6. Graph Algorithms

Graphen

(Endliche) gerichtete Graphen

Besteht aus:

1. (Endliche) Kontenmenge V
2. (Endliche) Kantenmenge $E \subseteq V \times V$; $(u, v) \in E$: Kante von Knoten u zu v

- Schleifen, Zyklen, isolierte Knoten möglich
- Keine Mehrfachkanten zwischen Knoten, Anordnung der Knoten irrelevant

Ungerichtete Graphen

Endlicher Graph, aber $(u, v) \in E \Leftrightarrow (v, u) \in E$
Alternative Darstellung: $\{u, v\}$ anstelle von $(u, v), (v, u)$

Pfadfinder

Konten v ist von Knoten u im Graphen $G = (V, E)$ erreichbar, wenn es Pfad $(w_1, \ldots, w_k) \in V^k$ gibt, sodass $(w_i, w_{i+1}) \in E$ für $i = 1, 2, \ldots, k-1, w_1 = u, w_k = v$
u ist immer von u mit leerem Pfad $k = 1$ erreichbar (Schleife)
Länge des Pfades = $k - 1$ = Anzahl Kanten
$(w_1, \ldots, w_k)$ ist kürzester Pfad von u nach v, wenn es keinen kürzeren Pfad gibt.
$shortest(u, v) :=$ Länge eines kürzesten Pfades von u nach v
Kürzester Pfad muss nicht eindeutig sein

Zusammenhänge

Ungerichteter Graph ist zusammenhängend, wenn jeder Knoten von jedem anderen Knoten aus erreichbar ist.
Gerichteter Graph ist stark zusammenhängend, wenn jeder Knoten von jedem anderen Knoten aus (gemäß Kantenrichtung) erreichbar ist.

Graphen und Bäume

Graphenbäume haben keine Ordnung auf Kindern
G ist ein Baum, wenn V leer ist oder es ein $r \in V$ (Wurzel) gibt, sodass jeder Knoten v von r per eindeutigem Pfad erreichbar ist.

Subgraphen

G' muss wieder ein Graph gleichen Typs (gerichtet/ungerichtet) sein
Graph $G' = (V', E')$ ist Subgraph/Teilgraph/Untergraph des Graphen $G = (V, E)$, wenn $V' \subseteq V$ und $E' \subseteq E$.

Darstellung von Graphen

Adjazenzmatrix

i Zeile, j Spalte

$$A[i,j] = \begin{cases} 1 & \text{wenn Kante von } i \text{ zu } j \\ 0 & \text{sonst} \end{cases}$$

- Bei ungerichteten Graphen ist Matrix spiegelsymmetrisch zur Hauptdiagonalen
- Speicherbedarf = $\Theta(|V|^2)$
- Eintrag $a_{i,j}^{(m)}$ der m-ten Potenz A^m der Adjazenzmatrix A eines Graphen gibt die Anzahl der Wege an, die von Knoten i zu j entlang genau m Kanten führen

Adjazenzliste

Array mit verketteten Listen (sortiert/unsortiert)
Speicherbedarf = $\Theta(|V| + |E|)$

Gewichtete Graphen

Besitzt zusätzlich Funktion $w : E \to \mathbb{R}$
Ungerichtete, gewichtete Graphen: $w((u,v)) = w((v,u))$ f.a. $(u,v) \in E$
Zu Kanten (u,v) ist noch Wert $w((u,v))$ abzuspeichern

Breitensuche/Breadth-First Search (BFS)

Besuche zuerst unmittelbare Nachbarn, dann deren Nachbarn usw.
Anwendungen: Web Crawling, Broadcasting, Garbage Collection, ...

