V. Graphalgorithmen

- Grundlegendes
  - Repräsentation von Graphen **22.1**
  - Breiten- und Tiefensuche **22.2, 22.3**
  - Anwendungen der Tiefensuche **22.4, 22.5**
- Minimale Spannbäume **23**
  - Algorithmus von Kruskal
- Kürzeste Wege **24,25**
  - Algorithmus von Dijkstra **24.3**
  - Bellman-Ford-Algorithmus **24.1**
  - Floyd-Warshall-Algorithmus **25.2**
- Flüsse in Netzwerken **26.1-26.3**

Repräsentation von Graphen

Graph $G = (V, E)$, wobei $E \subseteq V \times V$ (gerichteter Graph)
bzw. $E \subseteq \binom{V}{2}$ (ungerichteter Graph).

Adjazenzlisten

Für jeden Knoten $v \in V$ werden in einer verketteten Liste $Adj[v]$
alle Nachbarn u mit $(v, u) \in E$ (bzw. mit $\{v, u\} \in E$) gespeichert.

Platzbedarf: $O(|V| + |E| \log |V|)$

Adjazenzmatrix

Es wird eine $|V| \times |V|$-Matrix $A = (a_{ij})$ gespeichert, mit

$$a_{ij} = 1 \quad \text{genau dann, wenn} \quad (v_i, v_j) \in E \; .$$

Platzbedarf: $O(|V|^2)$

Breitensuche

Gegeben: Graph G (als Adjazenzlisten), ausgezeichneter Startknoten $s \in V$.

Gesucht für jeden Knoten $v \in V$: kürzester Pfad zu s und Distanz $d[v]$.

Speichere für jedes $v \in V$ den Vorgänger $\pi[v]$ auf kürzestem Pfad zu s.

Initialisiere $\pi[v] = \text{NIL}$ und $d[v] = \infty$ für alle $v \in V$.

Setze $d[s] = 0$ und speichere s in einer *FIFO-queue* Q.

while $Q \neq \emptyset$
  do $v \leftarrow get(Q)$
    for each $u \in Adj[v]$ with $d[u] = \infty$
      do $d[u] \leftarrow d[v] + 1$
        $\pi[u] \leftarrow v$
        $put(Q, u)$

Tiefensuche

Depth-First-Search (DFS):

Sucht jeden Knoten einmal auf, sondert eine Teilmenge der Kanten aus, die einen Wald (den DFS-Wald) bilden.

Hilfsmittel Färbung:

- Weiß $\hat{=}$ noch nicht besucht.
- Grau $\hat{=}$ schon besucht, aber noch nicht abgefertigt
- Schwarz $\hat{=}$ abgefertigt, d.h. der gesamte von hier erreichbare Teil wurde durchsucht.

Speichert außerdem für jeden Knoten v:

- Zeitpunkt des ersten Besuchs $d[v]$ (*discovery time*)
- Zeitpunkt der Abfertigung $f[v]$ (*finishing time*)

Tiefensuche: Pseudocode

DFS(G)

initialisiere $color[v] \leftarrow white$ und $\pi[v] \leftarrow$ NIL für alle $v \in V$
$time \leftarrow 0$
for each $v \in V$
  do if $color[v] = white$
    then DFS-Visit(G, v)

DFS-Visit(G, v)

$color[v] \leftarrow grey$
$d[v] \leftarrow ++time$
for each $u \in Adj[v]$ with $color[u] = white$
  do $\pi[u] \leftarrow v$
    DFS-Visit(G, u)
$color[v] \leftarrow black$
$f[v] \leftarrow ++time$

Klammerungseigenschaft

Seien $u, v \in V$ und $u \neq v$. Die folgenden drei Fälle sind möglich:

- $d[u] < d[v] < f[v] < f[u]$ und v ist Nachfahre von u im DFS-Wald.
- $d[v] < d[u] < f[u] < f[v]$ und u ist Nachfahre von v im DFS-Wald.
- $[d[u], f[u]] \cap [d[v], f[v]] = \emptyset$ und weder ist u Nachfahre von v im DFS-Wald noch umgekehrt.

Insbesondere ist die Konstellation $d[u] < d[v] < f[u] < f[v]$ unmöglich und der DFS-Wald lässt sich aus den Aktivitätsintervallen $[d[v], f[v]]$ eindeutig rekonstruieren.

Klassifikation der Kanten

Durch DFS werden die Kanten eines Graphen in die folgenden vier Typen klassifiziert.

