# Graphenalgorithmen

Vorlesung gehalten im WS 1996/97

Oliver Vornberger

Fachbereich Mathematik/Informatik
Universität Osnabrück

**Literatur**

A.V. Aho, J.E. Hopcroft, J.D. Ullman: *Data Structures and Algorithms*
J. Clark, D. Holton: *Graphentheorie*
N. Christofides: *Graph Theory: An Algorithmic Approach*
S. Even: *Graph Algorithms*
S. Horowitz, S. Sahni: *Fundamentals of Computer Algorithms*
M.R. Garey, D.S. Johnson: *Computers and Intractability*
E.L. Lawler: *Combinatorial Optimization: Networks and Matroids*
E. Minieka: *Optimization Algorithms for Networks and Graphs*
R. Sedgewick: *Algorithms*
R. Tarjan: *Data Structures and Network Algorithms*

**Danksagung**

Ich danke ...

... Herrn Andreas Landt und Herrn Andreas Siegner für die Erstellung der Skriptvorlage nach ihrer Vorlesungsmitschrift,

... Frau Marianne Gausmann für den Textsatz mit LaTeX $2_\varepsilon$ und das Erstellen der Grafiken,

... Herrn Frank M. Thiesing für seine engagierte Mitarbeit bei der inhaltlichen und äußerlichen Gestaltung des Textes.

... Herrn Viktor Herzog für die Konvertierung des Skripts nach HTML.

**HTML-Version**
Der Inhalt dieser Vorlesung kann online betrachtet werden unter
`http://www-lehre.informatik.uni-osnabrueck.de/~graph`

Osnabrück, im September 1998

(Oliver Vornberger)

# Inhaltsverzeichnis

# Kapitel 1

# Einführung

Graphenalgorithmen dienen in der Praxis zum Lösen von kombinatorischen Problemen. Dabei geht man wie folgt vor:

1. Modelliere das Problem als Graph.
2. Formuliere die Zielfunktion als Eigenschaft dieses Graphen.
3. Löse jetzt mit Hilfe eines Graphenalgorithmus.

## 1.1 Modellierungsmöglichkeiten von Graphen

Der Graph wird als 2-Tupel, bestehend aus der Knotenmenge $V$ und der Kantenmenge $E$, definiert:

$$G = (V, E)$$

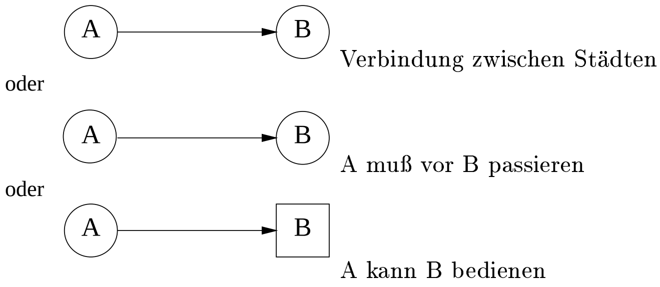
Die Knoten symbolisieren hierbei die jeweiligen Objekte, und die Kanten modellieren binäre Beziehungen zwischen diesen Objekten.

Man unterscheidet nun zwischen gerichteten und ungerichteten Graphen. Im gerichteten Fall ist die Kantenmenge $E$ eine Teilmenge des kartesischen Produkts der Knotenmenge mit sich selbst: $E \subseteq V \times V$. Eine Kante ist also ein gerichtetes Paar, man schreibt $(x, y) \in E$.

Im ungerichteten Fall ist $E$ eine Teilmenge von $\mathbf{P}_2(V)$ (Menge aller 2-elementigen Teilmengen von $V$): $E \subseteq \mathbf{P}_2(V)$. Eine Kante ist dann also eine 2-elementige Teilmenge von $V : \{x, y\} \in E$. Häufig schreibt man aber — wie im gerichteten Fall — $(x, y) \in E$ (und somit auch $(y, x) \in E$).

In gegebenen Fällen werden die Kanten des Graphen noch zusätzlich mit einer Gewichtung $c : E \to \mathbb{R}$ versehen, die z.B. Kosten einer Verbindung modellieren.

**Beispiele:**




**Aufgabe:**

Gegeben: 3 Wassereimer à 7 Liter (voll), 3 Liter und 1 Liter (beide leer).

Gesucht: Folge von Umfüllaktionen, die letztendlich in einem Eimer 5 Liter abmessen.

**Modell:**

Knoten entsprechen Zuständen, z.B. Start mit $(7, 0, 0)$.

Kanten werden für Umfüllaktionen eingeführt, z.B. $(7, 0, 0) \rightarrow (4, 3, 0)$.

Evtl. können die Kanten noch mit dem Eimergewicht des umgefüllten Eimers als Gewichtung versehen werden.

Eine Lösung des Problems ist jetzt gleichzusetzen mit einem Weg in dem beschriebenen Graphen von $(7, 0, 0)$ nach $(5, \cdot, \cdot)$.

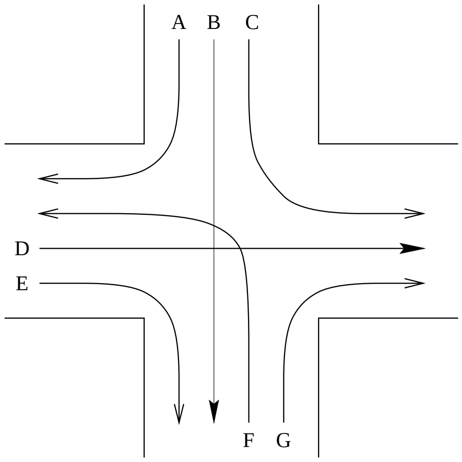
Für diesen Weg sind nun — neben der Existenz eines solchen Weges — evtl. verschiedene Optimalitätsbedingungen wichtig:

- Man sucht einen Weg mit möglichst wenigen Umfüllaktionen, also einen Pfad in $G$ mit möglichst wenig Kanten.

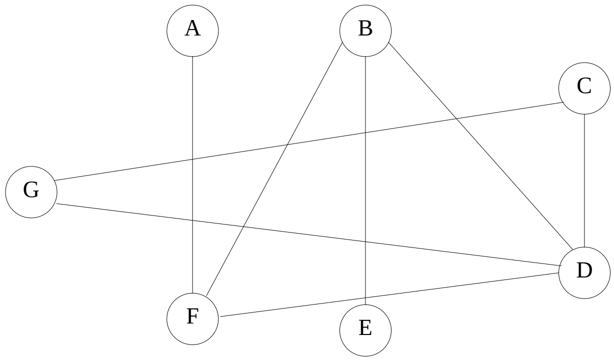
- Man sucht einen "kraftsparenden" Weg in $G$, d.h. einen Pfad mit möglichst geringer Summe der o.a. Kantenkosten.

**Aufgabe:**

Straßenkreuzung mit verschiedenen Abbiegemöglichkeiten




Dieses Problem modellieren wir nun durch einen ungerichteten Graphen, dessen Knoten die verschiedenen Abbiegemöglichkeiten ($A-G$) und dessen Kanten evtl. Konflikte zwischen zwei Möglichkeiten simulieren. Wir erhalten also folgenden (ungerichteten) Graphen:

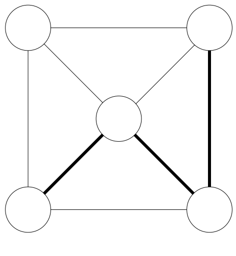



Als Problem gilt es jetzt, die Anzahl der Ampelphasen zu minimieren. Dies ist gleichbedeutend mit dem Problem, im obigen Graphen eine Färbung der Knoten mit minimaler Farbzahl zu finden. Bei der Einfärbung der Knoten muß man dabei beachten, daß benachbarte Knoten unterschiedliche Farben haben müssen.

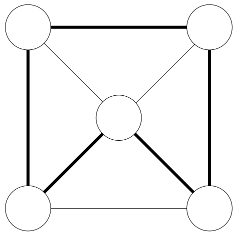
## 1.2 Grundtypen bei der Zielfindung

Bei der Zielfindung in einem Graphen kann man folgende Grundtypen unterscheiden:

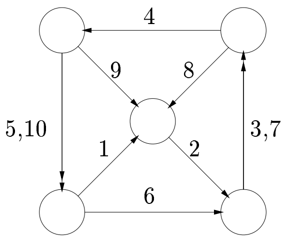
- **Kanten auswählen:**
  - Shortest Path:



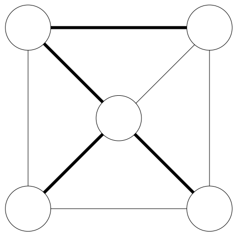
- Travelling Salesman:
  (Jeden Knoten besuchen)



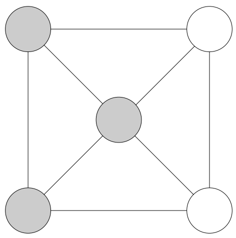
- Chinese Postman:
  (Jeden Knoten besuchen)



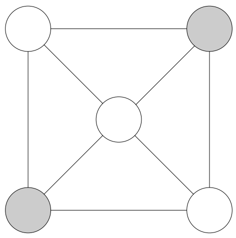

- Spanning Tree:



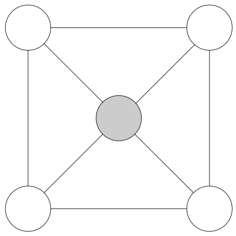
- **Knoten auswählen:**
  - größte Clique finden:



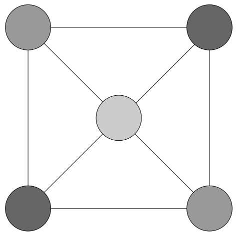
  - Independent Set:



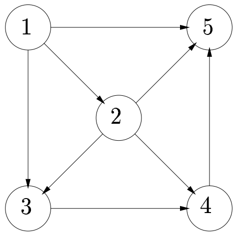
– Zentrum finden:



- **Kanten Werte zuordnen**
  - Maximum Flow: In einem Graphen versucht man unter Einhaltung von Kantenkapazitäten möglichst viele Einheiten (Güter etc.) von einer Quelle zu einer Senke fließen zu lassen.
  - Minimum Cost Flow: Hier wird versucht, bestimmte Nachfragen in Knoten durch Angebote in anderen Knoten durch einen Fluß zu befriedigen und die dabei entstehenden Kosten minimal zu halten.
- **Knoten Werte zuwiesen:**
  - Coloring (Färbung):



  - Scheduling:




## 1.3 Lösungsstrategien

Bei den Algorithmen zur Lösung der Graphenprobleme unterscheidet man verschiedene Strategien:

- **Greedy:** sukzessive Bestimmung der Lösungsvariablen
- **Divide & Conquer:** aufteilen, lösen, Lösungen vereinigen
- **Dynamic Programming:** Berechne Folge von Teillösungen.

- **Enumeration:** Erzeuge alle Permutationen und überprüfe sie.
- **Backtracking:** Teillösungen werden systematisch erweitert.
- **Branch & Bound:** Erweitere Teillösung an der vielversprechendsten Stelle.

