Grundzüge der Informatik 1

Vorlesung 21

Überblick Vorlesung

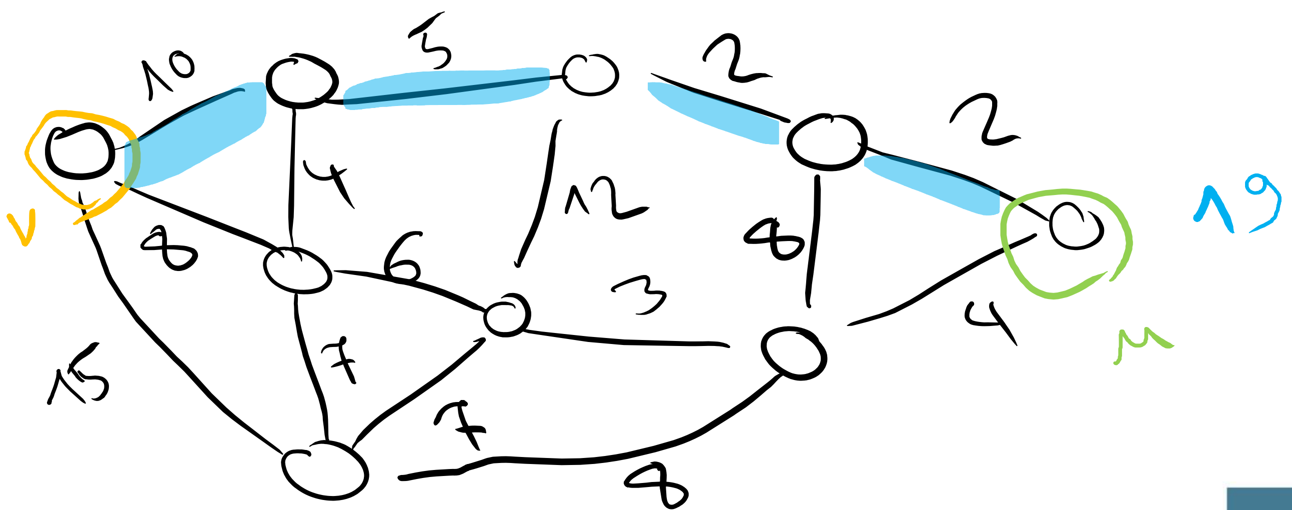
Graphenalgorithmen

- Wiederholung:
  - Breitensuche
- Kürzeste Wege in gewichteten Graphen
  - Simulation der Breitensuche
  - Dijkstras Algorithmus

Graphenalgorithmen

Kürzeste Wege in Graphen

- Gegeben (möglicherweise gewichteter) Graph G=(V,E)
- Frage: Was ist der kürzeste Weg von Knoten v nach Knoten u?
- Länge des Weges: Summe der Kantengewichte (bzw. Anzahl Kanten, wenn ungewichtet)



Graphenalgorithmen

SSSP in ungewichteten Graphen mit Breitensuche

- Graph in Adjazenzlistendarstellung
- Startknoten s
- Nutze Kanten von G, um alle Knoten zu finden, die von s aus erreichbar sind
- Finde kürzeste Distanz (Anzahl Kanten) zu jedem anderen Knoten

Idee

- Bearbeitet den Graphen „schichtweise" nach Entfernung vom Startknoten: Besucht zuerst alle Knoten mit Entfernung 1; dann alle mit Entfernung 2; usw.

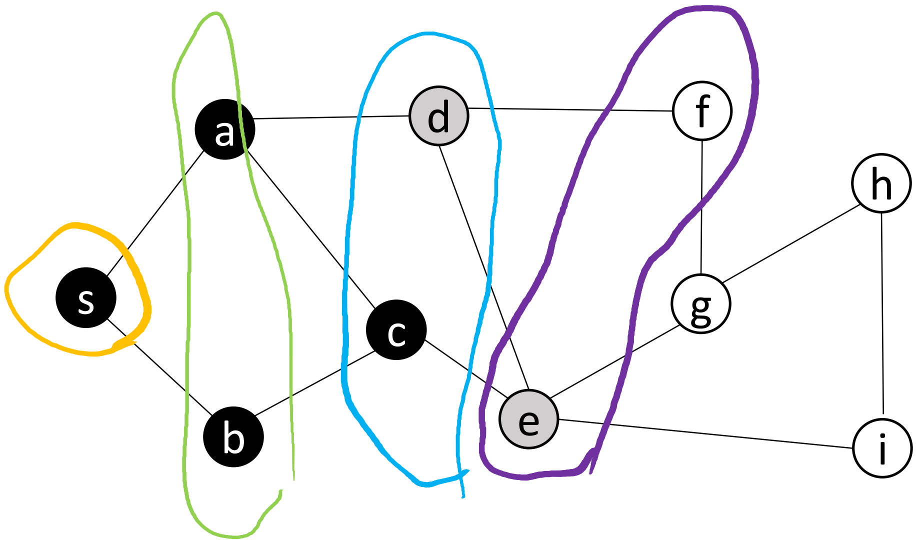
Graphenalgorithmen

Technische Invariante (Breitensuche)

- Knoten haben 3 Farben: weiß, grau und schwarz
- Zu Beginn: Alle Knoten sind weiß; Knoten s ist grau
- Ein nicht-weißer Knoten heißt „entdeckt“
- Unterscheidung grau-schwarz dient zur Steuerung des Algorithmus
- Wenn Knoten schwarz ist, dann sind seine benachbarten Knoten grau oder schwarz
- Graue Knoten können benachbarte weiße Knoten haben

Graphenalgorithmen

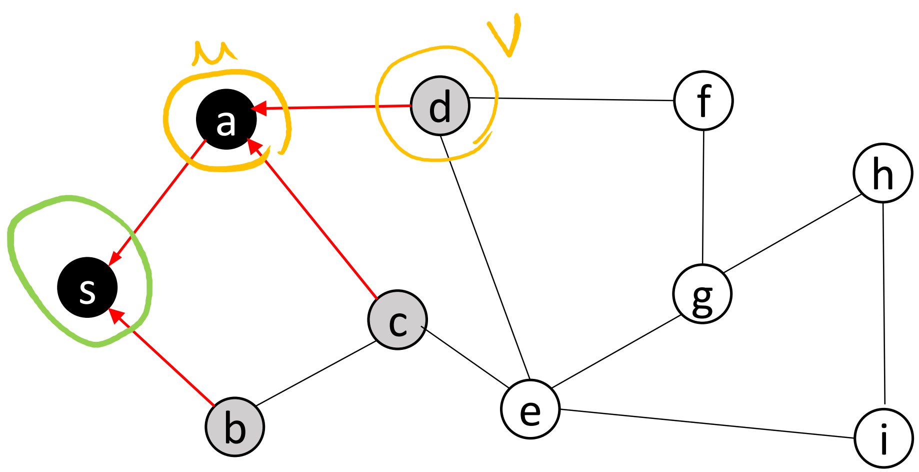
Beispiel Invariante



Graphenalgorithmen

Breitensuche

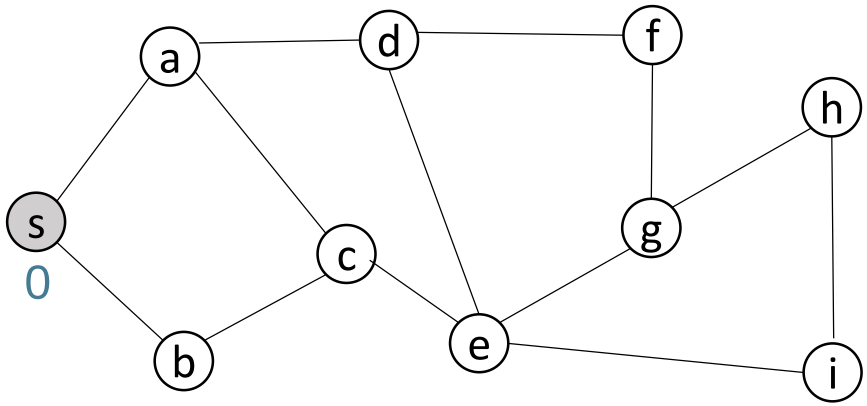
- Baut Breitensuche-Baum (BFS-Baum)
- Zu Beginn enthält der Baum nur die Wurzel, nämlich s
- Wenn weißer Knoten v beim Durchsuchen der Adjazenzliste eines bereits entdeckten Knotens u entdeckt wird, dann werden v und (u,v) dem Baum hinzugefügt
- u ist dann Vaterknoten von v



Graphenalgorithmen

BFS(G,s)

1. „initialisiere BFS“
2. **while** Q≠∅ **do**
3.   u = head[Q]
4.   **for** each v∈Adj[u] **do**
5.   **if** color[v] = weiß **then**
6.   color[v] = grau
7.   d[v] = d[u]+1; π[v] = u
8.   enqueue(Q,v)
9.   dequeue(Q)
10.   color[u] = schwarz

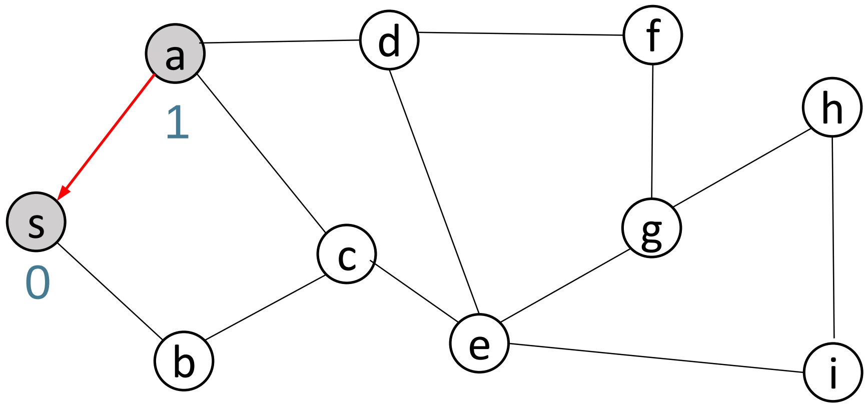


Q: s

Graphenalgorithmen

BFS(G,s)

1. „initialisiere BFS“
2. **while** Q≠∅ **do**
3. u = head[Q]
4. **for** each v∈Adj[u] **do**
5. **if** color[v] = weiß **then**
6. color[v] = grau
7. d[v] = d[u]+1; π[v] = u
8. enqueue(Q,v)
9. dequeue(Q)
10. color[u] = schwarz

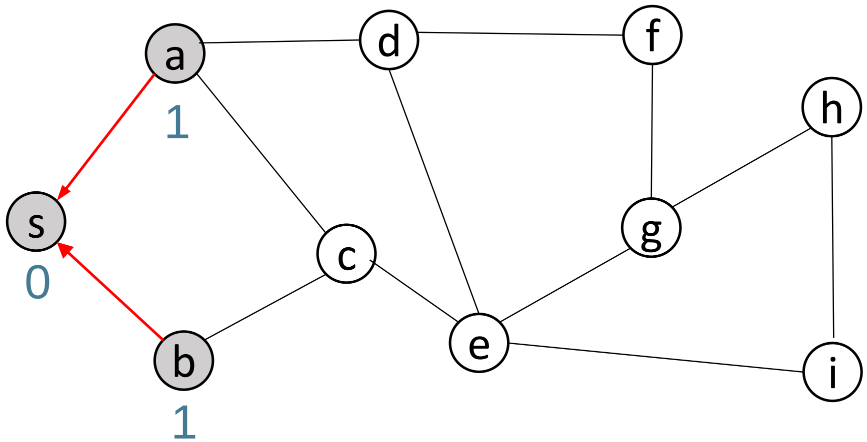


Q: s, a

Graphenalgorithmen

BFS(G,s)