Algorithmus

```
BFS(G,s) //G=(V,E), s=source node in V
        FOREACH u in V-{s} DO
                u.color=WHITE; u.dist=+∞; u.pred=NIL;
        s.color=GRAY; 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;
```

- `dist`: Distanz von `s`
- `pred`: Vorgängerknoten
- `WHITE`: Knoten noch nicht besucht
- `GRAY`: in Queue für nächsten Schritt
- `BLACk`: fertig
- `adj(G,u)`: Liste aller Knoten $v \in V$ mit $(u, v) \in E$, Reihenfolge egal

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;
```

Laufzeit ohne BFS = $O(|V|)$

Abgeleiteter BFS-Baum

Definiere Subgraph $G^s_{pred} := V^s_{pred}, E^s_{pred}$ von G durch:

- $V^s_{pred} := \{v \in V : v.pred \neq NIL\} \cup \{s\}$
- $E^s_{pred} := \{(v.pred, v) : v \in (V^s_{pred} \backslash \{s\})\}$ und $(v, v.pred)$ für ungerichtete Graphen G^s_{pred} ist BFS-Baum zu G:
- Enthält alle von s aus erreichbaren Knoten in G.
- F.a. $v \in V^s_{pred}$ existiert genau ein Pfad von s in G^s_{pred}, der ein kürzester Pfad von s zu v in G ist.

Tiefensuche/Depth-First Search (DFS)

Zuerst alle noch nicht besuchten Nachfolgeknoten besuchen
(So weit wie möglich vom aktuellen Knoten weglaufen)

```
DFS(G) //G=(V,E)
        FOREACH u in V DO
                u.color=WHITE;
                u.pred=NIL;
        time=0;
        FOREACH u in V DO
                IF u.color==WHITE THEN
                        DFS-VISIT(G,u)

DFS-VISIT(G,u)
        time=time+1;
        u.disc=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;
```

- `time`: globale Variable
- `disc`: discovery time
- `finish`: finish time
  Laufzeit = $O(|V| + |E|)$
  Standardordnung der Knoten ist gemäß der Nummer

DFS-Wald = Menge von DFS-Bäumen

Subgraph $G_{pred} = (V, E_{pred})$ von G:
$E_{pred} := \{(v.pred, v) : v \in V, v.pred \neq NIL\}$ (ungerichtete Graphen: auch $(v, v.pred)$)
DFS-Baum gibt nicht unbedingt den kürzesten Weg wieder

Charakterisierung der Kanten in G

Zeichne in DFS-Baum G_{pred} auch restliche Kanten ein, dann ergeben sich folgende Typen:

1. Baumkante:
   alle Kanten in G_{pred}
2. Vorwärtskante:
   alle Kanten in G zu Nachkommen in G_{pred}, die keine Baumkante sind
3. Rückwärtskante:

alle Kanten in G zu Vorfahren in G_{pred}, die keine Baumkante sind, alle Schleifen

4. Kreuzkante:
   Alle anderen Kanten in G

Kantenart erkennen

Man kann zu jedem Zeitpunkt während der DFS die soeben betrachte Kante (u, v) untersuchen. Sie ist:

1. Baumkante, wenn `v.color==WHITE`
2. Rückwärtskante, wenn `v.color==GRAY`
3. Vorwärtskante, wenn `v.color==BLACK` und `u.disc<v.disc`
4. Kreuzkante, wenn `v.color==BLACK` und `v.disc<u.disc`

Kantenarten in ungerichteten Graphen

In einem ungerichteten Graphen G entstehen durch DFS nur Baum- und Rückwärtskanten.

DFS Anwendungen

Job Scheduling

Graph:

- Knoten sind Jobs
- Kante: Job X muss vor Job Y beendet sein
  Problem: In welcher Reihenfolge sollen die Jobs bearbeitet werden?

Anwendungen

- Spreadsheets: Formeln aktualisieren
- makefiles
- Tensorflow: Computation Graphs

Abstrakte Modellierung: Topologische Sortierung

- Topologische Sortierung funktioniert nur für dags (gerichtete Graphen ohne Zyklen).
- Topologische Sortierung eines dag $G = (V, E)$:
  Sortiere Knoten in linearer Ordnung, so dass für alle Knoten $u, v \in V$ gilt, dass u vor v in der Ordnung kommt, wenn $(u, v) \in E$.
- Kanten gehen nur nach rechts
- Sortierung muss nicht eindeutig sein

Topologisches Sortieren mittels DFS

