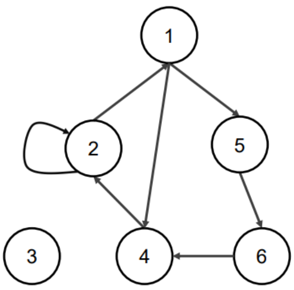
1 Graph Algorithms

1.1 Graphen

- **(Endlicher) gerichteter Graph**
  - (endlicher) gerichteter Graph $G = (V, E)$
  - besteht aus (endlicher) Knotenmenge V
  - besteht aus (endlicher) Kantenmenge $E \subseteq VxV$
  - $(u, v) \in E$: Kanten von Knoten u zu v
  - Kanten haben eine Richtung
- **Ungerichtete Graphen**
  - (endlicher) ungerichteter Graph $G = (V, E)$
  - besteht aus (endlicher) Knotenmenge V
  - besteht aus (endlicher) Kantenmenge $E \subseteq VxV$, sodass $(u, v) \in E \Leftrightarrow (v, u) \in E$
  - Kanten haben keine Richtung
- **Pfadfinder**
  - Knoten v ist von Knoten u erreichbar, wenn es einen Pfad gibt
  - u ist immer von u per leerem Pfad (k=1) erreichbar
  - Länge des Pfades = $k - 1$ = Anzahl Kanten
- **Zusammenhänge**
  - Ungerichtet: Zusammenhängend wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist
  - Gerichtet: **Stark** zusammenhängend, wenn obiges auch gemäß Kantenrichtung gilt
- **Bäume und Subgraphen**
  Graph G ist ein Baum, wenn V leer ist oder wenn es einen Knoten in V gibt,
  von dem aus jeder andere Knoten eindeutig erreichbar ist (Wurzel).
  Graph $G' = (V', E')$ ist Subgraph von $G = (V, E)$, wenn $V' \subseteq V$ und $E' \subseteq E$.
- **Darstellung von Graphen**
  - Als Adjazentmatrix (1, wenn Kante von i zu j / 0, wenn keine Kante)
  - Bei ungerichteten Graphen ist Matrix spiegelsymmetrisch zur Hauptdiagonalen
  - Speicherbedarf: $\Theta(|V^2|)$

$$A = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$



  - Auch darstellbar als Array mit verketteten Listen
  - Speicherbedarf: $\Theta(|V| + |E|)$

- **Gewichtete Graphen**
  - gewichteter gerichteter Graph $G = (V, E)$
  - besitzt zusätzlich Funktion $w : E \to R$
  - Abspeichern des Werts einer Kante $w((u, v))$

1.2 Breadth-First Search (BFS)

- **Idee**
  - Besuche zuerst alle unmittelbaren Nachbarn, dann deren Nachbarn, usw.
  - Anwendung: Webcrawling, Garbage Collection,..
- **Algorithmus**

BFS(G,s) //G=(V,E) s = source node in V

```
FOREACH u in V-{s} DO
    u.color = WHITE;            // Weiß = noch nicht besucht
    u.dist = +∞                 // Setzen der Distanzen auf Unendlich
    u.pred = nil;               // Setzen der Vorgänger auf nil
s.color = GRAY;                 // Anfang bei Startnode
s.dist = 0;
s.pred = nil;
newQueue(Q);
enqueue(Q,s);
WHILE !isEmpty(Q) DO
u = dequeue(Q);
FOREACH v in adj(G,u) DO
    IF v.color == WHITE THEN
        v.color == GRAY;
        v.dist = u.dist+1;
        v.pred = u;
        enqueue(Q,v);
u.color = BLACK;                // Knoten abgearbeitet
```

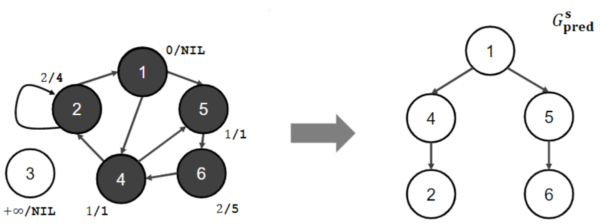
  - Laufzeit: $O(|V| + |E|)$
  - Nach Algorithmus steht in v die kürzeste Distanz von s nach v