- Baumkanten sind die Kanten des DFS-Waldes, also (u, v) mit $\pi[v] = u$.

  Kennzeichen: Beim ersten Durchlaufen ist v weiß.

- Rückwärtskanten (u, v), wo v Vorfahre von u im DFS-Wald ist.

  Kennzeichen: Beim ersten Durchlaufen ist v grau.

- Vorwärtskanten (u, v), wo v Nachkomme von u im DFS-Wald ist.

  Kennzeichen: Beim ersten Durchlaufen ist v schwarz, und $d[u] < d[v]$.

- Querkanten sind alle übrigen Kanten (u, v).

  Kennzeichen: Beim ersten Durchlaufen ist v schwarz, und $d[u] > d[v]$.

Bei ungerichteten Graphen kommen nur Baum- und Rückwärtskanten vor.

Topologische Sortierung

Eine topologische Ordnung eines gerichteten, azyklischen Graphen (dag) ist eine lineare Ordnung der Knoten

$$v_1 \prec v_2 \prec \ldots \prec v_n$$

so dass für jede Kante $(u, v) \in E$ gilt $u \prec v$.

Lemma: Ein gerichteter Graph ist genau dann azyklisch, wenn bei DFS keine Rückwärtskanten entstehen.

Satz: Eine topologische Ordnung auf einem dag G erhält man durch absteigende Sortierung nach den *finishing times* $f[v]$ nach Ausführung von DFS(G).

Zusammenhang

Weg (oder Pfad) von v_0 nach v_k:

$$p = \langle v_0, \ldots, v_k \rangle \text{ mit } (v_i, v_{i+1}) \in E \text{ für alle } i < k \ .$$

Schreibweise: $p : v_0 \rightsquigarrow v_k$.

Für ungerichtete Graphen:

Zwei Knoten u und v heißen zusammenhängend, wenn es einen Weg $p : u \rightsquigarrow v$ gibt.

Für gerichtete Graphen:

Zwei Knoten u und v heißen stark zusammenhängend, wenn es Wege $p : u \rightsquigarrow v$ und $q : v \rightsquigarrow u$ gibt.

Die Äquivalenzklassen bzgl. dieser Äquivalenzrelation heißen (starke) Zusammenhangskomponenten (SCC).

Starke Zusammenhangskomponenten

Definition: der zu $G = (V, E)$ transponierte Graph ist $G^T := (V, E^T)$,
wobei $(u, v) \in E^T$ gdw. $(v, u) \in E$.

Folgender Algorithmus zerlegt G in seine starken Zusammenhangskomponenten:

- Zuerst wird DFS(G) aufgerufen.
- Sortiere die Knoten nach absteigender *finishing time*.
- Berechne G^T.
- Rufe DFS(G^T) auf, wobei die Knoten im Hauptprogramm in der Reihenfolge der obigen Sortierung behandelt werden.
- Starke Zusammenhangskomponenten von G sind die Bäume des im zweiten DFS berechneten DFS-Waldes.

Minimale Spannbäume

Gegeben: Zusammenhängender, ungerichteter Graph $G = (V, E)$,
Gewichtsfunktion $w : E \to \mathbb{R}$.

Gesucht: Minimaler Spannbaum $T \subseteq E$ mit:

- T ist azyklisch.
- T spannt den Graphen auf:

  je zwei Knoten $u, v \in V$ sind durch einen Pfad in T verbunden.
- Gewicht $\sum_{e \in T} w(e)$ ist minimal.

Definition: Sei $A \subseteq E$ Teilmenge eines Minimalen Spannbaumes.
Kante e heißt sicher für A, falls $A \cup \{e\}$ Teilmenge eines Minimalen Spannbaumes ist.

Vorgehensweise: Beginne mit $A = \emptyset$, füge dann sukzessive Kanten hinzu, die sicher für A sind.

Finden sicherer Kanten

Definition: Für $S \subseteq V$ und $e \in E$ sagen wir "e kreuzt S", falls $e = \{u, v\}$ mit $u \in S$ und $v \in V \setminus S$.

Satz:

Sei A Teilmenge eines minimalen Spannbaumes,
sei $S \subseteq V$ mit der Eigenschaft: keine Kante in A kreuzt S,
und sei e eine Kante minimalen Gewichtes, die S kreuzt.

Dann ist e sicher für A.

