

1 Graph Algorithms

1.1 Graphen

Definition – (Endlicher) gerichteter Graph

- (endlicher) gerichteter Graph $G = (V, E)$
- besteht aus (endlicher) Knotenmenge V
- besteht aus (endlicher) Kantenmenge $E \subseteq V \times V$
- $(u, v) \in E$: Kanten von Knoten u zu v
- Kanten haben eine Richtung

Definition – Ungerichteter Graph

- (endlicher) ungerichteter Graph $G = (V, E)$
- besteht aus (endlicher) Knotenmenge V
- besteht aus (endlicher) Kantenmenge $E \subseteq V \times V$, sodass $(u, v) \in E \Leftrightarrow (v, u) \in E$
- Kanten haben keine Richtung

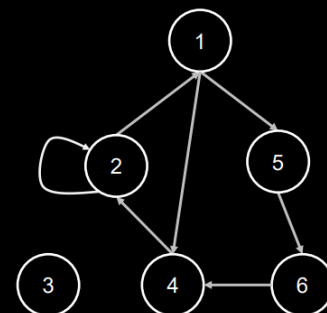
Darstellung von Graphen

- Als Adjazenzmatrix (1, wenn Kante von i zu j bzw. 0, wenn keine Kante)
- Bei ungerichteten Graphen ist Matrix spiegelsymmetrisch zur Hauptdiagonalen

\Rightarrow Speicherbedarf: $\Theta(|V|^2)$

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

(a) Darstellung als Adjazenzmatrix



(b) Grafische Darstellung

Abbildung 1: Beispielhafte Darstellung eines Graphen

- Auch darstellbar als Array mit verketteten Listen

\Rightarrow Speicherbedarf: $\Theta(|V| + |E|)$

Pfadfinder

- Knoten v ist von Knoten u erreichbar, wenn es von u aus einen Pfad über n Knoten nach v
- u ist immer von u per leerem Pfad ($k=1$) erreichbar
- Länge des Pfades $= k - 1 = \text{Anzahl Kanten}$

Definition – Zusammenhängende Graphen

- 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 $E' \subseteq E$.

Definition – Gewichtete Graphen

- gewichteter gerichteter oder ungerichteter Graph $G = (V, E)$
- besitzt zusätzlich Funktion $w : E \rightarrow R$
- Angabe des Gewichts einer Kante u nach v durch $w((u, v))$

1.2 Breadth-First Search (BFS)

Idee – Breadth-First Search

- 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

```
1 BFS(G,s) //G=(V,E) s = source node in V
2 FOREACH u in V-{s} DO
3     u.color = WHITE;           // Weiß = noch nicht besucht
4     u.dist = +∞;               // Setzen der Distanzen auf Unendlich
5     u.pred = nil;             // Setzen der Vorgänger auf nil
6 s.color = GRAY;               // Anfang bei Startnode
7 s.dist = 0;
8 s.pred = nil;
9 newQueue(Q);
10 enqueue(Q,s);
11 WHILE !isEmpty(Q) DO
12     u = dequeue(Q);
13     FOREACH v in adj(G,u) DO
14         IF v.color == WHITE THEN
15             v.color == GRAY;
16             v.dist = u.dist+1;
17             v.pred = u;
18             enqueue(Q,v);
19     u.color = BLACK;           // Knoten abgearbeitet
```

Farben:

- WHITE: Knoten noch nicht besucht
 - GRAY: Knoten in Queue für nächsten Schritt
 - BLACK: Knoten ist fertig
-
- 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
```