# Kapitel 2

# Implementation von Graphen

1. **Adjazenzmatrix:**

$$m[i,j] = \begin{cases} 1, & \text{falls } (i,j) \in E \\ 0, & \text{sonst} \end{cases}$$

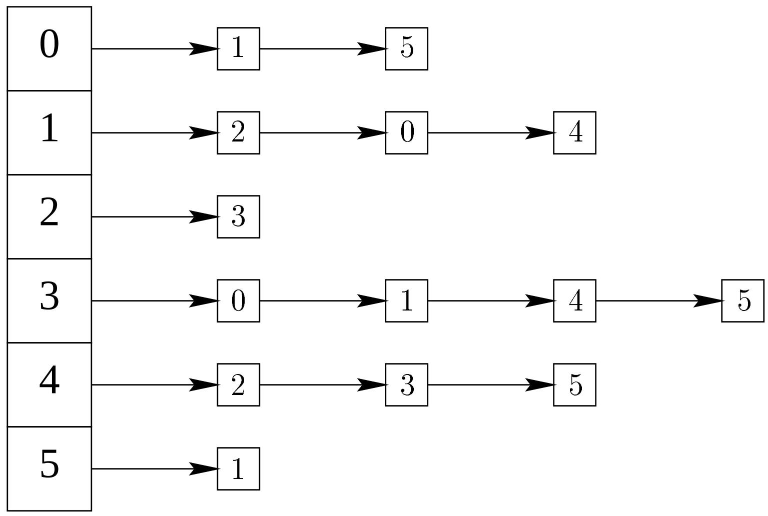
Es ist auch möglich, eventuelle Kantenkosten dabei zu berücksichtigen:

$$m[i,j] = \begin{cases} c(i,j), & \text{für } (i,j) \in E \\ 0, & \text{für } i = j \\ \infty & \text{sonst} \end{cases}$$

Diese Implementation ist hauptsächlich geeignet für:

- Random Access auf Kanten (in konstanter Zeit möglich !)
- dichtbesetzte Graphen

2. **Adjazenzlisten:**



Geeignet für:

- Algorithmen, die die Nachbarn eines Knotens abarbeiten.
- dünnbesetzte Graphen

Auf diesen Listen sind folgende Operationen möglich:

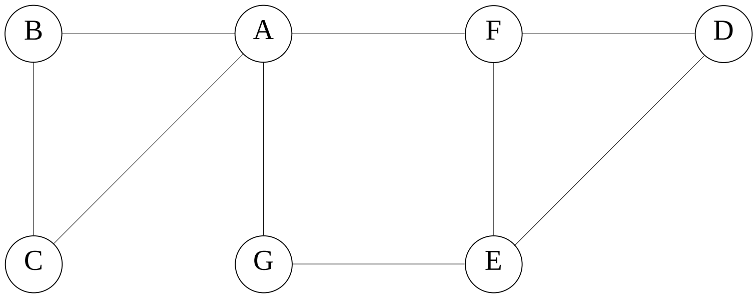
- **elem** (gibt momentanes Element aus
- **advance** (rücke einen vor)
- **reset** (rücke auf Listenanfang zurück)
- **insert** (füge Element ein)
- **delete** (lösche momentanes Element)
- **createlist** (lege neue Liste an)
- **emptylist** (gibt an, ob Liste leer ist)
- **endpos** (gibt an, ob Element am Listenende steht)

Der nachfolgende Algorithmus führt auf Basis dieser Adjazenzliste eine Tiefensuche durch:

```
VAR g: Graph, id: INTEGER, val: ARRAY [0..N-1] OF INTEGER,
   besucht: ARRAY [0..N-1] OF BOOLEAN;
(* val[i] = j ⇔ Knoten i bekommt Nr. j *)
FOR i := 0 TO N-1 DO besucht[i]:= FALSE END;
FOR i := 0 TO N-1 DO
   IF NOT besucht[i] THEN visit(i) END;
END;

PROCEDURE visit(k: INTEGER)
BEGIN
   id := id + 1;
   val[k] := id;
   besucht[k] := TRUE;
   reset (g[k]);
   WHILE NOT endpos (g[k]) DO
      y := elem (g[k]);
      IF NOT besucht[y] THEN visit(y) END;
      advance (g[k]);
   END
END
```

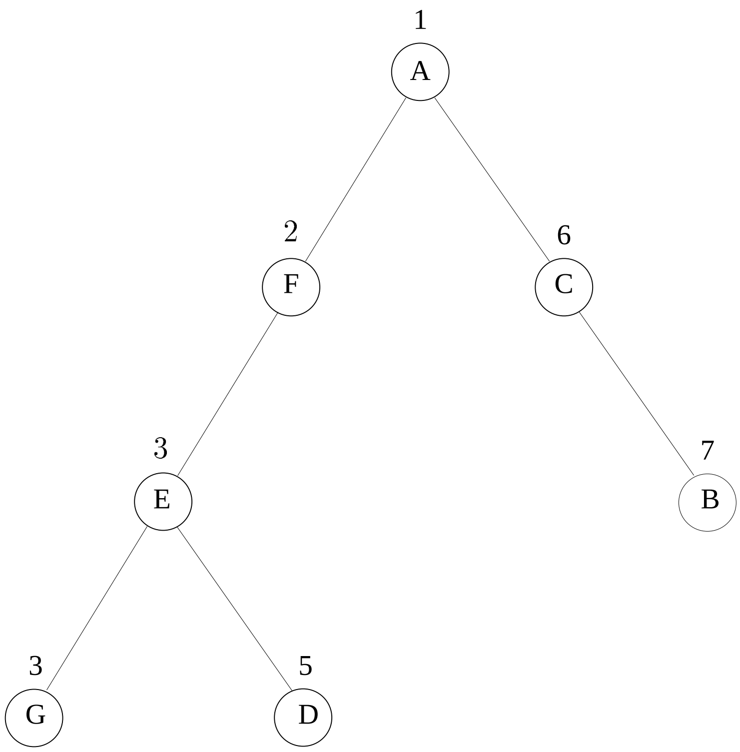
**Beispiel:**




Die zugehörige Adjazenzliste sieht dann wie folgt aus:

$$\begin{array}{ll} A: & F, C, B, G \\ B: & A, C \\ C: & A, B \\ D: & F, E \\ E: & G, F, D \\ F: & A, E, D \\ G: & E, A \end{array}$$

Der obige Algorithmus liefert dann den folgenden Tiefensuchebaum mit der an den Knoten notierten Besuchreihenfolge:




Zur Laufzeit des Algorithmus ist folgendes zu bemerken:

Jede Kante im Graphen $G = (V, E)$ wird genau zweimal besucht. Der Algorithmus berechnet also die Tiefensuchreihenfolge in $O(|V| + |E|)$.

Liegt hingegen eine Adjazenz**matrix** vor, so ersetze im Algorithmus reset $(g[k])$ und die WHILE-Schleife durch:

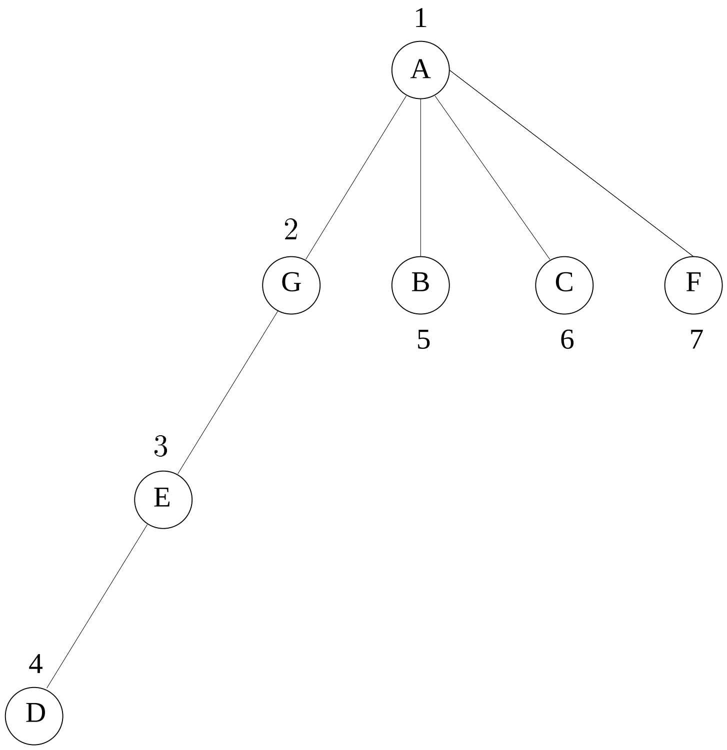
```
FOR y:=0 TO N-1 DO
   IF m[k,y] AND NOT besucht [y] THEN visit(y) END
END
```

Man sieht leicht, daß der Aufwand des Algorithmus nun in $O(|V|^2)$ liegt, also im Mittel schlechter geworden ist.

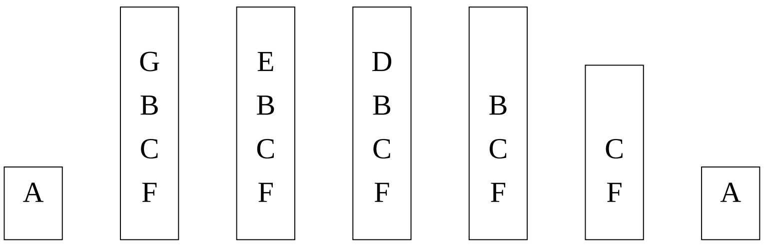
Die Prozedur `visit` ist mit Hilfe eines Kellers auch iterativ zu formulieren, sie sieht dann wie folgt aus:

```
PROCEDURE visit(k: INTEGER);
BEGIN
   push(s,k);
   gesichtet[k] := TRUE;
   WHILE NOT emptystack(s) DO
      k := top(s);
      pop(s);
      id := id + 1;
      val[k] := id;
      reset(g[k]);
      WHILE NOT endpos(g[k]) DO
         y := elem(g[k]);
         IF NOT gesichtet[y] THEN push(s,y);
            gesichtet[y] := TRUE;
         END;
         advance(g[k]);
      END;
   END;
END;
```

Im vorangegangenen Graphen ergibt sich dann folgender Tiefensuchebaum:




Der Keller speichert dabei im Laufe der Abarbeitung folgende Werte:

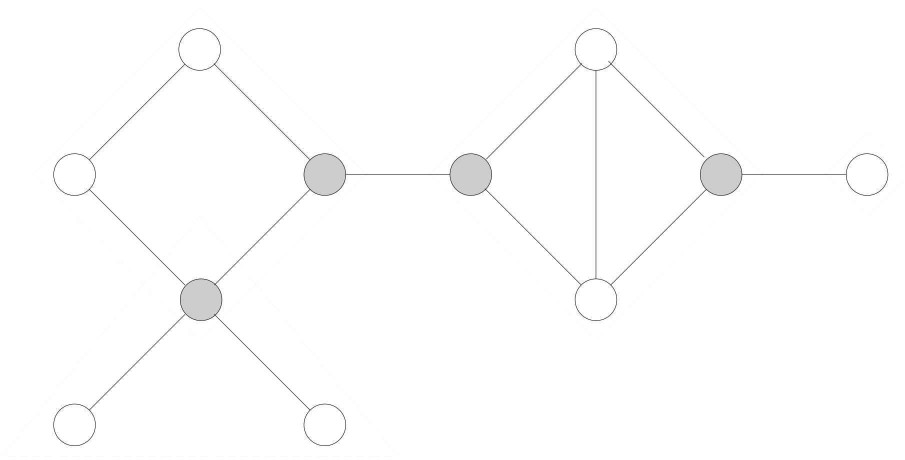


# Kapitel 3

# Zusammenhangskomponenten

## 3.1 Artikulationspunkt

Ein Knoten in einem zusammenhängenden Netzwerk heißt **Artikulationspunkt**, wenn durch sein Entfernen der Graph zerfällt, d.h. sich die Zahl der Zusammenhangskomponenten erhöht. Im nachfolgenden Beispiel sind die Artikulationspunkte dunkel eingefärbt:



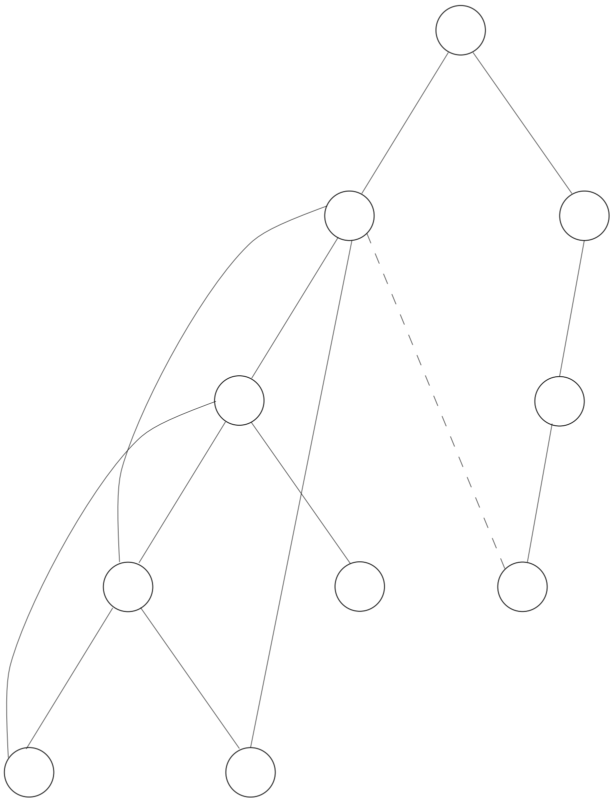
## 3.2 2-fach zusammenhängend

Ein Graph heißt **2-fach zusammenhängend**, wenn er nur aus einem Knoten besteht oder (für $|V| > 2$) wenn er keine Artikulationspunkte hat.

In einem 2-fach zusammenhängenden Graphen ist jedes Knotenpaar durch mindestens zwei kanten- und knotendisjunkte Wege verbunden.

Ein **nicht** 2-fach zusammenhängender Graph zerfällt in 2-fach Zusammenhangskomponenten, das sind die maximalen 2-fach zusammenhängenden Teilgraphen (siehe Umrahmungen oben).

**Ergebnis einer Tiefensuche:**



Die dünnen Kanten stellen "Rückwärtskanten" dar, d.h. Kanten, über die zurückgesprungen wird, wenn der Algorithmus in eine Sackgasse gelaufen ist. Solche Kanten wie die gestrichelte Kante sind im Ausgangsgraphen nicht möglich!

Für jede Kante $(x, y)$ gilt: $x$ ist Vorfahre von $y$ **oder** $y$ ist Vorfahre von $x$.

### Behauptung:

Die Wurzel ist Artikulationspunkt $\iff$ Sie hat 2 oder mehrere Söhne.

### Beweis:

"$\Rightarrow$": Die Wurzel ist Artikulationspunkt.
$\Rightarrow$ Die Wurzel hat Knotengrad $\geq 2$.
$\Rightarrow$ Die Wurzel hat 2 oder mehrere Söhne.

"$\Leftarrow$": Die Wurzel hat 2 oder mehrere Söhne.
$\Rightarrow$ Jeder Sohn der Wurzel ist nach deren Entfernen Wurzel eines Teilbaums.
$\Rightarrow$ Der Graph zerfällt in 2 oder mehrere Zusammenhangskomponenten.
$\Rightarrow$ Die Wurzel ist ein Artikulationspunkt.

### Behauptung:

Ein Knoten $k \neq$ Wurzel ist Artikulationspunkt

$\Longleftrightarrow$ Es gibt einen Sohn $x$ von $k$, der nicht von sich oder seinen Nachfahren einen Vorfahren von $k$ erreichen kann.

### Beweis:

"$\Rightarrow$": $k$ sei ein Artikulationspunkt.
*Annahme:* Alle Söhne von $k$ erreichen einen Vorfahren von $k$.
$\Rightarrow$ Der Graph zerfällt nicht! (Widerspruch zur Voraussetzung)
$\Rightarrow$ Behauptung

"$\Leftarrow$": Von $x$ gehe keine Kante zu einem Vorfahren von $k$.
$\Rightarrow$ Dieser Vorfahre von $k$ wird durch das Herausnehmen von $k$ von $x$ und dessen Teilbaum vollständig getrennt.
$\Rightarrow$ Der Graph zerfällt in mehrere Zusammenhangskomponenten.
$\Rightarrow$ $k$ ist Artikulationspunkt.

Aus diesen Eigenschaften läßt sich demzufolge ein Verfahren zur Ermittlung der Artikulationseigenschaft eines jeden Knotens ableiten:

> Berechne während der Tiefensuche für jeden Knoten $k$ den höchsten Knoten, den seine Söhne mit Rückkanten erreichen können. Dadurch kann man entscheiden, ob $k$ ein Artikulationspunkt ist oder nicht.

Es folgt der Algorithmus, der diesen höchsten erreichbaren Knoten berechnet:

```
PROCEDURE visit (k: INTEGER): INTEGER;
BEGIN
id := id + 1; val[k] := id; besucht[k] := TRUE; min := val[k];
reset(g[k]);
WHILE NOT endpos(g[k]) DO
   x := elem(g[k]);
   IF NOT besucht[x] THEN m:= visit(x);
      IF m < min THEN min := m;
      IF m ≥ val[k] THEN Write(4k ist Artikulationspunkt.4); END
   ELSE (* Rückkante *)
      IF val[x] < min THEN min := val[x]; END;
   END;
   advance(g[k]);
END (* WHILE *)
RETURN min;
END (* visit *)
```

Dies ist also ein Beispiel dafür, wie man eine Tiefensuche **sinnvoll** nutzen kann!

# Kapitel 4

# Union/Find

**Aufgabe:**

Verwaltung von disjunkten Mengen $S_1, ..., S_k \subset \{1, ..., n\}$

1.) Zu $x \in \{1, ..., n\}$ finde ein $y \in \{1, ..., k\}$ mit $x \in S_y$. Bei gleichbleibenden Mengen $S_1, ..., S_k$ ist das kein Problem!

**Aber:**

2.) Zu $x, y \in \{1, ..., n\}$ finde $i, j \in \{1, ..., k\}$ mit $x \in S_i$ und $y \in S_j$, dann bilde $S_i \cup S_j$.

Idee: Für jedes Element aus $\{1, ..., n\}$ führe einen Knoten ein. Diese Knoten faßt man in Bäumen zusammen, die durch ihre Wurzel repräsentiert werden.

Die dazu benötigte Datenstruktur: VAR vater: ARRAY[0..n-1] OF INTEGER,

Dabei gilt: $vater[x] = y \iff y$ ist Vater von $x$
$vater[x] = x \iff x$ ist Wurzel $\iff x$ ist Repräsentant.

Daraus ergeben sich folgende Prozeduren:

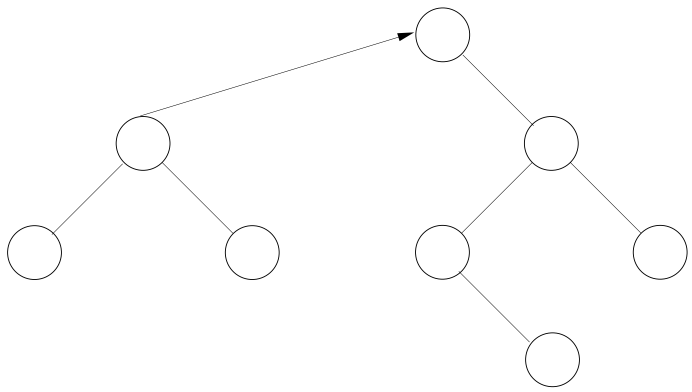
```
PROCEDURE find(x: INTEGER): INTEGER
(* liefert Repräsentanten von x *)
BEGIN
   WHILE vater[x] ≠ x DO
      x := vater[x];
   END;
RETURN x;
END;

PROCEDURE union(x, y: INTEGER);
BEGIN
   vater[find(x)] := find(y);
END;
```

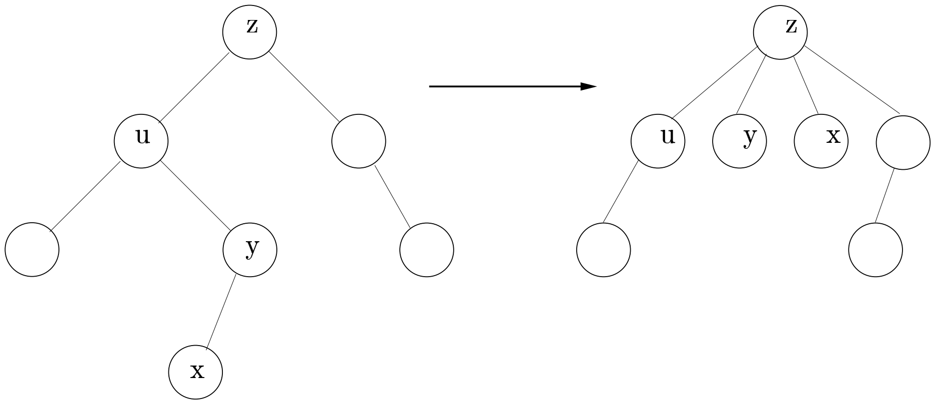
Die optimale Situation für die `find`-Prozedur liegt dann vor, wenn der Baum möglichst flach ist, also alle Knoten direkte Söhne der Wurzel sind. Die Prozedur `union` zerstört diese Situation aber, indem sie die Bäume vereinigt und sie untereinander setzt.

Um diesen Schaden allerdings möglichst klein zu halten, gibt es zwei verschiedene Ansätze:

1. Hänge den flacheren Baum unter die Wurzel des höheren Baums, merke also die Tiefe eines jeden Baumes in der Wurzel.



2. Ordne allen Knoten auf dem Weg zur Wurzel die Wurzel als neuen Vater zu:



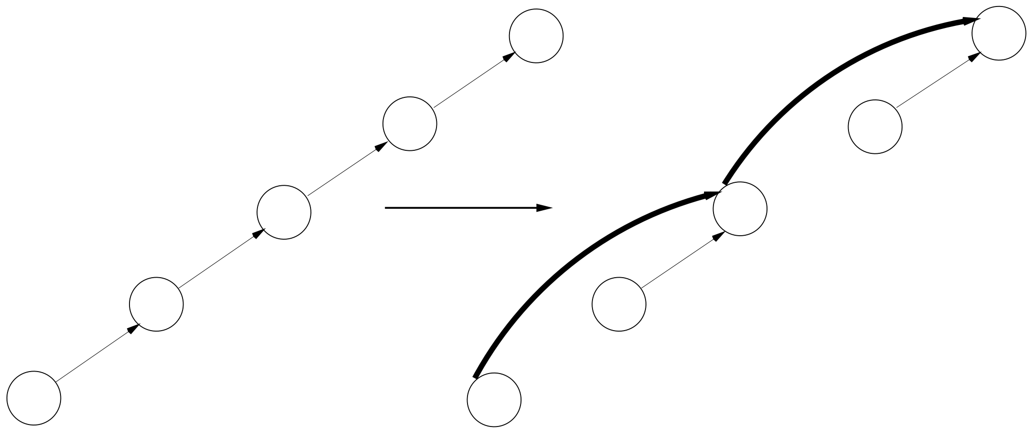

### 4.0.1 Satz von Tarjan

Der Aufwand um m-mal Union/Find durchzuführen beträgt $m \cdot \alpha(m,n)$, wobei $\alpha$ die inverse Ackermannfunktion darstellt, eine stark steigende Funktion.

Das bedeutet, daß Union/Find in fast konstanter Zeit abläuft.

### 4.0.2 Path Halving

Dieses Verfahren ermöglicht es, beim Hinaufgehen in einem Baum die Weglänge zu halbieren, indem man jeden zweiten Knoten auf seinen Großvater zeigen läßt:



Dieses wird durch die folgende Prozedur realisiert:

```
WHILE vater[x] ≠ x DO
   vater[x] := vater[vater[x]];
   x := vater[x];
END;
```

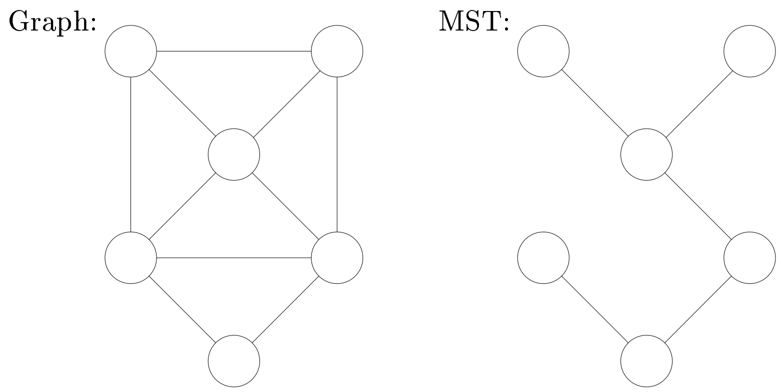
# Kapitel 5

# Minimum Spanning Tree

Eine Anwendung für den Union/Find-Algorithmus ist der **Minimum Spanning Tree** (MST).

Ein **Spannbaum** ist eine minimale Menge von Kanten eines Graphen, die alle Knoten des Graphen verbindet.

**Beispiel:**

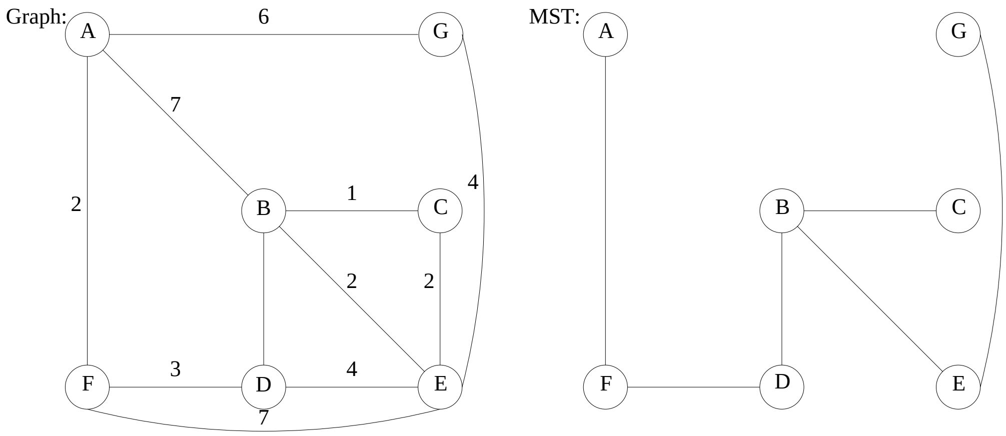



## 5.1 Algorithmus von Kruskal

Ein Algorithmus zur Berechnung des MST ist der **Algorithmus von Kruskal:**

```
Initialisiere einen Wald mit n isolierten Knoten.
Initialisiere einen Heap mit allen Kanten.
REPEAT
   entferne die billigste Kante (x,y) aus dem Heap
   union(x,y) (*füge (x,y) in den Wald ein*)
UNTIL der Wald besteht nur aus einem Baum
```

### Beispiel:




Sei $|E|$ nun die Anzahl der Kanten im Graphen.

Dann gilt: Der Aufwand für den Kruskal-Algorithmus liegt in $O(|E| \cdot \log |E|)$.

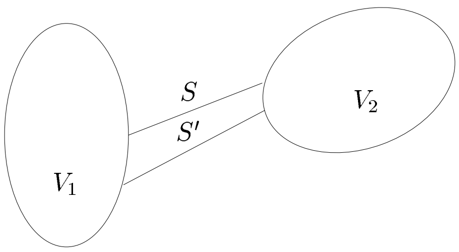
Das $\log |E|$ kommt dabei vom Entfernen der jeweils billigsten Kante und der erforderlichen Neuordnung des Heaps.

Die Korrektheit des Kruskal-Algorithmus folgt aus dem folgenden Satz:

#### Satz:

Gegeben: eine Partition $V_1, V_2$ der Knotenmenge $V$. Sei $s$ die billigste Kante zwischen $V_1$ und $V_2$. Dann gibt es einen MST der $s$ enthält. ($s$ ist sogar in allen MST enthalten).

### Beweis:




#### Annahme:

$s$ sei nicht im MST enthalten. Dann füge $s$ in den MST ein. Es entsteht ein Zyklus, der über $s'$ läuft. Entferne nun $s'$ aus dem MST. Es entsteht ein neuer **billigerer** MST. Dies ist ein Widerspruch zur Minimalität des Ausgangs-MST.

## 5.2 Algorithmus von Prim

Ein noch schnellerer Algorithmus zur Berechnung des MST ist der **Algorithmus von Prim**. Er ist ein gutes Beispiel für einen Greedy-Algorithmus:

> Starte mit einem Knoten.
>
> Wähle die billigste Kante, die den bisher aufgebauten Baum mit dem Rest verbindet.

genauer:

```
Initialisiere einen Baum mit dem Knoten A
Initialisiere Heap mit den Nachbarn von A
REPEAT
   entferne den billigsten Knoten x aus dem Heap
   füge x dem Baum hinzu
   update den Heap
UNTIL heap_is_empty
```

Der verwendete Heap muß dabei leistungsfähiger als ein normaler Heap sein (siehe später). In unserem vorangegangenen Beispiel ergibt sich dann folgender Ablauf:

| **Baumknoten** | **Rand mit Bewertung** $p(x)$ | **Rest** |
|---|---|---|
| A | F G B<br>2 6 7 | C D E |
| A, F | D G B E<br>3 6 7 7 | C |
| A, F, D | B E G<br>1 4 6 | C |
| A, F, D, B | C E G<br>1 2 6 | |
| A, F, D, B, C | E G<br>2 6 | |
| A, F, D, B, C, E | G<br>4 | |
| A, F, D, B, C, E, G | | |

Für einen Nachbarn $y$ von $x$ ist also folgendes zu tun:

- falls $y$ im Baum: tue nichts
- falls $y$ im Heap: falls $p(y) > c(x,y)$ setze $p(y) := c(x,y)$ dann führe Heapupdate durch (hier tauchen Probleme auf, die ein normaler Heap nicht bewältigen kann)

- falls y im Rest: füge $y$ in den Heap ein mit $p(y) := c(x, y)$

Der Aufwand des Prim-Algorithmus liegt in $O(|E| \cdot \log |V|)$, da jede Kante mit Aufwand $O(\log |V|)$ zweimal bearbeitet wird.

Bei dichtbesetzten Graphen ($|E| \simeq |V|^2, |V| = n$) ergibt sich für die Aufwände von Kruskal und Prim:

- Kruskal: $n^2 \cdot \log n^2 = n^2 \cdot \log n \cdot 2$
- Prim: $n^2 \cdot \log n$

# Kapitel 6

# Kürzeste Wege Probleme

Gegeben: $G = (V, E)$ ungerichteter Graph

Gesucht: der kürzeste Weg vom Knoten $A$ zum Knoten $B$

Bedeutet "kürzester Weg" der Weg mit minimaler Kantenzahl, so läßt sich dieser kürzeste Weg mit einer Breitensuche mit Wurzel $A$ berechnen. Falls "kurz" jedoch bedeutet, daß der Weg mit minimaler Summe der Kantenkosten gesucht wird, so ist ein brauchbarer Algorithmus nicht ohne weiteres zu finden.

Idee: Konstruiere einen Baum mit Wurzel $A$ und füge jeweils den Knoten mit kleinstem Abstand zu $A$ hinzu.

Analog zu Prim bearbeiten wir für einen Knoten $x$ alle seine Nachbarn $y$ und ersetzen ihre Bewertung $m(y)$ durch $m(y) := \min\{m(y), m(x) + c(x, y)\}$.

**Behauptung:**

Alle Knoten im so entstehenden Baum sind mit den korrekten kürzesten Wegekosten zu $A$ notiert. Im Rand und im Rest sind die Knoten mit den kürzesten Wegekosten zu $A$ markiert, wobei die Wege aber nur über Baumknoten laufen.

**Beweis: (durch Induktion)**

Verankerung: klar!

Schritt: Das Minimum $x$ in Rand und Rest kann endgültig markiert werden, da der kürzeste Weg von $A$ zu $x$ nur über Baumknoten laufen kann. Wichtig hierfür sind allerdings **positive Kantenkosten**!

Die Laufzeit eines auf dieser Idee basierenden Algorithmus läge also in $O(|E| \cdot \log |V|)$.

## 6.1 Algorithmus von Dijkstra

Eine bessere Variante dieses Algorithmus stammt von **Dijkstra**:

```
markiere alle Knoten als vorläufig ∞;
markiere den Startknoten (A) als 0; (endgültig)
REPEAT
   Sei x der zuletzt endgültig markierte Knoten;
   modifiziere alle Nachbarn y von x durch
         m(y) := min{m(y),m(x)+c(x,y)};
   markiere den Knoten mit der kleinsten vorläufigen
         Markierung als endgültig;
UNTIL alle Knoten sind markiert
```

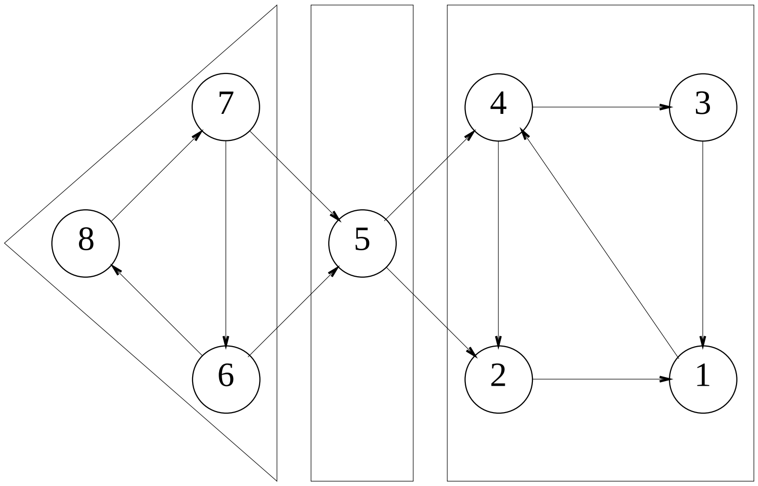
Die Laufzeit des Dijkstra-Algorithmus liegt in $O(|V|^2)$, was bei dicht besetzten Graphen eine erhebliche Verbesserung des ersten generischen Algorithmus bedeutet.

Allerdings ist auch beim Dijkstra-Algorithmus auf stets positive Kantenkosten zu achten, da der Algorithmus sonst nicht korrekte kürzeste Wege berechnet. Bei negativen Kantenkosten wären nämlich auch negative Zyklen möglich, die die Wohldefiniertheit des kürzesten Wege Problems verletzen würden.

## 6.2 Strenge Zusammenhangskomponenten

Eine **strenge Zusammenhangskomponente** (sZHK) ist ein maximaler Teilgraph, in dem je zwei Knoten durch gerichtete Wege verbunden sind.

**Beispiel:**




Ein Algorithmus zur Bestimmung der sZHK in $G = (V, E)$ sieht folgendermaßen aus:

1. Führe eine Tiefensuche in $G$ durch, wobei die Id eines Knotens vor dem Ende seiner Rekursion vergeben wird (nicht - wie ursprünglich - am Anfang).
2. Bilde $G'$ durch Umkehrung aller Kanten des Graphen $G$.
3. Führe nun eine Tiefensuche in $G'$ durch, wobei man jeweils mit den höchsten Ids aus Phase 1.) beginnt.
4. Die Tiefensuchebäume aus Phase 3.) sind dann die sZHK vom Graphen $G$.

Den Beweis der Korrektheit des Verfahrens liefert die folgende Behauptung:

**Behauptung:**

$v, w$ in einer sZHK von $G \iff v, w$ in einem Tiefensuchebaum von $G'$.

**Beweis:**

$\Rightarrow$: $v, w$ in einer sZHK von $G$.

$\Rightarrow$ Es existiert ein Weg von $v$ nach $w$ und ein Weg von $w$ nach $v$ in $G$.

$\Rightarrow$ Es existiert ein Weg von $v$ nach $w$ und ein Weg von $w$ nach $v$ in $G'$.

$\Rightarrow$ Bei Besuch von $v$ wird auch $w$ erreicht und umgekehrt.

$\Rightarrow$ $v$ und $w$ liegen in einem Tiefensuchebaum von $G'$.

$\Leftarrow$: Sei $x$ die Wurzel des Tiefensuchebaums, der $v$ und $w$ enthält.

$\Rightarrow$ Es existiert ein Weg von $v$ nach $x$ und ein Weg von $w$ nach $x$ in $G$.
Wegen des Weges von $v$ nach $x$ kann $v$ erst nach $x$ besucht worden sein.

$\Rightarrow$ $v$ wurde während der Bearbeitung von $x$ besucht.

$\Rightarrow$ Es existiert ein Weg von $x$ nach $v$ in $G$.

Analog: $\exists$ Weg von $x$ nach $w$ in $G \iff v$ und $w$ sind in einer sZHK.

## 6.3 Transitiver Abschluß

Gegeben: $G = (V, E)$ errichtet und unbewertet

Gesucht: der **transitive Abschluß** von $G$, d.h. die boolesche Matrix $a$, für die gilt:

$$a[i,j] = \text{ TRUE } \iff \text{ es gibt einen Weg von } i \text{ nach } j \text{ in } G$$

Der nun folgende Algorithmus zur Bestimmung der transitiven Hülle von $G$ ist ein typisches Beispiel für das Dynamic Programming:

Bestimme $r^k(i,j) = \text{TRUE} \iff \exists$ Weg von $i$ nach $j$, der nur über Knoten $\{1, ..., k\}$ läuft.
Initial seien die $r^0(i,j)$-Werte durch $r^0(i,j) = \text{TRUE} \iff (i,j) \in E$ gegeben.

Der Algorithmus sieht dann wie folgt aus:

```
FOR k := 1 TO n DO
   FOR ALL i, j DO
      r^k(i,j) := r^(k-1)(i,j) OR (r^(k-1)(i,k) AND r^(k-1)(k,j))
   END;
END;
```

Die Matrix $(r^n)_{i,j}$ ist dann das gewünschte Ergebnis.

Die Laufzeit dieses Algorithmus liegt in $O(n^3)$.

## 6.4 All pairs shortest path

Gesucht: $d[i,j] :=$ Länge des kürzesten Weges von $i$ nach $j$.

Die Technik zur Berechnung dieser $d[i,j]$-Werte entspricht der obigen und zwar berechnet man $d^k[i,j] :=$ Länge des kürzesten Weges von $i$ nach $j$, der nur über Knoten $\{1,...,k\}$ läuft. Initial setzt man $d^0[i,j] := c(i,j)$.

Rekursiv bestimmt man nun $d^k[i,j] := \min\{d^{k-1}[i,j], d^{k-1}[i,k]+d^{k-1}[k,j]\}$. $d^n[i,j]$ gibt dann schließlich die **Länge** des kürzesten Weges von $i$ nach $j$ in $G$ an, der **Weg** selber ist damit aber noch nicht berechnet! Dies bewerkstelligt man durch fortwährendes

Berechnen einer Matrix $w[i,j]$:

$$w[i,j] := w[i,k], \text{ falls } d^{k-1}[i,k] + d^{k-1}[k,j] < d^{k-1}[i,j] \text{ (Init.} : w[i,j] := j, \text{ falls } (i,j) \in E)$$

$w[i,j]$ gibt also den ersten Knoten auf dem (bisher) kürzesten Weg von $i$ nach $j$ an.

Der kürzeste Weg von $i$ nach $j$ ergibt sich also, indem man quasi durch die Matrix $w[i,j]$ "springt", also $w[\ldots[w[i,j],j]\ldots]$ berechnet.

### 6.4.1 Algorithmus von Floyd

Der hier beschriebene Algorithmus ist der **Algorithmus von Floyd**. Zwei Anwendungen dieses Algorithmus bzw. der Variante von Dijkstra sollen nun vorgestellt werden:

1. **Bottleneck:** Der Bottleneck eines Weges ist die Kante des Weges mit den billigsten Kosten. Dies ist z.B. dann sinnvoll, wenn die Kosten die Kapazitäten oder Tragfähigkeiten modellieren. Gesucht ist dann also ein Weg von $A$ nach $B$, dessen billigste Kante möglichst teuer (groß) ist. Diesen Weg findet man durch Anwendung des Floyd- bzw. Dijkstra-Algorithmus, nur daß man statt die Kantenkosten aufzuaddieren sie **minimiert** und statt den billigsten den **teuersten** Weg vorzieht.

2. **Zuverlässigkeitsproblem:** In einem Kommunikationsnetzwerk sei $p(i,j)$ die Erfolgswahrscheinlichkeit für die Verbindung von $i$ nach $j$ für alle $(i,j) \in E$. Die Sicherheit einer Verbindung ist dann gleich dem Produkt der Erfolgswahrscheinlichkeiten $p(i,j)$

der beteiligten Kanten. Gesucht ist dann also der sicherste Weg von $A$ nach $B$. Dazu modifiziert man die Kantenkosten $p(i,j)$ in $-\log(p(i,j))$. Der kürzeste Weg von $A$ nach $B$ in diesem modifizierten Graphen ist dann die sicherste Verbindung.

Bei kürzesten Wege Problemen haben wir bisher immer **positive** Kantenkosten vorausgesetzt und so die Existenz von negativen Zyklen ausgeschlossen. Im folgenden soll nun ein Algorithmus zur Berechnung der kürzesten Wege vorgestellt werden, der auch bei **negativen** Kantenkosten - im Gegensatz zu den bisherigen Algorithmen - korrekt arbeitet: Der Algorithmus von Floyd ist eine Alternative zu Dijkstra bei single source shortest path.

Gegeben: $G = (V, E)$ mit zum Teil negativen Kosten

Gesucht: kürzester Weg von Knoten $A$ zum Knoten $B$

Ähnlich wie bei Dijkstra ersetzt man $m[y]$ durch $m[x] + c(x,y)$ solange es eine Kante $(x,y)$ mit $m[x] + c(xy) < m[y]$ gibt.

Im $i$-ten Durchgang bestimmt man

$$m^i[y] := \min\{m^{i-1}[y], \min\{m^{i-1}[x] + c(x,y) \mid x \text{ ist Vorgänger von } y\}\}$$

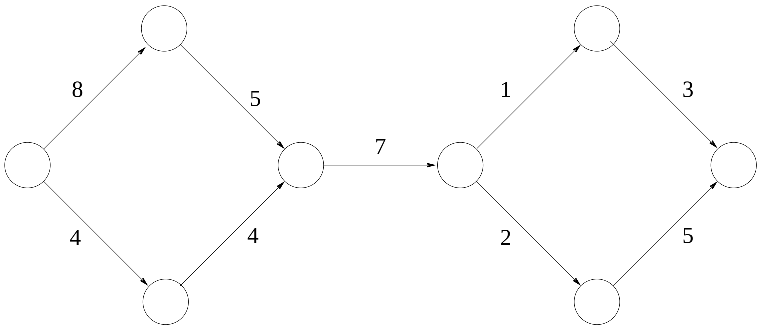
Falls nach dem $(n-1)$-ten Durchgang immer noch eine Verbesserung der $m^i[y]$-Werte entsteht, so liegt ein negativer Zyklus in $G$ vor, das Problem ist also unbeschränkt.

Auch im all shortest path Algorithmus nach Floyd, welcher negative Kantenkosten verkraftet, muß man überprüfen, ob ein negativer Zyklus vorliegt. Dies ist genau dann der Fall, wenn in der Hauptdiagonalen aller $d^k$-Matrizen ein negativer Eintrag zu finden ist. Das Problem ist in diesem Fall nicht mehr sinnvoll.

## 6.5 Pert

Eine weitere Anwendung von kürzeste Wege Berechnungen ergibt sich durch die sog. **PERT** (**p**rogram **e**valuation **r**eview **t**echnique). Bei diesen Problemen hat man es mit azyklischen, gerichteten und bewerteten Graphen $G$ zu tun. Diese Graphen veranschaulichen Produktionsabläufe, wie z.B. das Erstellen von Häusern etc. Die Kanten von $G$ stellen dabei bestimmte Tasks also Teilproduktionen dar, die Länge der Kanten modelliert die benötigte Zeit für diese Task.

**Beispiel:**



Gesucht ist in einem solchen PERT-Graphen nun der längste Weg vom Startknoten zum Zielknoten. Dessen Länge entspricht dann nämlich der benötigten Gesamtproduktionszeit.

Der Algorithmus, der dieses berechnet, lautet:

```
REPEAT
   finde Knoten y, dessen sämtliche Vorgänger markiert sind.
   markiere y mit m[y]:=max{m[x]+c(x,y)|x ist Vorgänger von y}
UNTIL Zielknoten ist markiert
```

Die Implementation basiert auf der Idee, sich eine Schlange $s$ mit Knoten, deren sämtliche Vorgänger markiert sind, zu halten. Initial ist die Quelle das einzige Element der Schlange $s$.

```
WHILE NOT emptyqueue(s) DO
   x := dequeue(s);
   WHILE NOT endpos(g[x]) DO
      y := elem(g[x]);
      m[y] := max m[y], m[x]+c(x,y);
      indegree[y] := indegree[y]-1;
      IF indegree[y] = 0 THEN enqueue (s, y) END;
      advance (g[x]);
   END;
END;
```

Die Gesamtlaufzeit dieses Algorithmus liegt in $O(|E|)$, denn auch die Initialisierung des indegree-Arrays läßt sich mit Hilfe der Adjazenzliste in $O(|E|)$ durchführen.

# Kapitel 7

# Location-Probleme

## 7.1 Zentrum

Das **Zentrum** eines Graphen $G$ ist der Knoten, dessen am weitesten entfernter Knoten in $G$ möglichst nahe liegt.

Das Ermitteln des Zentrums eines Graphen ist unter Anwendung des Floyd-Algorithmus zur Bestimmung aller kürzesten Wege in $G$ sehr leicht:

Bestimme mit Floyd alle $d(i,j)$-Werte

Errechne $\max[i] := \max\{d(i,j) \mid j \in G\}$

Suche das $i$ mit minimalem $\max[i]$, dieses $i$ ist das Zentrum von $G$.

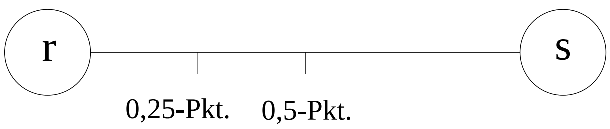
### 7.1.1 Generelles Zentrum

Das **generelle Zentrum** eines Graphen $G$ ist der Knoten, dessen entferntester Kantenpunkt möglichst nahe liegt.

Definere:

Für $0 \leq f \leq 1$ sei der $f$-Punkt der Kante $(r,s)$ der Punkt, der $f \cdot c(r,s)$ Einheiten von $r$ bzw. $(1-f) \cdot c(r,s)$ Einheiten von $s$ entfernt ist.

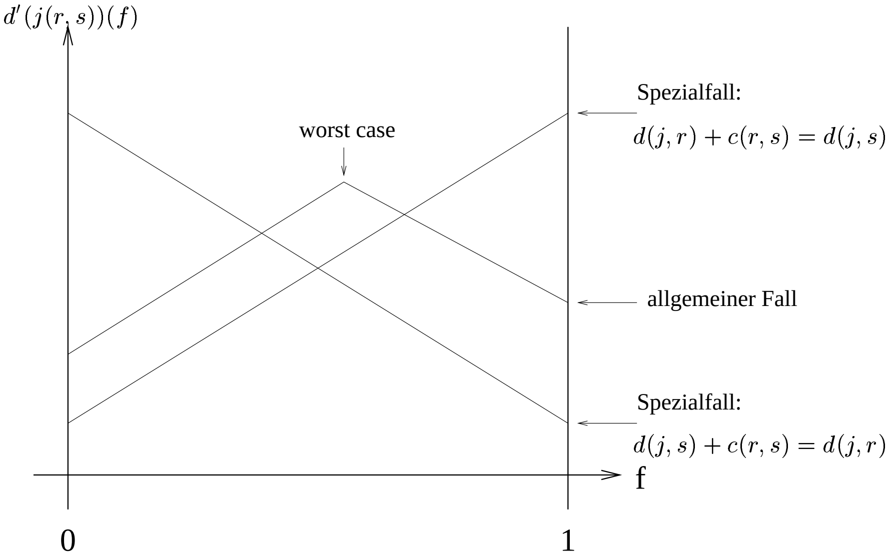
**Beispiel:**




Betrachte nun die kürzeste Entfernung von einem Knoten $j$ zu allen $f$-Punkten der Kante $(r,s)$. Es ergibt sich eine Funktion $(d'(j,(r,s)))(f)$, wobei $f \in [0,1]$ ist. Wir unterscheiden nun in zwei Fälle:

**a) $(\mathbf{r},\mathbf{s})$ ungerichtet:** In diesem Fall ist es möglich, daß der kürzeste Weg von $j$ zu einem $f$-Punkt auf $(r,s)$ entweder über $r$ oder auch über $s$ läuft. Auf jeden Fall läuft er aber bis zu diesem Punkt auf dem kürzesten $j-r$- bzw. $j-s$-Weg mit der Länge $d(j,r)$ bzw. $d(j,s)$. Für $d'(j,(r,s))$ sind also folgende Graphenverläufe möglich:

**Beispiel:**



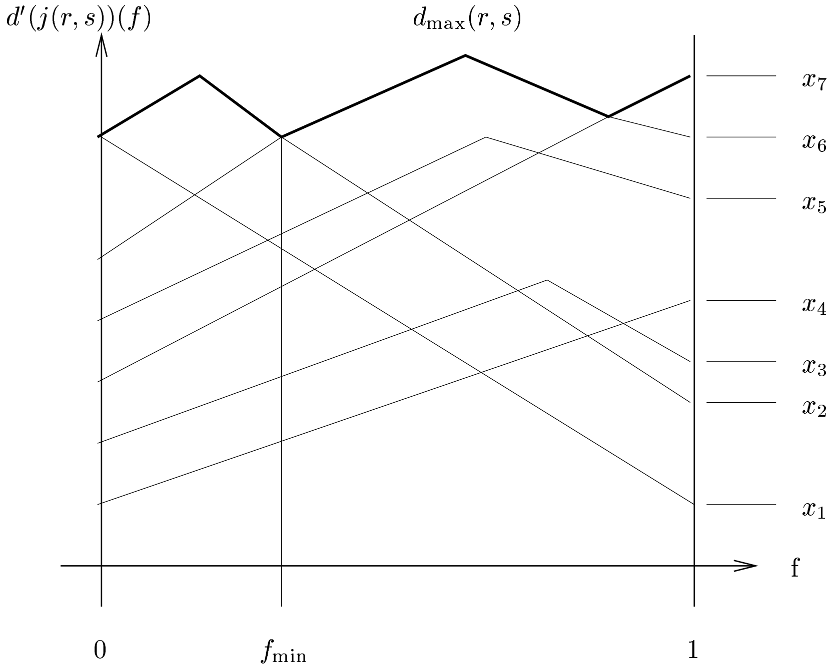

Die Frage ist nun, wie man den worst case, also die maximale Entfernung von $j$ zu einem Kantenpunkt $f$, berechnet. Es leuchtet ein, daß diese maximale Entfernung $\max[j,(r,s)]$ gegeben ist durch $\max[j,(r,s)] = \frac{1}{2}[d(j,r)+c(r,s)+d(j,s)]$.

Nun ist es leicht, das generelle Zentrum von $G=(V,E)$ zu bestimmen, es ist der Knoten $i$ mit minimalem $\max\{\max[i,(x,y)] | (x,y) \in E\}$.

**b) $(\mathbf{r},\mathbf{s})$ gerichtet:** Dieser Fall ist hier der einfachere, da jeder kürzeste Weg von $j$ zu einem Kantenpunkt von $(r,s)$ immer über $r$ läuft. Daraus folgt, daß der $f$-Punkt mit maxi maler Entfernung zu $j$ der Punkt der Kante $(r,s)$ ist, der beliebig nahe an $s$ liegt. Diese $r$ $f$-Punkt hat aber trivialerweise die Entfernung $\max[j,(r,s)] = d(j,r)+c(r,s)$ von $j$. Die Berechnung des generellen Zentrums in $G$ läuft nun analog zu a).

### 7.1.2 Absolutes Zentrum

Das **absolute Zentrum** eines Graphen $G$ ist der Kantenpunkt, dessen entferntester Knoten möglichst nahe liegt. Zur Berechnung des absoluten Zentrums betrachten wir zunächst eine Kante $(r,s)$. Für jeden $f$-Punkt dieser Kante kann man die Entfernung zu einem Knoten $x$ als Funktion $d'((r,s),x)$ ausdrücken. Berechne dann $d'((r,s),x)$ für **alle** Knoten $x \in V$. Es ergeben sich folgende Funktionsverläufe:




Bilde nun das Maximum $d_{\max}(r,s) = \max\{d'((r,s),x) | x \in V\}$. Diese Funktion ist im vorigen Schaubild als dicke Linie gezeichnet. Suche nun den $f$-Punkt $f_{\min}$ mit $d_{\max}(r,s)(f_{\min})$ minimal. Auch dieser Punkt ist in diesem Schaubild eingezeichnet. Sei nun $d_{opt}(r,s) = d_{\max}(r,s)(f_{\min})$, also der Funktionswert von $f_{\min}$. Bildet man nun das Minimum aller $d_{opt}$-Werte über alle Kanten des Graphen, so ermittelt man damit die Kante, auf der das absolute Zentrum liegen muß. Der zu dieser Kante gehörige $f_{\min}$-Wert legt dann das absolute Zentrum von $G$ exakt fest. Eine Schwierigkeit in diesem Verfahren liegt allerdings in der Berechnung der Funktionen $d_{\max}$ für alle Kanten des Graphen. Einen Algorithmus mit annehmbarer Laufzeitentwicklung für dieses Problem zu finden dürfte in jedem Fall schwer fallen.

### 7.1.3 Absoluter Median

Das **absolute Median** Eines Graphen $G$ ist der Kantenpunkt, dessen **durchschnittliche** Entfernung zu allen Knoten minimal ist.

#### Behauptung:

Der absolute Median liegt in jedem Fall in einem Knoten.

#### Beweis:

Es gilt:

(i) Die Summe konkaver Funktionen ist wieder eine konkave Funktion.

(ii) Das Minimum einer konkaven Funktion wird an einem der Ränder angenommen.

Da die einzelnen Funktionen, die bei der Berechnung auftreten, auch konkav sind, liegt das gesuchte Minimum auch hier in einem Rand einer Kante, also in einem Knoten selbst.

Der Algorithmus zur Berechnung des absoluten Medians sieht also so aus:

Berechne mit dem Algorithmus von Floyd alle $d(i,j)$-Werte.

Bilde für alle Knoten $i$ aus $G$ $sum[i] = \sum_{j=1}^{n} d(i,j)$

Finde das $i$ mit minimalem $sum[i]$. Dieser Knoten $i$ ist dann der absolute Median.

Der Algorithmus ist also erstaunlich einfach, relativ zur Komplexität des Problems gesehen.

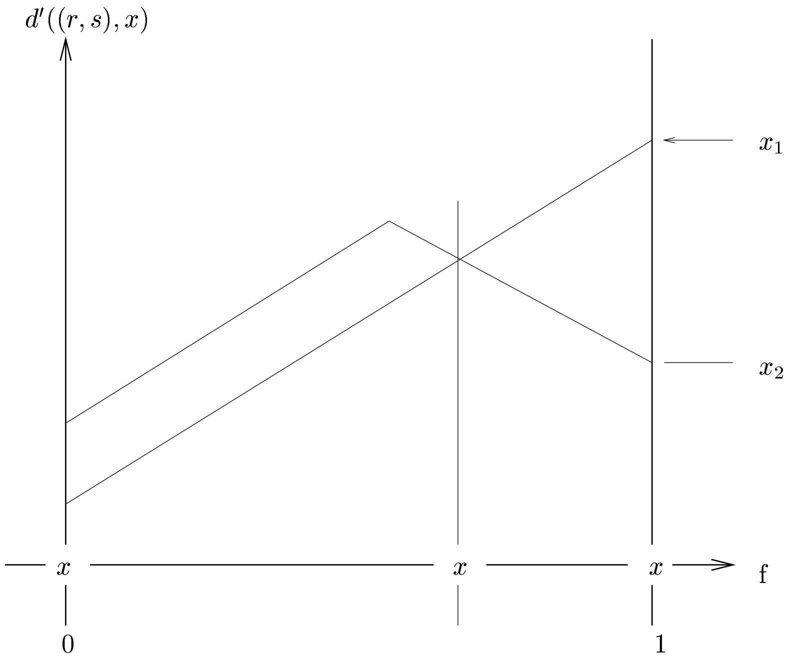
### 7.1.4 m-Zentrum

Als letztes Location-Problem betrachten wir nun das **m-Zentrum mit Abstand $\Delta$**. Das ist eine $m$-elementige Menge aus Kantenpunkten und Knoten, von der aus alle Knoten mit höchstens Abstand $\Delta$ erreicht werden können. Dabei ist die Zahl $m$ vorgegeben und das $\Delta$ läßt sich dann berechnen, wobei es natürlich möglichst klein sein soll.

#### Behauptung:

O.B.d.A. liegt jeder Punkt eines $m$-Zentrums so, daß er zu zwei Knoten denselben Abstand hat (oder in einem Knoten selbst).

Der Beweis dieser Behauptung folgt aus der folgenden Skizze:




Diese Skizze entspricht der Darstellung der Funktionen, die bei der Berechnung des absoluten Zentrums eine Rolle spielen. Es ist einleuchtend, daß ein potentieller $m$-Zentrums-Punkt nur einer der mit einem Kreuz markierten Punkte sein kann. Für diese Punkte gilt aber genau die oben gemachte Behauptung.

Es ergibt sich also nur eine **endliche** Menge von Zentrumskandidaten $K = \{p_1, \ldots, p_k\}$. Bilde jetzt die Matrix $w_{ij} := d(p_j, v_i)$ für alle $v_i \in V, p_j \in K$ und wähle $m$ Kandidaten $\{p_1, \ldots, p_m\} := M$ als erstes vorläufiges m-Zentrum. Sei $g_M := \max\{\min\{w_{ij} | p_j \in M\} | i \in V\}$.

Die entscheidende Frage ist nun, ob dieses $g_M$ noch zu unterbieten ist.

Um dieser Frage nachzugehen, bilden wir nun eine boolesche Matrix $a_{ij}$, deren Element gleich 1 ist, wenn $w_{ij} < g_M$ gilt, und gleich 0 ist im anderen Fall. In dieser Matrix suchen wir nun eine Spaltenüberdeckung, d.h. eine Menge $M_1$ von Spalten mit $\sum\limits_{j \in M_i} a_{ij} > 0 \forall i$ und minimaler Kardinalität von $M_1$.

Falls die errechnete Spaltenüberdeckung mehr als $m$ Spalten hat, so war $M$ bereits optimal, d.h., $g_M$ war schon der minimale Wert für $\Delta$.

Finden wir jedoch eine Spaltenüberdeckung mit $m$ oder weniger Spalten, so haben wir ein neues, besseres $m$-Zentrum gefunden. Dieses neue $m$-Zentrum unterziehen wir nun erneut dieser Prozedur, bis wir kein neues $m$-Zentrum mehr finden. Dieser Abbruch erfolgt auf jeden Fall nach endlich vielen Iterationen, da in jeder Iteration die Anzahl der Einsen in $(a)_{ij}$ abnimmt.

# Kapitel 8

# Maximale Flüsse Probleme

Gegeben: $G = (V, E)$ gerichtet und gewichtet mit $c : E \to \mathbb{N}$. Die Gewichte $c(i, j)$ werden hier als Kantenkapazitäten interpretiert. Zwei Knoten aus $V$ sind hier besonders ausgezeichnet, nämlich die Quelle $s$ und die Senke $t$.

## 8.1 Fluß

Ein **Fluß** ist dann eine Funktion $f : E \to \mathbb{R}$ mit:

1. $0 \leq f((i, j)) \leq c(i, j)$
2. für alle $i \in V \backslash \{s, t\}$ gilt: $\sum\limits_{a \in V(i)} f(a, i) = \sum\limits_{b \in N(i)} f(i, b)$, wobei $V(i)[N(i)]$ alle Vorgänger [Nachfolger] von $i$ im Graphen $G$ sind.

Gesucht ist nun der maximale Gesamtfluß $F = \sum\limits_{b \in N(s)} f(s, b) = \sum\limits_{a \in V(t)} f(a, t)$.

## 8.2 Erweiternder Weg

Ein **erweiternder** Weg in $G$ ist eine Folge von Knoten, beginnend bei $s$ und endend bei $t$, wobei für zwei aufeinanderfolgende Knoten $i$ und $j$ gilt:

(i) Es existiert eine Vorwärtskante $(i, j) \in E$ mit $f(i, j) < c(i, j)$

oder

(ii) Es existiert eine Rückwärtskante $(j, i) \in E$ mit $f(j, i) > 0$.

Längs so eines erweiternden Weges kann der Fluß vergrößert werden, indem man durch die Vorwärtskanten zusätzliche Einheiten fließen läßt oder man den Fluß in den Rückwärtskanten verringert. Beides ist nach der Definition des erweiternden Weges möglich.

## 8.3 Algorithmus von Ford-Fulkerson

Aus dieser Eigenschaft läßt sich ein erster generischer Algorithmus zur Berechnung des maximalen Flusses ableiten: **der Algorithmus von Ford-Fulkerson (1962)**

```
Finde einen zulässigen Fluß f in G
REPEAT
   erhöhe diesen Fluß f
UNTIL eine Erhöhung von f ist nicht möglich
```

Die Korrektheit dieses Algorithmus folgt unmittelbar aus dem folgenden Satz:

$$\text{Ein Fluß } f \text{ ist maximal} \iff \text{Es gibt keinen erweiternden Weg.}$$

**Beweis:**

$\Rightarrow$: klar !

$\Leftarrow$: Der Beweis wird dem Leser hier noch schuldig geblieben und folgt später.

## 8.4 Cut

Ein **Cut** des Graphen $G = (V, E)$ ist eine Partition der Knotenmenge $V$ in $S$ und $S'$, so daß gilt: $s \in S$ und $t \in V \backslash S = \overline{S}$. Sei dann $(S, \overline{S}) := \{(x, y) \in E | x \in S, y \in \overline{S}\}$. Die Kapazität eines Cuts ist dann festgelegt durch $c(S) := \sum\limits_{e \in (S,\overline{S})} c(e)$.

Daraus folgt unmittelbar der folgende Satz:

Sei $f$ ein Fluß der Größe $F$, sei $S$ ein Cut von $G$

$$\Rightarrow F = \sum_{e \in (S,\overline{S})} f(e) - \sum_{e \in (\overline{S},S)} f(e).$$

Der Fluß ist also an jedem beliebigen Cut meßbar.

Aus diesem Satz folgt aber direkt:

Sei $f$ ein Fluß der Größe $F$, sei $S$ ein Cut von $G$

$$\Rightarrow F \leq c(S),$$

d.h. der Fluß ist höchstens so hoch wie die Kapazität eines Cuts.

$\Rightarrow$

Ist ein Fluß $f$ so groß wie die Kapazität eines Cuts $S$, so ist $f$ ein maximaler Fluß und $S$ ist ein minimaler Cut bezogen auf seine Kapazität c(S).

Betrachte nun also folgende Situation:

Es gibt in $G$ keine erweiternden Wege mehr. Dann enden also alle von $s$ ausgehenden erweiternden Wege - genauer gesagt deren Anfangsstücke - entweder bei einer saturierten Vorwärtskante oder bei einer Rückwärtskante mit Fluß 0. Durch diese Kanten wird also ein Cut $S$ impliziert, dessen Kapazität gleich dem momentanen Fluß ist. Daher muß der momentane Fluß aufgrund des letzten Satzes maximal sein. Somit ist der fehlende Beweis von vorhin erbracht.