1. „initialisiere BFS“
2. **while** Q≠∅ **do**
3. u = head[Q]
4. **for** each v∈Adj[u] **do**
5. **if** color[v] = weiß **then**
6. color[v] = grau
7. d[v] = d[u]+1; π[v] = u
8. enqueue(Q,v)
9. dequeue(Q)
10. color[u] = schwarz

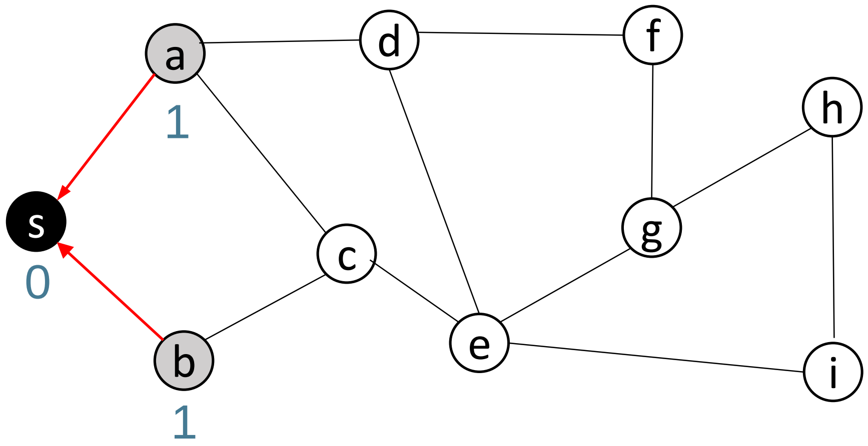


Q: s, a, b

Graphenalgorithmen

BFS(G,s)

1. „initialisiere BFS“
2. **while** Q≠∅ **do**
3. u = head[Q]
4. **for** each v∈Adj[u] **do**
5. **if** color[v] = weiß **then**
6. color[v] = grau
7. d[v] = d[u]+1; π[v] = u
8. enqueue(Q,v)
9. dequeue(Q)
10. color[u] = schwarz

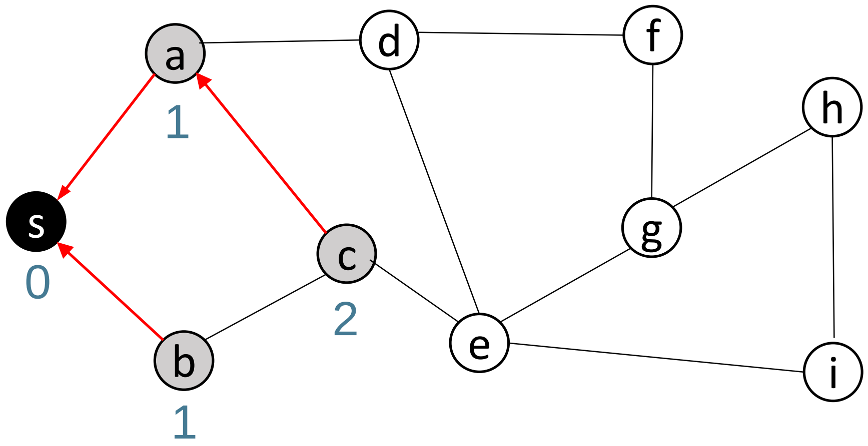


Q: a, b

Graphenalgorithmen

BFS(G,s)

1. „initialisiere BFS“
2. **while** Q≠∅ **do**
3. u = head[Q]
4. **for** each v∈Adj[u] **do**
5. **if** color[v] = weiß **then**
6. color[v] = grau
7. d[v] = d[u]+1; π[v] = u
8. enqueue(Q,v)
9. dequeue(Q)
10. color[u] = schwarz

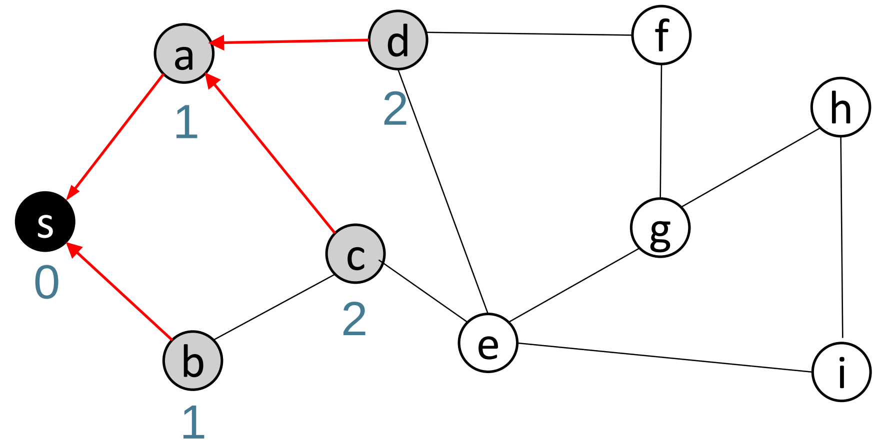


Q: a, b, c

Graphenalgorithmen

BFS(G,s)

1. „initialisiere BFS“
2. **while** Q≠∅ **do**
3. u = head[Q]
4. **for** each v∈Adj[u] **do**
5. **if** color[v] = weiß **then**
6. color[v] = grau
7. d[v] = d[u]+1; π[v] = u
8. enqueue(Q,v)
9. dequeue(Q)
10. color[u] = schwarz

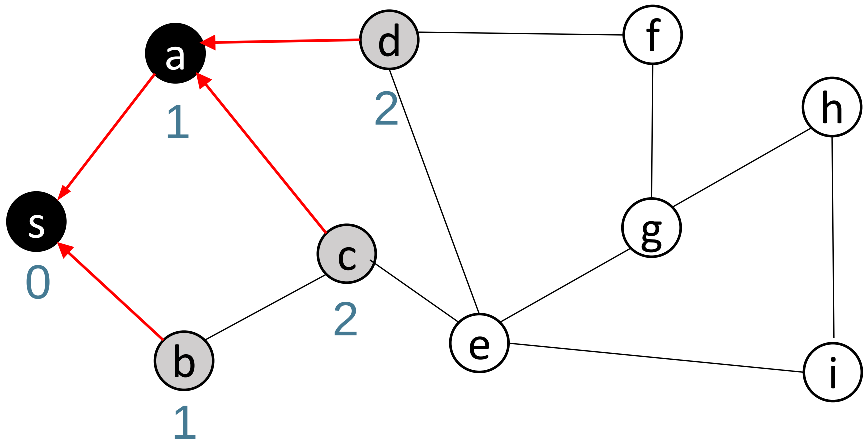


Q: a, b, c, d

Graphenalgorithmen

BFS(G,s)

1. „initialisiere BFS“
2. **while** Q≠∅ **do**
3. u = head[Q]
4. **for** each v∈Adj[u] **do**
5. **if** color[v] = weiß **then**
6. color[v] = grau
7. d[v] = d[u]+1; π[v] = u
8. enqueue(Q,v)
9. dequeue(Q)
10. color[u] = schwarz

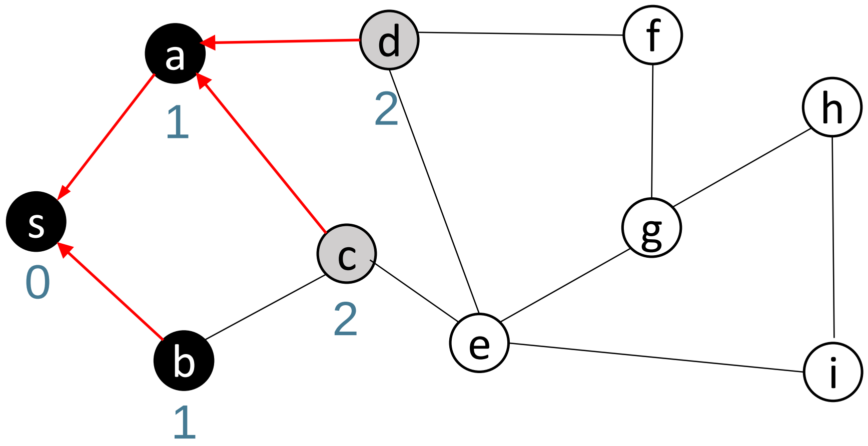


Q: b, c, d

Graphenalgorithmen

BFS(G,s)

1. „initialisiere BFS“
2. **while** Q≠∅ **do**
3. u = head[Q]
4. **for** each v∈Adj[u] **do**
5. **if** color[v] = weiß **then**
6. color[v] = grau
7. d[v] = d[u]+1; π[v] = u
8. enqueue(Q,v)
9. dequeue(Q)
10. color[u] = schwarz

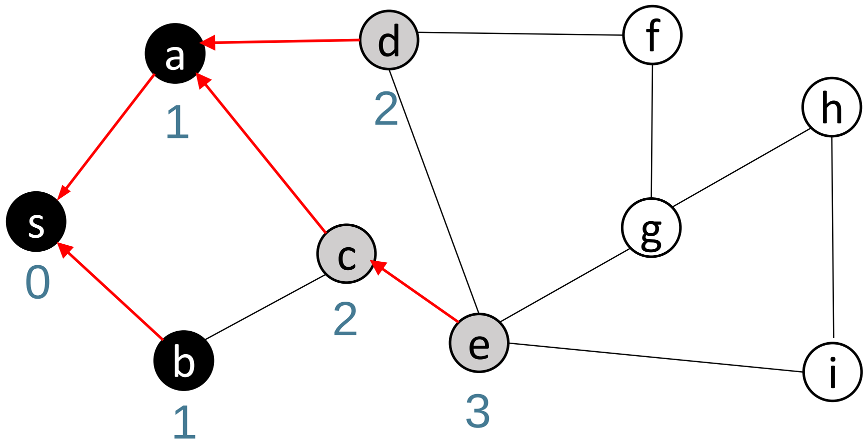


Q: c, d

Graphenalgorithmen

BFS(G,s)

```
„initialisiere BFS“
while Q≠∅ do
    u = head[Q]
    for each v∈Adj[u] do
        if color[v] = weiß then
            color[v] = grau
            d[v] = d[u]+1; π[v] = u
            enqueue(Q,v)
    dequeue(Q)
color[u] = schwarz
```

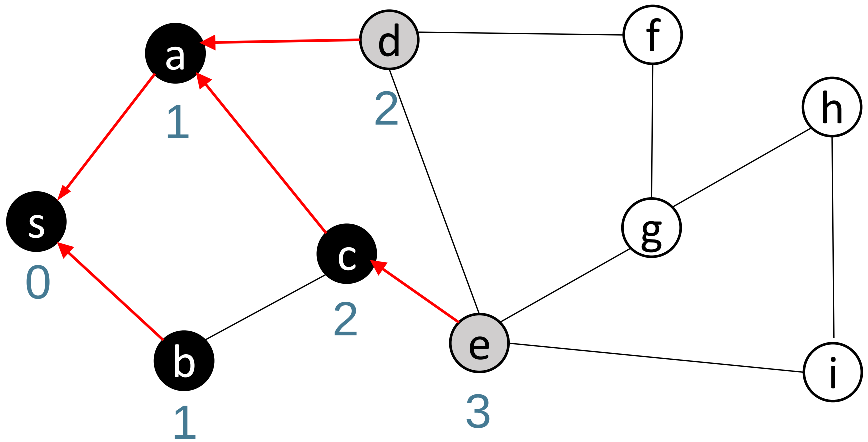


Q: c, d, e

Graphenalgorithmen

BFS(G,s)

