Kürzeste Wegealgorithmen

TECHNISCHE UNIVERSITÄT DARMSTADT

Berechnung kürzester Wege

- Ein gewichteter Graph G ist ein Tupel (V,E) zusammen mit einer Gewichtsfunktion f, wobei $E \subseteq V \times V$ und $f: E \rightarrow \mathbb{Q}$
- Seien $u,v \in V$. Ein kürzester Weg von u nach v ist ein Weg mit kleinstmöglichem Gewicht von u nach v. Das Gewicht eines Weges ist dabei die Summe der Kantengewichte auf diesem Weg.

Kürzeste Wegealgorithmen

TECHNISCHE UNIVERSITÄT DARMSTADT

Berechnung kürzester Wege

- Kürzeste Wege von einem Startknoten s ausgehend (Single Source Shortest Path Problem)
  - gegeben: Gewichteter Graph G und Startknoten s
  - gesucht: für alle Knoten v die Distanz δ(s,v) sowie ein kürzester Weg

- Kürzeste Wege zwischen allen Knotenpaaren

- Satz: Sei G ein gerichteter Graph mit nicht-negativen Kantengewichten. Wenn ein Weg w von s nach t ein kürzester Weg ist, und v ein Knoten auf dem Weg, sind auch die Wege von s nach v und von v nach t kürzeste Wege.

  Argument: Wenn es einen kürzeren Weg w‘ von s nach v gäbe, könnte man mit seiner Hilfe einen kürzeren Weg als w von s nach t erzeugen.

Kürzeste Wegealgorithmen

TECHNISCHE UNIVERSITÄT DARMSTADT

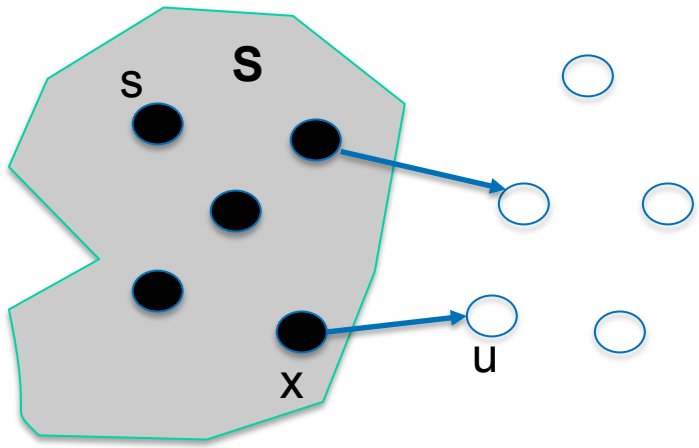
Dijkstras Algorithmus

- Bezeichnungen
  - gerichteter Graph G=(V,E), alle Kantengewichte positiv
  - G ist repräsentiert durch Adjazenzlisten
  - Startknoten s, Zielknoten t
  - S Menge von Knoten, deren endgültige Entfernung zu s schon bekannt ist
  - dist[v] Distanzvariable von v
  - π[v] Vorgängerknoten von v. In π lässt sich am Ende der Weg ablesen.
  - A = V \ S

TECHNISCHE UNIVERSITÄT DARMSTADT

Dijkstras Algorithmus

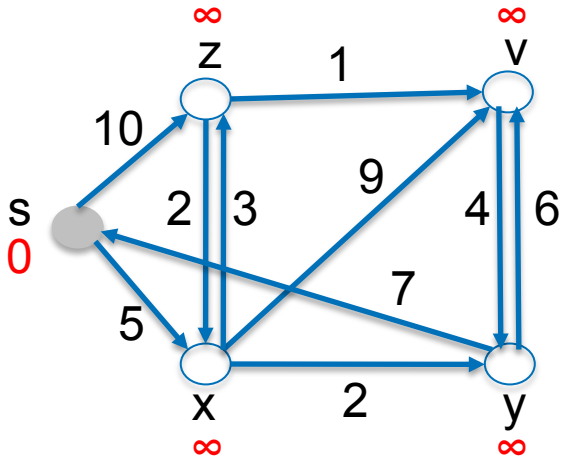
```
Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
S := ∅;
A := V;
while   A ≠ ∅  do
    u := argmin{ dist[a] | a∈A }
    S := S ∪ {u};
    for each node v ∈ Adj[u] do
        if dist[v] > dist[u] + f(u,v) then
            dist[v] := dist[u] + f(u,v);
            π[v] := u;
```



TECHNISCHE UNIVERSITÄT DARMSTADT

Dijkstras Algorithmus

```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: S := ∅;
 3: A := V;
 4: while   A ≠ ∅  do
 5:    u := argmin{ dist[a] | a∈A }; A:= A \ {u}
 6:    S := S ∪ {u};
 7:    for each node v ∈ Adj[u] do
 8:       if dist[v] > dist[u] + f(u,v) then
 9:          dist[v] := dist[u] + f(u,v);
10:          π[v] := u;
```



u aus Zeile 5

A

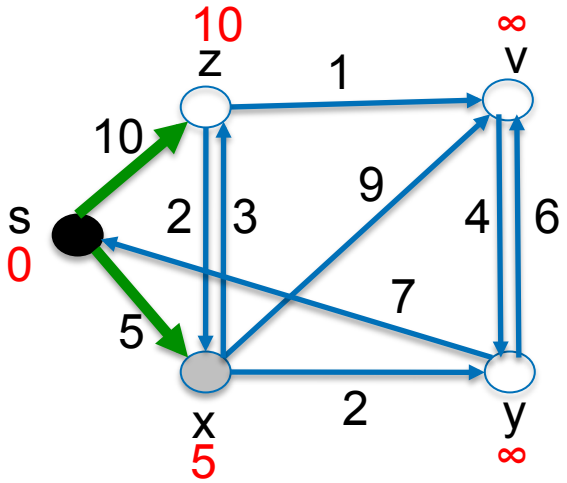
S

π[v]=u

TECHNISCHE UNIVERSITÄT DARMSTADT

Dijkstras Algorithmus

```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: S := ∅;
 3: A := V;
 4: while   A ≠ ∅  do
 5:    u := argmin{ dist[a] | a∈A }; A:= A \ {u}
 6:    S := S ∪ {u};
 7:    for each node v ∈ Adj[u] do
 8:       if dist[v] > dist[u] + f(u,v) then
 9:          dist[v] := dist[u] + f(u,v);
10:          π[v] := u;
```



u aus Zeile 5

A

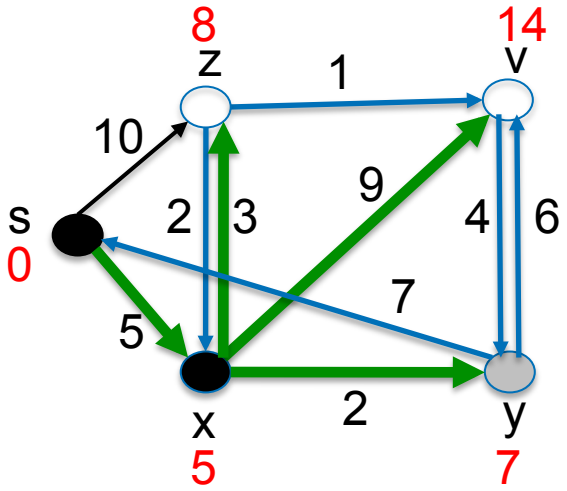
S

π[v]=u

TECHNISCHE UNIVERSITÄT DARMSTADT

Dijkstras Algorithmus

```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: S := ∅;
 3: A := V;
 4: while   A ≠ ∅  do
 5:    u := argmin{ dist[a] | a∈A }; A:= A \ {u}
 6:    S := S ∪ {u};
 7:    for each node v ∈ Adj[u] do
 8:       if dist[v] > dist[u] + f(u,v) then
 9:          dist[v] := dist[u] + f(u,v);
10:          π[v] := u;
```



u aus Zeile 5

A

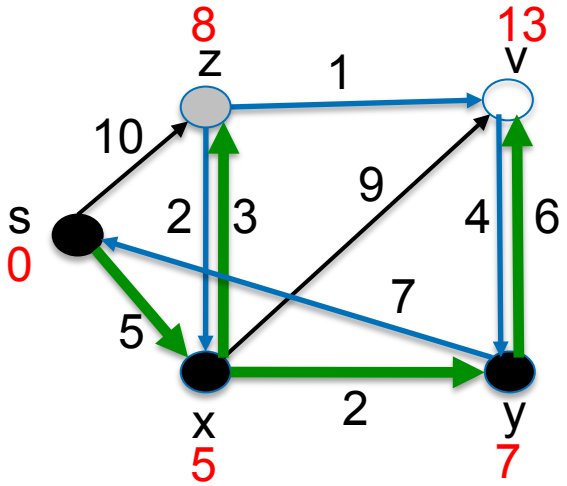
S

π[v]=u

TECHNISCHE UNIVERSITÄT DARMSTADT

Dijkstras Algorithmus

```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: S := ∅;
 3: A := V;
 4: while   A ≠ ∅  do
 5:    u := argmin{ dist[a] | a∈A }; A:= A \ {u}
 6:    S := S ∪ {u};
 7:    for each node v ∈ Adj[u] do
 8:       if dist[v] > dist[u] + f(u,v) then
 9:          dist[v] := dist[u] + f(u,v);
10:          π[v] := u;
```



u aus Zeile 5

A

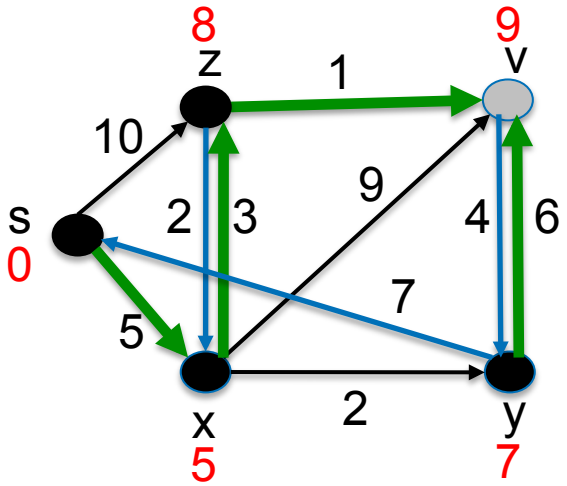
S

π[v]=u

TECHNISCHE UNIVERSITÄT DARMSTADT

Dijkstras Algorithmus

```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: S := ∅;
 3: A := V;
 4: while   A ≠ ∅  do
 5:    u := argmin{ dist[a] | a∈A }; A:= A \ {u}
 6:    S := S ∪ {u};
 7:    for each node v ∈ Adj[u] do
 8:       if dist[v] > dist[u] + f(u,v) then
 9:          dist[v] := dist[u] + f(u,v);
10:          π[v] := u;
```



u aus Zeile 5

A

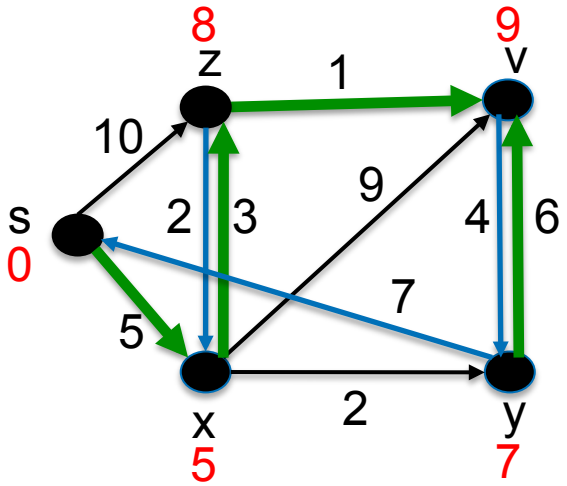
S

π[v]=u

TECHNISCHE UNIVERSITÄT DARMSTADT

Dijkstras Algorithmus

```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: S := ∅;
 3: A := V;
 4: while   A ≠ ∅  do
 5:    u := argmin{ dist[a] | a∈A }; A:= A \ {u}
 6:    S := S ∪ {u};
 7:    for each node v ∈ Adj[u] do
 8:       if dist[v] > dist[u] + f(u,v) then
 9:          dist[v] := dist[u] + f(u,v);
10:          π[v] := u;
```



u aus Zeile 5

A

S

π[v]=u

Kürzeste Wegealgorithmen

TECHNISCHE UNIVERSITÄT DARMSTADT

Dijkstras Algorithmus

```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: S := ∅;
 3: A := V;
 4: while   A ≠ ∅  do
 5:    u := argmin{ dist[a] | a∈A }
 6:    S := S ∪ {u};
 7:    for each node v ∈ Adj[u] do
 8:       if dist[v] > dist[u] + f(u,v) then
 9:          dist[v] := dist[u] + f(u,v);
10:          π[v] := u;
```

Laufzeit: $O(|E| \cdot O(\text{Zeile 9}) + |V| \cdot O(\text{Zeile 5}))$
Einfach: $O(|E| + |V| \cdot |V|) = O(|V|^2)$

Kürzeste Wegealgorithmen

TECHNISCHE UNIVERSITÄT DARMSTADT

Dijkstras Algorithmus, Korrektheit

Dijkstras Algorithmus

```
 1: Initialize(G,s)
 2: S := ∅;
 3: A := V;
 4: while   A ≠ ∅  do
 5:    u := argmin{ dist[a] | a∈A }; A:= A \ {u}
 6:    S := S ∪ {u};
 7:    for each node v ∈ Adj[u] do
 8:       if dist[v] > dist[u] + f(u,v) then
 9:          dist[v] := dist[u] + f(u,v);
10:          π[v] := u;
```

- Lemma Dijk1: dist[v] ≥ δ(s,v)
  Bew. Über Induktion über die Anzahl Aufrufe der Zeilen 8 u. 9
  - IA: Zu Beginn ist dist[s] = 0 und dist[u] = ∞ für alle anderen u.
  - IV: Beh. gilt bis zum k-ten Aufruf der Zeilen 8 u. 9
  - IS: Sei v der erste Knoten, für den die Beh. nicht gilt, erzeugt durch den (k+1)-ten Aufruf der Zeilen 8 u. 9.
    Dann gilt nach diesem (k+1)-ten Aufruf:
    dist[u]+f(u,v) = dist[v] < δ(s,v) [nach Annahme]. Und es gilt
    δ(s,v) ≤ δ(s,u) + f(u,v) [Kürzeste-Wege-Eigenschaft]
    Insgesamt also dist[u]+f(u,v) < δ(s,u) + f(u,v). Da aber dist[u] im (k+1)-ten Aufruf gar nicht verändert wurde, galt schon vorher dist[u]< δ(s,u).
    → Annahme dist[v] < δ(s,v) für v war falsch.

Dijkstras Algorithmus, Korrektheit

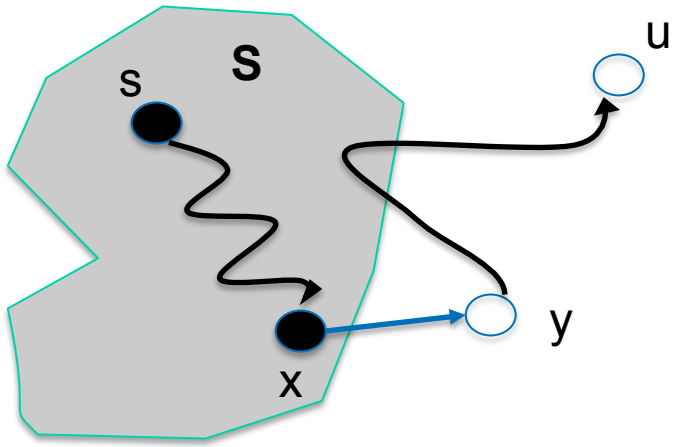
Satz DijkKor: Wenn man Dijksta‘s Algorithmus auf einem gewichteten, gerichteten Graphen mit nicht-negativer Gewichtsfunktion w und Startknoten s laufen läßt, gilt für alle Knoten $u \in S$: $dist[u] = \delta(s,u)$.

Beweis: Widerspruchsargument: Sei u der erste Knoten, für den gilt $dist[u] \neq \delta(s,u)$, wenn er in S eingefügt wird. [Der Beweis ist fertig, wenn wir zeigen konnten, dass für dieses erste u doch gilt $dist[u] = \delta(s,u)$.]

Nun:
- $u \neq s$, denn dist[s] wird korrekt eingefügt und nicht mehr verändert.
- wegen $u \neq s$ gilt direkt vorm Einfügen von u, dass $S \neq \varnothing$
- zudem muss es einen Weg von s nach u geben (denn sonst gilt $dist[u] = \delta(s,u) = \infty$)
    - also gibt es einen kürzesten Pfad p von s nach u.

Dijkstras Algorithmus, Korrektheit

• p verbindet den Knoten s in S mit einem Knoten u in A (=V \ S). Sei y der erste Knoten entlang p so dass y∈V\S und sei x der Vorgänger von y. p lässt sich aufteilen in p_1 und p_2, so dass p_1 komplett in S liegt.

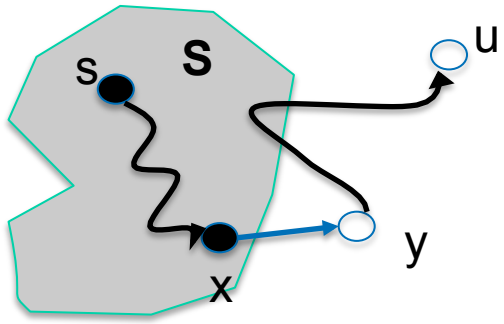


Kürzeste Wegealgorithmen

TECHNISCHE UNIVERSITÄT DARMSTADT

Dijkstras Algorithmus, Korrektheit

- Lemma Dijk2: dist[y] ≤ δ(s,y)
  Bew.
  Da x∈S und y im kürzesten Weg vor u liegt, wurde y mit Hilfe der Zeilen 8 u. 9 auf den endgültigen Wert gesetzt, bevor dies für u passierte.



Dijkstras Algorithmus

```
 1: Initialize(G,s)
 2: S := ∅;
 3: A := V;
 4: while   A ≠ ∅  do
 5:    u := argmin{ dist[a] | a∈A }; A:= A \ {u}
 6:    S := S ∪ {u};
 7:    for each node v ∈ Adj[u] do
 8:       if dist[v] > dist[u] + f(u,v) then
 9:          dist[v] := dist[u] + f(u,v);
10:          π[v] := u;
```

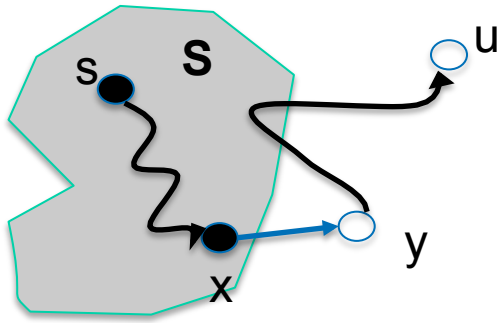
Damit gilt aber: dist[y] ≤ dist[x] + f(x,y) [wegen Zeilen 8 u. 9]
= δ(s,x) + f(x,y) [wegen x∈S und Annahme in Satz DijkKor]
= δ(s,y), weil s → x → y kürzester Weg ist ✔

TECHNISCHE UNIVERSITÄT DARMSTADT

Dijkstras Algorithmus, Korrektheit

- Lemma Dijk3: Mit Hilfe der Lemmata Dijk1 und Dijk2 wissen wir also für die gegebene Situation:

  In dem Moment, wenn u in S eingefügt wird gilt dist[y] = δ(s,y).



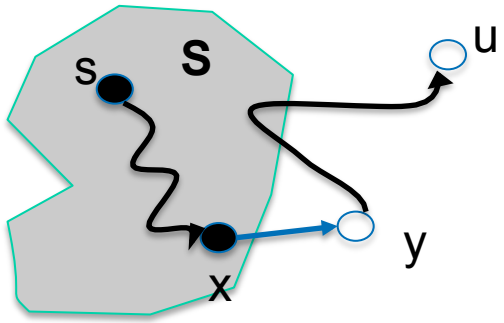
Dijkstras Algorithmus

```
 1: Initialize(G,s)
 2: S := ∅;
 3: A := V;
 4: while   A ≠ ∅  do
 5:    u := argmin{ dist[a] | a∈A }; A:= A \ {u}
 6:    S := S ∪ {u};
 7:    for each node v ∈ Adj[u] do
 8:       if dist[v] > dist[u] + f(u,v) then
 9:          dist[v] := dist[u] + f(u,v);
10:          π[v] := u;
```

Kürzeste Wegealgorithmen

TECHNISCHE UNIVERSITÄT DARMSTADT

Dijkstras Algorithmus, Korrektheit



Dijkstras Algorithmus

```
 1: Initialize(G,s)
 2: S := ∅;
 3: A := V;
 4: while   A ≠ ∅  do
 5:    u := argmin{ dist[a] | a∈A }; A:= A \ {u}
 6:    S := S ∪ {u};
 7:    for each node v ∈ Adj[u] do
 8:       if dist[v] > dist[u] + f(u,v) then
 9:          dist[v] := dist[u] + f(u,v);
10:          π[v] := u;
```

- Nun gilt:
  dist[y] = δ(s,y) ≤ δ(s,u) ≤ dist[u] und dist[u] ≤ dist[y] wegen Zeile 5 und y,u∈A

  kürzester Weg | Lemma Dijk1

  insgesamt also dist[u] ≤ dist[y] ≤ δ(s,u) ≤ dist[u],
  also dist[u] = δ(s,u) → Widerspruch zur Annahme in Satz DijkKor

Kürzeste Wegealgorithmen

TECHNISCHE UNIVERSITÄT DARMSTADT

Nochmal Dijkstras Algorithmus

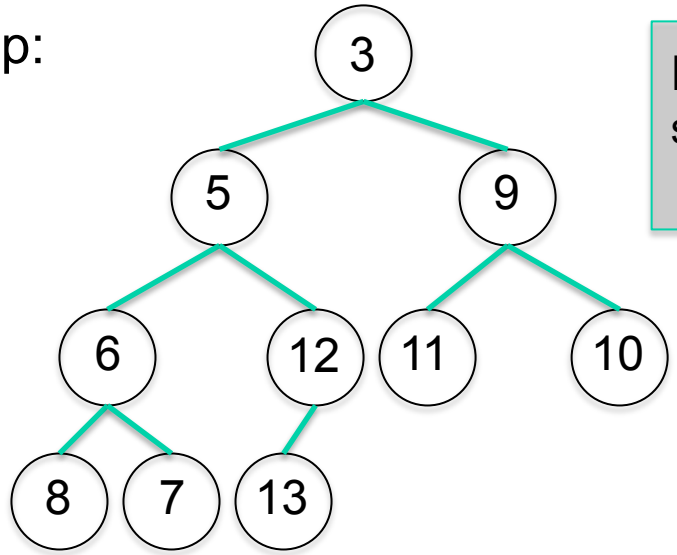
```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: S := ∅;
 3: A := V;
 4: while   A ≠ ∅  do
 5:    u := argmin{ dist[a] | a∈A }
 6:    S := S ∪ {u};
 7:    for each node v ∈ Adj[u] do
 8:       if dist[v] > dist[u] + f(u,v) then
 9:          dist[v] := dist[u] + f(u,v);
10:          π[v] := u;
```

Laufzeit: $O(|E| \cdot O(\text{Zeile 9}) + |V| \cdot O(\text{Zeile 5}))$
Mit Hilfe von Heaps (neuer abstrakter Datentyp wie Queue oder Stack) kann erreicht werden: $O(|E| \cdot \log(|V|) + |V| \cdot \log(|V|))$ und sogar $O(|E| + |V| \cdot \log(|V|)$

Die Heap Datenstruktur

TECHNISCHE UNIVERSITÄT DARMSTADT

Ein Heap:



H:

3	5	9	6	12	11	10	8	7	13
1	2	3	4	5	6	7	8	9	10

size:10

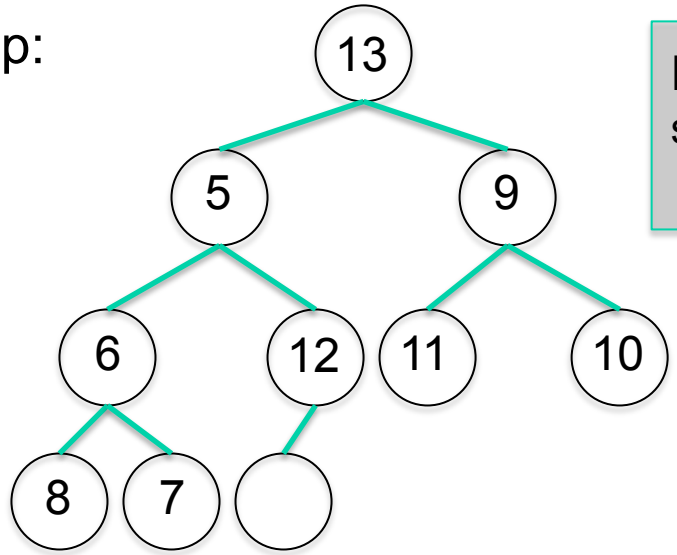
Parent(i): return $\lfloor i/2 \rfloor$
Left(i): return $2i$
Right(i): return $2i+1$

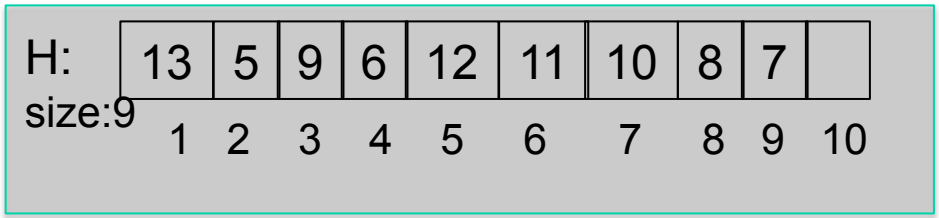
- binärer Baum
- jeder Knoten entspricht einem Element
- Baum wird ebenenweise aufgefüllt
- Baum wird meist in Array gespeichert
- es gilt die Heapeigenschaft: die Werte in den Nachfolgern v_1, v_2 eines Knotens v sind größer als das Element im Knoten v selber

Die Heap Datenstruktur

TECHNISCHE UNIVERSITÄT DARMSTADT

Ein Heap:





Parent(i): return $\lfloor i/2 \rfloor$
Left(i): return 2i
Right(i): return 2i+1

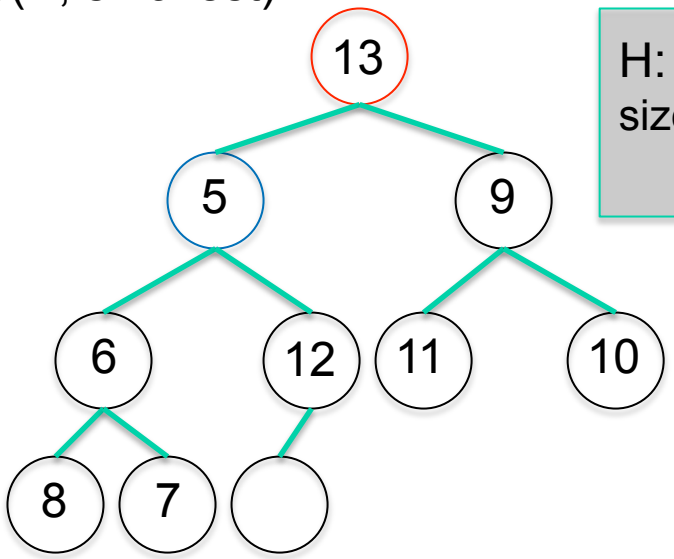
- Operationen sind
  - BuildHeap erstellt aus einer Menge von Elementen einen Heap
  - Insert fügt ein zusätzliches Element ein
  - ExtractMin nimmt das kleinste Element heraus
  - Heapify stellt auf einem Pfad von Wurzel zu einem Blatt die Heapeigenschaft her
  - DecreaseKey(A,i,newkey) verkleinert ein Element und stellt Heapeigenschaft her

Die Heap Datenstruktur

TECHNISCHE UNIVERSITÄT DARMSTADT

Heapify(A,i) **// Start mit i=1**

1. **if** Left(i) ≤ size **and** A[Left(i)] < A[i] **then** smallest := Left(i)
2. **else** smallest := i
3. **if** Right(i) ≤ size **and** A[Right(i)] < A[smallest] **then** smallest := Right(i)
4. **if** smallest ≠ i **then**
5. exchange(A[i], A[smallest])
6. Heapify(A, smallest)



H: size:9

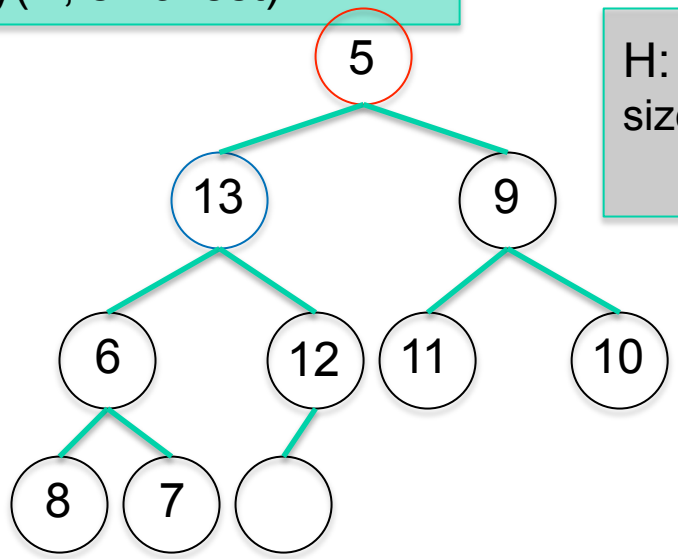
13	5	9	6	12	11	10	8	7	
1	2	3	4	5	6	7	8	9	10

Die Heap Datenstruktur

TECHNISCHE UNIVERSITÄT DARMSTADT

Heapify(A,i) **// Start mit i=1**

1. **if** Left(i) ≤ size **and** A[Left(i)] < A[i] **then** smallest := Left(i)
2. **else** smallest := i
3. **if** Right(i) ≤ size **and** A[Right(i)] < A[smallest] **then** smallest := Right(i)
4. **if** smallest ≠ i **then**
5. exchange(A[i], A[smallest])
6. Heapify(A, smallest)