```
TOPOLOGICAL-SORT(G) // G=(V,E) dag
        newLinkedList(L);
        run DFS(G) but, each time a node is finished, insert in front of L
        return L.head
```

Laufzeit = $O(|V| + |E|)$, da Einfügen in Liste vorne in Zeit $O(1)$
Wenn die Anzahl eingehender Kanten bekannt ist:

```
Kahn1962(G) //G=(V,E) dag, inbound[u]=#edges to u
        WHILE !isEmpty(V) DO
                pick vertex u with inbound[u]==0
                add u at the end of a list L
                inbound[v]=inbound[v]-1 for each v with (u,v) in E
                remove u from V and all (u,*) from E
```

Laufzeit = $O(|V| + |E|)$

Starke Zusammenhangskomponenten

Eine starke Zusammenhangskomponente eines gerichteten Graphen $G = (V, E)$ ist eine Knotenmenge $C \subseteq V$, sodass

1. es zwischen je zwei Knoten $u, v \in C$ einen Pfad von u nach v gibt
2. es keine Menge $D \subseteq V$ mit $C \subset D$ gibt, für die 1. auch gilt (C ist maximal)
   Ein Graph kann mehrere starke Zusammenhangskomponenten (strongly connected components, SCCs) haben.

Eigenschaften

Verschiedene SCCs sind disjunkt
Zwei SCCs sind nur in eine Richtung verbunden:
Seien C, D SCCs, $u, v \in C, w, x \in D$ und gebe es einen Pfad $u \to w$.
Dann kann es keinen Pfad $x \to v$ geben (C, D sonst identisch).

SCC Algorithmus

Lasse zweimal DFS laufen:
Auf G und auf transponiertem Graphen $G^T := (V, E^T)$ mit $E^T := \{(v, u) : (u, v) \in E\}$ (Kanten umgedreht)

SCCs im transponierten Graphen

- SCCs bleiben identisch in G, G^T
- nur Übergänge drehen sich um

Algorithmus

```
SCC(G) // G=(V,E) directed graph
        run DFS(G)
        compute G_T
        run DFS(G_T) but visit vertices in main loop in descending finish time
from 1
        output each DFS tree in 3 as one SCC
```

- `G_T` ist G^T
- Laufzeit = $O(|V| + |E|)$
- Graph muss nicht zusammenhängend sein

Idee

- Nach 1. Durchführung der DFS liegt Knoten mit höchster finish time in einem SCC
- In transponiertem Graphen kann man diesen SCC nicht verlassen
- Nachdem nun erster SCC abgearbeitet ist, wird der verbleibende Knoten mit der höchsten finish time verwendet, wiederhole bis alle Knoten abgearbeitet sind
- Gib alle Bäume aus

Algorithmendesign

- Kosarajus Algorithmus:
  Hier betrachtet, zwei DFS Ausführungen
- Tarjans Algorithmus, Pfad-basierter Algorithmus:
  jeweils nur eine DFS-Ausführung, speichern sich mehr Informationen unterwegs
- Asymptotisch alle gleich schnell
- Tarjans und pfad-basierter Algorithmus schneller in Praxis

Minimale Spannbäume

Für einen zusammenhängenden, ungerichteten, gewichteten Graphen $G = (V, E)$ mit Gewichten w ist der Subgraph $T = (V, E_T)$ von G ein Spannbaum, wenn T azyklisch ist und alle Knoten verbindet.
Der Spannbaum ist minimal, wenn $w(T) := \sum_{\{c,v\} \in E_T} w(\{u, v\})$ minimal f.a. Spannbäume von G ist.
MST muss nicht unbedingt minimale Kantenanzahl haben, da das Gesamtgewicht minimiert wird.

Anwendung: Broadcast in Netzwerken

Broadcast: Verteile Nachricht an alle Switches
Zu verhindern: "Broadcast Storm": Nachricht stets zyklisch weiterverteilt
Spanning Tree Protocal:
Wähle "Root Bridge" als Wurzel des Spannbaums
Gewicht abhängig von Geschwindigkeit und Entfernung von Root Bridge

Allgemeiner MST-Algorithmus