```
1 IF v == s THEN
2   print s;
3 ELSE
4   IF v.pred == nil THEN
5     print "no path from s to v"
6   ELSE
7     print-path(G,s,v.pred);
8     print v;
```

Abgeleiteter BFS-Baum



Abbildung 2: Beispiel BFS-Baum

- Subgraph $G_{pred}^s = (V_{pred}^s, E_{pred}^s)$ von G :
 - $V_{pred}^s = \{v \in V | v.pred \neq nil\} \cup \{s\}$
 - $E_{pred}^s = \{(v.pred, v) | v \in V_{pred}^s - \{s\}\}$
- G_{pred}^s 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 – Depth-First Search**
- Besuche zuerst alle noch nicht besuchten Nachfolgeknoten
 - "Laufe so weit wie möglich weg vom aktuellen Knoten"

Algorithmus

DFS(G)

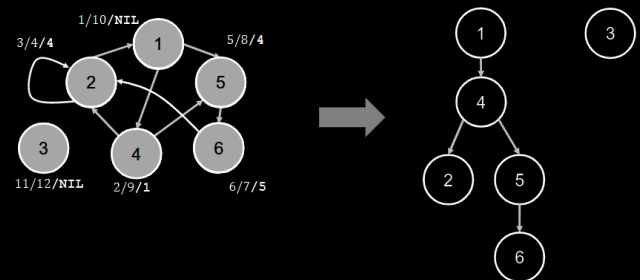
```
1  FOREACH u in V DO
2      u.color = WHITE;
3      u.pred = nil;
4  time = 0; // time hier als globale Variable
5  FOREACH u in v DO
6      IF u.color == WHITE THEN
7          DFS-VISIT(G,u) // Start eines rekursiven Aufrufs
```

DFS-VISIT(G,u)

```
1  time = time + 1;
2  u.disc = time; // discovery time
3  u.color = GRAY;
4  FOREACH v in adj(G,u) DO
5      IF v.color == WHITE THEN
6          v.pred = u;
7          DFS-VISIT(G,v);
8  u.color = BLACK;
9  time = time + 1;
10 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 keine Baumkante sind

Rückwärtskanten alle Kanten in G zu Vorfahren in G_{pred} , die keine Baumkante sind (inkl. Schleifen)

Kreuzkanten alle anderen Kanten in G

Abbildung 3: Beispiel DFS-Wald

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



Abbildung 4: Beispiel Topologisches Sortieren

TOPOLOGICAL-SORT(G)

```
1 new LinkedList(L);
2 run DFS(G) but, each time a node is finished, insert in front of L
3 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)

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

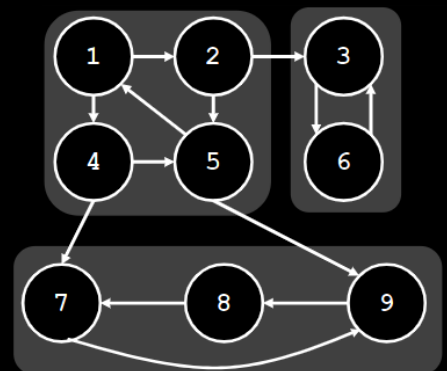


Abbildung 5: Beispiel Starke Zusammenhangskomponenten

1.4 Minimale Spannbäume

Definition – Minimaler Spannbaum

- Verbindung aller Knoten miteinander
- Minimaler Spannbaum \Rightarrow Minimales Gewicht

Allgemeiner Algorithmus

genericMST(G,w)

```
1 A =  $\emptyset$ 
2 WHILE A does not form a spanning tree for G DO
3     find safe edge {u,v} for A
4     A = A  $\cup$  {{u,v}}
5 return A
```

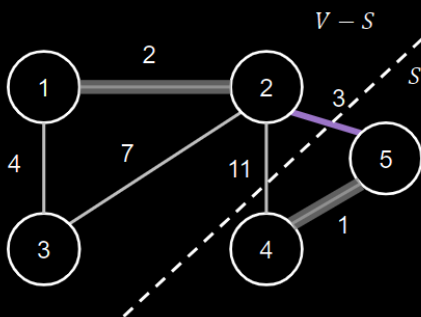


Abbildung 6: Beispiel Ausführung genericMST()

Terminologie:

- Schnitt (S, V-S) partitioniert 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

Idee – Algorithmus von Kruskal Suchen der „kleinsten“ Kante und Zusammenfügen von Mengen, falls Mengen ungleich sind

- Lässt parallel mehrere Unterbäume eines MST wachsen
- Laufzeit: $O(|E| \cdot \log|E|)$

MST-Kruskal(G,w)

