

Graphen- minimale Spannbäume und kürzeste Wege

Karsten Klein
Lehrstuhl XI Algorithm Engineering

21. Vorlesung

22.06.06

Überblick

- MST: Algorithmus von **Prim**
- Kürzeste Wege in Graphen: Algorithmus von **Dijkstra**

Beide nutzen ähnliches Konzept:

- **Greedy**
- Priority Queue
- Lassen einzelnen Baum wachsen

Greedy-Verfahren

(Greedy = gierig, gefräßig)

Greedy-Paradigma

Greedy-Verfahren treffen **lokale** Entscheidungen
Sie wählen in jedem Schritt die aktuell günstigste
Auswahl ohne Vorausschau



Greedy-Paradigma

Greedy-Verfahren treffen **lokale** Entscheidungen
Sie wählen in jedem Schritt die aktuell günstigste
Auswahl ohne Vorausschau



*Greedy-Algorithmen sind in der Regel nicht optimal,
können aber in einigen wichtigen Anwendungen
dennoch gute oder gar optimale Lösungen erzeugen!*

Greedy Methode

- Beliebig schlecht: TSP (Traveling Salesman Problem)
- Gut: Bin Packing ($\leq 2x$ Optimum)
- Optimal: MST (Minimum Spanning Trees)

Nötig für Optimalität:

- **Optimale Substruktur** (Optimallösung besteht aus optimalen Teillösungen)
- **Greedy Choice** Eigenschaft (lokal optimale Entscheidungen können Optimum erzeugen), keine Abhängigkeit von anderen Teillösungen

Priority-Queue

Wdh: ADT Priority Queue

- Dyn. Verwaltung von Elementmenge mit *Prioritäten*
- Operationen:
 - INSERT(p, v) : Position
 - DELETE(pos)
 - MINIMUM(): Position
 - EXTRACTMIN() : $P \times V$
 - DECREASEPRIORITY(pos, p)
- Bekannt: Binary (Min)Heap in Array

Kap. 6.5 Minimale Spannbäume

Minimaler Spannbaum

- $G=(V,E)$ ein ungerichteter, zshgd. Graph. Ein **Untergraph** $T=(V,E_T)$ von G heißt **Spannbaum** von G , falls T ein **Baum** ist.
- **Gewicht** eines Spannbaums: Gewichtsfunktion $w : E \rightarrow \mathbb{R}_+^+$
 $w(T) := w(E_T) := \sum_{e \in E_T} w(e)$

Minimum Spanning Tree (MST)

Gesucht: ein Spannbaum T von G mit minimalem Gewicht $w(T)$

MST Eigenschaft

Aussichtsreiche Menge T von Kanten:
 \exists MST, der alle Kanten von T enthält.



MST Eigenschaft

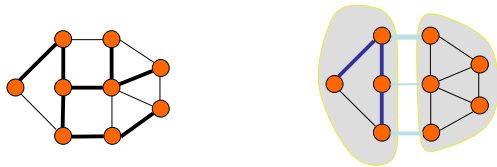
Lemma 1:

Sei $G=(V,E)$ zshgd. mit Gewichtsfunktion $w : E \rightarrow \mathbb{R}$. Seien:

- $S \subset V$
 - $T \subset E$ aussichtsreich und *keine* Kante aus T verlässt S
 - $e \in E$ Kante mit minimalem Gewicht, die S verlässt
- Dann ist $T \cup \{e\}$ aussichtsreich.

MST Eigenschaft

Kantenmenge T , aussichtsreich



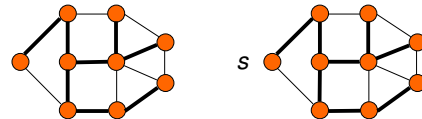
Knotenmenge S

MST sind nicht eindeutig!!

Algorithmus von Prim

Vgl. mit Kruskal:

- Kruskal läßt Wald wachsen mit Union-Find bis Spannbaum erzeugt
- Prim läßt von Startknoten s einen einzelnen Baum wachsen



Schema: Algorithmus von Prim

Wähle Startknoten $s \in V$

$S := \{s\}; T := \emptyset$ // S Unterbaumknoten, T Kanten

while $S \neq V$ **do**

 Wähle $e = (u, v)$, e verläßt S mit *min. Gewicht*

$T := T \cup \{e\}; S := S \cup \{v\}$

end while

Korrektheit Prim

Induktion über Kanten e_1, \dots, e_k , die zu T hinzugefügt werden:

Zeige: $T_i = \{e_1, \dots, e_i\}$ aussichtsreich für $0 \leq i \leq k$

$i = 0: T_0 = \emptyset \Rightarrow$ aussichtsreich

$1 \leq i \leq k: T_{i-1}$ aussichtsreich, S Knotenmenge vor Hinzunahme von e_i , keine Kante aus T_{i-1} verläßt S