1. „initialisiere BFS“
2. **while** Q≠∅ **do**
3. u = head[Q]
4. **for** each v∈Adj[u] **do**
5. **if** color[v] = weiß **then**
6. color[v] = grau
7. d[v] = d[u]+1; π[v] = u
8. enqueue(Q,v)
9. dequeue(Q)
10. color[u] = schwarz

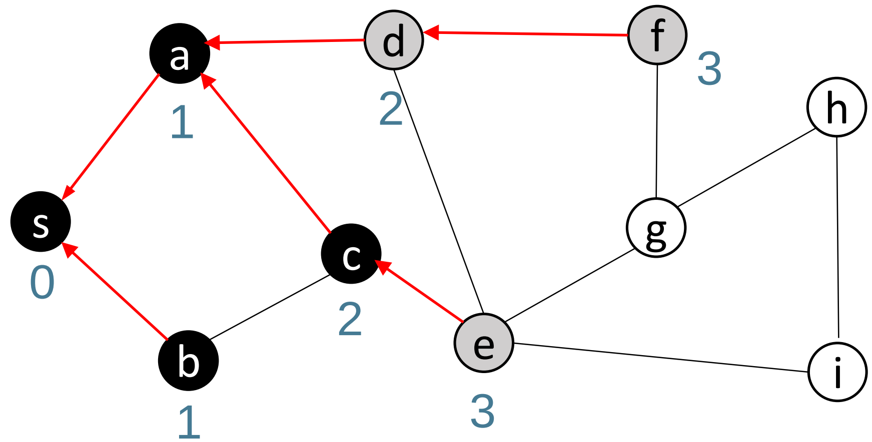


Q: d, e

Graphenalgorithmen

BFS(G,s)

```
„initialisiere BFS“
while Q≠∅ do
    u = head[Q]
    for each v∈Adj[u] do
        if color[v] = weiß then
            color[v] = grau
            d[v] = d[u]+1; π[v] = u
            enqueue(Q,v)
    dequeue(Q)
    color[u] = schwarz
```

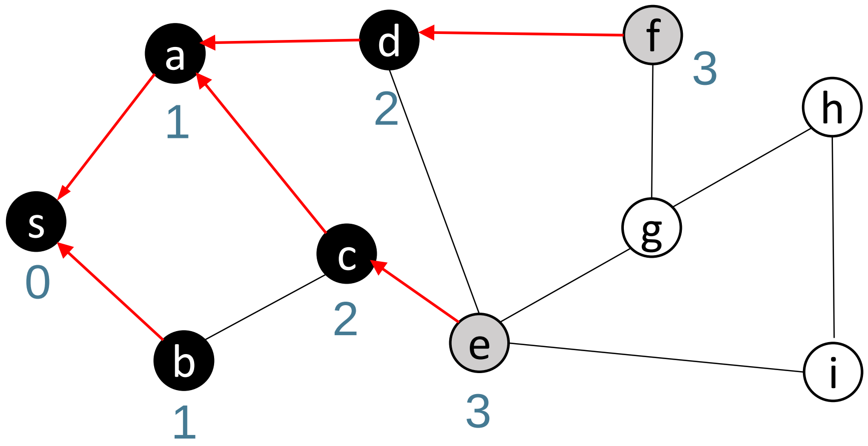


Q: d, e, f

Graphenalgorithmen

BFS(G,s)

1. „initialisiere BFS“
2. **while** Q≠∅ **do**
3. u = head[Q]
4. **for** each v∈Adj[u] **do**
5. **if** color[v] = weiß **then**
6. color[v] = grau
7. d[v] = d[u]+1; π[v] = u
8. enqueue(Q,v)
9. dequeue(Q)
10. color[u] = schwarz

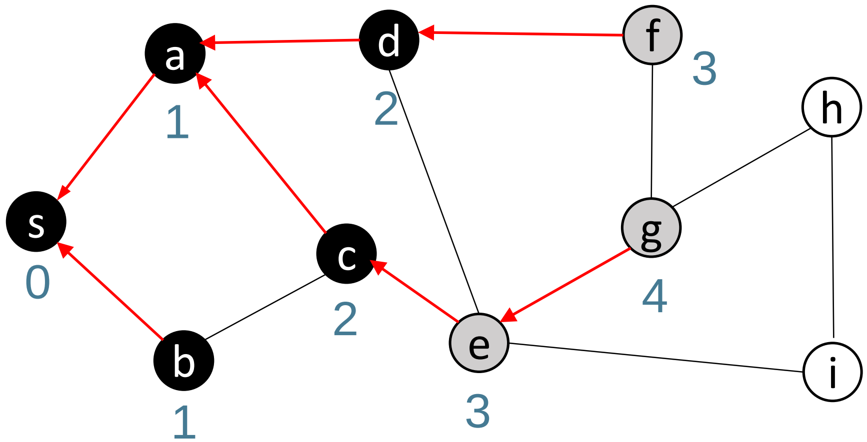


Q: e, f

Graphenalgorithmen

BFS(G,s)

```
„initialisiere BFS“
while Q≠∅ do
    u = head[Q]
    for each v∈Adj[u] do
        if color[v] = weiß then
            color[v] = grau
            d[v] = d[u]+1; π[v] = u
            enqueue(Q,v)
    dequeue(Q)
color[u] = schwarz
```

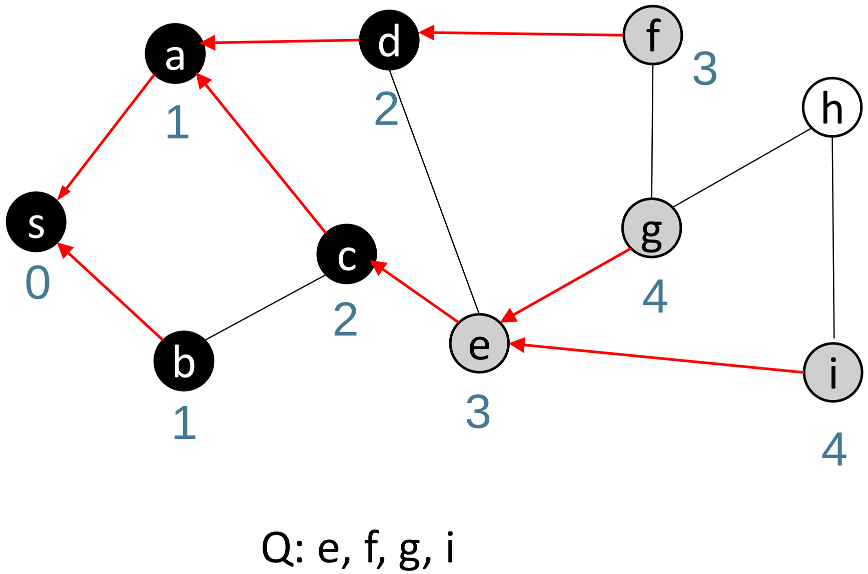


Q: e, f, g

Graphenalgorithmen

BFS(G,s)

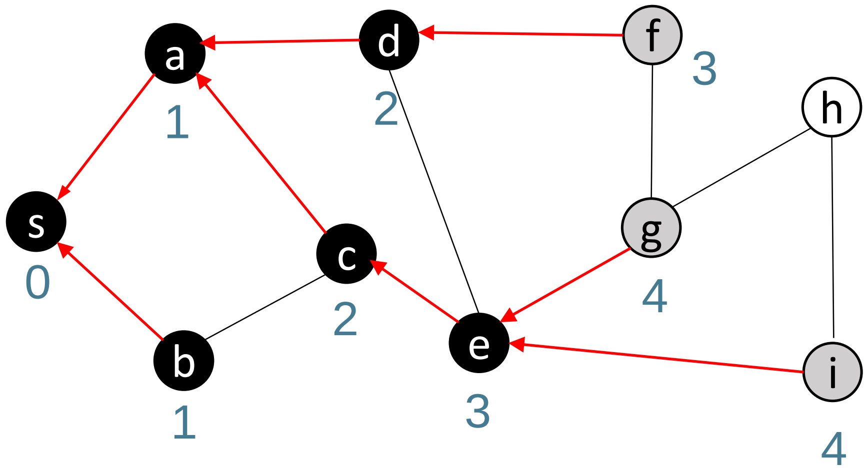
1. „initialisiere BFS“
2. **while** Q≠∅ **do**
3. u = head[Q]
4. **for** each v∈Adj[u] **do**
5. **if** color[v] = weiß **then**
6. color[v] = grau
7. d[v] = d[u]+1; π[v] = u
8. enqueue(Q,v)
9. dequeue(Q)
10. color[u] = schwarz



Graphenalgorithmen

BFS(G,s)

```
„initialisiere BFS“
while Q≠∅ do
    u = head[Q]
    for each v∈Adj[u] do
        if color[v] = weiß then
            color[v] = grau
            d[v] = d[u]+1; π[v] = u
            enqueue(Q,v)
    dequeue(Q)
color[u] = schwarz
```

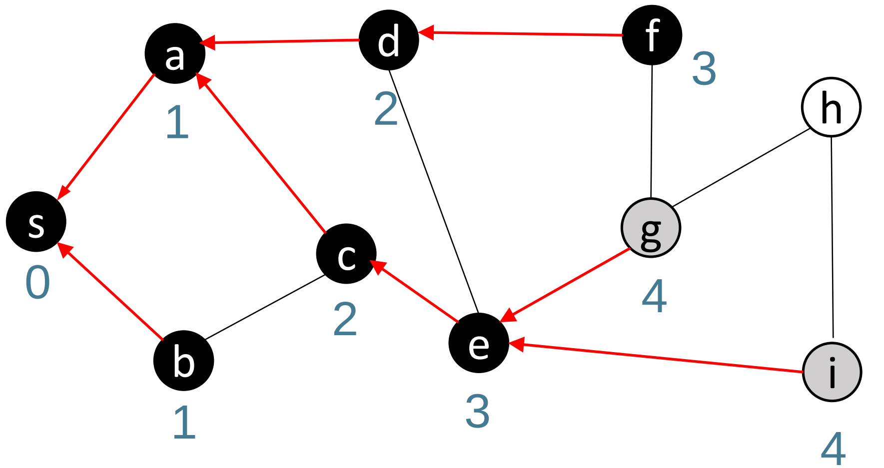


Q: f, g, i

Graphenalgorithmen

BFS(G,s)

```
„initialisiere BFS“
while Q≠∅ do
    u = head[Q]
    for each v∈Adj[u] do
        if color[v] = weiß then
            color[v] = grau
            d[v] = d[u]+1; π[v] = u
            enqueue(Q,v)
    dequeue(Q)
    color[u] = schwarz
```

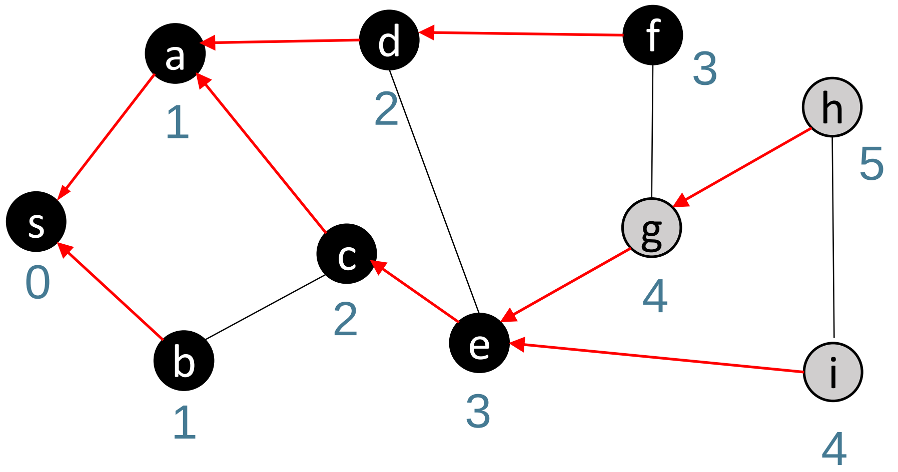


Q: g, i

Graphenalgorithmen

BFS(G,s)

1. „initialisiere BFS“
2. **while** Q≠∅ **do**
3.   u = head[Q]
4.   **for** each v∈Adj[u] **do**
5.   **if** color[v] = weiß **then**
6.   color[v] = grau
7.   d[v] = d[u]+1; π[v] = u
8.   enqueue(Q,v)
9.   dequeue(Q)
10.   color[u] = schwarz

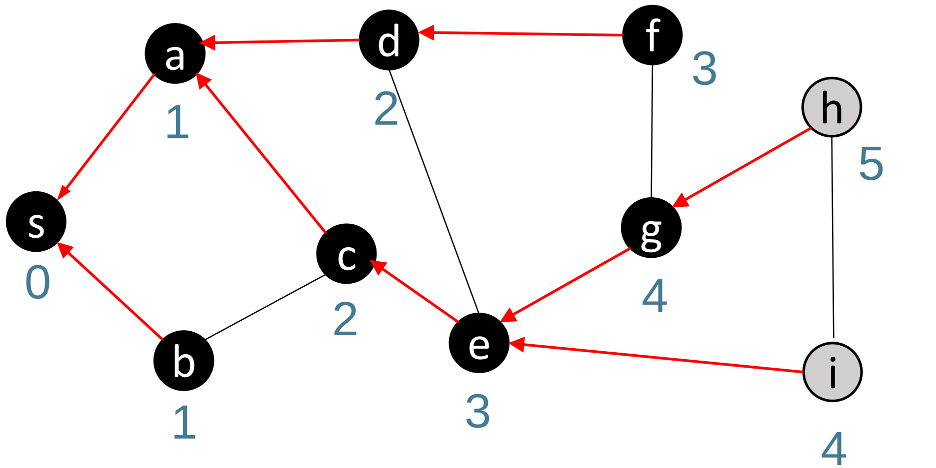


Q: g, i, h

Graphenalgorithmen

BFS(G,s)

```
„initialisiere BFS“
while Q≠∅ do
    u = head[Q]
    for each v∈Adj[u] do
        if color[v] = weiß then
            color[v] = grau
            d[v] = d[u]+1; π[v] = u
            enqueue(Q,v)
    dequeue(Q)
    color[u] = schwarz
```

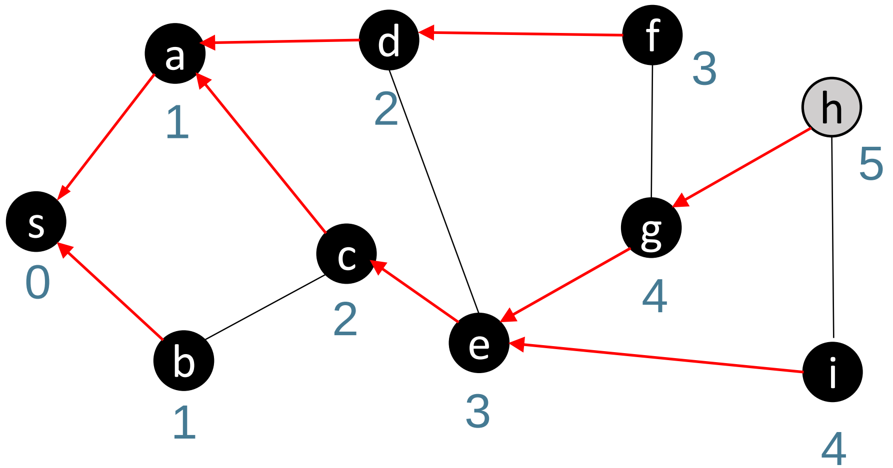


Q: i, h

Graphenalgorithmen

BFS(G,s)

1. „initialisiere BFS“
2. **while** Q≠∅ **do**
3.   u = head[Q]
4.   **for** each v∈Adj[u] **do**
5.   **if** color[v] = weiß **then**
6.   color[v] = grau
7.   d[v] = d[u]+1; π[v] = u
8.   enqueue(Q,v)
9.   dequeue(Q)
10.   color[u] = schwarz

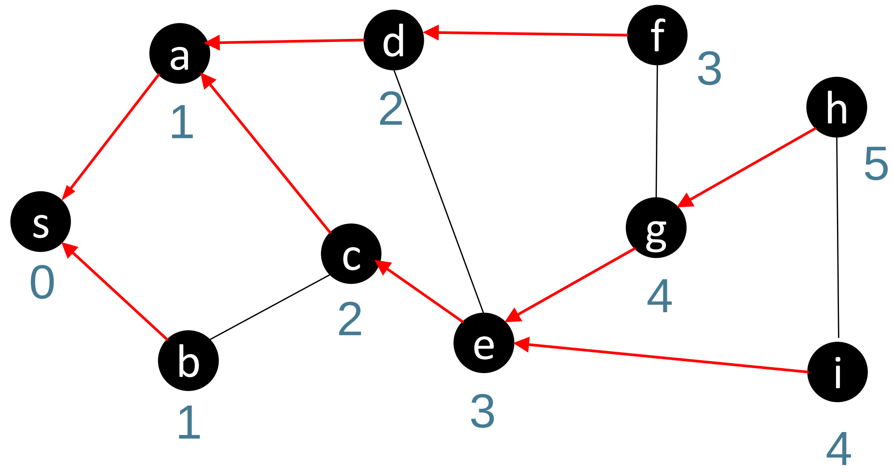


Q: h

Graphenalgorithmen

BFS(G,s)

1. „initialisiere BFS“
2. **while** Q≠∅ **do**
3. u = head[Q]
4. **for** each v∈Adj[u] **do**
5. **if** color[v] = weiß **then**
6. color[v] = grau
7. d[v] = d[u]+1; π[v] = u
8. enqueue(Q,v)
9. dequeue(Q)
10. color[u] = schwarz



Q:

Graphenalgorithmen

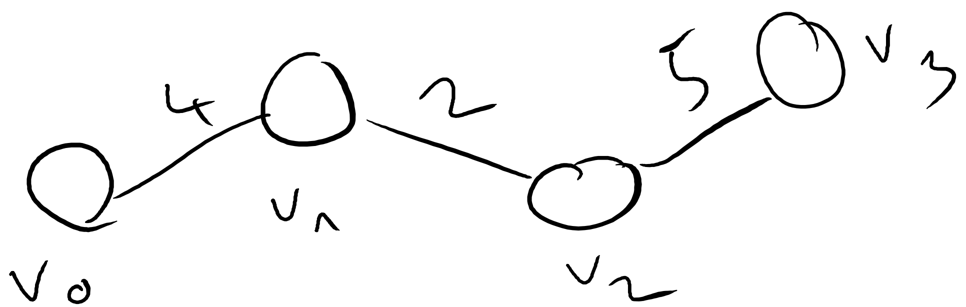
Zusammenfassung

- Breitensuche traversiert einen Graph in O(|V|+|E|) Zeit
- Die Breitensuche kann zur Berechnung der kürzesten Wege in ungewichteten Graphen verwendet werden

Graphenalgorithmen

Kürzeste Wege in gewichteten Graphen

- G=(V,E)
- w: $E \rightarrow \mathbb{R}$; w(e) ist Länge der Kante e; w(u,v) ist Länge der Kante (u,v)
- Für Weg p=<v_0, v_1,…, v_k> ist Länge gegeben durch $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$
- δ(u,v) $= \min_{u-v-\text{Wege}\ p} w(p)$, falls es Weg von u nach v gibt
- δ(u,v) = ∞, sonst

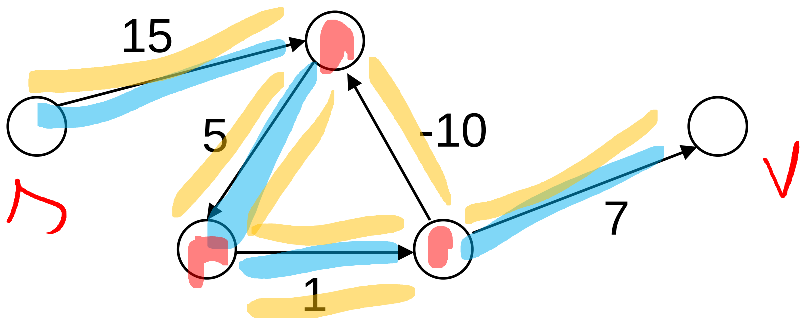


11

Graphenalgorithmen

Negative Kantengewichte

- Manchmal hat man Instanzen mit negativen Kantenlängen

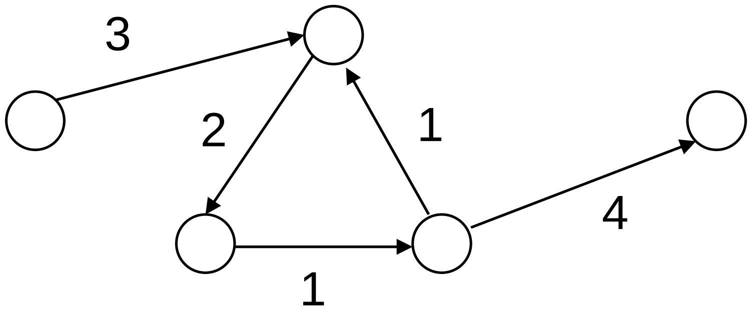


- Bei ungerichteten Graphen kann man Kante immer wieder vorwärts und rückwärts durchlaufen
- Kürzester Weg u.U. nicht wohldefiniert
- Erstmal nichtnegative Kantengewichte

Graphenalgorithmen

Dijkstras Algorithmus

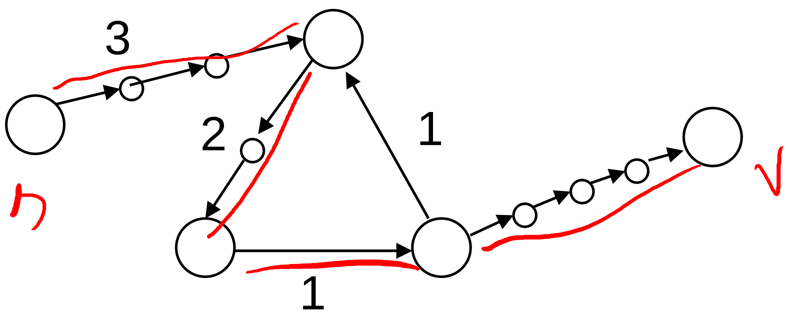
- Graph in Adjazenzlistendarstellung
- Keine negativen Kantenlängen
- Modifiziert Idee der Breitensuche auf gewichtete Graphen