H: size:9

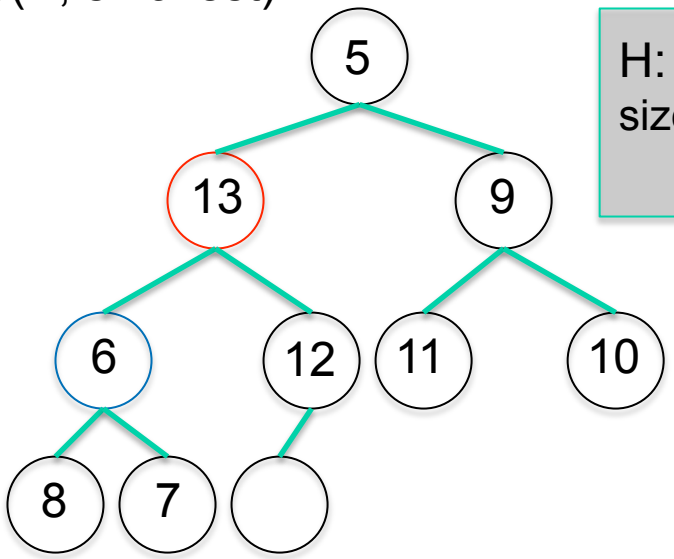
5	13	9	6	12	11	10	8	7	
1	2	3	4	5	6	7	8	9	10

Die Heap Datenstruktur

TECHNISCHE UNIVERSITÄT DARMSTADT

Heapify(A,i) **// jetzt mit i=2**

1. **if** Left(i) ≤ size **and** A[Left(i)] < A[i] **then** smallest := Left(i)
2. **else** smallest := i
3. **if** Right(i) ≤ size **and** A[Right(i)] < A[smallest] **then** smallest := Right(i)
4. **if** smallest ≠ i **then**
5. exchange(A[i], A[smallest])
6. Heapify(A, smallest)



H: size:9

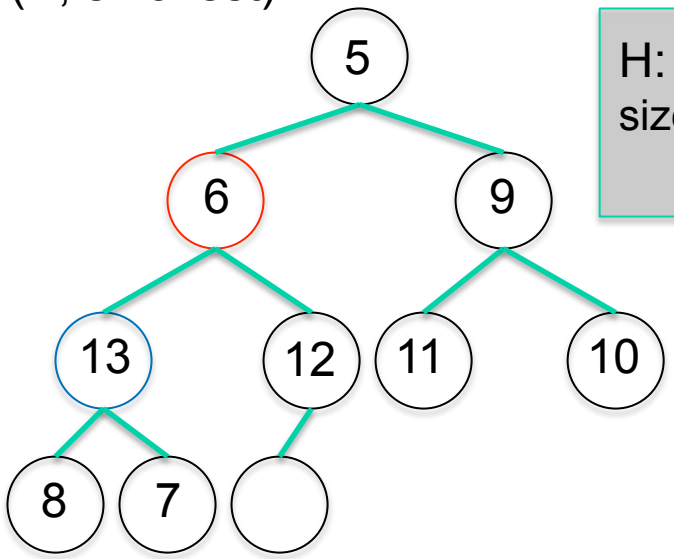
1	2	3	4	5	6	7	8	9	10
5	13	9	6	12	11	10	8	7	

Die Heap Datenstruktur

TECHNISCHE UNIVERSITÄT DARMSTADT

Heapify(A,i) **// jetzt mit i=2**

1. **if** Left(i) ≤ size **and** A[Left(i)] < A[i] **then** smallest := Left(i)
2. **else** smallest := i
3. **if** Right(i) ≤ size **and** A[Right(i)] < A[smallest] **then** smallest := Right(i)
4. **if** smallest ≠ i **then**
5. exchange(A[i], A[smallest])
6. Heapify(A, smallest)



H:

5	6	9	13	12	11	10	8	7	
1	2	3	4	5	6	7	8	9	10

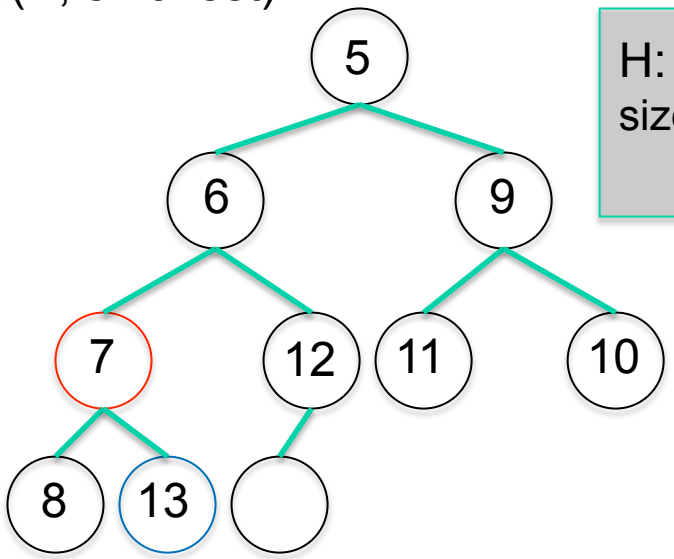
size:9

Die Heap Datenstruktur

TECHNISCHE UNIVERSITÄT DARMSTADT

Heapify(A,i) **// jetzt mit i=4**

1. **if** Left(i) ≤ size **and** A[Left(i)] < A[i] **then** smallest := Left(i)
2. **else** smallest := i
3. **if** Right(i) ≤ size **and** A[Right(i)] < A[smallest] **then** smallest := Right(i)
4. **if** smallest ≠ i **then**
5. exchange(A[i], A[smallest])
6. Heapify(A, smallest)



H: size:9

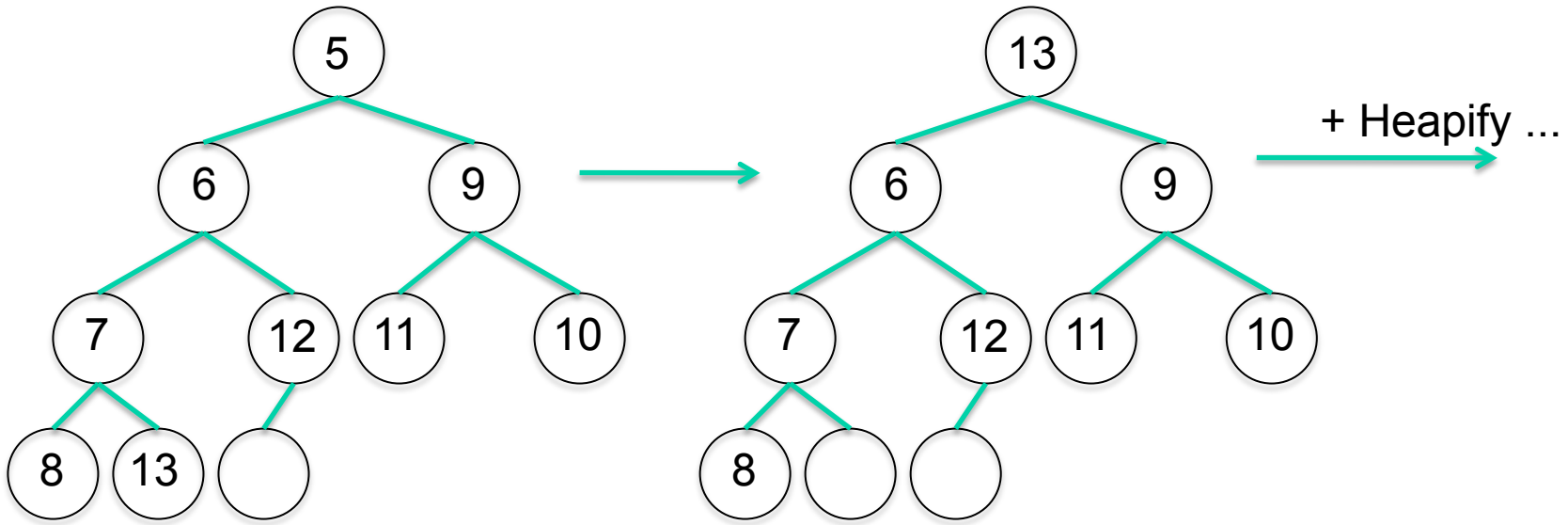
5	6	9	7	12	11	10	8	13	
1	2	3	4	5	6	7	8	9	10

Die Heap Datenstruktur

TECHNISCHE UNIVERSITÄT DARMSTADT

ExtractMin(A)

1. **nimm das Wurzelement heraus. Es ist das Kleinste.**
2. **nimm das size-te Element und tue es in die Wurzel**
3. **size := size -1**
4. **Heapify(A,1)**

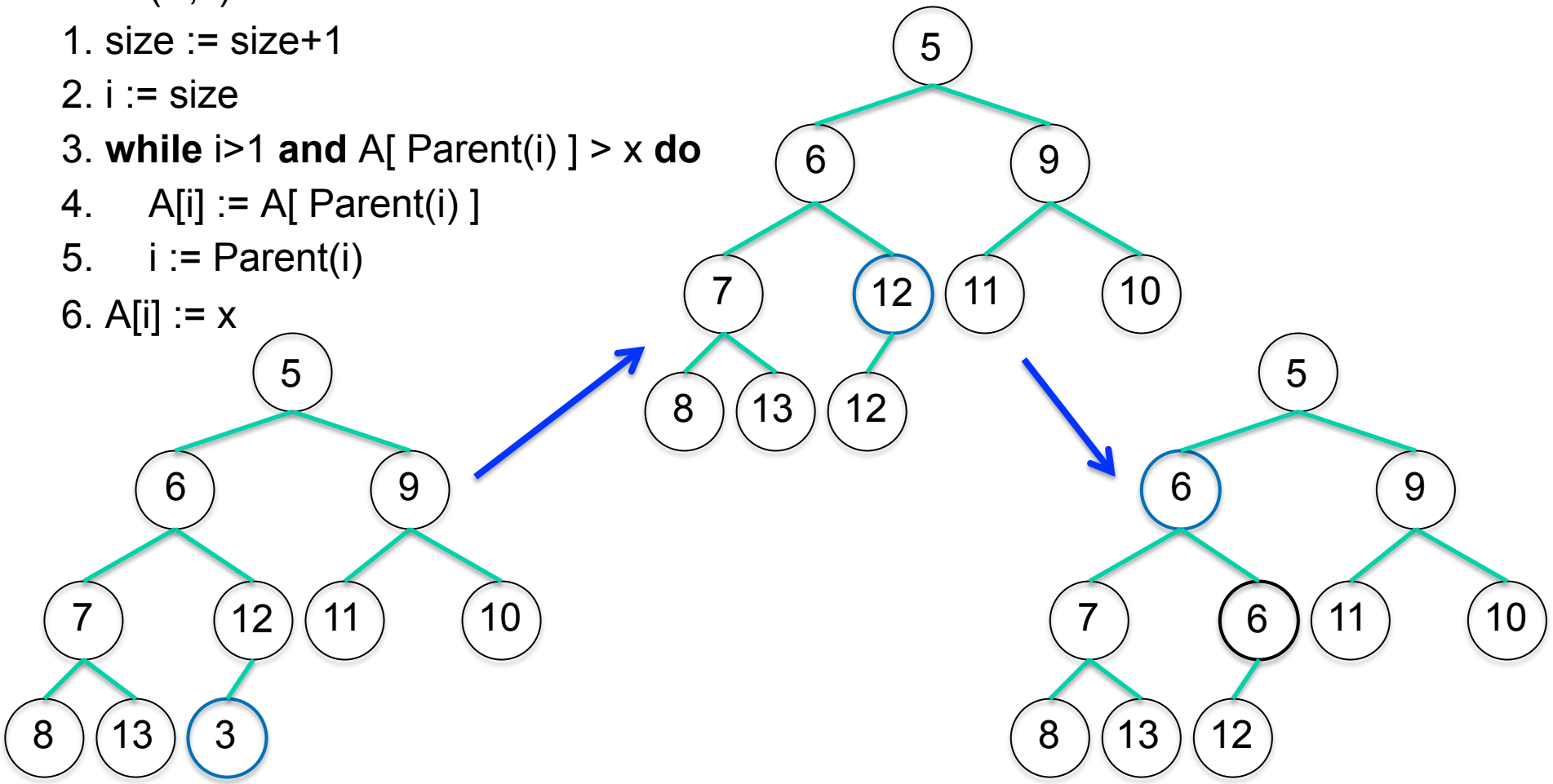


Die Heap Datenstruktur

TECHNISCHE UNIVERSITÄT DARMSTADT

Insert(A,x)

1. size := size+1
2. i := size
3. **while** i>1 **and** A[Parent(i)] > x **do**
4. A[i] := A[Parent(i)]
5. i := Parent(i)
6. A[i] := x



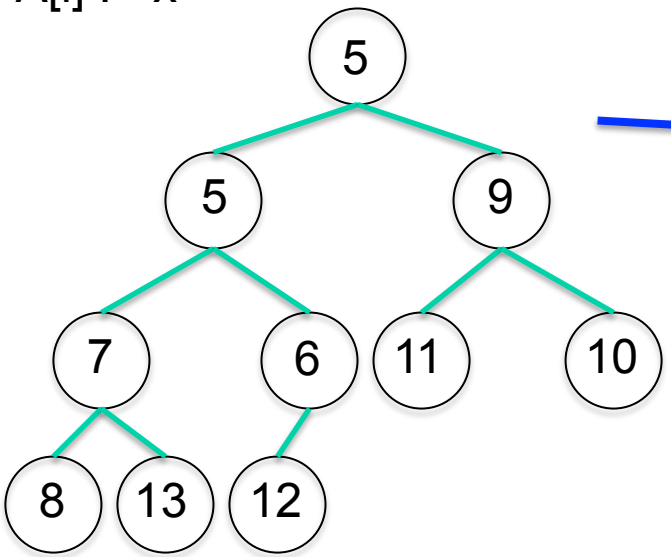
Die Heap Datenstruktur

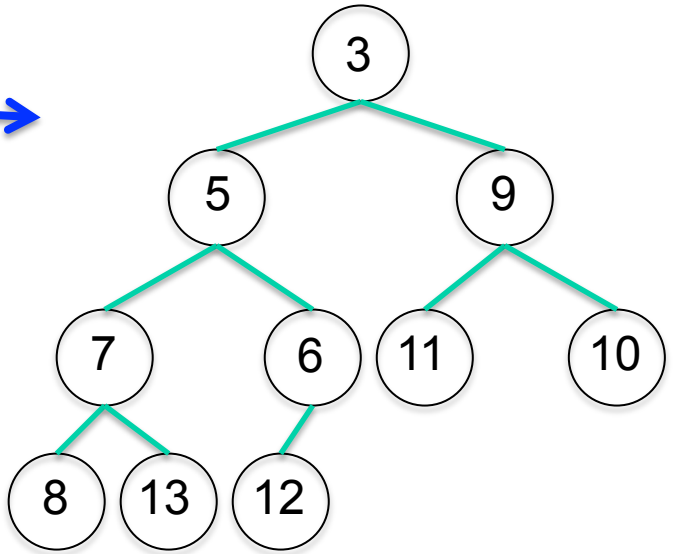
TECHNISCHE UNIVERSITÄT DARMSTADT

Insert(A,x)

1. size := size+1
2. i := size
3. **while** i>1 **and** A[Parent(i)] > x **do**
4. A[i] := A[Parent(i)]
5. i := Parent(i)
6. A[i] := x

Korrektheit: Wenn ein Element x eines Knotens v in einen Nachfolger kopiert wird, dann deshalb, weil das neu einzufügende Element kleiner als x ist. x wird also die Heapeigenschaft am Ende nicht zerstören. Die heruntergezogenen Elemente aber auch nicht.





☺

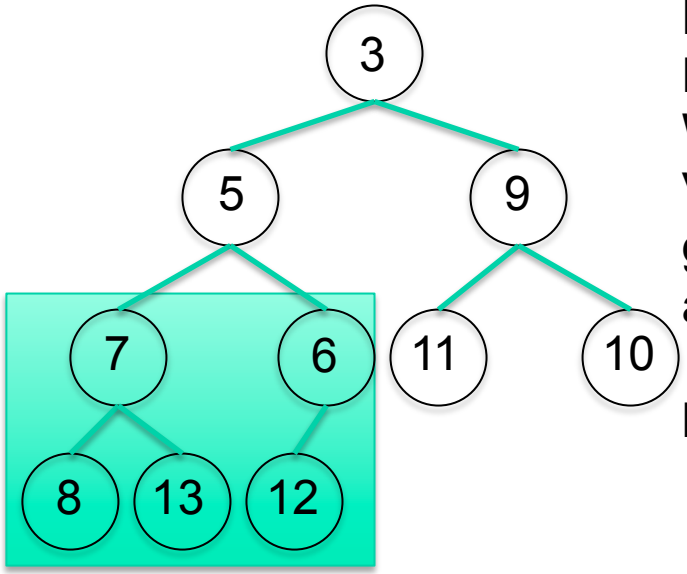
Die Heap Datenstruktur

TECHNISCHE UNIVERSITÄT DARMSTADT

BuildHeap(A) // alle n Elemente liegen unsortiert im Array (Baum)

1. size := Anzahl Elemente n
2. for i := ⌊size/2⌋ downto 1 do
3. Heapify(A,i)

Einfache Laufzeitschranke: O(n log n). Genauer: O(n) (ohne Beweis)



Korrektheitsüberlegung: Heapify macht aus den Bäumen der letzten 2 Ebenen sicher Heaps. Werden nun 2 Heaps zusammengefast, indem ein Vaterknoten für die 2 vorhandenen Subheaps geschaffen wird, können 2 Fälle auftreten:

a) Der Wert im neuen Knoten ist kleiner als die Sohnwerte. Dann ist die Heapeigenschaft erfüllt. Oder
b) der Wert im neuen Vaterknoten v ist größer als einer der Sohnwerte. Dann ist die Heapeigenschaft an v verletzt, sonst nirgends. Dann repariert heapify den Heap.

Die Heap Datenstruktur

TECHNISCHE UNIVERSITÄT DARMSTADT

DecreaseKey(A,i,newkey)

1. A[i] := newkey
2. **while** i>1 **and** A[Parent(i)] > A[i] **do**
3. Exchange(A[i], A[Parent(i))
4. i := Parent(i)

Korrektheitsüberlegung wie bei Insert(A,x) auch.

Kürzeste Wegealgorithmen

TECHNISCHE UNIVERSITÄT DARMSTADT

Nochmal Dijkstras Algorithmus

```
1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
2: S := ∅;
3: A := V;
   BuildHeap(A) mit Werten dist[a] für alle a∈A
4: while   A ≠ ∅  do
5:    u := ExtractMin(A)
6:    S := S ∪ {u};
7:    for each node v ∈ Adj[u] do
8:       if dist[v] > dist[u] + f(u,v) then
9:          dist[v] := dist[u] + f(u,v);
            DecreaseKey(A,v,dist[v])
10:         π[v] := u;
```

Laufzeit: $O((|E|+|V|) \cdot \log(|V|)$

Noch besser wird‘s mit so genannten Fibonacci-Heaps: $O(|E| + |V| \cdot \log(|V|)$, nicht Teil dieser Vorlesung.

Kürzeste Wegealgorithmen

TECHNISCHE UNIVERSITÄT DARMSTADT

Andere Kürzeste-Wege Probleme

- Kürzeste Wege in gerichteten Graphen mit allgemeinen Gewichten
  → **Bellman-Ford Algorithmus** gibt kürzeste Wege ausgehend von einem Startknoten s, oder „es gibt negativen Kreise, der von Startknoten s aus erreichbar ist“ aus.
  → ganz allgemein: NP-vollständig

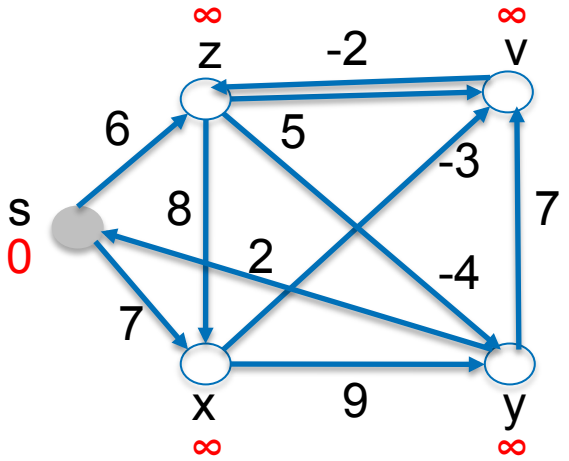
- Kürzeste Wege in gerichteten Graphen ohne Kreise
  → gibt schnellen Algorithmus, auch bei allgemeinen Kantengewichten

- **Längste Wege**
  - gibt es einen einfachen Pfad von zwei ausgezeichneten Knoten s nach t im Graphen G, so dass jeder Knoten genau einmal besucht wird? Nennt sich **Hamiltonpfadproblem**. → NP-vollständig
  - Gibt es einen **Hamiltonkreis**, also einen Hamiltonpfad von s nach s im Graphen G? → NP-vollständig

TECHNISCHE UNIVERSITÄT DARMSTADT

Bellman-Ford Algorithmus

```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: for i := 1 to |V| - 1 do
 3:     for each edge (u,v)∈E do
 4:         if dist[v] > dist[u] + f(u,v) then
 5:             dist[v] := dist[u] + f(u,v);
 6:             π[v] := u;
 7:  for each edge (u,v)∈E do
 8:      if dist[v] > dist[u] + f(u,v)
 9:             return false
10:  return true
```



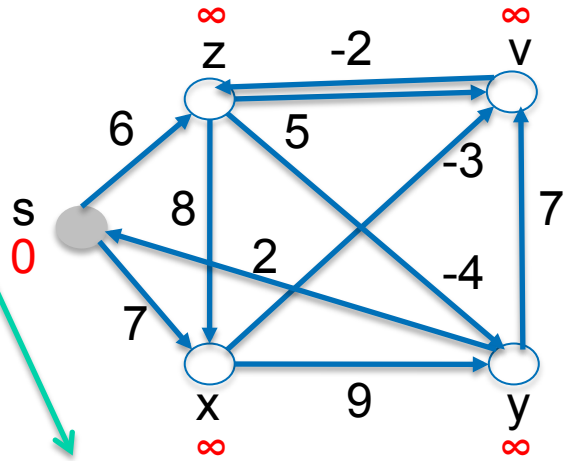
Kantenreihenfolge, Zeile 3: (v,z), (x,v), (x,y), (y,v), (y,s), (z,v), (z,x), (z,y), (s,x), (s,z)

Rechenzeit: O(|V||E|)

TECHNISCHE UNIVERSITÄT DARMSTADT

Bellman-Ford Algorithmus

```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: for i := 1 to |V| - 1 do          → i=1
 3:     for each edge (u,v)∈E do
 4:         if dist[v] > dist[u] + f(u,v) then
 5:             dist[v] := dist[u] + f(u,v);
 6:             π[v] := u;
 7:  for each edge (u,v)∈E do
 8:      if dist[v] > dist[u] + f(u,v)
 9:          return false
10:  return true
```



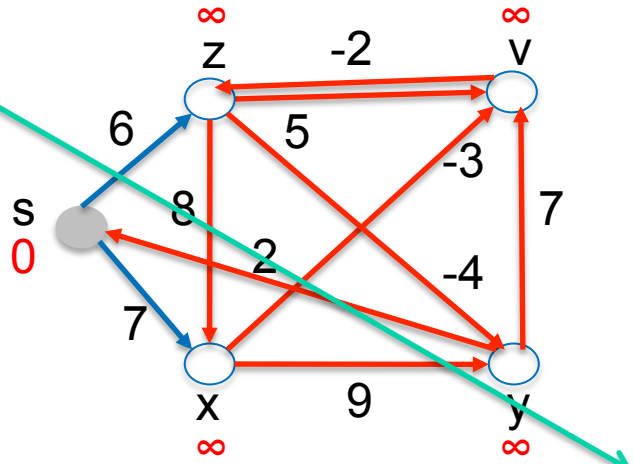
Kantenreihenfolge, Zeile 3: **(v,z), (x,v), (x,y), (y,v), (y,s), (z,v), (z,x), (z,y)**, (s,x), (s,z)

Rechenzeit: O(|V||E|)

TECHNISCHE UNIVERSITÄT DARMSTADT

Bellman-Ford Algorithmus

```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: for i := 1 to |V| - 1 do          → i=1
 3:     for each edge (u,v)∈E do
 4:         if dist[v] > dist[u] + f(u,v) then
 5:             dist[v] := dist[u] + f(u,v);
 6:             π[v] := u;
 7:  for each edge (u,v)∈E do
 8:      if dist[v] > dist[u] + f(u,v)
 9:          return false
10:  return true
```



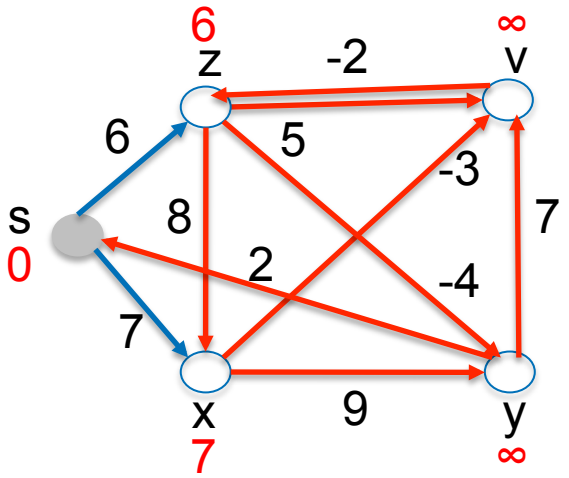
Kantenreihenfolge, Zeile 3: (v,z), (x,v), (x,y), (y,v), (y,s), (z,v), (z,x), (z,y), **(s,x), (s,z)**

Rechenzeit: O(|V||E|)

TECHNISCHE UNIVERSITÄT DARMSTADT

Bellman-Ford Algorithmus

```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: for i := 1 to |V| - 1 do          → i=1
 3:     for each edge (u,v)∈E do
 4:         if dist[v] > dist[u] + f(u,v) then
 5:             dist[v] := dist[u] + f(u,v);
 6:             π[v] := u;
 7:  for each edge (u,v)∈E do
 8:      if dist[v] > dist[u] + f(u,v)
 9:          return false
10:  return true
```



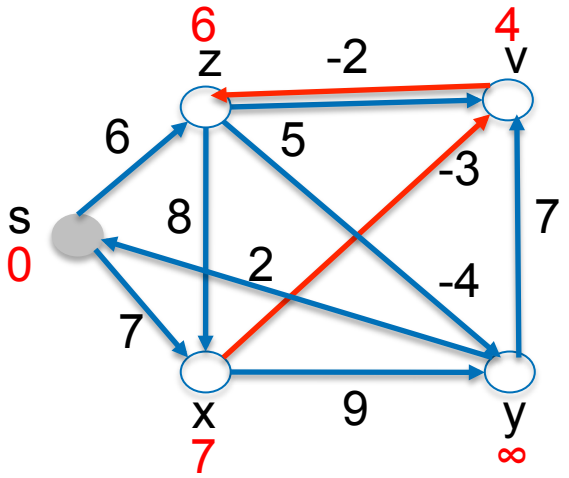
Kantenreihenfolge, Zeile 3: (v,z), (x,v), (x,y), (y,v), (y,s), (z,v), (z,x), (z,y), **(s,x), (s,z)**

Rechenzeit: O(|V||E|)

TECHNISCHE UNIVERSITÄT DARMSTADT

Bellman-Ford Algorithmus

```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: for i := 1 to |V| - 1 do                    → i=2
 3:     for each edge (u,v)∈E do
 4:         if dist[v] > dist[u] + f(u,v) then
 5:             dist[v] := dist[u] + f(u,v);
 6:             π[v] := u;
 7:  for each edge (u,v)∈E do
 8:      if dist[v] > dist[u] + f(u,v)
 9:          return false
10:  return true
```



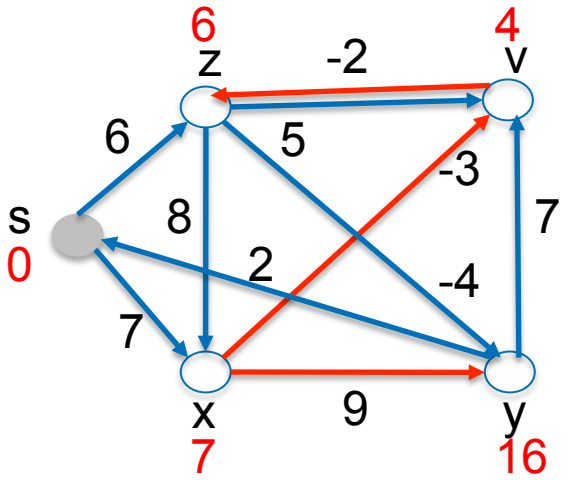
Kantenreihenfolge, Zeile 3: **(v,z), (x,v),** (x,y), (y,v), (y,s), (z,v), (z,x), (z,y), (s,x), (s,z)

Rechenzeit: O(|V||E|)

TECHNISCHE UNIVERSITÄT DARMSTADT

Bellman-Ford Algorithmus

```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: for i := 1 to |V| - 1 do                    → i=2
 3:     for each edge (u,v)∈E do
 4:         if dist[v] > dist[u] + f(u,v) then
 5:             dist[v] := dist[u] + f(u,v);
 6:             π[v] := u;
 7:  for each edge (u,v)∈E do
 8:      if dist[v] > dist[u] + f(u,v)
 9:          return false
10:  return true
```



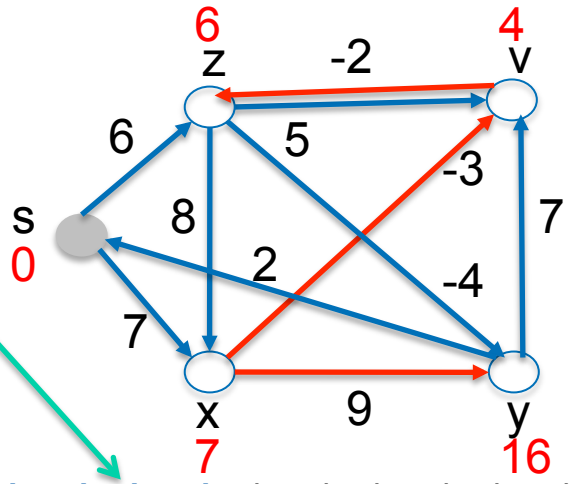
Kantenreihenfolge, Zeile 3: (v,z), (x,v), **(x,y),** (y,v), (y,s), (z,v), (z,x), (z,y), (s,x), (s,z)

Rechenzeit: O(|V||E|)

TECHNISCHE UNIVERSITÄT DARMSTADT

Bellman-Ford Algorithmus

```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: for i := 1 to |V| - 1 do                    i=2
 3:     for each edge (u,v)∈E do
 4:         if dist[v] > dist[u] + f(u,v) then
 5:             dist[v] := dist[u] + f(u,v);
 6:             π[v] := u;
 7:  for each edge (u,v)∈E do
 8:      if dist[v] > dist[u] + f(u,v)
 9:          return false
10:  return true
```



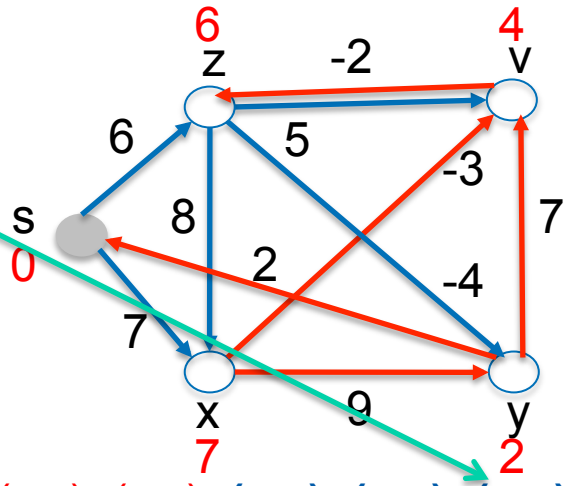
Kantenreihenfolge, Zeile 3: (v,z), (x,v), **(x,y), (y,v), (y,s),** (z,v), (z,x), (z,y), (s,x), (s,z)

Rechenzeit: O(|V||E|)

TECHNISCHE UNIVERSITÄT DARMSTADT

Bellman-Ford Algorithmus

```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: for i := 1 to |V| - 1 do                    → i=2
 3:     for each edge (u,v)∈E do
 4:         if dist[v] > dist[u] + f(u,v) then
 5:             dist[v] := dist[u] + f(u,v);
 6:             π[v] := u;
 7:  for each edge (u,v)∈E do
 8:      if dist[v] > dist[u] + f(u,v)
 9:          return false
10:  return true
```



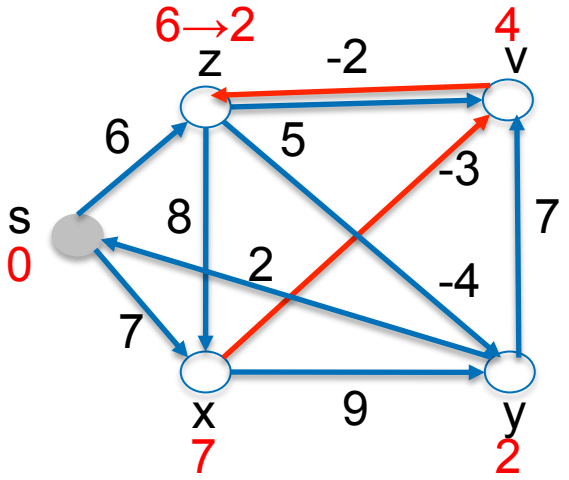
Kantenreihenfolge, Zeile 3: (v,z), (x,v), (x,y), (y,v), (y,s), **(z,v), (z,x), (z,y),** (s,x), (s,z)

Rechenzeit: O(|V||E|)

TECHNISCHE UNIVERSITÄT DARMSTADT

Bellman-Ford Algorithmus

```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: for i := 1 to |V| - 1 do                    → i=3
 3:     for each edge (u,v)∈E do
 4:         if dist[v] > dist[u] + f(u,v) then
 5:             dist[v] := dist[u] + f(u,v);
 6:             π[v] := u;
 7:  for each edge (u,v)∈E do
 8:      if dist[v] > dist[u] + f(u,v)
 9:          return false
10:  return true
```



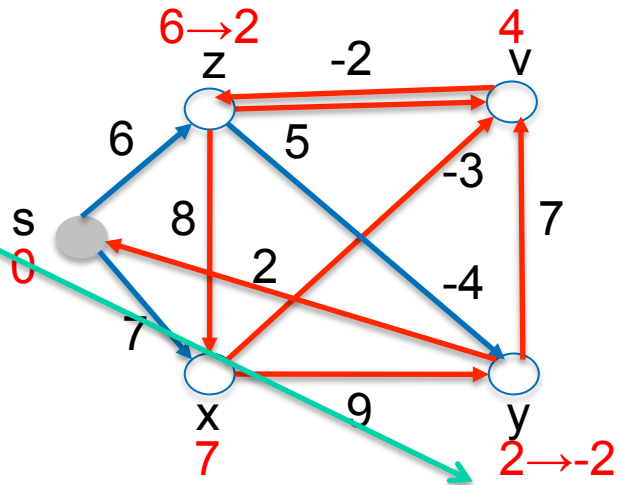
Kantenreihenfolge, Zeile 3: **(v,z), (x,v),** (x,y), (y,v), (y,s), (z,v), (z,x), (z,y), (s,x), (s,z)

Rechenzeit: O(|V||E|)

TECHNISCHE UNIVERSITÄT DARMSTADT

Bellman-Ford Algorithmus

```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: for i := 1 to |V| - 1 do                    → i=3
 3:     for each edge (u,v)∈E do
 4:         if dist[v] > dist[u] + f(u,v) then
 5:             dist[v] := dist[u] + f(u,v);
 6:             π[v] := u;
 7:  for each edge (u,v)∈E do
 8:      if dist[v] > dist[u] + f(u,v)
 9:          return false
10:  return true
```



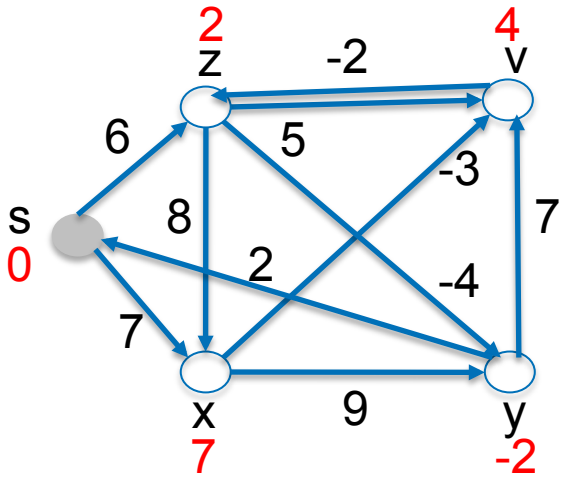
Kantenreihenfolge, Zeile 3: **(v,z), (x,v),** (x,y), (y,v), (y,s), (z,v), (z,x), **(z,y),** (s,x), (s,z)

Rechenzeit: O(|V||E|)

TECHNISCHE UNIVERSITÄT DARMSTADT

Bellman-Ford Algorithmus

```
 1: Initialize(G,s)  // für alle Knoten v≠s: π[v]:=nil; dist[v]:=∞; dist[s]:=0;π[s]:=nil;
 2: for i := 1 to |V| - 1 do                    i=4,5
 3:     for each edge (u,v)∈E do
 4:         if dist[v] > dist[u] + f(u,v) then
 5:             dist[v] := dist[u] + f(u,v);
 6:             π[v] := u;
 7:  for each edge (u,v)∈E do
 8:      if dist[v] > dist[u] + f(u,v)
 9:          return false
10:  return true
```



Kantenreihenfolge, Zeile 3: (v,z), (x,v), (x,y), (y,v), (y,s), (z,v), (z,x), (z,y), (s,x), (s,z)

Rechenzeit: O(|V||E|)

Bellman-Ford Algorithmus

Lemma BF1: Sei G=(V,E) ein gewichteter gerichteter Graph mit Startknoten s und Gewichtsfunktion f: E→IR, und habe G keine negativen Kreise, die von s aus erreicht werden können. Dann gilt nach Beendigung des Bellman-Ford Algorithmus: dist[v]=δ(s,v) für alle Knoten v, die von s aus erreichbar sind.

Beweis: Sei v ein Knoten, der von s aus erreichbar ist, und sei $p=<s=v_0,v_1,...,v=v_k>$ eine kürzester Weg von s nach v. p ist ein einfacher Weg und deshalb gilt $k \leq |V|-1$.
Zu zeigen über Induktion: $dist[v_i] = \delta(s,v_i)$ nach i-tem Durchlauf der Zeilen 3-6.

IA: zu Beginn gilt: dist[s] = 0 = δ(s,s). Da es nach Voraussetzung keinen negativen Kreis gibt, wird dist[s] nicht mehr verändert.
IV: $dist[v_{i-1}] = \delta(s,v_{i-1})$ nach (i-1)-tem Durchlauf der Zeilen 3-6.
IS: i-1→i: analog zu Beweis von Lemma Dijk2

Bellman-Ford Algorithmus

Satz BFKor: Werde der Bellman-Ford Algorithmus auf einem gerichteten, gewichteten Graphen G=(V,E) mit Startknoten s ausgeführt. Falls G einen negativen Kreis enthält, der von s aus erreichbar ist, gibt der Algorithmus false zurück. Ansonsten gibt er true zurück und es gilt für alle Knoten dist[v]=δ(s,v).

Beweis: Wenn es keine von s aus erreichbaren negativen Kreise gibt, gilt für alle erreichbaren Knoten v: dist[v]=δ(s,v) wegen Lemma BF1. Ist v nicht erreichbar von s, bleibt dist[v] offenbar ∞. Da nach Beendigung für alle Knoten gilt:

dist[v] = δ(s,v) ≤ δ(s,u) + f(u,v) = dist[u] + f(u,v),

wird Zeile 9 niemals ausgeführt.

gilt immer für kürzeste Wege

```
...
 7:  for each edge (u,v)∈E do
 8:      if dist[v] > dist[u] + f(u,v)
 9:          return false
10:  return true
```

TECHNISCHE UNIVERSITÄT DARMSTADT

Bellman-Ford Algorithmus

Satz BFKor: (Forts.)
Andersherum, enthalte G nun einen negativen Kreis $c=<v_0,v_1,...,v_k>$ der von s aus erreichbar ist, und für den gilt: $v_0=v_k$. Es gilt dann

$$\sum_{i=1}^{k} f(v_{i-1},v_i) < 0$$

Nehmen wir an, Bellman-Ford gibt true zurück, d.h., es gilt für alle i=1,2,...,k:
$dist[v_i] \leq dist[v_{i-1}]+f(v_{i-1},v_i)$. Dann gilt auch für die Summen:

$$\sum_{i=1}^{k} dist[v_i] \leq \sum_{i=1}^{k} dist[v_{i-1}] + \sum_{i=1}^{k} f(v_{i-1},v_i)$$

Da c ein Kreis ist, kommt jeder Summand in den beiden ersten Summen vor. Damit folgt:

$$\sum_{i=1}^{k} dist[v_i] = \sum_{i=1}^{k} dist[v_{i-1}] \quad und\ damit \quad 0 \leq \sum_{i=1}^{k} f(v_{i-1},v_i)$$

Widerspruch zu

Minimale Spannbäume

TECHNISCHE UNIVERSITÄT DARMSTADT

- Definition MiSpa1
  - **Gewichteter ungerichteter Graph** (G,f): ungerichteter Graph G=(V,E) mit Gewichtsfunktion f: E → IR.
  - Ist H=(U,F), U⊆V, F⊆E, ein Teilgraph von G, so ist das Gewicht f(H) von H definiert als

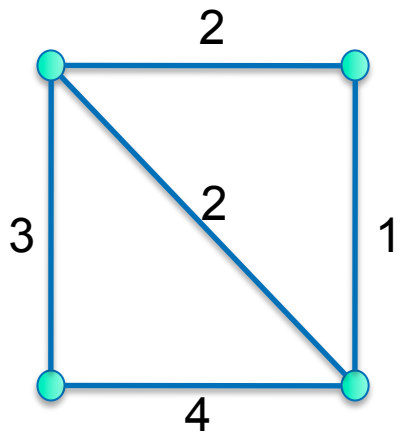
$$w(H) = \sum_{e \in F} w(e)$$

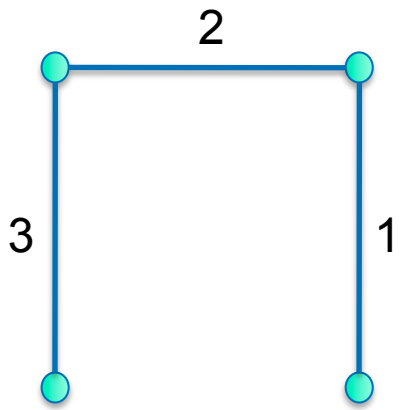
  - Ein Teilgraph H eines ungerichteten Graphen G heißt **Spannbaum** von G, wenn H ein Baum ist.
  - Ein Spannbaum S eines gewichteten ungerichteten Graphen G heißt minimaler Spannbaum von G, Wenn S minimales Gewicht unter allen Spannbäumen von G besitzt.

Minimale Spannbäume

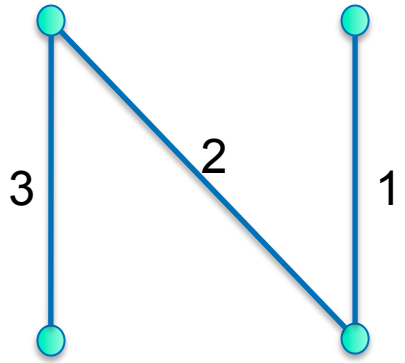
TECHNISCHE UNIVERSITÄT DARMSTADT

Graph G

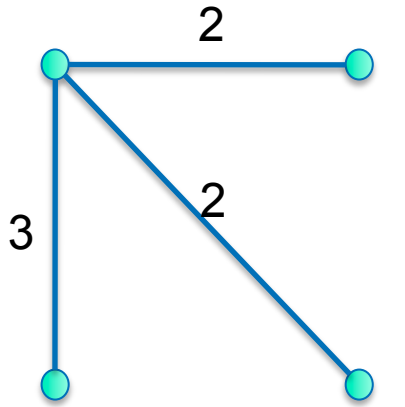




Spannbaum für G, minimal



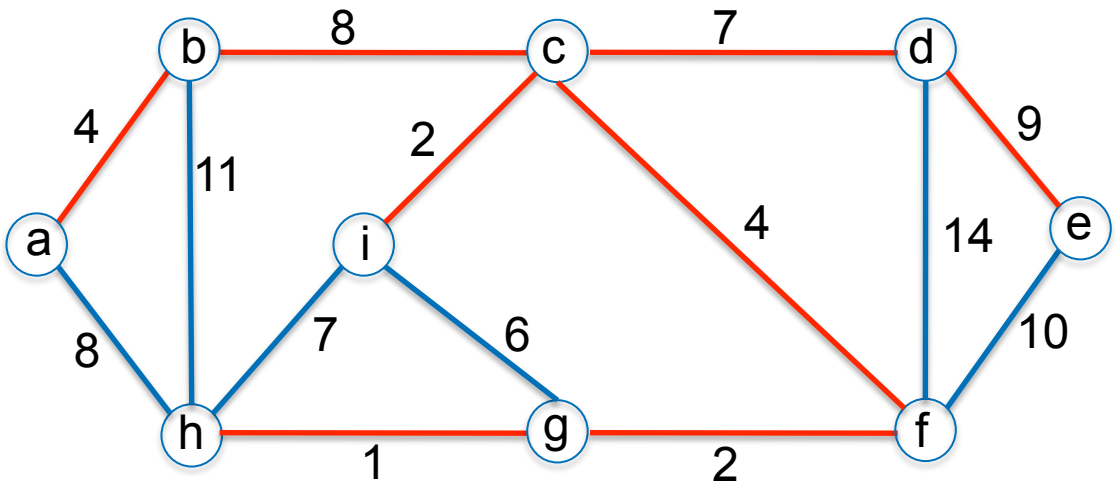
Spannbaum für G, minimal



Spannbaum für G, nicht minimal

Minimale Spannbäume

TECHNISCHE UNIVERSITÄT DARMSTADT



Minimale Spannbäume

TECHNISCHE UNIVERSITÄT DARMSTADT

- Ziel: Gegeben ein gewichteter ungerichteter Graph (G,f) mit G=(V,E). Finde effizient einen minimalen Spannbaum von (G,f).