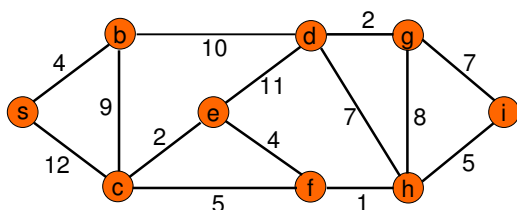
$\Rightarrow e_i$ leichteste S verlassende Kante

\Rightarrow *Mit Lemma 1: $T_i = T_{i-1} \cup \{e_i\}$ aussichtsreich*

Und: Keine Kante aus T_i verläßt neues S

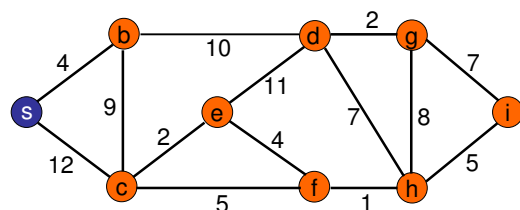
Am Ende hat T $|V|-1$ Kanten \Rightarrow MST

Algorithmus von Prim



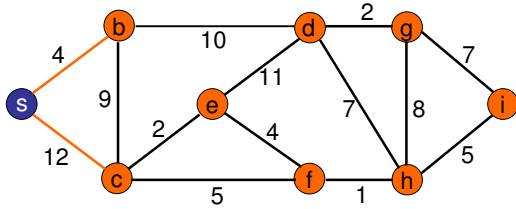
Knotenmenge S , Kantenmenge T :
In Teilbaum T_i

Algorithmus von Prim



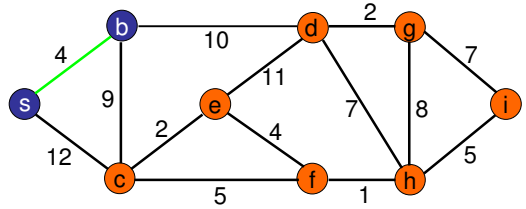
Knotenmenge S , Kantenmenge T :
In Teilbaum T_i

Algorithmus von Prim



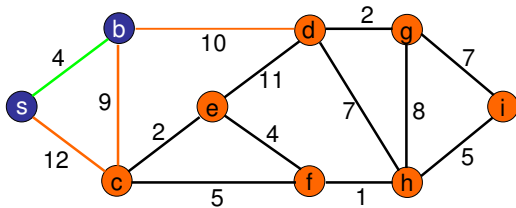
Knotenmenge S, Kantenmenge T:
In Teilbaum T_i

Algorithmus von Prim



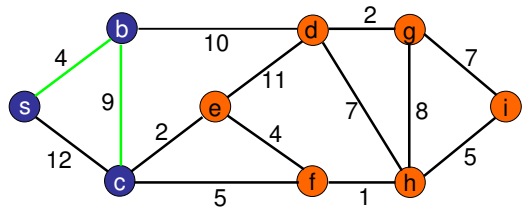
Knotenmenge S, Kantenmenge T:
In Teilbaum T_i

Algorithmus von Prim



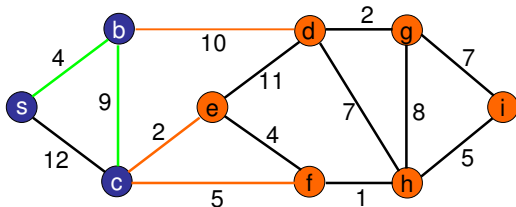
Knotenmenge S, Kantenmenge T:
In Teilbaum T_i

Algorithmus von Prim



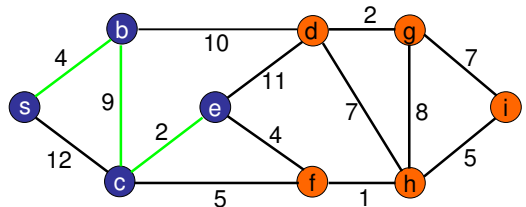
Knotenmenge S, Kantenmenge T:
In Teilbaum T_i

Algorithmus von Prim



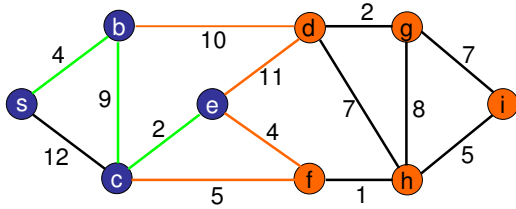
Knotenmenge S, Kantenmenge T:
In Teilbaum T_i