```
genericMST(G,w) // G=(V,E) undirected, connected graph, w weight function
        A=∅
        WHILE A does not form a spanning tree for G DO
                find safe edge {u,v} for A
                A = A U{{u,v}}
        return A
```

- `A` Teilmenge der Kanten eines MST
- Kante `{u,v}` ist sicher, wen $A \cup \{\{u,v\}\}$ noch Teilmenge eines MST ist

Terminologie

- Schnitt $(S, V \backslash S)$ partitioniert Knoten des Graphen in 2 Mengen
- $\{u,v\}$ überbrückt Schnitt $(S, V \backslash S)$, wenn $u \in S, v \in (V \backslash S)$
- Schnitt $(S, V \backslash S)$ respektiert $A \subseteq E$, wenn keine Kante $\{u,v\} \in A$ Schnitt überbrückt
- $\{u,v\}$ leichte Kante für $(S, V \backslash S)$, wenn $w(\{u,v\})$ minimal f.a. den Schnitt überbrückenden Kanten

Leicht = sicher

$\{u,v\}$ sicher für A, wenn $A \cup \{\{u,v\}\}$ Teilmenge eines MST
Sei A Teilmenge eines MST, $(S, V \backslash S)$ Schnitt, der A respektiert, $\{u,v\}$ eine leichte Kante, die den Schnitt überbrückt.
Dann ist $\{u,v\}$ sicher für A.

Algorithmendesign

Leicht = sicher $\rightsquigarrow$ Greedy-Strategie für konkrete Implementierung

- Kruskal lässt parallel mehrere Unterbäume eines MST wachsen
- Prim konstruiert MST Knoten für Knoten
  Beide Algorithmen funktionieren auch für negative Kantengewichte

Algorithmus von Kruskal

```
MST-Kruskal(G,w) // G=(V,E) undirected, connected graph, w weight function
        A=∅
        FOREACH v in V DO set(v)={v};
        Sort edges according to weight in nondecreasing order
        FOREACH {u,v} in E according to order DO
                IF set(u)!=set(v) THEN // u and v connected otherwise,
                        A = A U{{u,v}}     // adding {u,v} would create cycle
                        UNION(G,u,v);
        return A
```

- Jeder Knoten hat Attribut `set`
- `UNION(G,u,v)` setzt $set(w) = set(u) \cup set(v)$ f.a. Knoten $w \in set(u) \cup set(v)$
- $set(u), set(v)$ sind disjunkt oder identisch

Laufzeit

Mit vielen Optimierungen (komplexere Datenstruktur, ...): Laufzeit = $O(|E| \cdot \log |E|)$
Laufzeit = $O(|E| \cdot \log |V|)$

Algorithmus von Prim