- **Kürzeste Pfade ausgeben**

print-path(G,s,v) // Assumes that BFS(G,s) has already been executed

```
IF v == s THEN
    print s;
ELSE
    IF v.pred == nil THEN
        print 'no path from s to v'
    ELSE
        print-path(G,s,v.pred);
        print v;
```

- **Abgeleiteter BFS-Baum**



  - Subgraph $G^s_{pred} = (V^s_{pred}, E^s_{pred})$ von G:
    * $V^s_{pred} = \{v \in V | v.pred \neq nil\} \cup \{s\}$
    * $E^s_{pred} = \{(v.pred, v) | v \in V^s_{pred} - \{s\}\}$
  - G^s_{pred} enthält alle von s aus erreichbaren Knoten in G
  - Außerdem handelt es sich hier nur um kürzeste Pfade

1.3 Depth-First Search(DFS)

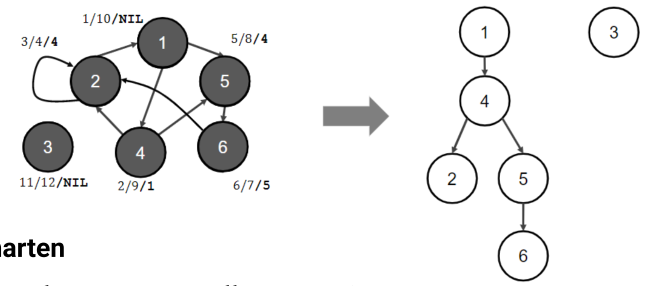
- **Idee**
  - Besuche zuerst alle noch nicht besuchten Nachfolgeknoten
  - "Laufe so weit wie möglich weg vom aktuellen Knoten"
- **Algorithmus**

DFS(G)

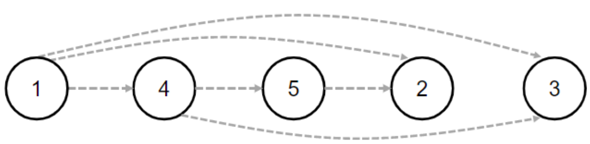
```
FOREACH u in V DO
    u.color = WHITE;
    u.pred = nil;
time = 0;                // time hier als globale Variable
FOREACH u in v DO
    IF u.color == WHITE THEN
        DFS-VISIT(G,u)   // Start eines rekursiven Aufrufs
```

```
DFS-VISIT(G,u)
time = time + 1;
u.disc = time;          // discovery time
u.color = GRAY;
FOREACH v in adj(G,u) DO
    IF v.color == WHITE THEN
        v.pred = u;
        DFS-VISIT(G,v);
u.color = BLACK;
time = time + 1;
u.finish = time;        // finish time
```

- **DFS-Wald = Menge von DFS-Bäumen**
  - Subgraph $G_{pred} = (V, E_{pred})$ von G
  - besteht aus $E_{pred} = (v.pred, v)|v \in V, v.pred \neq nil$
  - DFS-Baum gibt nicht unbedingt den kürzesten Weg wieder



- **Kantenarten**
  - Baumkanten: alle Kanten in G_{pred}
  - Vorwärtskanten: alle Kanten in G zu Nachkommen in G_{pred}, die nicht Baumkante
  - Rückwärtskanten: alle Kanten in G zu Vorfahren in G_{pred}, die nicht Baumkante
  - Kreuzkanten: alle anderen Kanten in G (inkl. Schleifen)
- **Anwendungen DFS**
  - Job Scheduling (Job X muss vor Job Y beendet sein)
  - Topologisches Sortieren
    * nur für dag (directed acyclic graph)
    * Kanten immer nur nach rechts
    * Sortierung aber nicht eindeutig

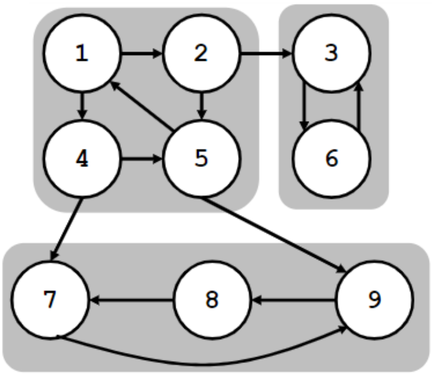



```
TOPOLOGICAL-SORT(G)
new LinkedList(L);
run DFS(G) but, each time a node is finished, insert in front of L
return L.head;
```

- **Starke Zusammenhangskomponenten**
  - Knotenmenge $C \subseteq V$, so dass

    es zwischen zwei Knoten $u, v \in C$ einen Pfad von u nach v gibt

    und es keine Menge $D \subseteq V$ mit $C \subsetneq D$ gibt, für die obiges auch gilt.



Eigenschaften:

  * Verschiedene SCC's sind disjunkt
  * Zwei SCC's sind nur in eine Richtung verbunden

  - Algorithmus:
    * DFS zweimal laufen lassen

      Einmal auf Graph G

      Einmal auf Graph $G^T = (V, E^T)$ (transponiert)
    * Dadurch bleiben die SCC's gleich, die Kanten drehen sich aber jeweils um
    * Code:

SCC(G)

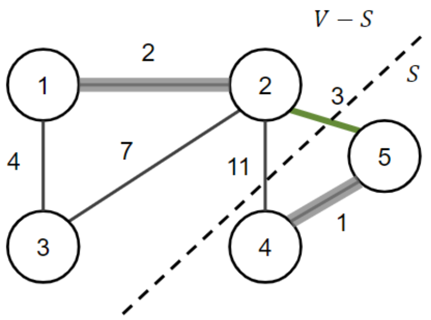
```
run DFS(G)
compute G^T
run DGS(G^T) but visit vertices in main loop
    in descending finish time from 1
output each DFS tree from above as one SCC
```

1.4 Minimale Spannbäume

- **Definition**
  - Verbindung aller Knoten miteinander
  - Minimaler Spannbaum ⇒ Minimales Gewicht
- **Allgemeiner Algorithmus**

genericMST(G,w)

```
A = ∅
WHILE A does not form a spanning tree for G DO
    find safe edge {u,v} for A
    A = A ∪{{u,v}}
return A
```



Terminologie:

* Schnitt (S, V-S) partioniert Knoten in zwei Mengen
* {u,v} überbrückt Schnitt, wenn $u \in S$ und $v \in V - S$
* Schnitt respektiert $A \subseteq E$, wenn keine Kante {u,v} aus A den Schnitt überbrückt
* {u,v} leichte Kante für (S, V-S), wenn w({u,v}) minimal für alle den Schnitt überbrückenden Kanten
* {u,v} sicher für A, wenn $A \cup \{\{u, v\}\}$ Teilmenge eines MST

- **Algorithmus von Kruskal**
  - Lässt parallel mehrere Unterbäume eines MST wachsen
  - In Worten: Suchen der "kleinsten" Kante und Zusammenfügen von Mengen, falls noch nicht geschehen
  - Laufzeit: $O(|E| \cdot log|E|)$

MST-Kruskal(G,w)

```
A = ∅
FOREACH v in V DO
    set(v) = {v};         // Menge mit sich selbst
Sort edges according to weight in nondecreasing order
FOREACH {u,v} in E according to order DO
    IF set(u) != set(v) THEN     // Mengen noch nicht verbunden
        A = A ∪ {{u,v}};
        UNION(G,u,v);    // Zusammenführen der Mengen aller Knoten aus den Sets
return A;
```

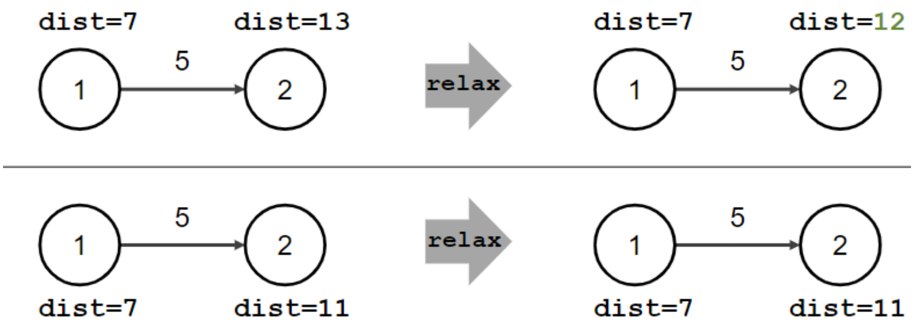
- **Algorithmus von Prim**
  - Konstruiert einen MST Knoten für Knoten
  - Fügt immer leichte Kante zu zusammenhängender Menge hinzu
  - Laufzeit: $O(|E| + |V| \cdot log|V|)$

MST-Prim(G,w,r) // r is given root

```
FOREACH v in V DO
    v.key = +∞;
    v.pred = nil;
r.key = -∞
Q = V;
WHILE !isEmpty(Q) DO
    u = EXTRACT-MIN(Q);     // smallest key value
    FOREACH v in adj(u) DO
        IF v∈Q and w({u,v})<v.key THEN
            v.key = w({u,v});
            v.pred = u;
```

1.5 Kürzeste Wege in (gerichteten) Graphen

- **Definition**
  - SSSP - Single-Source Shortest Path
  - Von Quelle s ausgehend die kürzesten Pfad zu allen anderen Knoten
  - Kürzester Pfad: Minimales Gewicht von einem zum anderen Knoten
  - BFS findet nur minimale Kantenwege (nicht Gewichtswege)
  - MST minimiert das Gesamtgewicht des Baumes (nicht zu einzelnen Kanten)
  - Negative Kantengewichte sind erlaubt, aber keine Zyklen mit negativem Gesamtgewicht
- **Gemeinsame Idee für Algorithmen - Relax**
  - Verringere aktuelle Distanz von Knoten v, wenn durch Kante (u, v) kürzer erreichbar



relax(G,u,v,w)

```
IF v.dist > u.dist + w((u,v)) THEN
    v.dist = u.dist + w((u,v));
    v.pred = u;
```

- **Bellman-Ford-Algorithmus**
  - Laufzeit: $\Theta(|E| \cdot |V|)$

Bellman-Ford-SSSP(G,s,w)

```
initSSSP(G,s,w);
FOR i = 1 TO |V|-1 DO
    FOREACH (u,v) in E DO
        relax(G,u,v,w);
FOREACH (u,v) in E DO    // Prüfung ob negativer Zyklus
    IF v.dist > u.dist+w((u,v)) THEN
        return false;
return true;
```

initSSSP(G,s,w)

```
FOREACH v in V DO
    v.dist = ∞;
    v.pred = nil;
s.dist = 0;
```

- **TopoSort für dag**
  - Erhalten des kürzesten Pfades durch das topologische Sortieren
  - Laufzeit: $\Theta(|E| + |V|)$

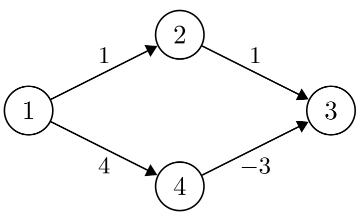
TopoSort-SSSP(G,s,w) // G muss dag sein

```
initSSSP(G,s,w);
execute topological sorting
FOREACH u in V in topological order DO
    FOREACH v in adj(u) DO
        relax(G,u,v,w);
```

- **Dijkstra-Algorithmus**
  - Voraussetzung: Keine negativen Kantengewichte
  - Laufzeit: $\Theta(|V| \cdot log|V| + |E|)$

Dijkstra-SSSP(G,s,w)

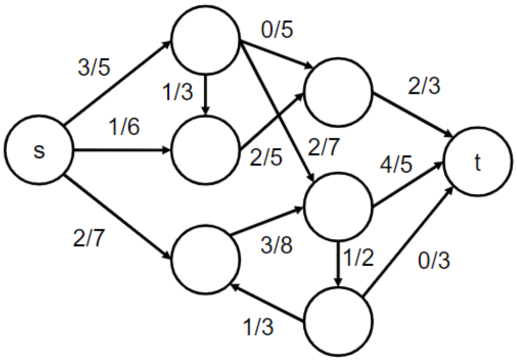
```
initSSSP(G,s,w);
Q = V;
WHILE !isEmpty(Q) DO
    u = EXTRACT-MIN(Q);      // smallest distance
    FOREACH v in adj(u) DO
        relax(G,u,v,w);
```

  - 

    * Beispiel für Problem mit negativen Kantengewisten bei Dijkstra: Dijkstra würde Pfad 1-2-3 liefern, da das Kantengewicht 4 größer als der andere Pfad ist.

1.6 Maximaler Fluss in Graphen

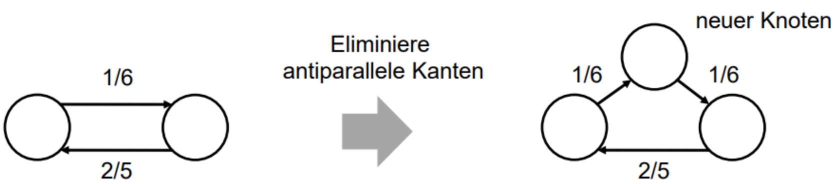
- **Idee**

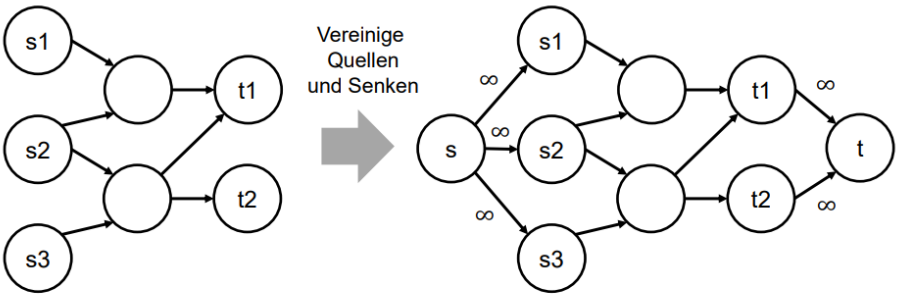


  * Kanten haben Flusswert und maximale Kapazität
  * Jeder Knoten (außer s und t) haben den gleichen eingehenden und ausgehenden Fluss
  * Ziel: Finde maximalen Fluss von s nach t
  * s: Source/Quelle
  * t: Target/Senke

  - **Flussnetzwerk:**
    Ein Flussnetzwerk ist ein gewichteter, gerichteter Graph $G = (V, E)$ mit Kapazität c, so dass $c(u, v) \geq 0$ für $(u, v) \in E$ und $c(u, v) = 0$ für $(u, v) \notin E$, mit zwei Knoten $s, t \in V$, so dass jeder Knoten von s aus erreichbar ist und t von jedem Knoten aus erreichbar ist. Damit gilt $|E| \geq |V| - 1$.
  - **Fluss:**
    Ein Fluss $f : VxV \to \mathbb{R}$ für ein Flussnetzwerk $G = (V, E)$ mit Kapazität c und Quelle s und Senke t erfüllt $0 \leq f(u, v) \leq c(u, v)$ für alle $u, v \in V$, sowie für alle $u \in V - \{s, t\}$:
    $\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$ (ausgehend = eingehend)
  - **Wert eines Flusses**
    Der Wert $|f|$ eines Flusses $f : VxV \to \mathbb{R}$ für ein Flussnetzwerk G ist:
    $|f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, s)$

- **Transformationen**

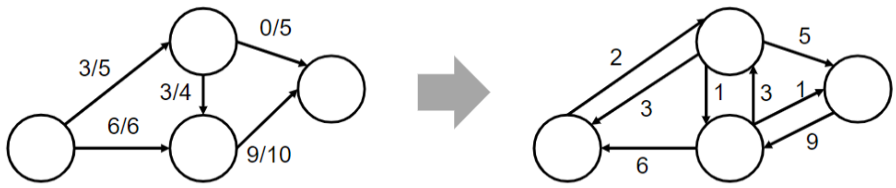




- **Restkapazitätsgraph**
  - Wird für Ford-Fulkerson benötigt
  - Restkapazität $c_f(u,v)$:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) & \text{falls } (u,v) \in E \\ f(v,u) & \text{falls } (v,u) \in E \\ 0 & \text{sonst} \end{cases}$$

  - $G_f = (V, E_f)$ mit $E_f = \{(u,v) \in VxV | c_f(u,v) > 0\}$



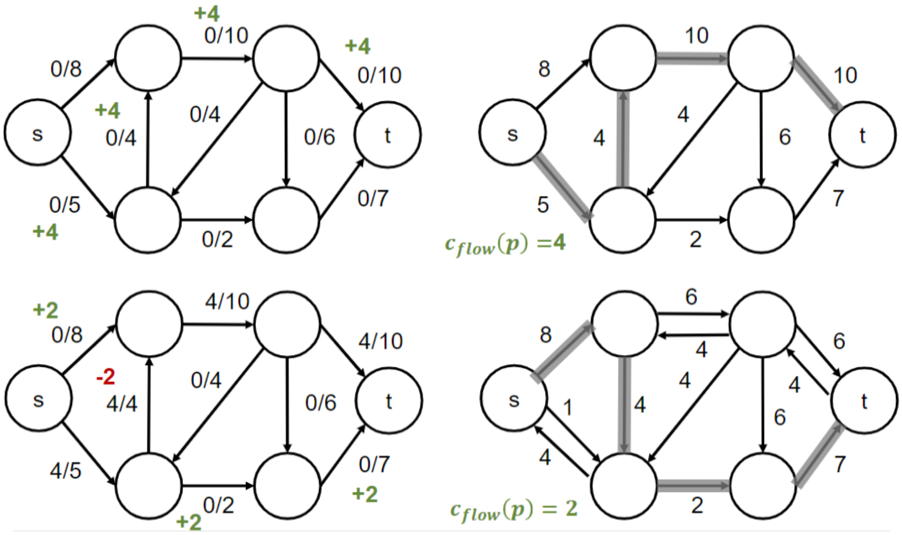
  - Suche eines Pfades von s nach t und Erhöhung aller Flüsse um niedrigsten möglichen Wert auf Pfad
- **Ford-Fulkerson-Algorithmus**
  - Idee: Suche Pfad von s nach t, der noch **erweiterbar** ist
  - Suche dieses Pfades im Restkapazitätsgraphen G_f (mögliche Zu- und Abflüsse)
  - Code:

Ford-Fulkerson(G,s,t,c)

```
FOREACH e in E do e.flow = 0;
WHILE there is path p from s to t in G_flow DO
    c_flow(p) = min {c_flow(u,v) : (u,v) in p}
    FOREACH e in p DO
        IF e in E THEN
            e.flow = e.flow + c_flow(p);
        ELSE
            e.flow = e.flow - c_flow(p);
```

  - Die Pfadsuche erfolgt z.B. per BFS oder DFS
  - Laufzeit: $O(|E| \cdot u \cdot |f^*|)$
    ($O(|V| \cdot |E|^2)$ Mit Verbesserung nach Edmonds-Karp)
    (wobei f^* maximaler Fluss und Fluss um bis zu $\frac{1}{u}$ pro Iteration wächst)

- Beispiel:



+4
0/10
+4
0/8
0/10
+4
0/4
0/4
s
0/6
t
0/5
0/7
+4
0/2
10
8
10
4
s
4
6
t
5
7
$c_{flow}(p)$ =4
2
+2
4/10
0/8
4/10
-2
0/4
s
4/4
0/6
t
4/5
0/7
+2
0/2
+2
6
8
4
6
4
4
s
1
4
6
t
4
7
$c_{flow}(p) = 2$
2