Algorithmus von Prim



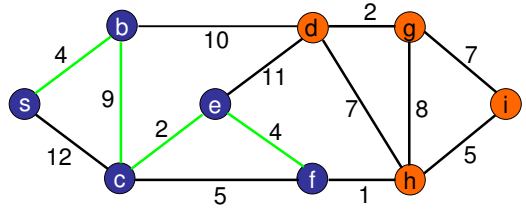
Knotenmenge S, Kantenmenge T:
In Teilbaum T_i

Algorithmus von Prim



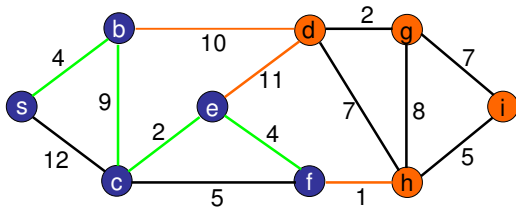
Knotenmenge S, Kantenmenge T:
In Teilbaum T_i

Algorithmus von Prim



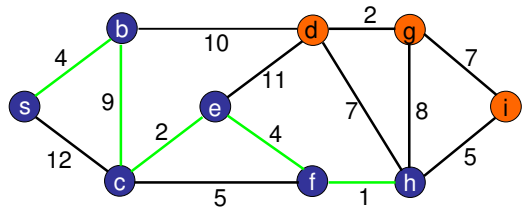
Knotenmenge S, Kantenmenge T:
In Teilbaum T_i

Algorithmus von Prim



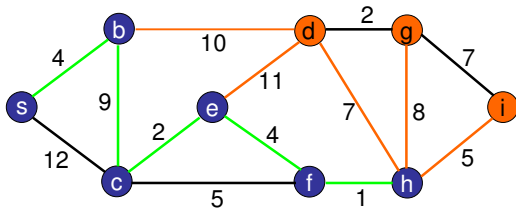
Knotenmenge S, Kantenmenge T:
In Teilbaum T_i

Algorithmus von Prim



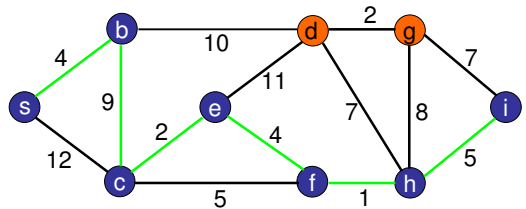
Knotenmenge S, Kantenmenge T:
In Teilbaum T_i

Algorithmus von Prim



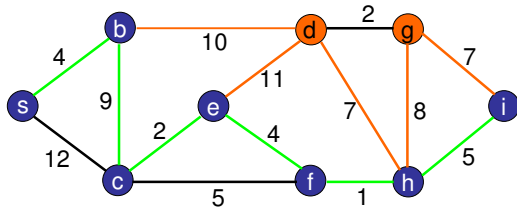
Knotenmenge S, Kantenmenge T:
In Teilbaum T_i

Algorithmus von Prim



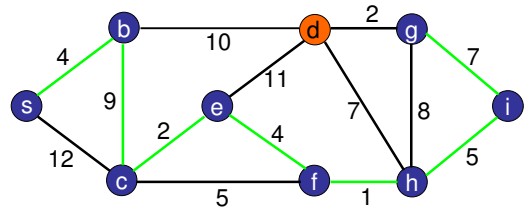
Knotenmenge S, Kantenmenge T:
In Teilbaum T_i

Algorithmus von Prim



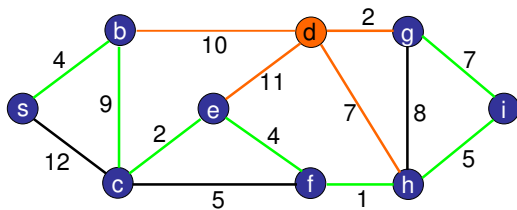
Knotenmenge S, Kantenmenge T:
In Teilbaum T_i

Algorithmus von Prim



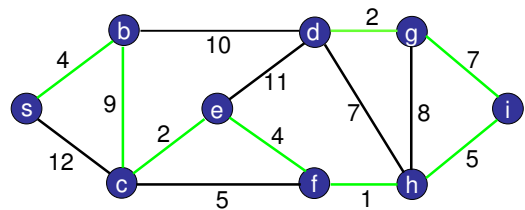
Knotenmenge S, Kantenmenge T:
In Teilbaum T_i

Algorithmus von Prim



Knotenmenge S, Kantenmenge T:
In Teilbaum T_i

Algorithmus von Prim



Knotenmenge S, Kantenmenge T:
In Teilbaum T_i

Realisierung Prim

Vgl. mit Kruskal:

- Kreistest (Union-Find Partition) nicht nötig
- Kein Sortieren nötig
- Auswahl leichtester Kante nötig \Rightarrow Priority Queue
- Wir speichern nicht die Kanten selbst, sondern die Knoten $v \in V \setminus S$. Priorität ist das Gewicht der Kante von S nach v

Pseudo-Code: Initialisierung

```

G=(V, E), float w[E]           // Graph und Gewichte
(1) var  $\pi[V]$ , PriorityQueue Q, pos[V] //  $\pi$  Vorgänger in
//MST, pos Pos. in Q
(2) for each  $u \in V \setminus \{s\}$  do
(3) pos[u] := Q.INSERT( $\infty$ , u)
(4) end for
(5) pos[s] := Q.INSERT(0, s)
(6)  $\pi[s] := nil$ 
    
```

Pseudo-Code Prim

```

(7) while not Q.ISEMPY() do
(8)    $(p, u) := Q.EXTRACTMIN()$ 
(9)    $pos[u] := nil$  // Minimum entfernen
(10)  for all  $e = (u, v) \in E(u)$  do
(11)    if  $pos[v] \neq nil$  and
(12)      $w(e) < Q.PRIORITY(pos[v], w(e))$  then
(13)       $Q.DECREASEPRIORITY(pos[v], w(e))$ 
(14)       $\pi[v] := u$ 
(15)    end if
(16)  end for
(17) end while
  
```

Analyse

- Aufbau des Heaps in $\Theta(|V|)$
- $|V|$ Durchläufe der while-Schleife mit EXTRACTMIN $\Rightarrow O(|V|\log|V|)$
- 2 Durchläufe der forall-Schleife für jede Kante
- $|E|$ Aufrufe von DECREASEPRIORITY max. $\Rightarrow O(|E|\log|V|)$

Gesamtlaufzeit $O(|E|\log|V|)$, da $|E| \geq |V| - 1$

Warum kann Greedy-Verfahren funktionieren?

- MST besteht aus Teilbäumen, die ebenfalls MSTs sind
- Lokal beste Entscheidung (leichteste Kante) führt zu MST

Kap. 6.6 Kürzeste Wege

Kürzeste Wege

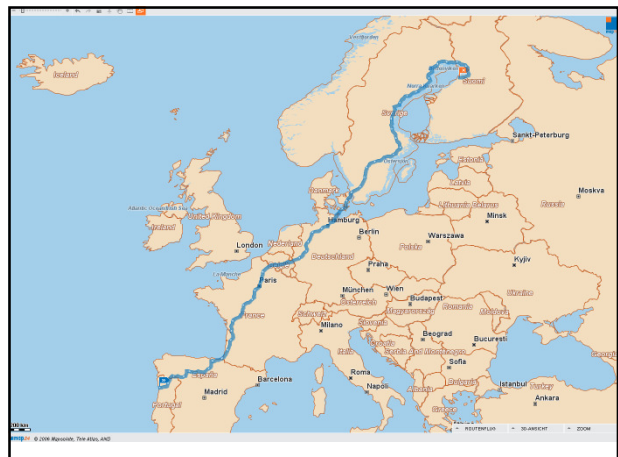
Achtung: in diesem Abschnitt **gerichtete gewichtete** Graphen!



Kürzeste Wege Problem

Gegeben: gerichteter Graph $G=(V,A)$
 Gewichtsfunktion $w : A \rightarrow \mathbb{R}$
 Keine Mehrfachkanten

Gesucht: Der bzgl. Gewicht w kürzeste Weg von Startknoten zu Zielknoten.



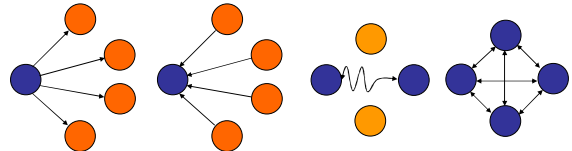
Anwendungen

Direkte und indirekte (Teilproblem) Anwendungen:

- Routenplaner (Streckenlänge)
- Auskunftssysteme für Bus und Bahn (Zeit)
- Berechnung minimaler Flüsse in Netzwerken
- DNA Sequenz Analyse
- ...