```
MST-Prim(G,w,r) // r root in V, MST given through v.pred values
        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;
```

- Algorithmus fügt, beginnend mit Wurzelknoten, immer leichte Kante zu zusammenhängender Menge hinzu
- Auswahl der nächsten Kante gemäß `key`-Wert, der stets aktualisiert wird
- A implizit definiert durch $A = \{\{v, v.pred\} : v \in (V \backslash (\{r\} \cup Q))\}$

Laufzeit

Laufzeit = $O(|E| + |V| \cdot \log |V|)$
mit vielen Optimierungen, speziell Fibonacci-Heaps

Kürzeste Wege in (gerichteten) Graphen

Single-Source Shortest Path (SSSP)

Finde von Quelle s aus jeweils den kürzesten Pfad zu allen anderen Knoten
Kürzester Pfad: Gesamtgewicht reduzieren, Länge egal
Länge eines Pfades $p := (v_1, \ldots, v_k) \in V^k$ von $u = v_1$ zu $v = v_k$:

- $w(p) := \sum_{i=1}^{k-1} w((v_i, v_{i+1}))$
- $shortest(u, v) := \begin{cases} \min\{w(p) : p \text{ Pfad von } u \text{ nach } v\} & \text{wenn } v \text{ erreichbar von } u \\ \infty & \text{sonst} \end{cases}$

SSSP vs. BFS, DFS, MST

- BFS, DFS beachten Kantengewichte nicht
- BFS findet kürzeste "Kantenwege", nicht kürzeste "Gewichtswege"
- MST (für ungerichtete Graphen) minimiert Gesamtgewicht des Baumes
  - Dies bedeutet nicht unbedingt eine Minimierung der Gewichtswege

Zyklen und negative Kantengewichte

- Negative Kantengewichte sind erlaubt
- Zyklen mit negativem Gesamtgewicht sind nicht erlaubt
- Kürzeste Pfade können keine Zyklen mit positivem Gesamtgewicht haben
- Kürzeste Pfade enthalten höchstens Zyklen mit Gesamtgewicht 0
- Es gibt stets einen kürzesten Pfad mit Kantenlänge $\leq |V| - 1$

Kürzeste Teilpfade

- Teilpfad $s \to x$ eines kürzesten Pfades $s \to x \to z$ ist auch kürzester Pfad von s nach x
- Sonst gäbe es ja kürzeren Pfad von s nach x

Algorithmen für SSSP

- Gemeinsame Idee: Lockerung/Relaxation
- Bellman-Ford funktioniert allgemein, auch ungerichtet
  - Laufzeit = $O(|V| \cdot |E|)$
- Algorithmus für dags (gerichtete, azyklische Graphen) funktioniert nur für dags
  - Laufzeit = $O(|V| + |E|)$
- Dijkstra funktioniert nur für nicht-negative Kantengewichte, auch ungerichtet
  - Laufzeit = $O(|V| \cdot \log |V| + |E|)$

relax, initSSSP

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

Verringere aktuelle Distanz von Knoten v, wenn durch Kante (u, v) kürzere Distanz erreichbar

```
initSSSP(G,s,w)
        FOREACH v in V DO
                v.dist=∞;
                v.pred=NIL;
        s.dist=0;
```

Zu Beginn: Distanz = ∞ f.a. Knoten $\neq s$

Bellman-Ford-Algorithmus

```
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 // check for negative cycle
                IF v.dist > u.dist+w((u,v)) THEN
                        return false;
        return true;
```

Laufzeit = $\Theta(|E| \cdot |V|)$ wegen geschachtelter `FOR`-Schleifen
Testet, ob negativer Zyklus erreichbar, gibt `false` zurück, falls dies zutrifft

SSSP mittels topologischer Sortierung

```
TopoSort-SSSP(G,s,w) // G dag
        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);
```

Kanten auf dem kürzesten Pfad werden nacheinander gelockert
Laufzeit = $\Theta(|E| + |V|)$

Dijkstra-Algorithmus

```
Dijkstra-SSSP(G,s,w)
        initSSSP(G,s,w);
        Q=V; // let S=V\Q, Q is a set
        WHILE !isEmpty(Q) DO
                u=EXTRACT-MIN(Q); // wrt. dist
                FOREACH v in adj(u) DO
                        relax(G,u,v,w);
```

Voraussetzung: $w((u,v)) \leq 0$ f.a. u, v, also alle Kanten
Laufzeit = $\Theta(|V| \cdot \log |V| + |E|)$ mittels Fibonacci-Heaps

Negative Kanten

Wenn man den absoluten Wert der kleinsten Kante zu allen Werten addiert, addiert man den Wert so oft, wie Anzahl Kanten auf dem kürzesten Weg

A*-Algorithmus

- Suche kürzesten Weg von s zu einem Ziel t
- Dijkstra sucht lokal vom gegenwärtigen Punkt aus günstigsten nächsten Schritt, ignoriert aber Zielrichtung
- Füge Heuristik hinzu, die vom Ziel her denkt