Insbesondere: Sei Z eine Zusammenhangskomponente von A. Ist e eine Kante minimalen Gewichts, die Z mit einer anderen Zusammenhangskomponente verbindet, so ist e sicher für A.

Der Algorithmus von Kruskal

Benutzt eine UNION-FIND-Datenstruktur.

Erinnerung: Diese Datenstruktur verwaltet ein System disjunkter Mengen von "Objekten" und bietet folgende Operationen an:

- INIT Initialisieren
- $Make - Set(x)$ Fügt eine neue Einermenge mit Inhalt x hinzu. Ist x schon in einer vorhandenen Menge enthalten, so passiert nichts.
- $Find(x)$ Ist x in einer Menge enthalten, so liefere diese in Form eines kanonischen Elementes zurück. Anderenfalls liefere NIL o.ä. zurück. Insbesondere kann man durch den Test $Find(x) = Find(y)$ feststellen, ob zwei bereits eingefügte Elemente in derselben Menge liegen.
- $Union(x, y)$: Sind x und y in zwei verschiedenen Mengen enthalten, so vereinige diese zu einer einzigen. Anschließend gilt also insbesondere $Find(x) = Find(y)$.

Beachte: man kann Mengen und Elemente nicht wieder entfernen oder auseinanderreißen.

Der Algorithmus von Kruskal

Kruskal's Algorithmus

- Setze $A := \emptyset$.
- Rufe MAKE-SET(v) für jeden Knoten $v \in V$ auf.
- Sortiere die Kanten aufsteigend nach Gewicht.
- Für jede Kante $e = \{u, v\}$, in der sortierten Reihenfolge, prüfe ob FIND-SET(u) $\neq$ FIND-SET(v).
- Falls ja, füge e zu A hinzu, und rufe UNION(u, v) auf, sonst weiter.

Komplexität bei geschickter Implementierung der Union-Find Struktur: $O(|E| \log |E|)$.

Kürzeste Wege

Gegeben: gerichteter Graph $G = (V, E)$ mit Kantengewichten $w : E \to \mathbb{R}$.

Für einen Weg $p : v_0 \rightsquigarrow v_k$

$$p = \langle v_0, \ldots, v_k \rangle \text{ mit } (v_i, v_{i+1}) \in E \text{ für alle } i < k \ .$$

sei das Gewicht des Weges p definiert als:

$$w(p) = \sum_{i=1}^{k} w\big((v_{i-1}, v_i)\big)$$

Minimaldistanz von u nach v:

$$\delta(u, v) = \begin{cases} \min\{w(p) \ ; \ p : u \rightsquigarrow v\} & \text{falls } v \text{ von } u \text{ erreichbar ist,} \\ \infty & \text{sonst.} \end{cases}$$

Kürzester Weg von u nach v:

Pfad $p : u \rightsquigarrow v$ mit $w(p) = \delta(u, v)$.

Eigenschaften kürzester Wege

Problem: Gibt es einen negativen Zyklus $p : v \rightsquigarrow v$ mit $w(p) < 0$, so ist $\delta(u, u')$ nicht wohldefiniert, falls es einen Weg von u nach u' über v gibt.

Algorithmen für kürzeste Wege von einem Startpunkt s:

Dijkstra: nimmt an, dass $w(e) \geq 0$ für alle $e \in E$.

Bellman-Ford: Entdeckt die Präsenz negativer Zyklen, und liefert korrekte kürzeste Wege, falls es keinen gibt.

Optimale Teillösungen: Ist $p = \langle v_0, \ldots, v_k \rangle$ ein kürzester Weg von v_0 nach v_k, so ist für alle $0 \leq i < j \leq k$ der Pfad

$$p_{ij} = \langle v_i, \ldots, v_j \rangle$$

ein kürzester Weg von v_i nach v_j.

Daher reicht es zur Angabe eines kürzesten Weges von s zu v für alle $v \in V$, für jeden Knoten $v \neq s$ einen Vorgänger $\pi[v]$ anzugeben.

Relaxierung

Algorithmen halten für jedes $v \in V$ eine Abschätzung $d[v] \geq \delta(s,v)$ und einen vorläufigen Vorgänger $\pi[v]$.