Graphenalgorithmen

Dijkstras Algorithmus

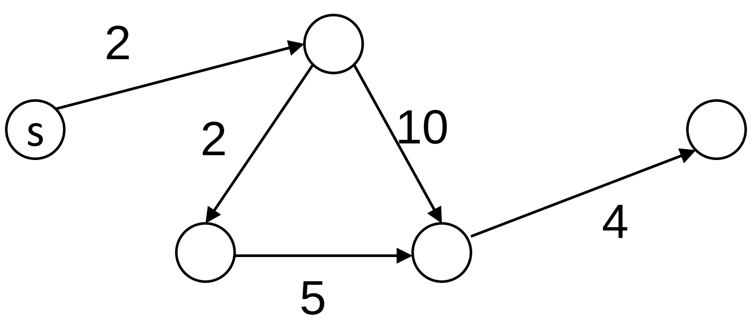
- Graph in Adjazenzlistendarstellung
- Keine negativen Kantenlängen
- Modifiziert Idee der Breitensuche auf gewichtete Graphen



Graphenalgorithmen

Erster Ansatz

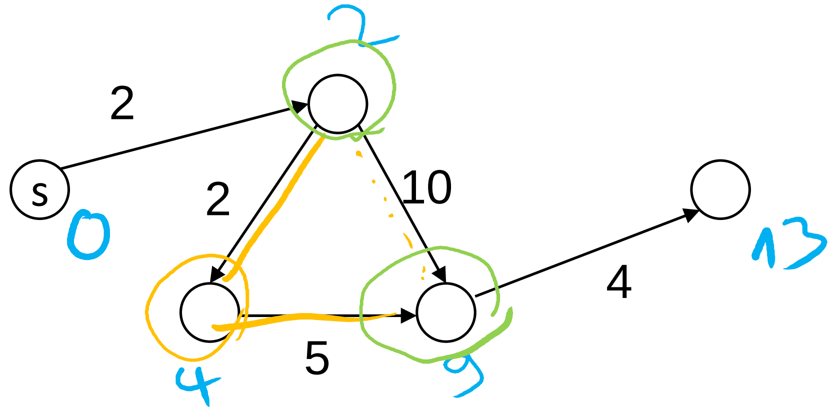
- Ersetze Kantenlängen durch mehrfache Kanten
- Probleme: Langsam bei großen Kantenlängen; nur ganzzahlige Längen
- Annahme: Zunächst ganzzahlige Längen.
- Idee: Simuliere Breitensuche effizient
- Aufgabe: Bestimme für jeden Knoten den Zeitpunkt, zu dem er entdeckt wird



Graphenalgorithmen

Erster Ansatz

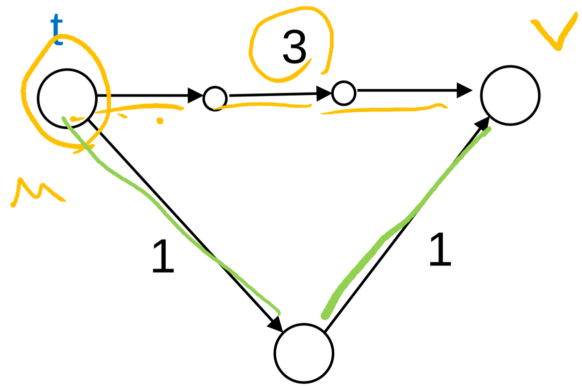
- Ersetze Kantenlängen durch mehrfache Kanten
- Probleme: Langsam bei großen Kantenlängen; nur ganzzahlige Längen
- Annahme: Zunächst ganzzahlige Längen
- Idee: Simuliere Breitensuche effizient
- Aufgabe: Bestimme für jeden Knoten den Zeitpunkt, zu dem er entdeckt wird



Graphenalgorithmen

Beobachtung zur Breitensuche

- Betrachte Breitensuche in der expandierten Version von G
- Wird ein Knoten u zum Zeitpunkt t (d.h. d[u]=t) entdeckt und ist Kante (u,v) mit Gewicht w(u,v) in G, so wird v spätestens zum Zeitpunkt t+w(u,v) entdeckt
- Unter Umständen wird v eher über einen anderen Knoten entdeckt

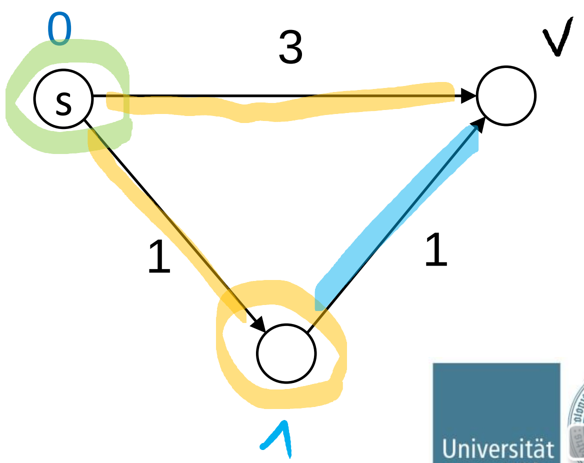


Wird zu Zeitpunkt ≤t+3 entdeckt

Graphenalgorithmen

Simulation der Breitensuche

- Simuliere die Breitensuche schrittweise, wobei jeder Zeitschritt der Abarbeitung einer Entfernungsebene entspricht (alle Knoten mit Distanz k werden bearbeitet)
- Priorität eines Knotens ist der nächste bekannte Zeitpunkt, an dem die Breitensuche diesen Knoten erreicht
- Im Laufe des Algorithmus können sich die Prioritäten verändern
- Startet z.B. der Algorithmus die simulierte Breitensuche im Graph rechts bei s, so wird die Priorität von v zunächst auf 3 gesetzt. Wird der untere Knoten entdeckt, so wird sie auf 2 reduziert



Graphenalgorithmen

Datenstruktur Prioritätenschlange

- Einfügen, Löschen
- ExtractMin: Entfernt Objekt mit der kleinsten Priorität aus der Prioritätenschlange und gibt diesen zurück
- DecreaseKey(v,p): Verringert die Priorität von Objekt v auf p

Realisierung durch Rot-Schwarz-Bäume

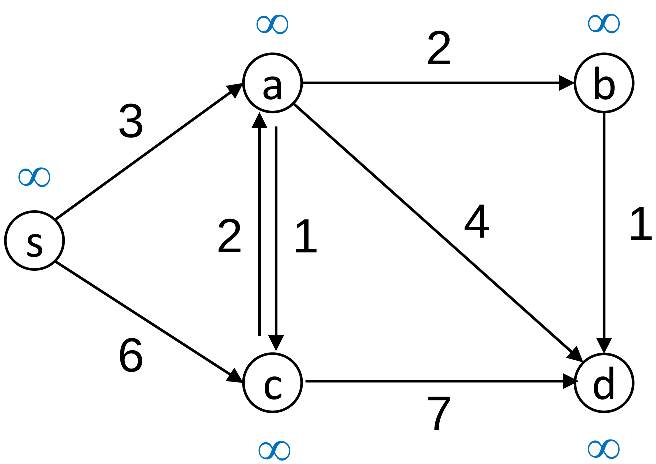
- Einfügen, Löschen, ExtrahiereMinimum, VerringereSchlüssel in O(log n) Zeit
- Bei Gleichheit von Schlüsseln wird Sortierung durch Zusatzinformation bestimmt (z.B. Nr. des zugehörigen Knotens)

Graphenalgorithmen

d[u]=∞ für alle u∈V
prio[u]=∞
prio[s]=0
color[u]=weiß für alle u∈V

BreitensucheSimulation(G,w,s)

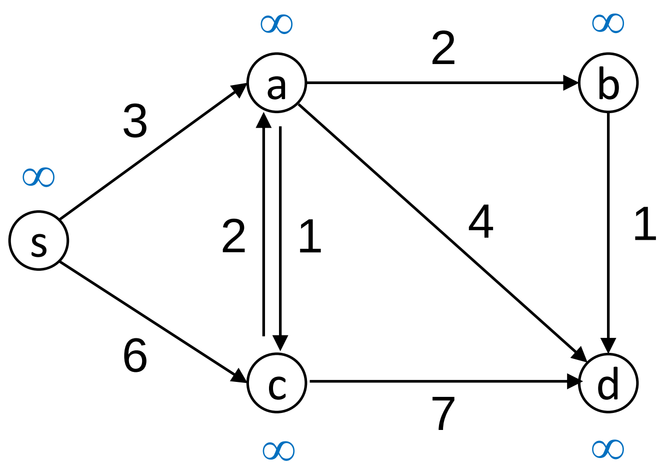
1. Initialisiere Simulation
2. Füge (s,prio[s]) mit Priorität prio[s] in Prioritätenschlange Q ein
3. **while** Q≠∅ **do**
4. (u, prio[u]) = ExtractMin(Q)
5. **if** color[u] = weiß **then**
6. color[u] = schwarz
7. d[u] = prio[u]
8. **for each** v∈Adj[u] **do**
9. prio[v] = d[u] + w(u,v)
10. Füge (v,prio[v]) mit Priorität prio[v] in Q ein



Q:

Graphenalgorithmen

```
BreitensucheSimulation(G,w,s)
Initialisiere Simulation
Füge (s,prio[s]) mit Priorität prio[s] in Prioritätenschlange Q ein
while Q≠∅ do
    (u, prio[u]) = ExtractMin(Q)
    if color[u] = weiß then
        color[u] = schwarz
        d[u] = prio[u]
        for each v∈Adj[u] do
            prio[v] = d[u] + w(u,v)
            Füge (v,prio[v]) mit Priorität prio[v] in Q ein
```

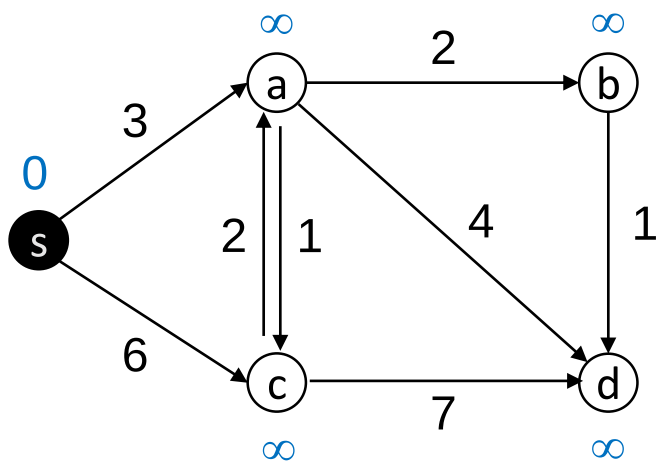


Q: (s,0)

Graphenalgorithmen

BreitensucheSimulation(G,w,s)