```
A*(G,s,t,w)
        init(G,s,t,w);
        Q=V; //let S=V-Q
        WHILE !isEmpty(Q) DO
                u=EXTRACT-MIN(Q);
                IF u==t THEN break;
                FOREACH v in adj(u) DO
                        relax(G,u,v,w);
```

- Jeder Knoten u bekommt Attribut `u.heur` zugewiesen, z.B. Abstand Luftlinie zum Ziel
- `EXTRACT-MIN` sucht Minimum über `u.dist + u.heur`
- Auch nicht-negative Kantengewichte
- Dijkstra ist A* mit Heuristik 0
- A* mit monotoner Heuristik ist Dijkstra mit Kantengewichten `w(u,v)+v.heur-u.heur` und `s.dist=s.heur`
  - A *und Dijkstra wählen dann gleiche Knoten, da* `u.dist+u.heur==u.dist`, `t.dist==t.dist` *(linke Seite A*, rechte Seite Dijkstra)

Bedingungen für optimale Lösung

1. Heuristik überschätzt nie Kosten: `u.heur` $\leq shortest(u,v)$, `t.heur == 0`
2. Heuristik ist monoton: F.a. $u,v \in E$ gilt `u.heur` $\leq$ `w(u,v) + v.heur`

Maximaler Fluss in Graphen

Netzwerkflüsse

Idee

Kanten haben (aktuellen) Flusswert und (maximale) Kapazität
Ziel: Finde maximalen Fluss von s nach t
Jeder Knoten außer s und t hat gleichen ein- und ausgehenden Fluss

Ein Flussnetzwerk ist ein gewichteter, gerichteter Graph $G = (V, E)$ mit Kapazität(sgewicht) c, sodass $c(u, v) \geq 0$ für $(u, v) \in E$ und$c(u, v) = 0$ für $(u, v) \notin E$; mit zwei Knoten $s, t \in V$ (Quelle, Senke), sodass jeder Knoten von s aus erreichbar ist und t von jedem Knoten aus erreichbar ist.
Ein Fluss $f : V \times V \to \mathbb{R}$ für ein Flussnetzwerk $G = (V, E)$ mit Kapazität c, Quelle s, Senke t erfüllt:

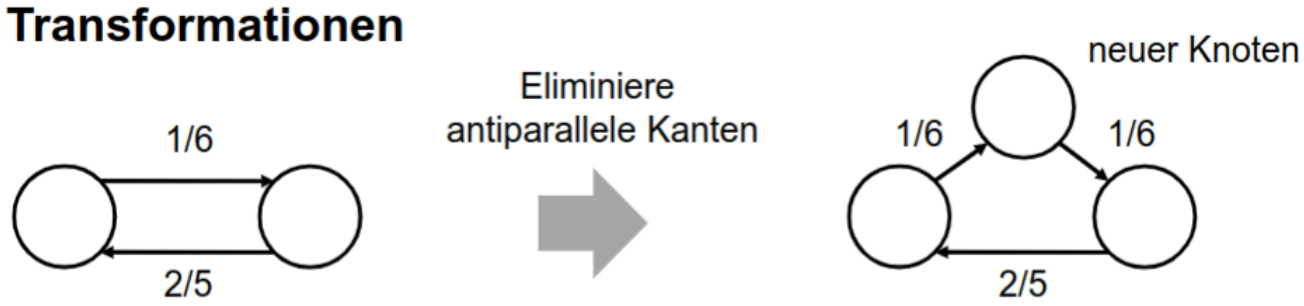
- $0 \leq f(u, v) \leq c(u, v)$ f.a. $u, v \in V$ (Fluss zwischen 0 und max. Kapazität)
- $\sum_{v \in V} f(u, v) = \sum_{v \in V} f(v, u)$ f.a. $u \in (V \backslash \{s, t\})$ (gleicher ein- und ausgehenden Fluss)

Maximale Flüsse

Der Wert $|f|$ eines Flusses $f : V \times V \to \mathbb{R}$ für ein Flussnetzwerk $G = (V, E)$ mit Kapazität c, Quelle s, Senke t ist $|f| := \sum\limits_{v \in V} f(s, v) - \sum\limits_{v \in V} f(v, s)$

Transformationen

Antiparallele Kanten sind durch neue Knoten zu elimieren:



Mehrere Quellen und Senken sind zu vereinigen, indem eine neue Quelle/Senke mit Verbindungen zu allen andere und unendlicher Kapazität kreiert wird.

Ford-Fulkerson-Methode

Suche Pfad von s nach t, der noch erweiterbar (bzgl. des Flusses) ist, Pfad wird im Restkapazitätsgraphen gesucht, der die möglichen Zu- und Abflüsse beschreibt

Reste

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

Fall 1: Wie viel eingehenden Fluss über (u,v) könnte man zu v hinzufügen?
Fall 2: Wie viel abgehenden Fluss über (u,v) kann man wegnehmen und so zu v hinzufügen?
Wohldefiniert, da nach Transformationen nicht mehr sowohl (u,v) als auch (v,u) im Netzwerk sind

Restkapazitäts-Graph

$G_f := (V, E_f)$ mit $E_f := \{(u,v) \in v \times V : c_f(u,v) > 0\}$
Im Restkapazitäts-Graphen sind antiparallele Kanten erlaubt.
#TODO Grafik einfügen

Restkapazitäten ausnutzen

Finde Pfad von s zu t in G_f und erhöhe (für Kanten in G) bzw. erniedrige (für Gegenrichtung) um Minimum $c_f(u,v)$ aller Werte auf dem Pfad in G

Ford-Fulkerson-Algorithmus