INITIALISE(G,s) :

for $v \in V$ **do**

$d[v] \leftarrow \infty; \pi[v] \leftarrow$ NIL

$d[s] \leftarrow 0$

RELAX(u,v,w) : ▷testet, ob der bisher gefundene kürzeste Pfad zu v

▷ durch die Kante (u,v) verbessert werden kann

if $d[v] > d[u] + w\big((u,v)\big)$

then $d[v] \leftarrow d[u] + w\big((u,v)\big)$

$\pi[v] \leftarrow u$

Eigenschaften der Relaxierung

Lemma: Wird für einen Graphen G und $s \in V$ erst INITIALIZE(G, s), und dann eine beliebige Folge von RELAX(u, v, w) für Kanten (u, v) ausgeführt, so gelten die folgenden Invarianten:

1. $d[v] \geq \delta(s, v)$ für alle $v \in V$.
2. Ist irgendwann $d[v] = \delta(s, v)$, so ändert sich $d[v]$ nicht mehr.
3. Ist v nicht errreichbar von s, so ist $d[v] = \delta(s, v) = \infty$.
4. Gibt es einen kürzesten Pfad von s zu v, der in der Kante (u, v) endet, und ist $d[u] = \delta(s, u)$ vor dem Aufruf RELAX(u, v, w), so ist danach $d[v] = \delta(s, v)$.
5. Enthält G keinen negativen Zyklus, so ist der Teilgraph G_π aus den Kanten $(\pi[v], v)$ mit $\pi[v] \neq$ NIL ein Baum mit Wurzel s. Die d-Einträge geben die Distanz von s bei ausschließlicher Verwendung der Kanten aus G_π an.

Folgerung: Gilt nach einer Folge von RELAX(u, v, w), dass $d[v] = \delta(s, v)$ für alle $v \in V$ ist, so enthält der Baum G_π für jeden Knoten v einen kürzesten Pfad zu s.

Der Algorithmus von Dijkstra

Benutzt eine *priority queue* Q, die Knoten $v \in V$ mit Schlüssel $d[v]$ hält, und eine dynamische Menge S.

DIJKSTRA(G, w, s)

- Rufe INITIALIZE(G, s) auf, setze $S \leftarrow \emptyset$ und $Q \leftarrow V$.
- Solange $Q \neq \emptyset$ ist, setze $u \leftarrow$ EXTRACT-MIN(Q) und füge u zu S hinzu.
- Für jedes $v \in Adj[u]$ führe Relax(u, v, w) aus.
  (Bemerke: dies beinhaltet DECREASE-KEY-Operationen.)
  Anschliessend nächste Iteration.

Korrektheit:

Nach Ausführung von DIJKSTRA(G, w, s) ist $d[v] = \delta(s, v)$ für alle $v \in V$.

Invariante: $d[u]$ ist für alle Knoten in S korrekt eingetragen; für $v \in Q$ hält $d[v]$ den kürzesten Weg, der nur innere Knoten in S benutzt.

Komplexität: Hängt von der Realisierung der queue Q ab. (Vgl. Prim)

Als Liste: $O(|V|^2)$ Als Heap: $O(|E| \log |V|)$

Der Algorithmus von Dijkstra

Als Fibonacci-Heap (s. Cormen): $O(|V| \log |V| + |E|)$.

Der Algorithmus von Bellman-Ford

BELLMAN-FORD(G, w, s)

- Rufe INITIALIZE(G, s) auf.
- Wiederhole $|V| - 1$ mal:

  Für jede Kante $(u, v) \in E$ rufe RELAX(u, v, w) auf.
- Für jede Kante $(u, v) \in E$, teste ob $d[v] > d[u] + w(u, v)$ ist.
- Falls ja für eine Kante, drucke "*negativer Zyklus vorhanden*",
  sonst brich mit Erfolg ab.

Korrektheit: Nach Ausführung von BELLMAN-FORD(G, w, s) gilt:

Ist kein negativer Zyklus von s erreichbar, dann ist
$d[v] = \delta(s, v)$ für alle $v \in V$, und der Algorithmus terminiert erfolgreich.

Andernfalls ist wird der negative Zyklus durch den Test entdeckt.

Beweisidee: Egal, ob negative Zyklen da sind oder nicht, enthält $d[v]$ nach der k-ten Iteration die Länge des kürzesten Pfades, der aus höchstens k Knoten besteht.

Komplexität ist offenbar $O(|V| \cdot |E|)$.

Kürzeste Wege zwischen allen Paaren

Aufgabe: Berechne $\delta(i,j)$ für alle Paare $i,j \in V = \{1,\ldots,n\}$.
Kantengewichte in Matrix $W = (w_{i,j})$, mit $w_{i,i} = 0$.