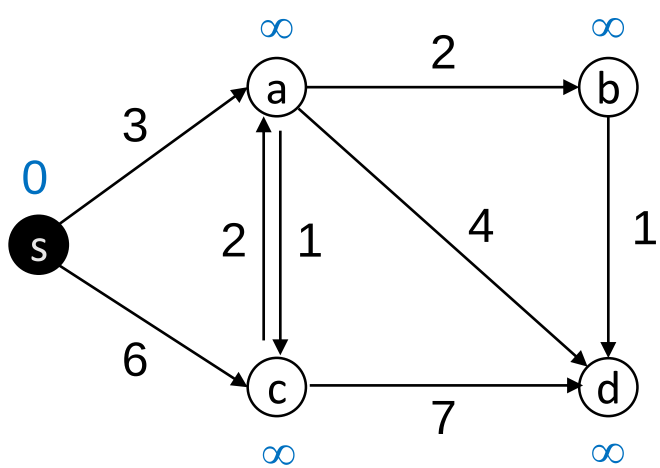
1. Initialisiere Simulation
2. Füge (s,prio[s]) mit Priorität prio[s] in Prioritätenschlange Q ein
3. **while** Q≠∅ **do**
4. (u, prio[u]) = ExtractMin(Q)
5. **if** color[u] = weiß **then**
6. color[u] = schwarz
7. d[u] = prio[u]
8. **for each** v∈Adj[u] **do**
9. prio[v] = d[u] + w(u,v)
10. Füge (v,prio[v]) mit Priorität prio[v] in Q ein



Q:

Graphenalgorithmen

```
BreitensucheSimulation(G,w,s)
Initialisiere Simulation
Füge (s,prio[s]) mit Priorität prio[s] in Prioritätenschlange Q ein
while Q≠∅ do
    (u, prio[u]) = ExtractMin(Q)
    if color[u] = weiß then
        color[u] = schwarz
        d[u] = prio[u]
        for each v∈Adj[u] do
            prio[v] = d[u] + w(u,v)
            Füge (v,prio[v]) mit Priorität prio[v] in Q ein
```

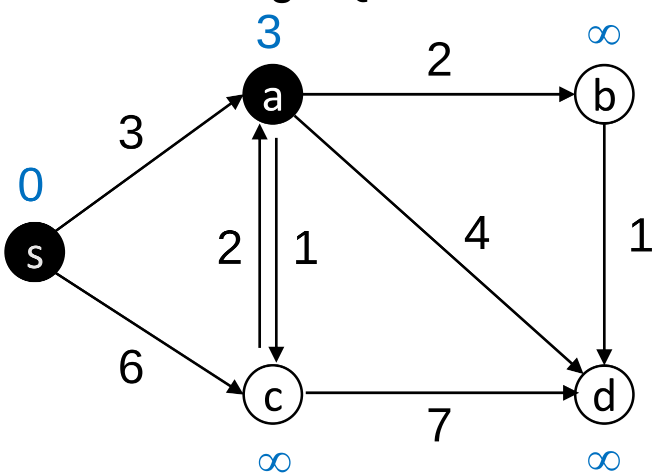


Q: (a,3), (c,6)

Graphenalgorithmen

BreitensucheSimulation(G,w,s)

1. Initialisiere Simulation
2. Füge (s,prio[s]) mit Priorität prio[s] in Prioritätenschlange Q ein
3. **while** Q≠∅ **do**
4. (u, prio[u]) = ExtractMin(Q)
5. **if** color[u] = weiß **then**
6. color[u] = schwarz
7. d[u] = prio[u]
8. **for each** v∈Adj[u] **do**
9. prio[v] = d[u] + w(u,v)
10. Füge (v,prio[v]) mit Priorität prio[v] in Q ein

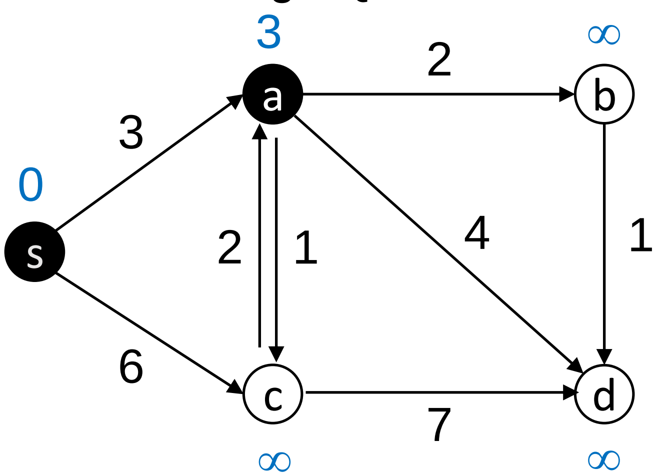


Q: (c,6)

Graphenalgorithmen

BreitensucheSimulation(G,w,s)

1. Initialisiere Simulation
2. Füge (s,prio[s]) mit Priorität prio[s] in Prioritätenschlange Q ein
3. **while** Q≠∅ **do**
4. (u, prio[u]) = ExtractMin(Q)
5. **if** color[u] = weiß **then**
6. color[u] = schwarz
7. d[u] = prio[u]
8. **for each** v∈Adj[u] **do**
9. prio[v] = d[u] + w(u,v)
10. Füge (v,prio[v]) mit Priorität prio[v] in Q ein

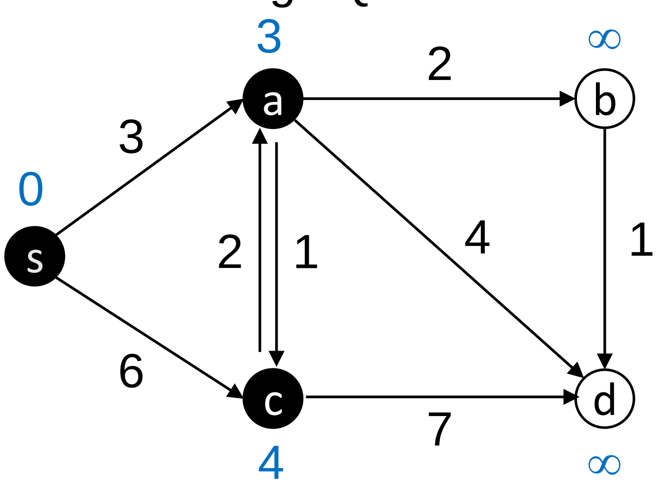


Q: (c,4), (b,5), (c,6), (d,7)

Graphenalgorithmen

BreitensucheSimulation(G,w,s)

1. Initialisiere Simulation
2. Füge (s,prio[s]) mit Priorität prio[s] in Prioritätenschlange Q ein
3. **while** Q≠∅ **do**
4.   (u, prio[u]) = ExtractMin(Q)
5.   **if** color[u] = weiß **then**
6.   color[u] = schwarz
7.   d[u] = prio[u]
8.   **for each** v∈Adj[u] **do**
9.   prio[v] = d[u] + w(u,v)
10.   Füge (v,prio[v]) mit Priorität prio[v] in Q ein

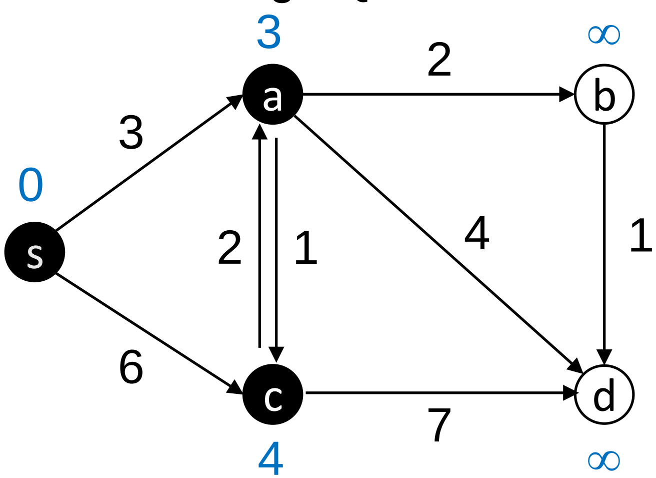


Q: (b,5), (c,6), (d,7)

Graphenalgorithmen

BreitensucheSimulation(G,w,s)

1. Initialisiere Simulation
2. Füge (s,prio[s]) mit Priorität prio[s] in Prioritätenschlange Q ein
3. **while** Q≠∅ **do**
4. (u, prio[u]) = ExtractMin(Q)
5. **if** color[u] = weiß **then**
6. color[u] = schwarz
7. d[u] = prio[u]
8. **for each** v∈Adj[u] **do**
9. prio[v] = d[u] + w(u,v)
10. Füge (v,prio[v]) mit Priorität prio[v] in Q ein

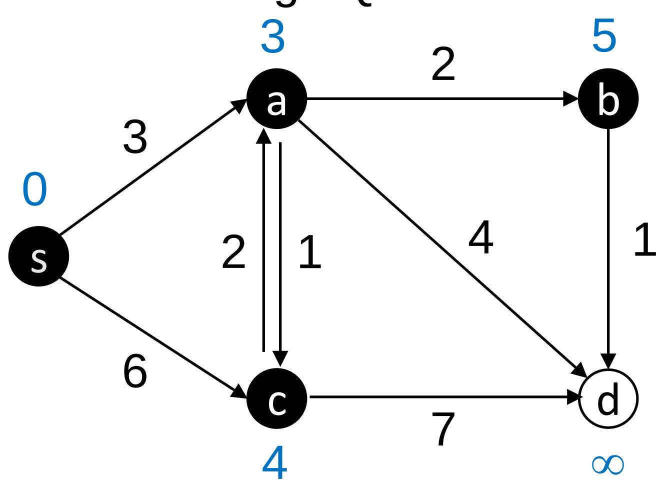


Q: (b,5), (a,6), (c,6), (d,7), (d,13)

Graphenalgorithmen

BreitensucheSimulation(G,w,s)

1. Initialisiere Simulation
2. Füge (s,prio[s]) mit Priorität prio[s] in Prioritätenschlange Q ein
3. **while** Q≠∅ **do**
4.   (u, prio[u]) = ExtractMin(Q)
5.   **if** color[u] = weiß **then**
6.   color[u] = schwarz
7.   d[u] = prio[u]
8.   **for each** v∈Adj[u] **do**
9.   prio[v] = d[u] + w(u,v)
10.   Füge (v,prio[v]) mit Priorität prio[v] in Q ein

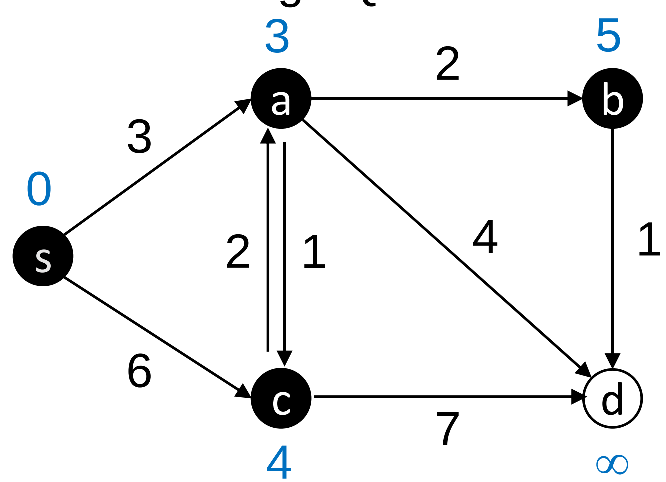


Q: (a,6), (c,6), (d,7), (d,13)

Graphenalgorithmen

BreitensucheSimulation(G,w,s)

1. Initialisiere Simulation
2. Füge (s,prio[s]) mit Priorität prio[s] in Prioritätenschlange Q ein
3. **while** $Q \neq \emptyset$ **do**
4.   (u, prio[u]) = ExtractMin(Q)
5.   **if** color[u] = weiß **then**
6.   color[u] = schwarz
7.   d[u] = prio[u]
8.   **for each** $v \in Adj[u]$ **do**
9.   prio[v] = d[u] + w(u,v)
10.   Füge (v,prio[v]) mit Priorität prio[v] in Q ein

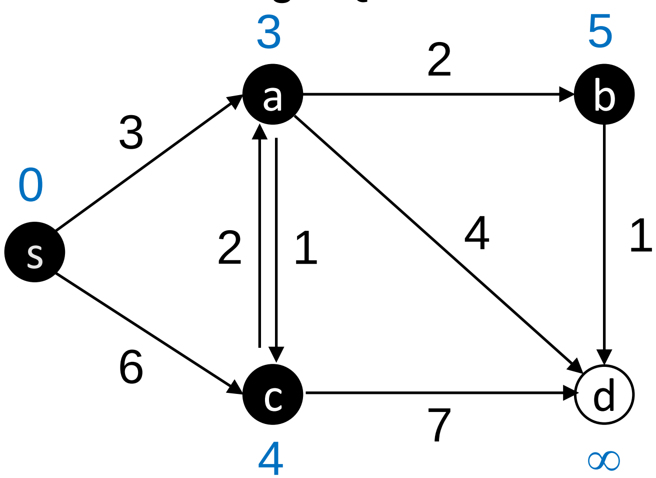


Q: (a,6), (c,6), (d,6), (d,7), (d,13)

Graphenalgorithmen

BreitensucheSimulation(G,w,s)

1. Initialisiere Simulation
2. Füge (s,prio[s]) mit Priorität prio[s] in Prioritätenschlange Q ein
3. **while** Q≠∅ **do**
4.   (u, prio[u]) = ExtractMin(Q)
5.   **if** color[u] = weiß **then**
6.   color[u] = schwarz
7.   d[u] = prio[u]
8.   **for each** v∈Adj[u] **do**
9.   prio[v] = d[u] + w(u,v)
10.   Füge (v,prio[v]) mit Priorität prio[v] in Q ein

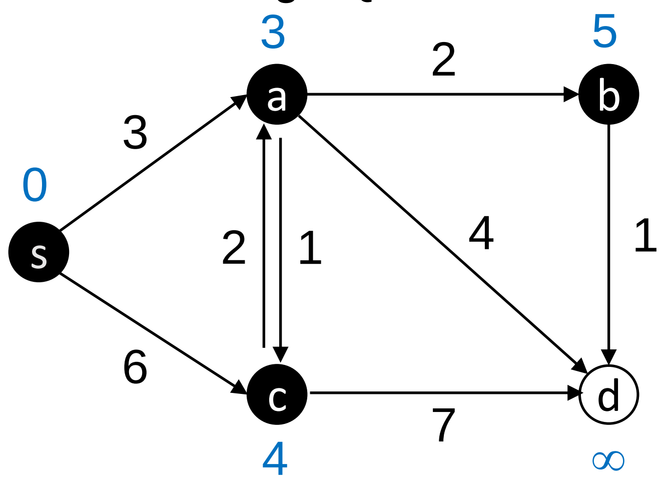


Q: (c,6), (d,6), (d,7), (d,13)

Graphenalgorithmen

BreitensucheSimulation(G,w,s)

1. Initialisiere Simulation
2. Füge (s,prio[s]) mit Priorität prio[s] in Prioritätenschlange Q ein
3. **while** $Q \neq \emptyset$ **do**
4. (u, prio[u]) = ExtractMin(Q)
5. **if** color[u] = weiß **then**
6. color[u] = schwarz
7. d[u] = prio[u]
8. **for each** $v \in Adj[u]$ **do**
9. prio[v] = d[u] + w(u,v)
10. Füge (v,prio[v]) mit Priorität prio[v] in Q ein

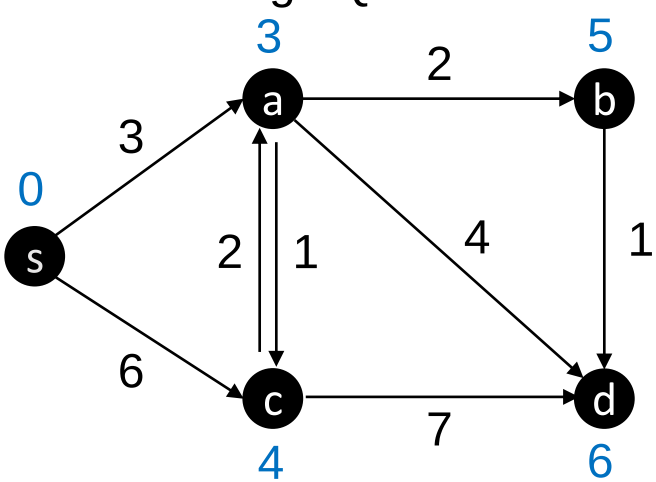


Q: (d,6), (d,7), (d,13)

Graphenalgorithmen

BreitensucheSimulation(G,w,s)

1. Initialisiere Simulation
2. Füge (s,prio[s]) mit Priorität prio[s] in Prioritätenschlange Q ein
3. **while** Q≠∅ **do**
4.   (u, prio[u]) = ExtractMin(Q)
5.   **if** color[u] = weiß **then**
6.   color[u] = schwarz
7.   d[u] = prio[u]
8.   **for each** v∈Adj[u] **do**
9.   prio[v] = d[u] + w(u,v)
10.   Füge (v,prio[v]) mit Priorität prio[v] in Q ein

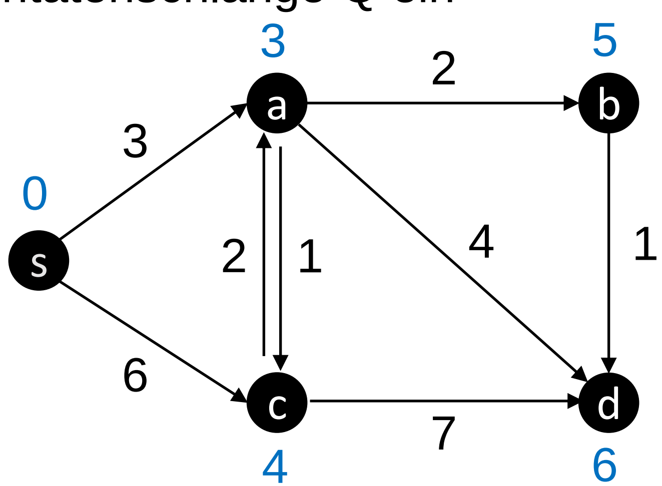


Q: (d,7), (d,13)

Graphenalgorithmen

BreitensucheSimulation(G,w,s)

1. Initialisiere Simulation
2. Füge (s,prio[s]) mit Priorität prio[s] in Prioritätenschlange Q ein
3. **while** Q≠∅ **do**
4. (u, prio[u]) = ExtractMin(Q)
5. **if** color[u] = weiß **then**
6. color[u] = schwarz
7. d[u] = prio[u]
8. **for each** v∈Adj[u] **do**
9. prio[v] = d[u] + w(u,v)
10. Füge (v,prio[v]) mit Priorität prio[v] in Q ein

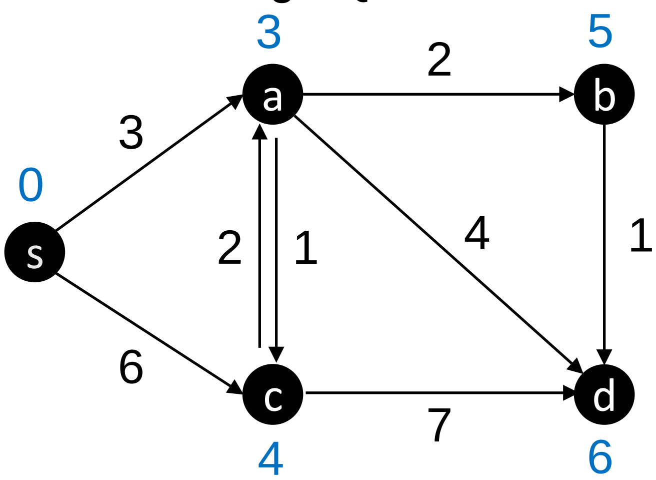


Q: (d,13)

Graphenalgorithmen

BreitensucheSimulation(G,w,s)

1. Initialisiere Simulation
2. Füge (s,prio[s]) mit Priorität prio[s] in Prioritätenschlange Q ein
3. **while** Q≠∅ **do**
4. (u, prio[u]) = ExtractMin(Q)
5. **if** color[u] = weiß **then**
6. color[u] = schwarz
7. d[u] = prio[u]
8. **for each** v∈Adj[u] **do**
9. prio[v] = d[u] + w(u,v)
10. Füge (v,prio[v]) mit Priorität prio[v] in Q ein



Q:

Graphenalgorithmen

Beobachtung:
Sind mehrere Paare (u,p) in der Prioritätenschlange, so ist nur das mit der geringsten Priorität relevant

BreitensucheSimulation(G,w,s)

1. Initialisiere Simulation
2. Füge (s,prio[s]) mit Priorität prio[s] in Prioritätenschlange Q ein
3. **while** Q$\neq\emptyset$ **do**
4. (u, prio[u]) = ExtractMin(Q)
5. **if** color[u] = weiß **then**
6. color[u] = schwarz
7. d[u] = prio[u]
8. **for each** v$\in$Adj[u] **do**
9. prio[v] = d[u] + w(u,v)
10. Füge (v,prio[v]) mit Priorität prio[v] in Q ein

Beobachtung:
d-Werte und Prioritäten sind fast identisch

Graphenalgorithmen

Dijkstra‘s Algorithmus(G,w,s)

1. Initialisiere SSSP
2. Q = V[G]
3. **while** Q≠∅ **do**
4. u = ExtractMin(Q)
5. **for each** v∈Adj[u] **do**
6. **if** d[u] + w(u,v)< d[v] **then**
7. d[v] = d[u] + w(u,v)
8. DecreaseKey(v,d[v])
9. π[v] = u
10. color[u] = schwarz

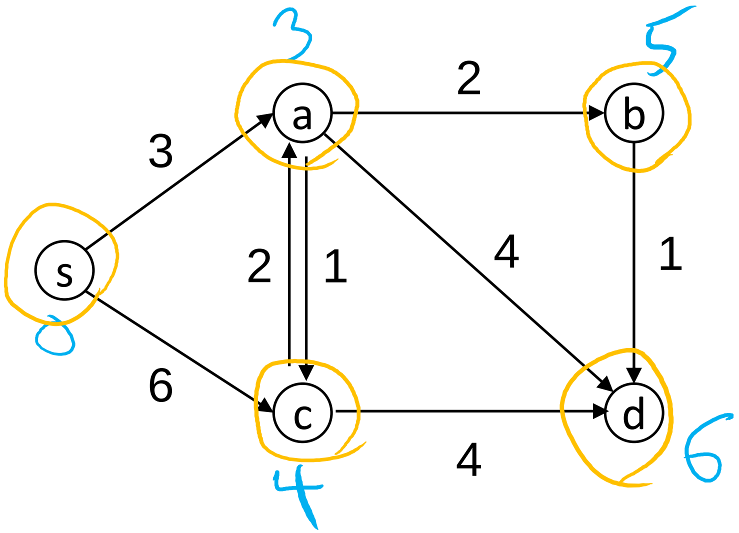
d[u]=∞ für alle u∈V-{s}
d[s]=0
color[u]=weiß für alle u∈V

Invariante:
Für alle schwarzen Knoten wurde die Distanz korrekt berechnet

Graphalgorithmen

Aufgabe

- Gehen Sie Dijkstras Algorithmus auf folgendem Graph durch



Graphenalgorithmen

Satz 21.1

- Die Laufzeit unserer Implementierung von Dijkstras Agorithmus ist O((|V|+|E|) log |V|).

Beweis

- Zu Beginn wird jeder Knoten in die Prioritätenschlange eingefügt (O(|V| log |V|) Zeit)
- In jedem Durchlauf der **while**-Schleife wird einmal ExtractMin aufgerufen und ein Knoten aus der Prioritätenschlange entfernt und es werden keine Knoten eingefügt
- Jeder Knoten von G tritt nur einmal als aktueller Knoten u in der Schleife auf
- In der **for**-Schleife werden alle Nachbarn des aktuellen Knotens durchlaufen
- Insgesamt wird die **for**-Schleife $\mathrm{O}(\sum_{v\in V}(1+\deg(v))) = O(|V| + \sum_{v\in V}\deg(v)) = O(|V| + |E|)$ mal durchlaufen

Graphenalgorithmen

Satz 21.1

- Die Laufzeit unserer Implementierung von Dijkstras Agorithmus ist $O((|V|+|E|) \log |V|)$.

Beweis

- In jedem Durchlauf wird maximal einmal DecreaseKey aufgerufen (und sonst nur Operationen mit konstanter Laufzeit)
- Daher ergibt sich insgesamt eine Laufzeit von $O((|V|+|E|)\log |V|)$

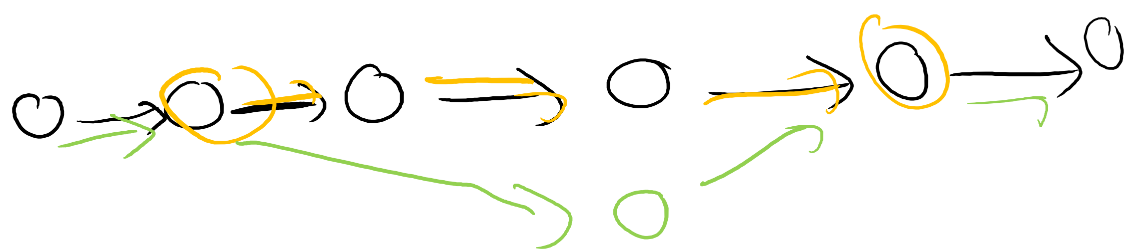
Graphenalgorithmen

Lemma 21.2

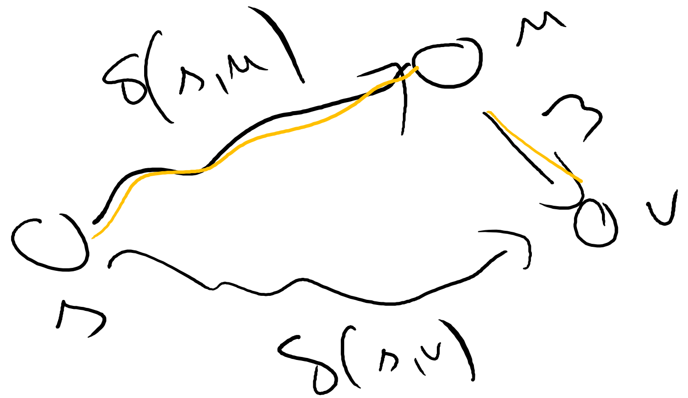
- Sei $G=(V,E)$ ein gewichteter Graph mit Kantengewichten $w(e)$ und sei $\langle v_1,\dots, v_k\rangle$ ein kürzester Weg von v_1 nach v_k. Dann ist für alle $1\leq i<j\leq k$ der Weg $\langle v_i,\dots, v_j\rangle$ ein kürzester Weg von v_i nach v_j.

Beweis

- Annahme: Es gäbe einen kürzeren Weg $\langle v_i, u_1,\dots,u_l ,v_j\rangle$ von v_i nach v_j . Dann wäre $\langle v_1,\dots, v_i, u_1, \dots, u_l , v_j ,\dots, v_k\rangle$ kürzer als $\langle v_1, \dots, v_k\rangle$.
- Widerspruch!



Graphenalgorithmen



Lemma 21.3

- Sei G=(V,E) ein gewichteter Graph und sei $s \in V$ ein beliebiger Knoten. Dann gilt für jede Kante $(u,v) \in E$:

$$\delta(s,v) \le \delta(s,u)+w(u,v)$$

Beweis

- (Argumentation identisch zu Lemma 20.2)
- Ist u erreichbar von s, dann ist es auch v
- Der kürzeste Weg von s nach v kann nicht länger sein, als der kürzeste Weg von s nach u gefolgt von der Kante (u,v). Damit gilt die Ungleichung.
- Ist u nicht erreichbar von s, dann ist $\delta(s,u)=\infty$ und die Ungleichung gilt.

Graphenalgorithmen

Lemma 21.4

- Zu jedem Ausführungszeitpunkt von Dijkstras Algorithmus gilt für jeden Knoten w:

$$d[w] \geq \delta(s,w).$$

Beweis

- Zeile 7 ist die einzige Zeile, in der d-Werte geändert werden.
- Wir zeigen per Induktion über die Ausführungen von Zeile 7, dass das Lemma gilt.
- Induktionsanfang: Vor der ersten Ausführung entsprechen die d-Werte den Werten direkt nach der Initialisierung. Für diese Werte gilt das Lemma.
- Induktionsannahme: Das Lemma gilt nach m Ausführungen von Zeile 7.

Graphenalgorithmen

Lemma 21.4

- Zu jedem Ausführungszeitpunkt von Dijkstras Algorithmus gilt für jeden Knoten w:

$$d[w] \geq \delta(s,w).$$

Beweis

- Induktionsschluss: Betrachte (m+1)ste Ausführung. Nach Induktionsannahme gilt $d[v] \geq \delta(s,v)$ und $d[u] \geq \delta(s,u)$. Wir setzen in Zeile 7 d[v] auf $\min\{d[v], d[u]+w(u,v)\}$.
- Es gilt $d[u] + w(u,v) \geq \delta(s,u) + w(u,v) \geq \delta(s,v)$ nach Lemma 21.3. Somit gilt auch hier das Lemma.

Graphenalgorithmen

Satz 21.5

- Wenn wir Dijkstras Algorithmus auf einem gewichteten Graph $G=(V,E)$ mit nichtnegativen Kantengewichten und Startknoten s ausführen, so gilt nach Terminierung $d[u] = \delta(s,u)$ für alle Knoten $u \in V$.

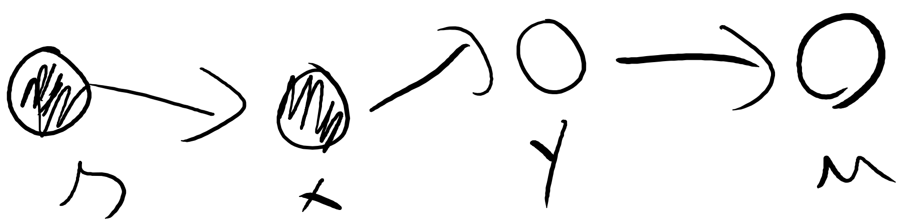
Beweis

- Jeder (erreichbare) Knoten wird im Verlauf des Algorithmus schwarz gefärbt.
- Da die Kantengewichte nichtnegativ sind, ist δ wohldefiniert
- Wir zeigen per Widerspruchsbeweis, dass für jeden Knoten $u \in V$ zum Zeitpunkt des Schwarzfärbens $d[u] = \delta(s,u)$ gilt.

Graphenalgorithmen

Beweis

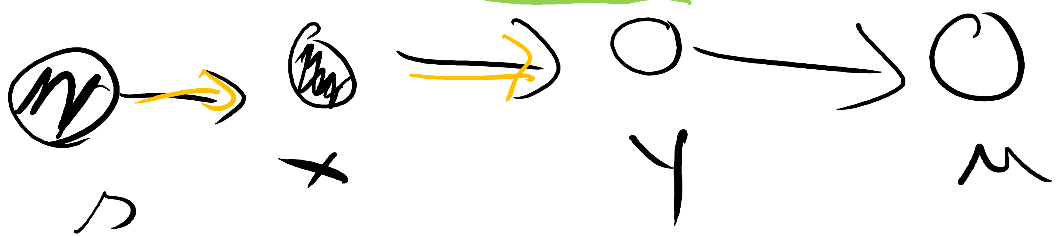
- Annahme: Es gibt einen Knoten u, für den zum Zeitpunkt des Schwarzfärbens $d[u] \neq \delta(s,u)$ gilt. Sei u der erste solche Knoten. Betrachte die Situation zu Beginn des Durchlaufs der **while**-Schleife, in dem u schwarz gefärbt wird. Es gilt $u \neq s$, da s als erster Knoten schwarz gefärbt wird und zu diesem Zeitpunkt $d[s]=0=\delta(s,s)$ gilt. (Widerspruch!)
- Sei nun y der erste weiße Knoten auf einem kürzesten Weg von s nach u und x sein Vorgänger.
- Es gilt $d[x]=\delta(s,x)$ nach Wahl von u.



Graphenalgorithmen

Beweis

- In Zeile 7 wird bei der Abarbeitung von x der Wert d[y] auf min{d[y], d[x]+w(x,y)} gesetzt. Nach Lemma 21.2 ist der Weg von s nach y über x ein kürzester Weg (da er „Teilweg“ des kürzesten Weges von s nach u ist). Somit ist d[x]+w(x,y)=δ(s,y) und d[y] wird auf diesen Wert gesetzt (wegen Lemma 21.3).
- Da die Kantengewichte nichtnegativ sind, folgt $\delta(s,y) \le \delta(s,u)$ und somit nach Lemma 21.3 $d[y]= \delta(s,y) \le \delta(s,u) \le d[u]$.
- Da aber u von ExtractMin aus der Prioritätenschlange entfernt wurde, gilt $d[u] \le d[y]$ und somit $d[y]= \delta(s,y) = \delta(s,u) = d[u]$. Widerspruch!



Graphenalgorithmen

Zusammenfassung

- Der Algorithmus von Dijkstra kann dazu genutzt werden, um das SSSP Problem in gewichteten Graphen mit nichtnegativen Kantengewichten zu lösen
- Dijkstras Algorithmus kann als Erweiterung der Breitensuche interpretiert werden
- Die Laufzeit von Dijkstras Algorithmus ist $O((|V| + |E|) \log |V|)$, wenn die Prioritätenschlange als Rot-Schwarz-Baum implementiert wird

Referenzen

- T. Cormen, C. Leisserson, R. Rivest, C. Stein. Introduction to Algorithms. The MIT press. Second edition, 2001.