```
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)
```

- G_f ist `G_flow`, c_f ist `c_flow`
- Pfadsuche z.B. per DFS/BFS
- Laufzeit = $O(|E| \cdot u \cdot |f^*|)$
  - f^* ist maximaler Fluss, Fluss wächst um bis zu $\frac{1}{u}$ pro Iteration
- Laufzeit = $O(|V| \cdot |E|^2)$ mit Verbesserung
- Es wird ein zufälliger Pfad in G_f verwendet

Max-Flow Min-Cut Theorem

Sei $f : V \times V \to \mathbb{R}$ Fluss für ein Flussnetzwerk $G = (V, E)$ mit Kapazität c, Quelle s, Senke t. Dann sind äquivalent:

1. f ist ein maximaler Fluss für G.
2. Der Restkapazitätsgraph G_f enthält keinen erweiterbaren Pfad.
3. $|f| = \min_{S \subseteq V : s \in S, t \notin S} c(S, V \backslash S)$ mit $s \in S$, $t \in (V \backslash S)$, wobei $c(S, V \backslash S)$ für $s \in S$, $t \in (V \backslash S)$ die Kapazität eines Schnitts $(S, V \backslash S)$ ist.

Beispielanwendung: Bipartites Matching

- Flussproblem ist hier overkill
- Bipartiter Graph:
  Knotenmenge zerfällt in zwei disjunkte Mengen, sodass Kanten nur zwischen den Mengen verlaufen
  - z.B. Menge an Käufern und Verkäufern
- Maximales bipartites Matching:
  Finde maximale Anzahl von Kanten, sodass jeder Käufer genau einem Verkäufer zugeordnet wird
  - Wende Transformationen an, um eine Quelle und Senke zu erhalten
  - Setze Kapazität = 1 f.a. Kanten
  - Fluss = 1 bedeutet, dass Kante aktiv ist
  - Wert des Flusses gibt an, wie viele aktive Kanten aus s ausgehen bzw. wie viele in t ankommen