Dynamische Programmierung: Berechne rekursiv die Werte

$d_{i,j}^{(m)}$ = minimales Gewicht eines Weges von i zu j, der $\leq m$ Kanten lang ist.

$$d_{i,j}^{(0)} = \begin{cases} 0 & \text{falls } i = j \\ \infty & \text{sonst} \end{cases}$$

$$d_{i,j}^{(m)} = \min\Big(d_{i,j}^{(m-1)}, \min_{k\neq j}\big(d_{i,k}^{(m-1)} + w_{k,j}\big)\Big)$$

$$= \min_{1\leq k\leq n}\big(d_{i,k}^{(m-1)} + w_{k,j}\big)$$

Keine negativen Zyklen $\quad\rightsquigarrow\quad \delta(i,j) = d_{i,j}^{(n-1)} = d_{i,j}^{(m)}$ für alle $m \geq n-1$.

Kürzeste Wege und Matrizenmultiplikation

Betrachte Matrizen $D^{(m)} = \left(d_{i,j}^{(m)}\right)$. Es gilt

$$D^{(m)} = D^{(m-1)} \odot W$$

wobei $\odot$ eine Art Multiplikation ist mit $\quad \min \mathrel{\hat{=}} \sum \quad$ und $\quad + \mathrel{\hat{=}} \times$.

Matrix $D^{(n-1)} = \left(\delta(i,j)\right)$ kann ausgerechnet werden in Zeit $\Theta(n^4)$.

Bessere Methode durch iteriertes Quadrieren:

Da für $m \geq n-1$ gilt $D^{(m)} = D^{(n-1)}$, und $\odot$ assoziativ ist,
berechne $D^m = D^{(n-1)}$ für $m = 2^{\lceil \log(n-1) \rceil}$ mittels

$$D^{(1)} = W$$
$$D^{(2k)} = D^{(k)} \odot D^{(k)}$$

Zeitkomplexität: nur $\Theta(n^3 \log n)$.

Der Algorithmus von Floyd-Warshall

Betrachte Weg von i nach j:

$$\langle i = v_0, v_1, \ldots, v_{\ell-1}, v_\ell = j \rangle$$

Knoten $v_1, \ldots, v_{\ell-1}$ sind die Zwischenknoten.

Dynamische Programmierung: Berechne rekursiv die Werte

$d_{i,j}^{(k)}$ = minimales Gewicht eines Weges von i zu j, der nur Zwischenknoten $\{1, \ldots, k\}$ verwendet.

$$d_{i,j}^{(0)} = w_{i,j}$$
$$d_{i,j}^{(k)} = \min\big(d_{i,j}^{(k-1)} , d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)} \big)$$

Klar: $\delta(i,j) = d_{i,j}^{(n)}$.

Matrix $D^{(n)} = \big(d_{i,j}^{(n)}\big) = \big(\delta(i,j)\big)$ kann in Zeit $\Theta(n^3)$ berechnet werden.

Flüsse in Netzwerken

Gegeben: gerichteter Graph $G = (V, E)$ mit Quelle $s \in V$ und Senke $t \in V$,
für $(u, v) \in E$ Kapazität $c(u, v) \geq 0$. Für $(u, v) \notin E$ sei $c(u, v) = 0$.

Gesucht: Ein Fluss durch G: Funktion $f : V \times V \to \mathbb{R}$ mit

1. $f(u, v) \leq c(u, v)$
2. $f(u, v) = -f(v, u)$
3. Für alle $u \in V \setminus \{s, t\}$:

$$\sum_{v \in V} f(u, v) = 0$$

Wert des Flusses f

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

soll maximiert weden.

Eigenschaften von Flüssen

Für $X, Y \subseteq V$ sei $f(X,Y) := \sum_{x \in X} \sum_{y \in Y} f(x,y)$.

Abkürzung: $f(v,X) = f(\{v\},X)$.

Eigenschaft 3 lautet damit: $f(u,V) = 0$.

Lemma: Für alle $X, Y, Z \subseteq V$ mit $Y \cap Z = \emptyset$ gilt:

- $f(X,X) = 0$
- $f(X,Y) = -f(Y,X)$
- $f(X, Y \cup Z) = f(X,Y) + f(X,Z)$
- $f(Y \cup Z, X) = f(Y,X) + f(Z,X)$

Restnetzwerke und Erweiterungspfade

Sei f ein Fluss in einem Netzwerk $G = (V, E)$ mit Kapazität c.