- Vorgehensweise: Erweitere iterativ eine Kantenmenge $A \subseteq E$ zu einem minimalen Spannbaum:
  - Def. MiSpa2: (u,v) heißt **A-sicher**, wenn $A \cup \{(u,v)\}$ zu einem minimalen Spannbaum erweitert werden kann.
  - Zu Beginn: A={}
  - Ersetze in jedem Schritt A durch $A \cup \{(u,v)\}$, wobei (u,v) eine A-sichere Kante ist
  - Wiederhole 1. und 2. so lange, bis |A| = |V| - 1

Minimale Spannbäume

TECHNISCHE UNIVERSITÄT DARMSTADT

- Generischer MST-Algorithmus (MST = Minimum Spanning Tree)

```
Generic-MST(G,f)
  1: A := {}
  2: while A ist noch kein Spannbaum
  3:     for each edge (u,v)∈E do
  4:         finde eine A-sichere Kante (u,v)
  5:             A := A ∪ {(u,v)}
  6:  return A;
```

Minimale Spannbäume

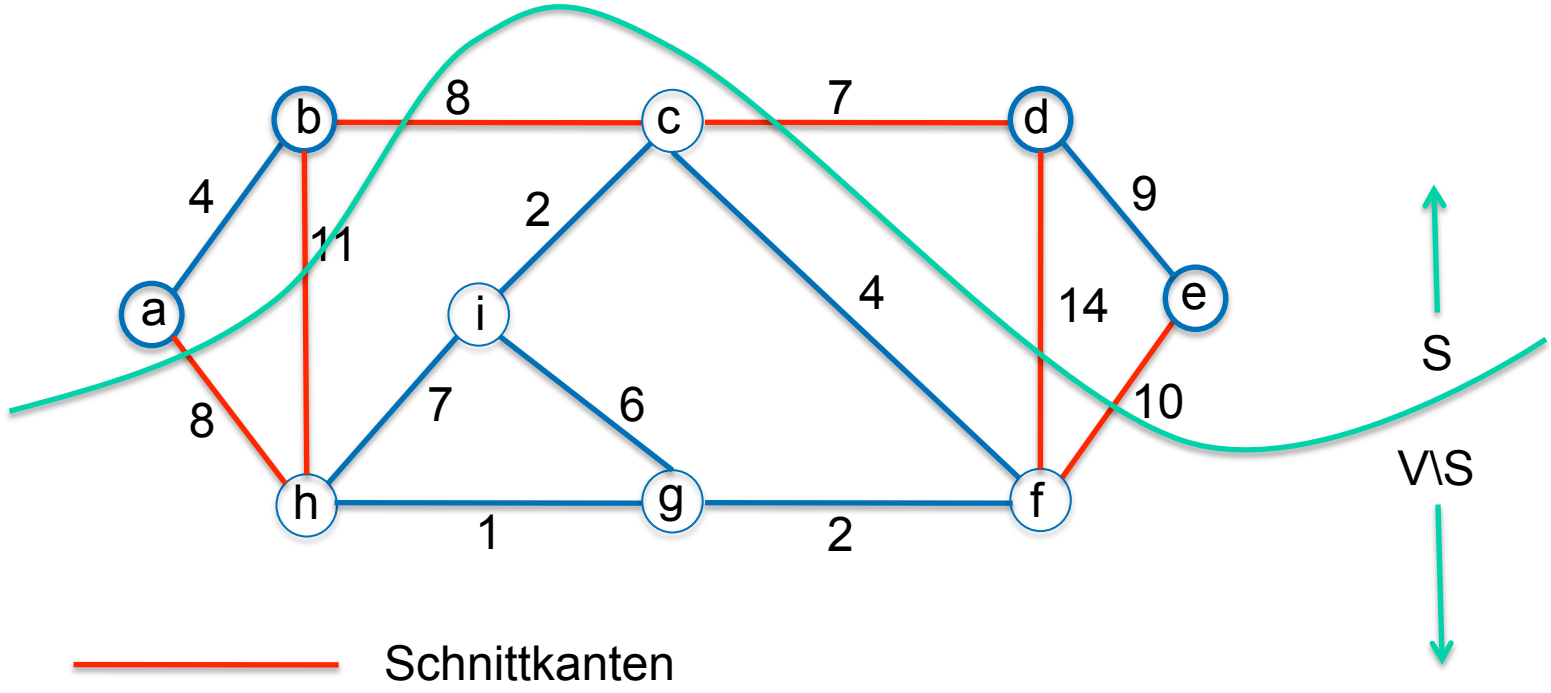
TECHNISCHE UNIVERSITÄT DARMSTADT

- Def. MiSpa3:
  - Ein **Schnitt** (C,V \ C) in einem Graphen G=(V,E) ist eine Partition der Knotenmenge V des Graphen G.
  - Eine Kante von G **kreuzt** einen Schnitt (C,V \ C), wenn ein Knoten der Kante in C, der andere in V \ C liegt.
  - Für einen Schnitt (C, V \ C) ist eine Teilmenge $A \subseteq E$ **verträglich**, wenn kein Element von A den Schnitt kreuzt.
  - Eine (C, V \ C) kreuzende Kante heißt **leicht**, wenn sie eine Kante minimalen Gewichts unter den (C, V \ C) kreuzenden Kanten ist.

Minimale Spannbäume

TECHNISCHE UNIVERSITÄT DARMSTADT

Schnitt:



Minimale Spannbäume

TECHNISCHE UNIVERSITÄT DARMSTADT

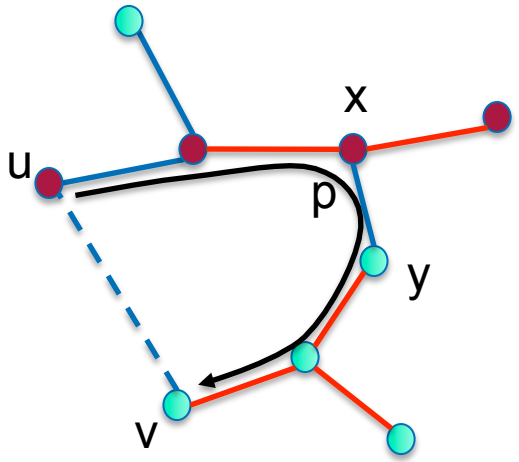
- Satz MiSpa1:
  Sei (G,f) ein gewichteter, ungerichteter Graph. Es gebe einen minimalen Spannbaum T in G, der die Kantenmenge $A \subseteq E$ enthalte. Sei (S, V \ S) ein mit A verträglicher Schnitt

  [...verträglich: kein Element von A kreuzt den Schnitt]

  und (u,v) sei eine leichte (S, V \ S) kreuzende Kante.

  [... leicht: Kante minimalen Gewichts unter den (C, V \ C) kreuzenden Kanten]

  Dann ist (u,v) eine A-sichere Kante.



- S V\S
- A
- (u,v) leichte Kante von S nach V \ S
  nur Kanten von T im Bild

- Spannbaum T‘, der (u,v) enthält, wird so konstruiert:
  entferne (x,y), füge (u,v) ein

Minimale Spannbäume

TECHNISCHE UNIVERSITÄT DARMSTADT

- Satz MiSpa2:
  Sei (G,f) ein gewichteter, ungerichteter Graph. Es gebe einen minimalen Spannbaum in G, der die Kantenmenge $A \subseteq E$ enthalte. Ist (u,v) eine leichte Kante minimalen Gewichts, die einen Baum B des Waldes G_A = (V,A) mit einem anderen Baum von G_A verbindet, so ist (u,v) A-sicher.

  Beweis:
  - Der Schnitt (B, V \ B) ist A-verträglich. (warum?)
    - B ist eine Zusammenhangskomponente von G_A = (V,A)
    - also ist B ein Baum, der keine Kanten in G_A zu V\B aufweist
    - also A-verträglich (Def)
  - (u,v) ist deshalb eine leichte Kante für den Schnitt (warum?)
    - (u,v) verbindet zwei Bäume, die Teil eines minimalen Spannbaums sind.
    - Billiger können die beiden Teilbäume nicht verbunden werden.
  - Also (u,v) A-sicher mit Satz MiSpa1

Minimale Spannbäume

TECHNISCHE UNIVERSITÄT DARMSTADT

- Algorithmus von Prim -- Idee

  – Zu jedem Zeitpunkt des Algorithmus besteht der Graph $G_A = (V,A)$ aus einem Baum T_A und einer Menge von isolierten Knoten I_A

  – Eine Kante minimalen Gewichts, die einen Knoten aus I_A mit T_A verbindet, wird zu A hinzugefügt

  – Die Knoten in I_A sind in einem Min-Heap organisiert. Dabei ist der Schlüssel key[v] eines Knotens $v \in I_A$ gegeben durch das minimale Gewicht einer Kante, die v mit T_A verbindet.

Minimale Spannbäume

TECHNISCHE UNIVERSITÄT DARMSTADT

- Algorithmus von Prim

```
Prim-MST(G,f,w)
 1: for all v ∈ V do
 2:     key[v] := ∞
 3:     π(v) := nil
 4: key[w] := 0
 5: Q:= Build-Heap(V)
 6: while Q ≠ {} do
 7:     u :=Extract-Min(Q)
 8:     for all v ∈ Adjazenzliste[u] do
 9:         if v ∈ Q and f(u,v) < key[v] then
10:             π[v] := u
11:             key[v] := f(u,v)
12:             Decrease-Key(Q,v,key[v])
```

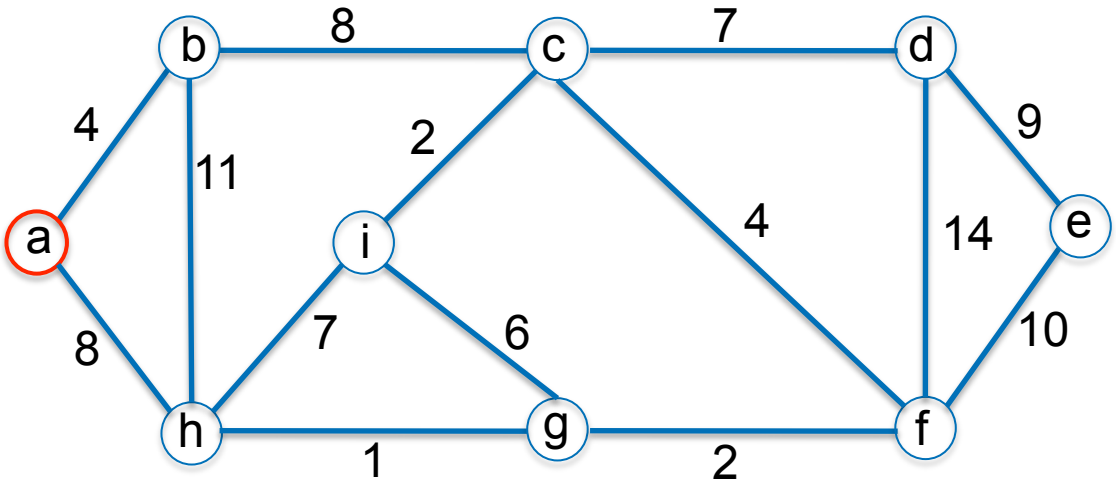
Laufzeit:

- |V|-viele while-Durchläufe
- Zeile 7: O(log |V|)
- Zeilen 9-12: O(2 · |E|) Durchläufe
- Zeile 12: O(log |V|)

- gesamt: O(|E| · log |V|)
- mit Fibonacci-Heaps
  O(|V| log |V| + |E|) möglich

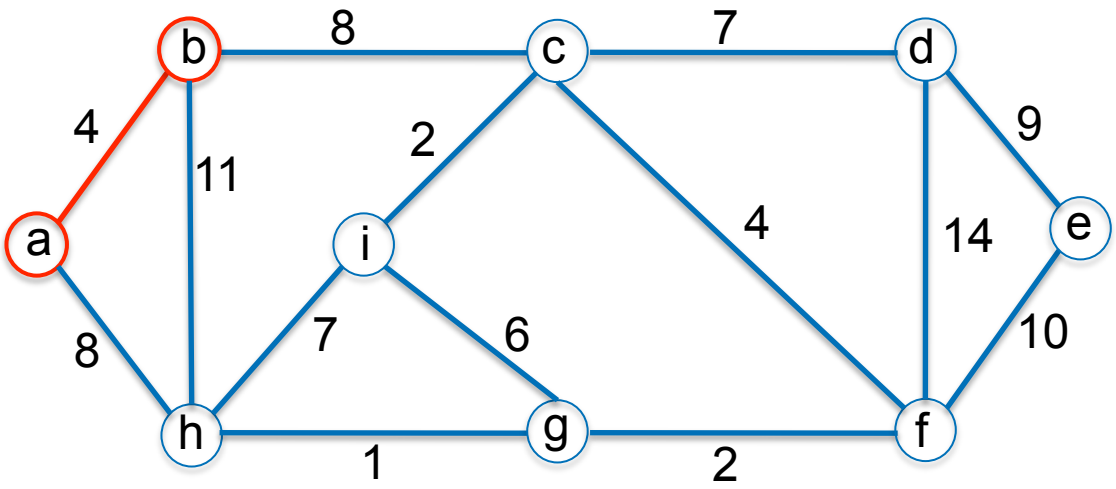
Minimale Spannbäume

TECHNISCHE UNIVERSITÄT DARMSTADT



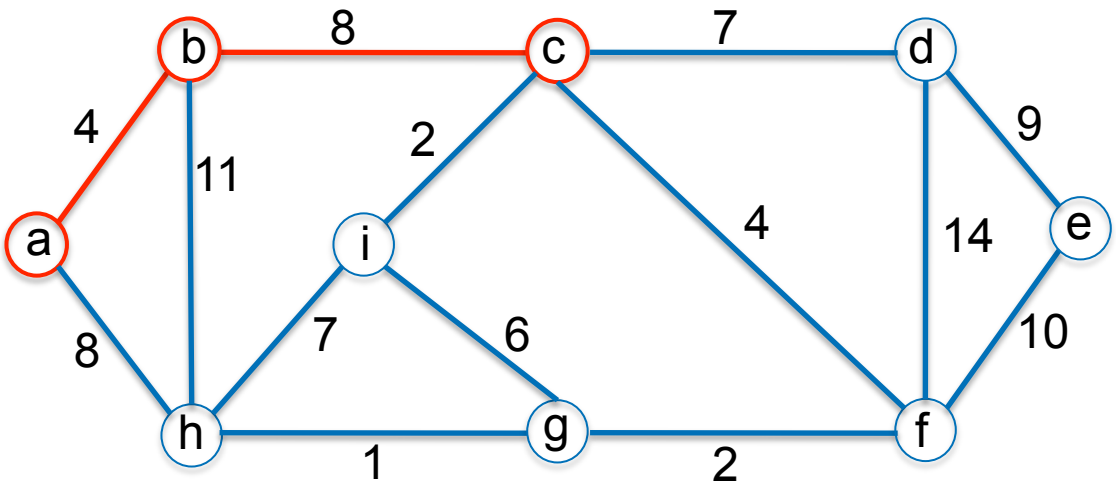
Minimale Spannbäume

TECHNISCHE UNIVERSITÄT DARMSTADT



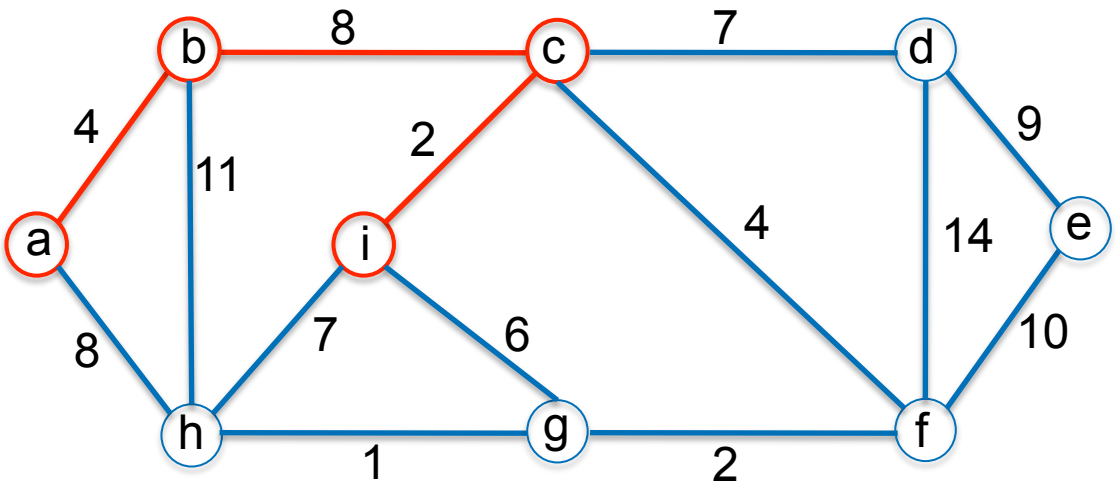
Minimale Spannbäume

TECHNISCHE UNIVERSITÄT DARMSTADT



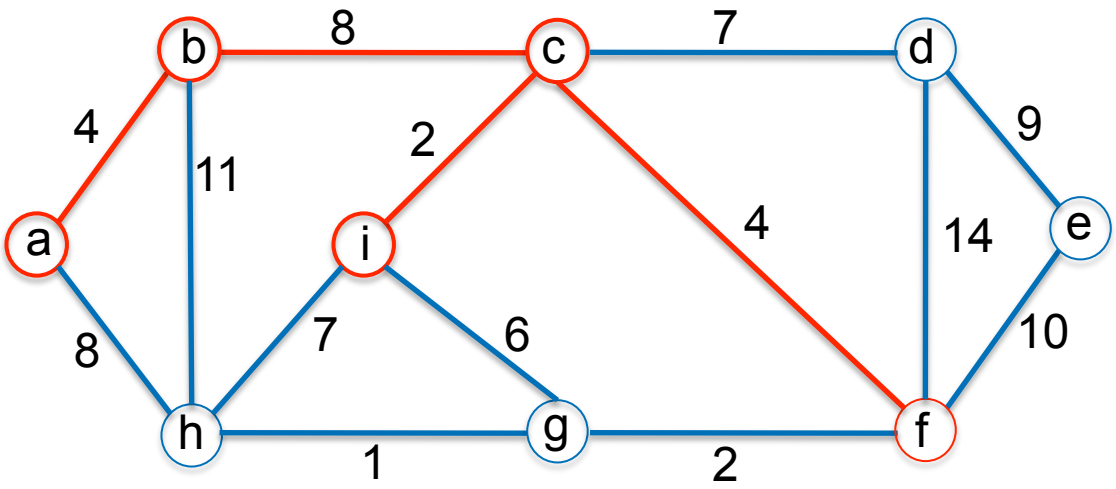
Minimale Spannbäume

TECHNISCHE UNIVERSITÄT DARMSTADT



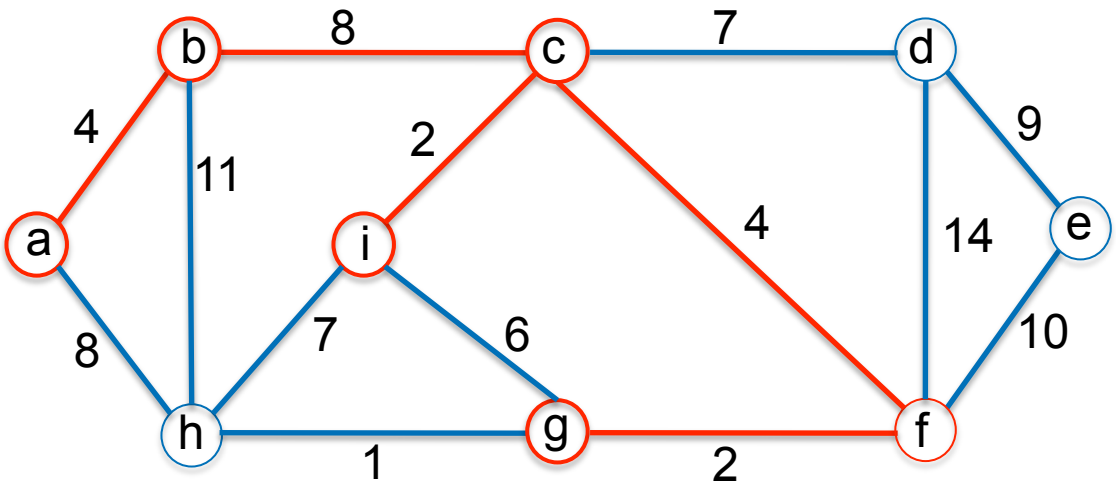
Minimale Spannbäume

TECHNISCHE UNIVERSITÄT DARMSTADT



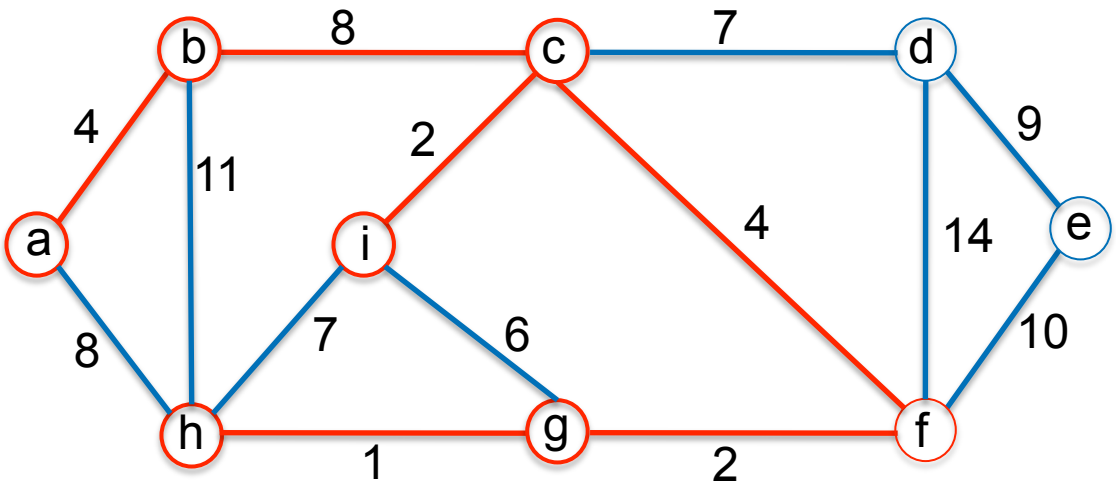
Minimale Spannbäume

TECHNISCHE UNIVERSITÄT DARMSTADT



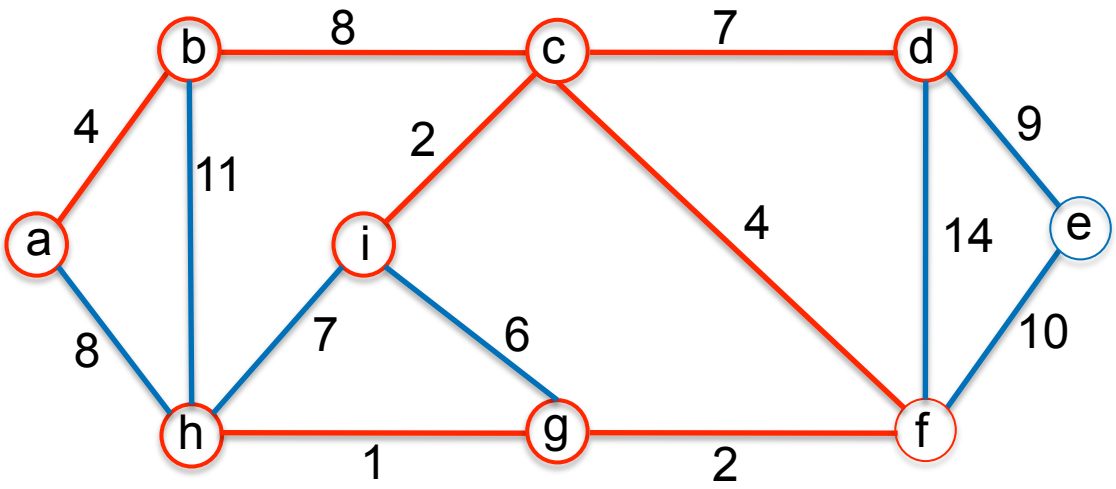
Minimale Spannbäume

TECHNISCHE UNIVERSITÄT DARMSTADT



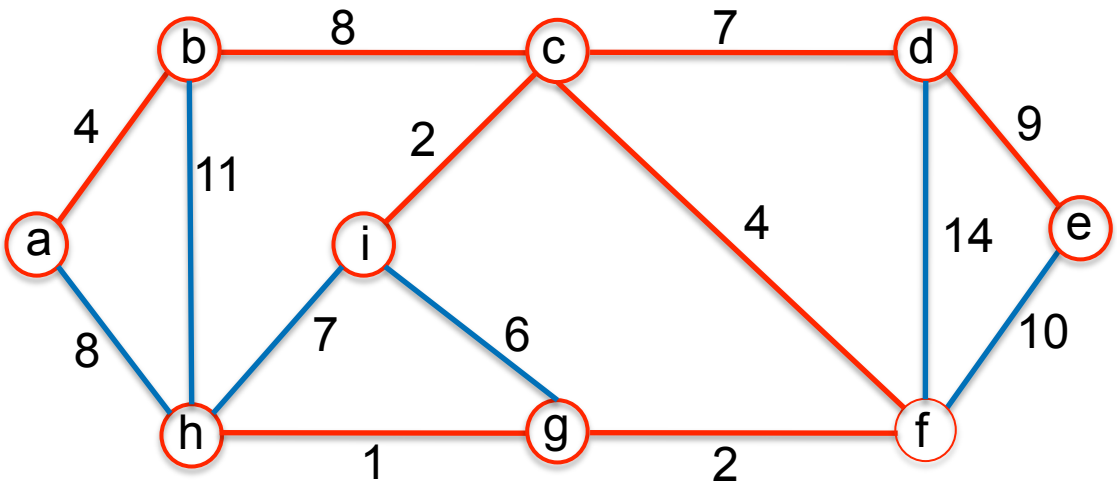
Minimale Spannbäume

TECHNISCHE UNIVERSITÄT DARMSTADT



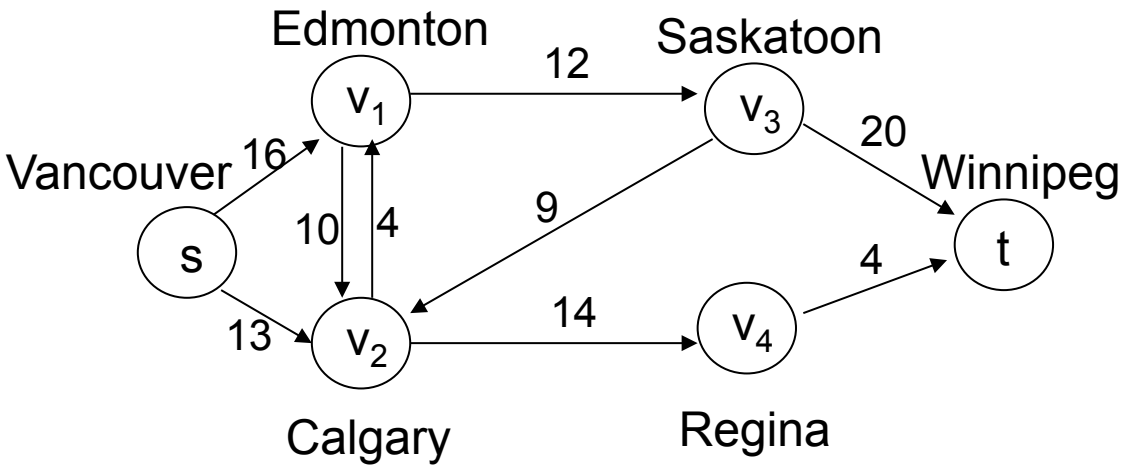
Minimale Spannbäume

TECHNISCHE UNIVERSITÄT DARMSTADT



Flussprobleme

TECHNISCHE UNIVERSITÄT DARMSTADT



Ein Flussproblem G=(V,E). Die Fabrik in Vancouver ist die Quelle, der Grossmarkt in Winnipeg die Senke t. Die Güter müssen über Straßen zum Zielort gebracht werden. Leider können nur $c_{u,v}$ viele pro Woche über (u,v) transportiert werden. Wie kriegt man also jede Woche so viele Einheiten des zu verkaufenden Guts über alle möglichen Wege von Vancouver nach Winnipeg?

Flussnetzwerke werden genutzt, um Verteilungsprobleme, Transport- und Umladeprobleme... zu modellieren. Transportierte Ware sind z.B. Wasser, Strom, Gas, Fahrzeuge, ...