```
1 A =  $\emptyset$ 
2 FOREACH v in V DO
3     set(v) = {v}; // Menge mit sich selbst
4 Sort edges according to weight in nondecreasing order
5 FOREACH {u,v} in E according to order DO
6     IF set(u)  $\neq$  set(v) THEN // Mengen noch nicht verbunden
7         A = A  $\cup$  {{u,v}};
8         UNION(G,u,v); // Zusammenführen der Mengen aller Knoten aus den Sets
9 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

```
1  FOREACH  $v$  in  $V$  DO
2       $v.key = +\infty$ ;
3       $v.pred = nil$ ;
4   $r.key = -\infty$ 
5   $Q = V$ ;
6  WHILE !isEmpty( $Q$ ) DO
7       $u = \text{EXTRACT-MIN}(Q)$ ;      // smallest key value
8      FOREACH  $v$  in  $\text{adj}(u)$  DO
9          IF  $v \in Q$  and  $w(\{u, v\}) < v.key$  THEN
10              $v.key = w(\{u, v\})$ ;
11              $v.pred = u$ ;
```

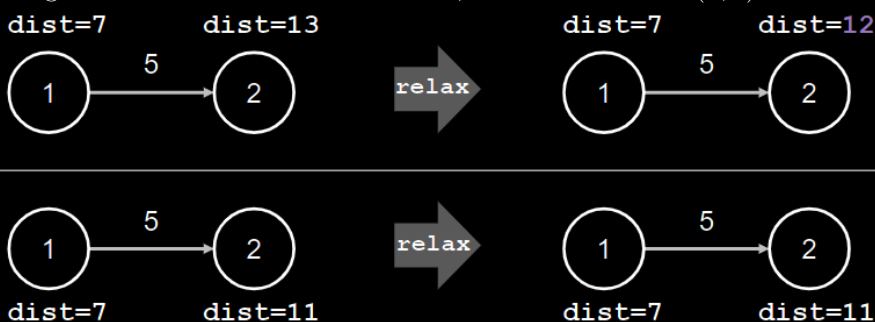
1.5 Kürzeste Wege in (gerichteten) Graphen

Definition – SSSP

- SSSP - Single-Source Shortest Path
- Von Quelle s ausgehend die kürzesten Pfad zu allen anderen Knoten
- Kürzester Pfad: Pfad mit minimalem Gesamtgewicht 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)

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


Bellman-Ford-Algorithmus

Laufzeit: $\Theta(|E| \cdot |V|)$

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

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

initSSSP(G,s,w)

```
1  FOREACH  $v$  in  $V$  DO
2       $v.dist = \infty$ ;
3       $v.pred = nil$ ;
4   $s.dist = 0$ ;
```

TopoSort für dag

Idee — TopoSort für dag Erhalten des kürzesten Pfades durch das topologische Sortieren

Laufzeit: $\Theta(|E| + |V|)$

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

```
1  initSSSP( $G,s,w$ );
2  execute topological sorting
3  FOREACH  $u$  in  $V$  in topological order DO
4      FOREACH  $v$  in adj( $u$ ) DO
5          relax( $G,u,v,w$ );
```

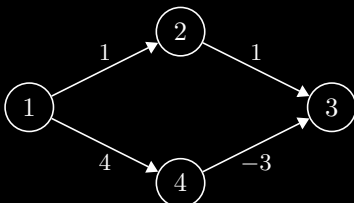
Dijkstra-Algorithmus

Voraussetzung: Keine negativen Kantengewichte

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

Dijkstra-SSSP(G,s,w)

```
1  initSSSP( $G,s,w$ );
2   $Q = V$ ;
3  WHILE !isEmpty( $Q$ ) DO
4       $u = \text{EXTRACT-MIN}(Q)$ ;    // smallest distance
5      FOREACH  $v$  in adj( $u$ ) DO
6          relax( $G,u,v,w$ );
```



- Beispiel für Problem mit negativen Kantengewichten 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 – Maximaler Fluss im Graphen

- 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

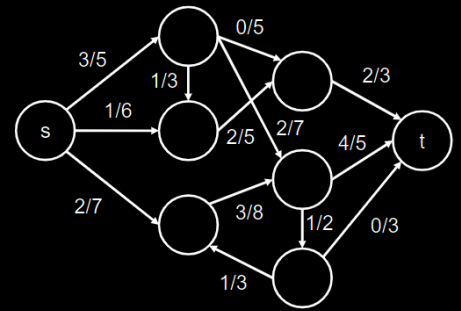


Abbildung 7: Beispiel Flussnetzwerk

Definition – 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 $|E| \geq |V| - 1$.