Kürzeste Wege Probleme

- Single Source Shortest Path (SSSP) ←
- Single Destination Shortest Path
- Single Pair Shortest Path
- All Pairs Shortest Path (APSP) ←



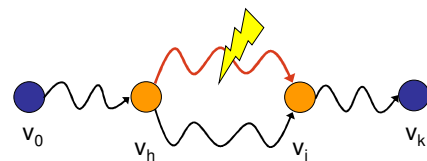
Kürzeste Wege

Single Source Shortest Path (SSSP)

- Bekannt: BFS für ungewichtete kürzeste Wege (USSSP)
- **Jetzt:** Kürzeste Wege mit Kantengewichten:
 - Gerichteter Graph $G = (V, A)$
 - Kantengewichte $w(e) \in \mathbf{R}$ (Strecke, Fahrzeit)
 - Startknoten s
 - Weglänge $w(p) := \sum_{i=1, \dots, k} w(e_i)$ für $p = v_0, e_1, \dots, e_k, v_k$

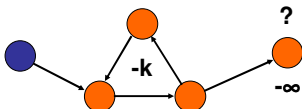
Optimale Substruktur

Pfad über $v_0=s, v_1, \dots, v_k$ kürzester Weg von s nach v_k
 \Rightarrow Teilweg $v_h \rightarrow v_j$ kürzester Weg von v_h nach v_j



Single Source Shortest Path

Keine negativen Kreise!!



\Rightarrow Kürzeste Wege von Knoten s bilden immer einen Baum

Spezialfall: $w(e) \geq 0$ (Strecke, Fahrzeit)

Algorithmus von Dijkstra

Analogien mit BFS und Prim:

- *BFS/Prim* läßt *einzelnen* Baum wachsen: Neue Kante verbindet Baum mit Rest
- *Dijkstra* bildet Kürzeste-Wege Baum (SPT) ausgehend von Wurzel s , aber andere Auswahl
- Greedy, Priority Queue

Bezeichnungen

- Vorgänger von Knoten v im SPT: $\pi[v]$
- Kanten, die Knotenmenge S verlassen: $A(S)$
- Min. Abstand vom Startknoten zu Knoten v : $d[v]$

Schema: Algor. von Dijkstra

```

S := {s}           // S Knoten im SPT
d[s] := 0; π[s] := nil // s Wurzel
while A(S) ≠ ∅ do // erreichbare Knoten
    // Optimale Substruktur
    Sei e = (u, v) ∈ A(S) mit d[u] + w(e) minimal
    S := S ∪ {v}
    d[v] := d[u] + w(e)
    π[v] := u
end while
    
```

Korrektheit

- Knoten ausserhalb von S sind nur über Knoten bzw. Pfade in S erreichbar
- Dijkstra wählt Minimum der Weglänge aus S hinaus und erweitert S
- w nicht-negativ: Spätere Weglängen können nur größer sein als Minimum!
- Optimale Substruktur: Kürzester Weg besteht aus kürzestem Weg plus Kante

Korrektheit

Induktion über die Kanten e_1, \dots, e_n , die der Algorithmus für v wählt, $e_i = (u_i, v_i)$, $v_0 := s$.

Zeige: v_0, \dots, v_i ist kürzester Weg von v_0 nach v_i

$i=0$: Keine Kante, korrekt

$1 \leq i \leq n$: Annahme v_0, \dots, v_j kürzester Weg für $j < i$.

Dijkstra wählt Weg mit Kosten $d[u_i] + w(e_i)$.

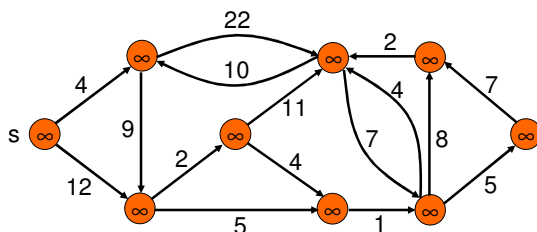
Jeder andere Weg $v_0 \rightarrow v_i$ beginnt mit Knoten aus $V_i := \{v_0, \dots, v_{i-1}\}$ gefolgt von Kante $(x, y) \in A(V_{i-1})$ und hat deshalb Kosten mind. $d[u_i] + w(e_i)$, da e_i als Minimum gewählt wurde.

- Alle erreichbaren Knoten werden auch erreicht

Realisierung von Dijkstra

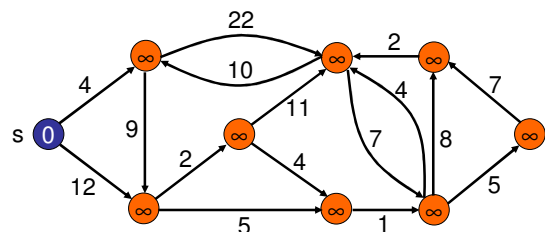
- Knotenmenge S: Kürzester Weg mit Länge $d[v]$ bereits ermittelt
 - Knotenmenge $V \setminus S$: Speichere *vorläufige* Werte für den Abstand zu s (obere Schranke $\delta(v) \geq d[v]$) in *Priority Queue*, und aktualisiere Priorität, falls günstigerer Weg gefunden wird, $\delta(s) = 0$
1. Wähle mit **EXTRACTMIN** Knoten u mit minimalem Abstandswert. Start: $\delta(s) = d[s] = 0$
 2. Aktualisiere für ausgehende Kanten (u, v) Abstandswerte der Endknoten („Relaxieren“) mit **DECREASEPRIORITY** und Vorgänger $\pi[v]$

Algorithmus von Dijkstra



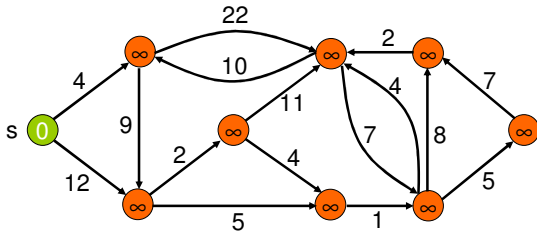
Priority Queue PQ: Knoten, Priorität Weglänge

Algorithmus von Dijkstra



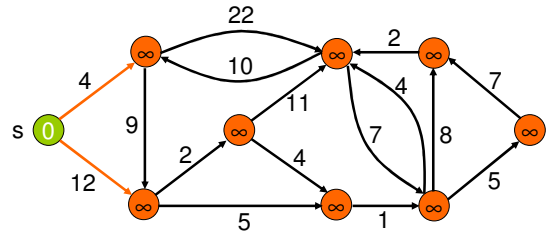
Priority Queue PQ: Knoten, Priorität Weglänge
Kandidatenmenge K in PQ: Weg von s gefunden

Algorithmus von Dijkstra



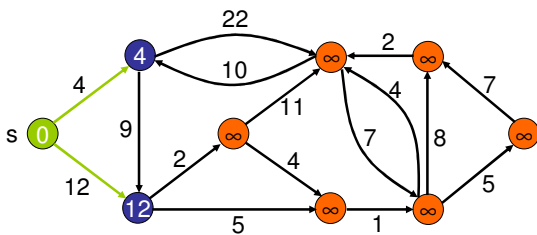
Priority Queue PQ: Knoten, Priorität Weglänge
 Kandidatenmenge K in PQ: Weg von s gefunden
 Abgeschlossene Knoten: Minimum aus PQ

Algorithmus von Dijkstra



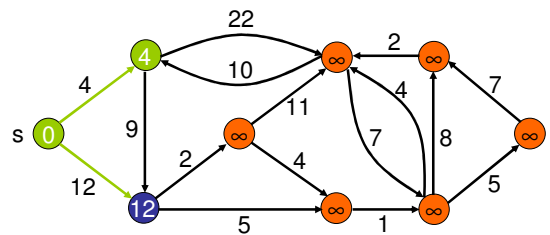
Ausgehende Kanten des gewählten Knoten

Algorithmus von Dijkstra



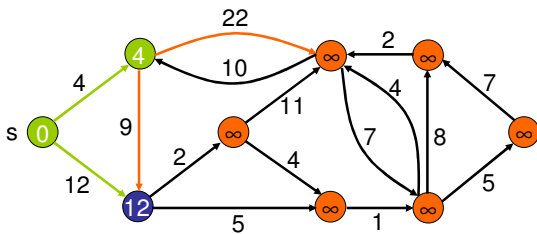
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



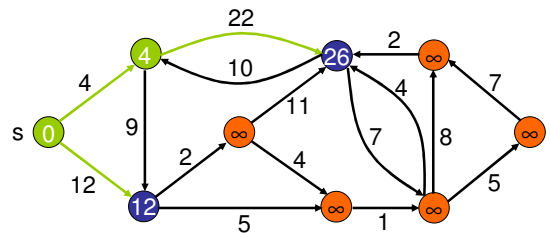
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



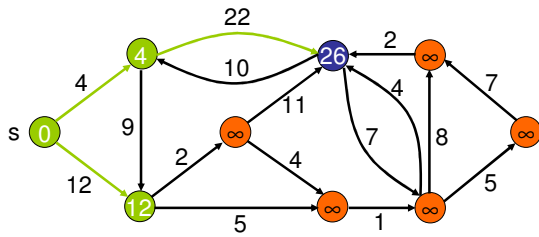
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



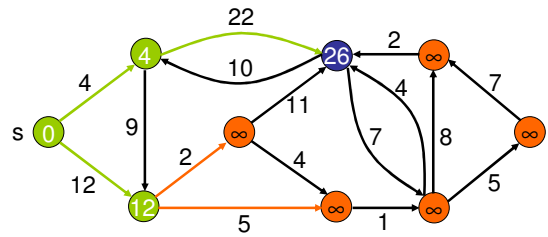
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



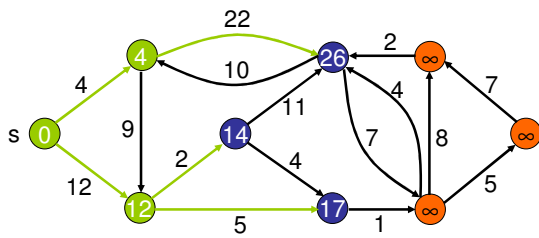
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



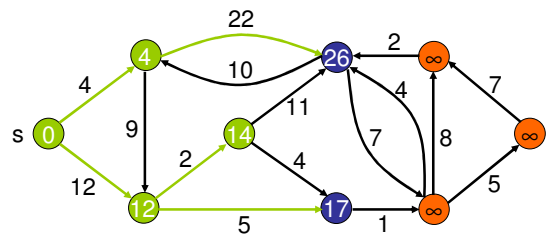
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



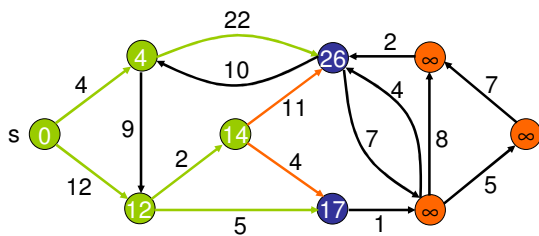
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



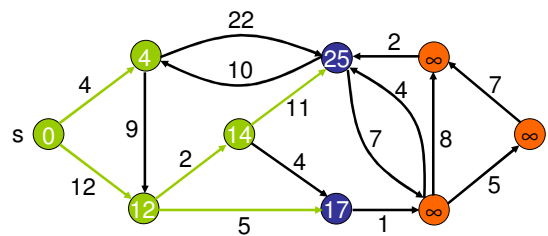
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



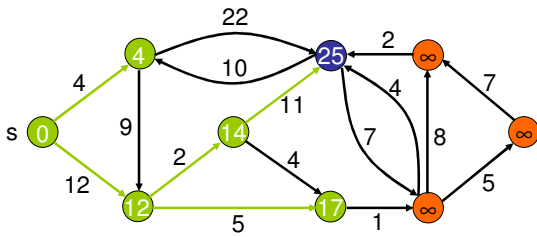
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



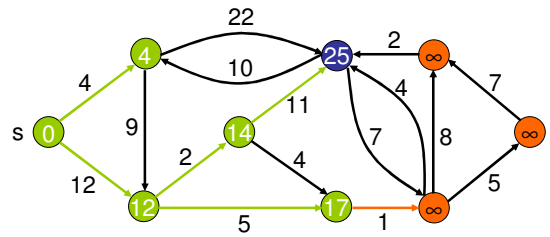
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



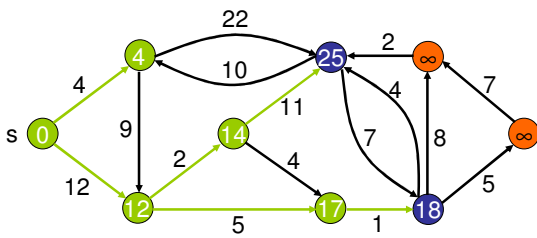
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



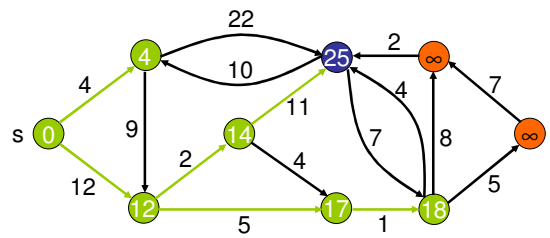
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



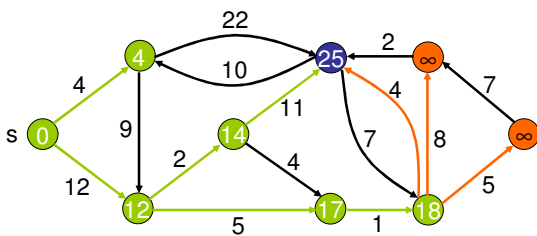
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



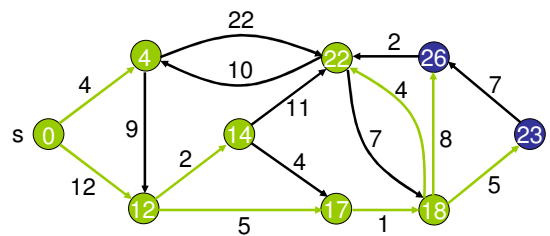
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



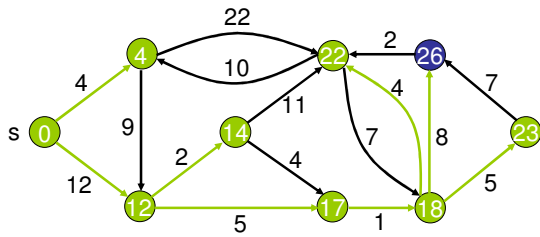
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



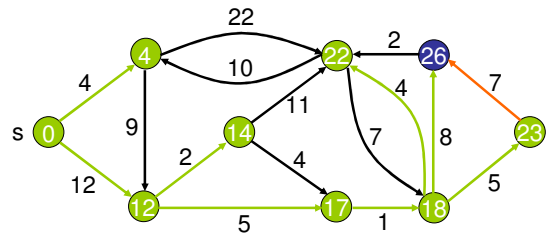
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



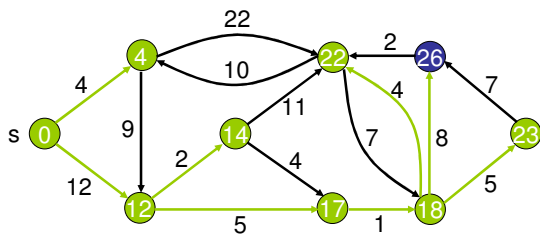
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



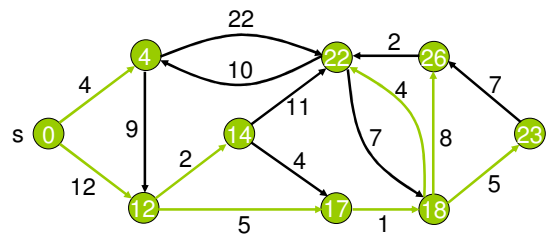
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



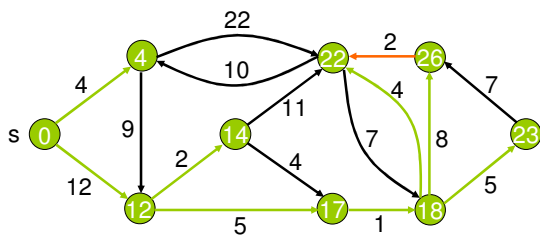
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



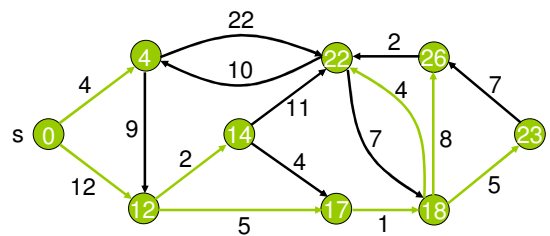
Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Algorithmus von Dijkstra



Ausgehende Kanten des gewählten Knoten
 Erreichte Kandidaten
 Vorgänger-Kanten: π

Pseudo-Code: Initialisierung

```
G=(V, A), w: A → R0+  
(1) var π[V], PriorityQueue Q, pos[V]  
(2) for each u ∈ V \ {s} do  
(3)   pos[u] := Q.INSERT(∞, u)  
(4)   π[u] := nil  
(5) end for  
(6) pos[s] := Q.INSERT(0, s)  
(7) π[s] := nil
```

Pseudo-Code

```
(8) while not Q.ISEMPTY() do  
(9)   (dv, u) := Q.EXTRACTMIN(); //dv Abstand s zu u  
(10)  pos[u] := nil //Minimum entfernt  
(11)  for all e = (u, v) ∈ A(u) do //Erreichbare Knoten  
(12)    if dv+w(e) < Q.PRIORITY(pos[v]) then  
(13)      Q.DECREASEPRIORITY(pos[v], dv + w(e))  
(14)      π[v] := u  
(15)    end if  
(16)  end for  
(17) end while
```

Analyse

Laufzeit abhängig von Datenstruktur:

- Kosten der ExtractMin und vor allem der DecreasePriority Operationen
- Wir betrachten Binary Heaps

Spezielle PQ mit amortisiert konstanter DecreasePriority Zeit:
Fibonacci-Heaps

Analyse

- Aufbau des Heaps in $O(|V|)$
- $|V|$ Durchläufe der while-Schleife mit EXTRACTMIN $\Rightarrow O(|V|\log|V|)$
- $|A|$ Aufrufe von DECREASEPRIORITY max. $\Rightarrow O(|A|\log|V|)$

Gesamtlaufzeit $O((|V|+|A|)\log|V|)$

Ende