Für $u, v \in V$ ist die Restkapazität $c_f(u, v) = c(u, v) - f(u, v)$.

Das Restnetzwerk $G_f = (V, E_f)$ ist gegeben durch

$$E_f := \{(u, v)\ ;\ c_f(u, v) > 0\}\ .$$

Lemma: Ist f' ein Fluss in G_f, so ist $f + f'$ ein Fluss in G mit Wert $|f| + |f'|$.

Ein Weg $p : s \rightsquigarrow t$ in G_f ist ein Erweiterungspfad, seine Restkapazität ist

$$c_f(p) = \min\{c_f(u, v)\ ;\ (u, v) \text{ Kante in } p\}$$

Für einen Erweiterungspfad p definiere

$$f_p(u, v) = \begin{cases} c_f(p) & (u, v) \text{ in } p \\ -c_f(p) & (v, u) \text{ in } p \\ 0 & \text{sonst} \end{cases}$$

Dann ist f_p ein Fluss in G_f.

Das Max-Flow-Min-Cut Theorem

Ein Schnitt in G ist eine Zerlegung (S, T) mit $s \in S \subseteq V$ und $t \in T = V \setminus S$.

Lemma: Ist (S, T) ein Schnitt, so ist $f(S, T) = |f|$.

Satz: Die folgenden Aussagen sind äquivalent:

1. f ist ein maximaler Fluss in G.
2. Im Restnetzwerk G_f gibt es keinen Erweiterungspfad.
3. Es gibt einen Schnitt (S, T) mit $|f| = c(S, T)$.

Der schwierigste Teil des Beweises ist 2→3. Er verwendet folgende Idee: Im Falle von 2 definiert man einen Schnitt S, T durch $S = \{v \mid v \text{ ist von } s \text{ in } G_f \text{ erreichbar.}\}$.

Die Ford-Fulkerson-Methode

FORD-FULKERSON(G, s, t, c)

- Initialisiere $f(u, v) = 0$ für alle $u, v \in V$.
- Solange es einen Erweiterungspfad p in G_f gibt:

  Setze für jede Kante (u, v) in p

$$f(u, v) \leftarrow f(u, v) + c_f(p) \quad ; \quad f(v, u) \leftarrow -f(u, v)$$

Korrektheit folgt aus dem Max-Flow-Min-Cut-Theorem.

Komplexität hängt davon ab, wie man nach Erweiterungspfaden sucht.

Ist $c(x, y) \in \mathbb{N}$ für alle $(x, y) \in E$, so ist die Laufzeit $O(|E| \cdot |f^*|)$, für einen maximalen Fluss f^*.

Der Algorithmus von Edmonds-Karp

Algorithmus von Edmonds-Karp:

Suche bei Ford-Fulkerson Erweiterungspfade mittels Breitensuche in G_f.

Für $v \in V$, sei $\delta_f(s, v)$ die Distanz von s zu v in G_f.

Lemma: Beim Ablauf des Algorithmus von Edmonds-Karp steigt $\delta_f(s, v)$ für jeden Knoten $v \in V \setminus \{s, t\}$ monoton an.

Satz: Die Zahl der Iterationen der äußeren Schleife beim Algorithmus von Edmonds-Karp ist $O(|V| \cdot |E|)$.

Damit: Laufzeit ist $O(|V| \cdot |E|^2)$.

Anwendung: Maximale Matchings

Sei $G = (V, E)$ ein ungerichteter Graph. Ein Matching in G ist $M \subseteq E$ mit

$$e_1 \cap e_2 = \emptyset \qquad \text{für alle } e_1, e_2 \in M \ .$$

Aufgabe: Gegeben ein bipartiter Graph $G = (V, E)$ mit $V = L \cup R$ und $E \subseteq L \times R$, finde ein Matching maximaler Größe.

Idee: Betrachte $G' = (V', E')$, wobei $V' = V \cup \{s, t\}$, und

$$E' = E \cup \{(s, \ell)\,;\, \ell \in L\} \cup \{(r, t)\,;\, r \in R\}$$

mit Kapazität $c(e) = 1$ für alle $e \in E'$.

Beobachtung: Jedes Matching M in G entspricht einem ganzzahligen Fluss in G' mit $|f| = |M|$, und umgekehrt.

Satz: Ist die Kapazitätsfunktion c ganzzahlig, so ist auch der mit der Ford-Fulkerson-Methode gefundene maximale Fluss ganzzahlig.