Flussprobleme

TECHNISCHE UNIVERSITÄT DARMSTADT

Flussnetzwerk:
- G=(V,E), gerichteter Graph,
- jedes (u,v)∈E besitzt nicht-negative Kapazitätsbeschränkungen c(u,v) > 0
- falls (u,v)∉E, gilt c(u,v)=0
- es gibt 2 ausgezeichnete Knoten: Quelle s und Senke t
- es gibt für jeden Knoten v einen Pfad von s nach v und von v nach t

Sei G=(V,E) ein Flussnetzwerk, sei s Quelle und t Senke. Ein **Fluss** in G ist eine Funktion f: V×V → IR mit:
- Kapazitätsbeschränkung: $f(u,v) \leq c(u,v)$ für alle $u,v \in V$
- Symmetrie: $f(u,v) = -f(v,u)$ für alle $u,v, \in V$
- Flusskonservierung: $\sum_{v \in V} f(u,v) = 0$

Der **Wert** eines Flusses ist
|f| = $\sum_{v \in V} f(s,v)$, Gesamtfluss aus s heraus

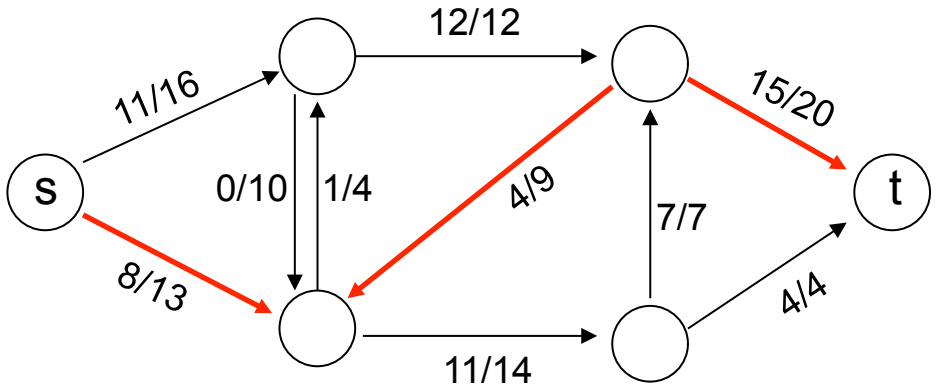
Flussproblem

Ford-Fulkerson Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT

Def.: Gegeben sei ein Flussnetzwerk und ein zulässiger Fluss x von s nach t. Ein „augmenting path“ (oder „verbessernder Pfad“) ist ein Pfad P von s nach t bei dem die Kantenrichtungen ignoriert werden mit folgenden Eigenschaften:

- Für jede Kante (a,b), die von P in Vorwärtsrichtung durchschritten (Vorwärtskante) wird gilt: f(a,b) < c(a,b). D.h., Vorwärtskanten haben freie Kapazitäten.
- Für jede Kante (b,a), die von P in Rückwärtsrichtung durchschritten wird (Rückwärtskante) gilt: f(a,b) > 0.



maximale Änderung entlang P:

$$\min_{\text{Kanten von P}} \begin{cases} c(a,b)-f(a,b) \text{ entlang Vorwärtskanten} \\ f(a,b) \text{ entlang Rückwärtskanten} \end{cases}$$

Flussproblem

Ford-Fulkerson Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT

Ford-Fulkerson(G,s,t)
1. initialisiere Fluss auf 0
2. while es gibt augmenting Pfad p do
3. verbessere den Fluss f entlang p
4. return f

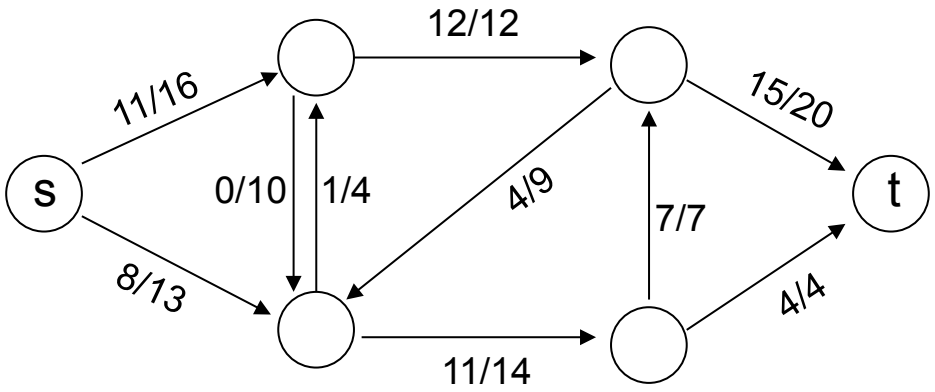
Flussproblem

Residuale Netzwerke

TECHNISCHE UNIVERSITÄT DARMSTADT

Sei f ein Fluss in G. $c_f(u,v) = c(u,v) - f(u,v)$ nennt man **residuale Kapazität**.

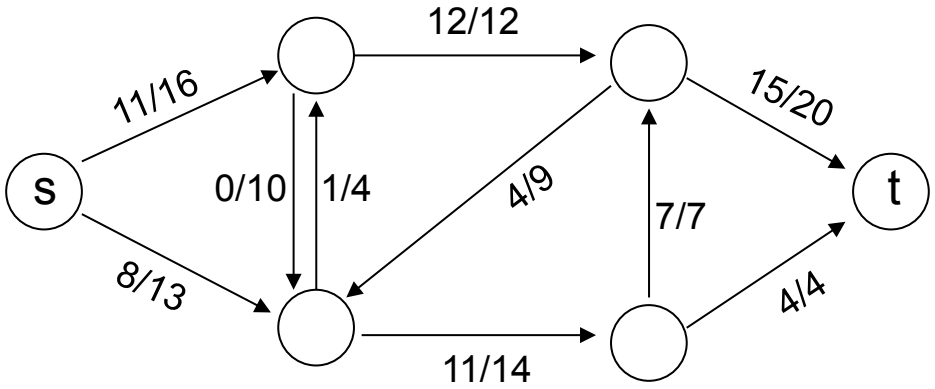
Sei G=(V,E) ein Flussnetzwerk und f ein Fluss. Das **residuale Netzwerk** ist dann $G_f = (V,E_f)$ mit $E_f = \{(u,v) \in V \times V \mid c_f(u,v) > 0\}$. Achtung, es können im residualen Graphen mehr Kanten sein, als im Originalen (warum?):

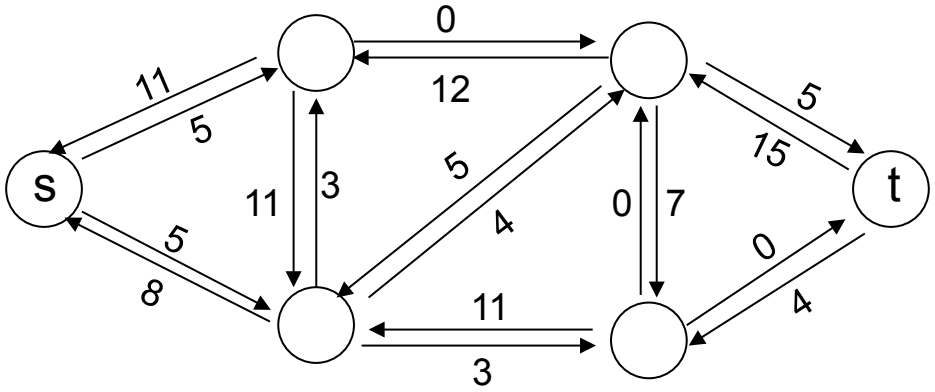


Flussproblem

Residuale Netzwerke

TECHNISCHE UNIVERSITÄT DARMSTADT

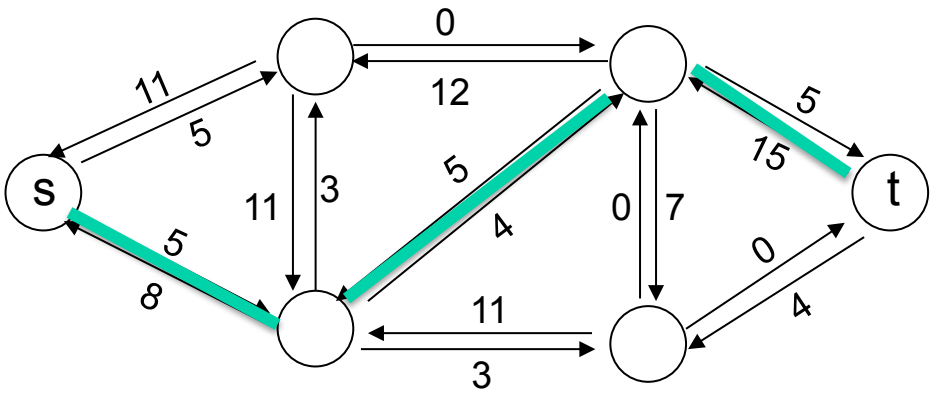


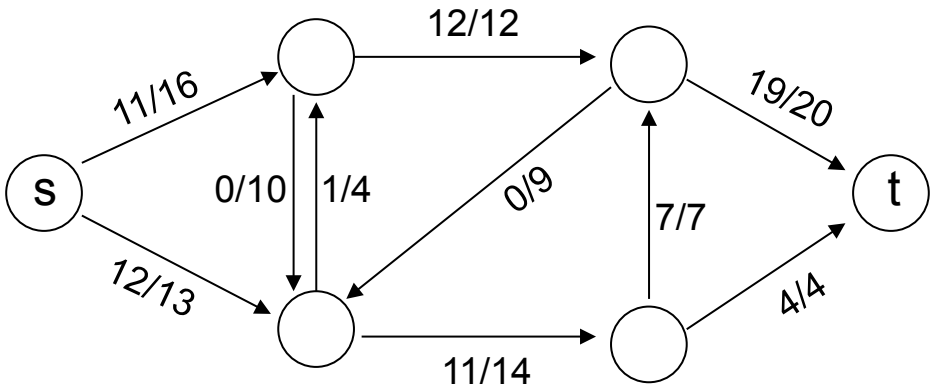


Flussproblem

Residuale Netzwerke

TECHNISCHE UNIVERSITÄT DARMSTADT





Flussprobleme

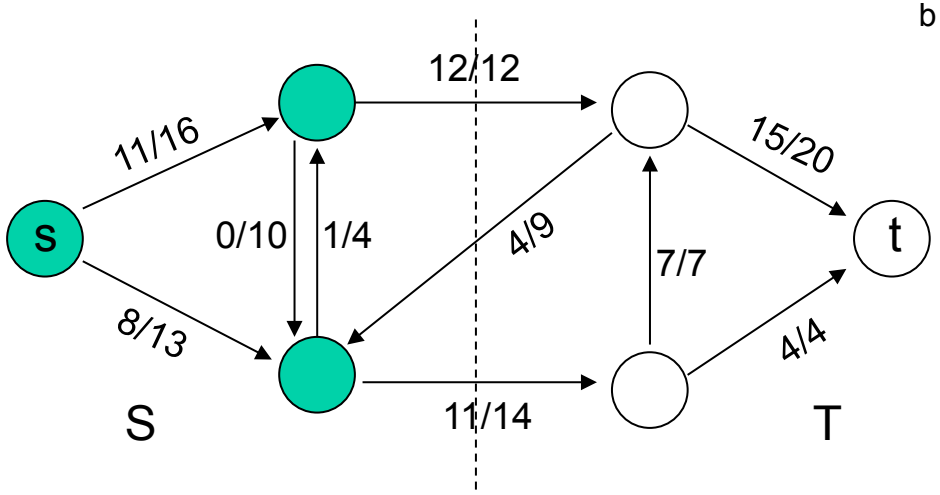
Flussproblem, **Schnitt:**

TECHNISCHE UNIVERSITÄT DARMSTADT

Ein **Schnitt** eines Flußnetzwerkes ist eine Aufteilung der Knoten V in die Mengen S und T = V \ S, so dass $s \in S$ und $t \in T$.

Wenn f ein Fluss ist, dann ist ein **Netzfluss über einen Schnitt (S,T):** $\sum_{\substack{(a,b)\in E \\ a \in S \\ b \in T}} f(a,b)$

Die **Kapazität** eines Schnittes (S,T) ist: $C(S,T) := \sum_{\substack{(a,b)\in E \\ a \in S \\ b \in T}} c(a,b)$



Netzfluss über (S,T) ist 19
Kapazität C(s,T) = 26

Flussproblem

Residuale Netzwerke

TECHNISCHE UNIVERSITÄT DARMSTADT

Satz ResNet1: Sei G = (V,E) ein Flussnetzwerk und sei f ein Fluss. Sei G‘ das residuale Netzwerk von G, und sei f‘ ein Fluss in G‘ entlang eines verbessernden Pfades. Dann gilt für die Summe der Flüsse f + f‘: |f + f‘| = |f| + |f‘|

Beweis: wegen der Konstruktion von G‘

Satz ResNet2: Wenn (S,T) ein Schnitt ist, kann der Fluss von S nach T nicht größer sein, als die Kapazität des Schnitts.
Beweis: Für jede einzelne Kante (u,v) von S nach T gilt, dass $f(u,v) \leq c(u,v)$. Also gilt auch für die Summe über alle Kanten von S nach T:
$\sum_{u \in S, v \in T} f(u,v) \leq \sum_{u \in S, v \in T} c(u,v)$

Flussproblem

Residuale Netzwerke

TECHNISCHE UNIVERSITÄT DARMSTADT

Max-flow min-cut Theorem

Sei f ein Fluss in einem Flussnetzwerk G=(V,E) mit Quelle s und Senke t. Dann sind folgende Aussagen äquivalent:

1. f ist ein maximaler Fluss
2. Das residuale Netzwerk G_f enthält keinen verbessernden Pfad
3. Es gibt einen Schnitt (S,T) so, dass $|f| = \sum_{u \in S, v \in T} c(u,v)$

1 ⇒ 2: Annahme, f sei ein maximaler Fluss, und G_f enthält verbessernden Pfad f'. der verbessernde Pfad ist aber gerade so gewählt, dass er hilft, den Fluss f zu vergrößern. Damit wäre also $|f + f'| > |f|$. Dann wäre aber f nicht maximal gewesen.

2 ⇒ 3: es gebe keinen verbessernden Pfad. Dann gibt es keinen Weg in G_f Von s nach t (, bei dem die Kantenkapazitäten > 0) sind. Sei nun

$S = \{v \in V$ mit: es gibt einen Pfad von s zu v in $G_f\}$

Dann ist (S,T=V\S) eine Partition und für jede Kante (u,v) mit $u \in S$ und $v \in T$ gilt $f(u,v)=c(u,v)$, da sonst $(u,v) \in E_f$.

Flussproblem
Residuale Netzwerke

TECHNISCHE UNIVERSITÄT DARMSTADT

Max-flow min-cut Theorem

Sei f ein Fluss in einem Flussnetzwerk G=(V,E) mit Quelle s und Senke t. Dann sind folgende Aussagen äquivalent:

1. f ist ein maximaler Fluss
2. Das residuale Netzwerk G_f enthält keinen verbessernden Pfad
3. Es gibt einen Schnitt (S,T) so, dass $|f| = \sum_{u \in S, v \in T} c(u,v)$

3 ⇒ 1: Es sei $|f| = \sum_{u \in S, v \in T} c(u,v)$, für S und T wie in Punkt 2. Wegen Satz ResNet2 gibt es keinen größeren Fluss.

Wie findet man einen verbessernden Pfad? Mit Breitensuche.

Flussproblem

Ford-Fulkerson Algorithmus

TECHNISCHE UNIVERSITÄT DARMSTADT

Satz FF1: Wenn der Ford-Fulkerson Algorithmus terminiert, terminiert er mit optimaler Lösung.
Bew.: Bilde nach der Terminierung die Mengen S und T wie im Max-flow min-cut Theorem. Alle Vorwärtskanten sind dann saturiert, alle Rückwärtskanten leer. (Sonst hätte der Algorithmus nicht terminiert) Der (S,T)-Schnitt hat den gleichen Wert, wie der Fluss, den der Algorithmus liefert.

Satz FF2: Der Ford-Fulkerson Algorithmus terminiert nach endlich vielen Schritten, sofern alle Inputparameter natürliche oder rationale Zahlen sind.

Bew.: natürliche Zahlen: klar, da der Fluss immer um ganzzahlige Einheiten erhöht wird.
rationale Zahlen: klar, da man vor Beginn der Berechnungen mit gemeinsamen Nenner multiplizieren kann.