Definition – Fluss

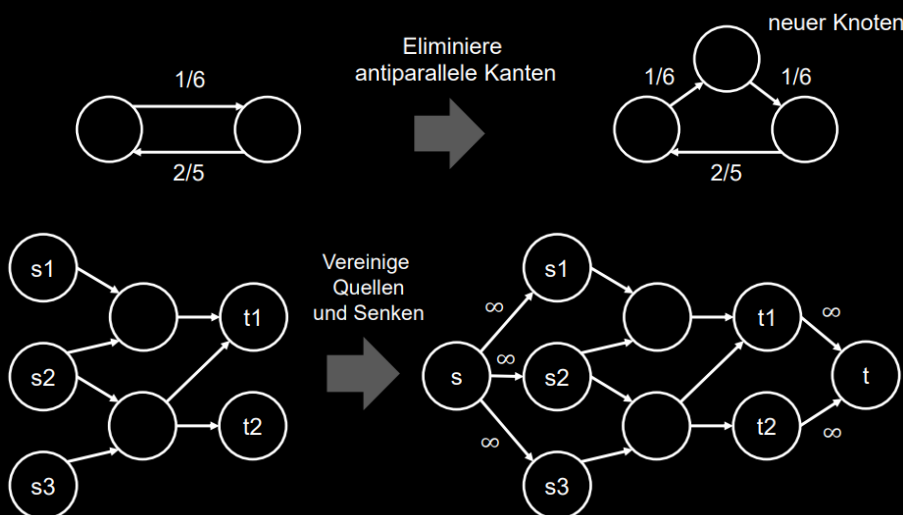
Ein Fluss $f : V \times V \rightarrow \mathbb{R}$ für ein Flussnetzwerk $G = (V, E)$ mit Kapazität c und Quelle s und Senke t erfüllt $0 \leq f(u, v) \leq c(u, v)$ für alle $u, v \in V$, sowie für alle $u \in V - \{s, t\}$:

$$\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u) \quad (\text{ausgehend} = \text{eingehend})$$

Definition – Wert eines Flusses Der Wert $|f|$ eines Flusses $f : V \times V \rightarrow \mathbb{R}$ für ein Flussnetzwerk G ist:

$$|f| = \sum_{v \in V} f(s, v) = \sum_{v \in V} f(v, t)$$

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 $E_f = \{(u, v) \in V \times V \mid c_f(u, v) > 0\}$ Suche eines Pfades von s nach t und Erhöhung aller Flüsse um

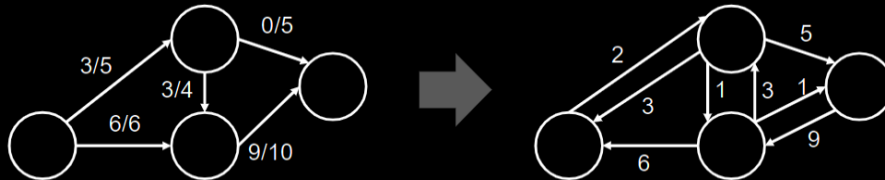


Abbildung 8: Beispiel Restkapazitätsgraph

niedrigsten möglichen Wert auf Pfad

Ford-Fulkerson-Algorithmus

Idee – Ford-Fulkerson-Algorithmus

- 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)

```
1  FOREACH e in E do e.flow = 0;
2  WHILE there is path p from s to t in  $G_{flow}$  DO
3       $c_{flow}(p) = \min \{c_{flow}(u, v) : (u, v) \text{ in } p\}$ 
4      FOREACH e in p DO
5          IF e in E THEN
6              e.flow = e.flow +  $c_{flow}(p)$ ;
7          ELSE
8              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:

