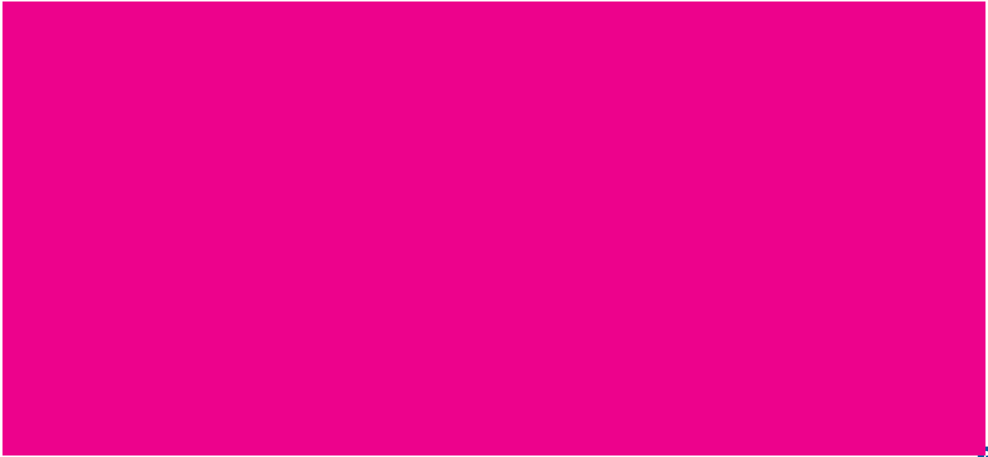
Der Algorithmus von Dijkstra

Algorithmus

procedure Dijkstra(s) :
$Q := V - \{ s \}$;
for v in Q **do** $d[v] :=$ infty **od**;
$d[s] := 0$;
while $Q \neq$ emptyset **do**
  choose v in Q with minimal d[v];
  $Q := Q - \{ v \}$;
  forall u adjacent **to** v **do**
    $d[u] := \min \{ d[u], d[v] + \text{length}(v, u) \}$
  od
od

Wie implementieren wir Q?

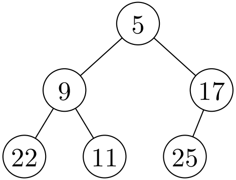
Der Algorithmus von Dijkstra – Beispiel



Priority Queues (Prioritätswarteschlangen)

Operationen einer Prioritätswarteschlange Q:

1. Einfügen von x mit Gewicht w (insert)
2. Finden und Entfernen eines Elements mit minimalem Gewicht (extract-min)
3. Das Gewicht eines Elements x auf w verringern (decrease-weight)

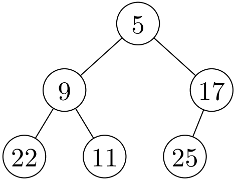


Heap: alle Operationen in $O(\log n)$ Schritten
(n ist die aktuelle Anzahl von Elementen im Heap)

Priority Queues (Prioritätswarteschlangen)

Operationen einer Prioritätswarteschlange Q:

1. Einfügen von x mit Gewicht w (insert)
2. Finden und Entfernen eines Elements mit minimalem Gewicht (extract-min)
3. Das Gewicht eines Elements x auf w verringern (decrease-weight)



Heap: alle Operationen in $O(\log n)$ Schritten
(n ist die aktuelle Anzahl von Elementen im Heap)

Algorithmus von Dijkstra – Laufzeit

Theorem

Der Algorithmus von Dijkstra berechnet die Abstände von s zu allen anderen Knoten in $O((|V| + |E|) \log |V|)$ *Schritten.*

Beweis.

Es werden $|V|$ Einfügeoperationen, $|V|$ extract-mins und $|E|$ decrease-keys ausgeführt. Verwenden wir einen Heap für die Prioritätswarteschlange, ergibt sich die verlangte Laufzeit. □

Ein Fibonacci-Heap benötigt für ein Einfügen und extract-min $O(\log n)$ und für decrease-key nur $O(1)$ amortisierte Zeit.

Dijkstra: $O(|V| \log |V| + |E|)$

Algorithmus von Dijkstra – Laufzeit

Theorem

Der Algorithmus von Dijkstra berechnet die Abstände von s zu allen anderen Knoten in $O((|V| + |E|) \log |V|)$ *Schritten.*

Beweis.

Es werden $|V|$ Einfügeoperationen, $|V|$ extract-mins und $|E|$ decrease-keys ausgeführt. Verwenden wir einen Heap für die Prioritätswarteschlange, ergibt sich die verlangte Laufzeit. $\square$

Ein Fibonacci-Heap benötigt für ein Einfügen und extract-min $O(\log n)$ und für decrease-key nur $O(1)$ amortisierte Zeit.

Dijkstra: $O(|V| \log |V| + |E|)$

Algorithmus von Dijkstra – Laufzeit

Theorem

Der Algorithmus von Dijkstra berechnet die Abstände von s zu allen anderen Knoten in $O((|V| + |E|) \log |V|)$ *Schritten.*

Beweis.

Es werden $|V|$ Einfügeoperationen, $|V|$ extract-mins und $|E|$ decrease-keys ausgeführt. Verwenden wir einen Heap für die Prioritätswarteschlange, ergibt sich die verlangte Laufzeit. $\square$

Ein Fibonacci-Heap benötigt für ein Einfügen und extract-min $O(\log n)$ und für decrease-key nur $O(1)$ amortisierte Zeit.

Dijkstra: $O(|V| \log |V| + |E|)$

Java

```
public static⟨V extends Comparable⟨V⟩⟩ Map⟨V, Double⟩
dijkstra(Graph⟨V⟩ G, V s, Map⟨Edge⟨V⟩, Double⟩ length, Map⟨V, V⟩ pred) {
  Map⟨V, Double⟩ dist = new HashMap⟨V, Double⟩();
  PriorityQueue⟨V, Double⟩ queue = new SplayPriorityQueue⟨V, Double⟩();
  for(V u : G.allNodes()) dist.put(u, Double.MAX_VALUE);
  dist.put(s, 0.0); queue.insert(s, 0.0);
  while(!queue.isEmpty()) {
    V u = queue.extractMin();
    for(V v : G.neighbors(u)) {
      Double l = length.get(G.edge(u, v));
      if(l == null) continue;
      double d = dist.get(u) + l;
      if(d < dist.get(v)) {
        queue.decreaseKey(v, d); dist.put(v, d); if(pred ≠ null) pred.put(v, u); }
    }
  }
  return dist;
}
```