Der Algorithmus kann also so notiert werden:

```
Sei f = 0 der initiale Fluß
REPEAT
   Suche bzgl. f einen erweiternden Weg von s nach t
   falls gefunden: erhöhe f längs dieses Weges.
UNTIL kein erweiternder Weg gefunden
Der Fluß f ist nun maximal.
```

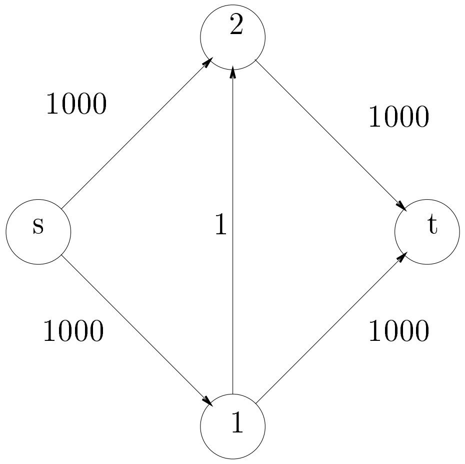
Die Idee für die Suche nach einem erweiternden Weg in $G$ ist eine Breitensuche in $G$, die beim Knoten $s$ beginnt:

```
Markiere s;
WHILE möglich AND t ist nicht markiert DO
   Sei x ein markierter Knoten;
   IF ∃ unmarkiertes y mit (x,y) ∈ E und f(x,y)<c(x,y) THEN
      markiere y und vermerke seinen Vorgänger x;
      δ(x,y) := c(x,y) - f(x,y);
   IF ∃ unmarkiertes y mit (y,x) ∈ E und f(y,x) > 0 THEN
      markiere y und vermerke seinen Vorgänger x;
      δ(x,y) := f(y,x);
END;
```

Eine interessante Eigenschaft von Netzwerken mit ganzzahligen Kapazitäten ist, daß auch die maximalen Flüsse in solchen Netzwerken immer ganzzahlig sind, da der obige Algorithmus nur ganzzahlige Erhöhungen durchführt.

**Anmerkung:** Bei irrationalen Kantenkapazitäten konvergiert der Algorithmus nicht zwingend. Solche Anwendungen sind in der Praxis aber auch sehr selten zu finden und daher ist diese Eigenschaft nicht von großer Bedeutung.

Betrachten wir nun das Laufzeitverhalten des Algorithmus anhand eines Beispiels:




Bei ungeschickter Wahl des ersten erweiternden Weges, nämlich $s, 1, 2, t$, ist in der ersten Iteration lediglich **eine** Erhöhung um eine Einheit möglich. Wir saturieren also lediglich die Kante $(1, 2)$. Als nächsten erweiternden Weg können wir nun $s, 2, 1, t$ wählen, wobei auch hier wieder nur eine Erhöhung um eine Einheit zu erreichen ist. Wiederholt man diese beiden Wahlen immer wieder, so benötigt man insgesamt 2000 (!) Iterationen, bis man den maximalen Fluß gefunden hat.

Die Laufzeit liegt also im worst case in $O(|E| \cdot v)$, wobei $v$ der maximale Fluß in $G$ ist. Der Algorithmus hat also ein exponentielles Laufzeitverhalten relativ zur Darstellungslänge des Graphen in einem Rechner.

Im weiteren werden wir uns nun damit beschäftigen, wie wir durch geschicktere Wahl der erweiternden Wege die Komplexität des Algorithmus erheblich verbessern können.

Dafür gibt es im wesentlichen zwei Ansätze:

1. Finde jeweils den erweiternden Weg mit der **größtmöglichen** Erhöhung.
2. Finde jeweils den **kürzesten** erweiternden Weg.

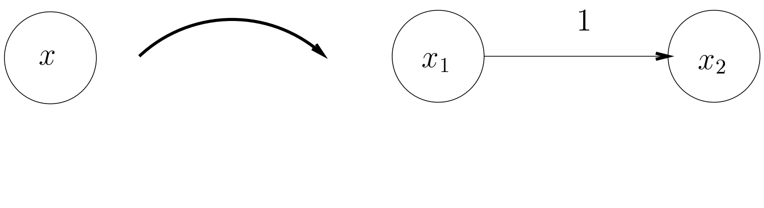
Im ersten Fall erreichen wir damit eine Verbesserung der Laufzeit auf $O(|E| \cdot \log v)$, in zweiten Ansatz erreichen wir eine Komplexität von $O(|V|^3 \cdot |E|)$, also eine Komplexität, die vollkommen unabhängig vom maximalen Fluß in $G$ ist.

Es existiert eine große Anzahl von Anwendungsmöglichkeiten für Max-Fluß-Probleme, von denen hier einige näher vorgestellt werden sollen:

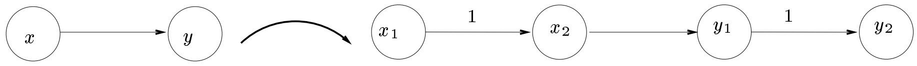
Gegeben: $G = (V, E)$ gerichtet, $s$, $t \in V$ seien die Quelle bzw. Senke von $G$.

Frage: Wieviel knotendisjunkte Wege gibt es von $s$ nach $t$?

Um diese Frage zu beantworten, modifiziert man das vorliegende Netzwerk, indem man jeden Knoten $x$ durch zwei Knoten $x_1$ und $x_2$ ersetzt, die mit einer Kante mit Kapazität 1 verbunden werden:



Außerdem ersetzt man jede Kante $(x, y)$ durch die neue Kante $(x_2, y_1)$:




Nun bestimmt man mit den bekannten Algorithmen den maximalen Fluß in diesem modifizierten Netzwerk. Dessen Größe entspricht dann offensichtlich der Anzahl der knotendisjunkten Wege von $s$ nach $t$.

# Kapitel 9

# Kostenminimale Flüsse

Gegeben: $G = (V, E)$ gerichtet, $s$, $t \in V$ Quelle bzw. Senke. Jeder Kante in $G$ wird eine Kapazität und die Kosten pro Flußeinheit zugeordnet:

$$c : E \to \mathbb{N} \text{ (Kapazität)}$$
$$a : E \to \mathbb{N} \text{ (Kosten)}$$

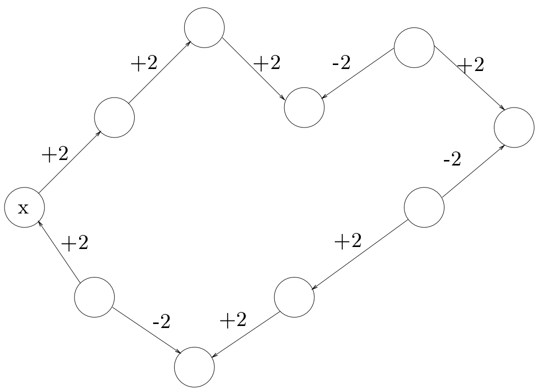
Gesucht: der Fluß $f$ der Größe $v$ mit minimalen Kosten.

Idee: Finde zunächst — z.B. mit Ford/Fulkerson — irgendeinen Fluß der Größe $v$. Lenke diesen Fluß nun so um, daß seine Größe gleich bleibt aber seine Kosten sinken.

## 9.1 Erweiternder Kreis

Ein **erweiternder** Kreis bzgl. eines Flusses $f$ ist ein erweiternder Weg von einem Knoten $x$ zu sich selbst.

**Beispiel:**



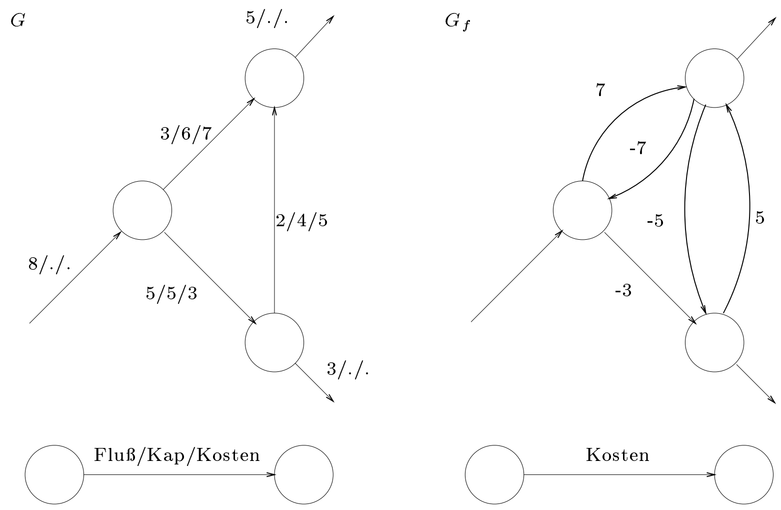

Offenbar läßt sich längs eines erweiternden Kreises ein Fluß umlenken, ohne seine Mächtigkeit zu verändern.

Daraus ergibt sich der folgende Satz: Fluß $f$ hat minimale Kosten $\iff$ Es gibt bzgl. $f$ keinen kostenmindernden erweitern den Kreis.

Konstruiere also aus $f$ und $G$ einen Graphen $G_f = (V, E_f)$ mit folgenden Kantenkosten:

$$a'(i,j) \begin{cases} a(i,j), & \text{falls } f(i,j) < c(i,j) \\ -a(j,i), & \text{falls } f(j,i) > 0 \end{cases}$$

**Beispiel:**



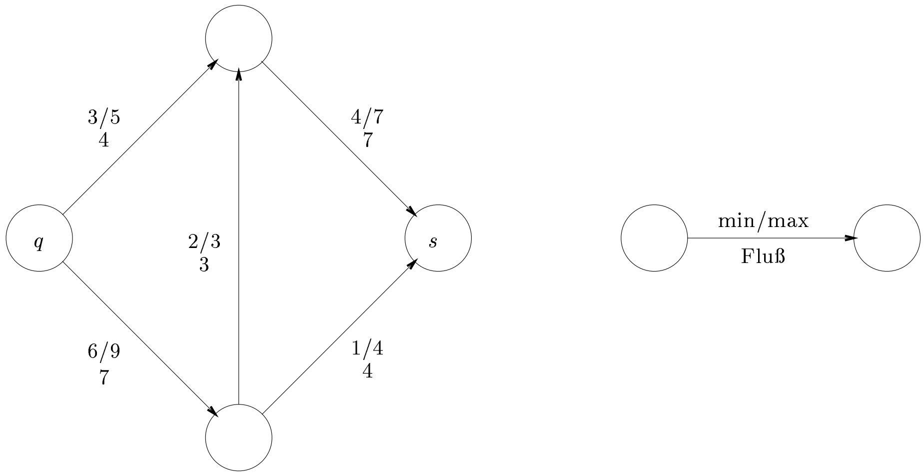

Ein kostensenkender Kreis in $G$ entspricht einem Kreis negativer Länge in $G_f$. Im obigen Beispiel haben wir z.B. in $G_f$ einen Kreis der Länge -1. Ändere nun den Fluß um das Minimum der Möglichkeiten, in unserem Fall also um $\min\{3, 2, 5\} = 2$. Die Ersparnis beträgt dann also: $-2 \cdot 7 + 2 \cdot 5 + 2 \cdot 3 = 2$. Zur Erkennung von negativen Zyklen in $G_f$ kann man z.B. den Algorithmus von Floyd verwenden, denn wenn im Laufe dieses Algorithmus in der Hauptdiagonalen der Entfernungsmatrix ein negativer Wert auftaucht, so liegt ein negativer Zyklus vor.

Gegeben: $G = (V, E)$ gerichtet mit oberen und unteren Schranken:

$$o : E \to \mathbb{Z}(\text{ob. Schr.})$$
$$u : E \to \mathbb{Z}(\text{unt. Schr.})$$

Gesucht: Ein zulässiger Fluß in $G$, d.h. ein Fluß, der die Schranken respektiert.

**Beispiel:**

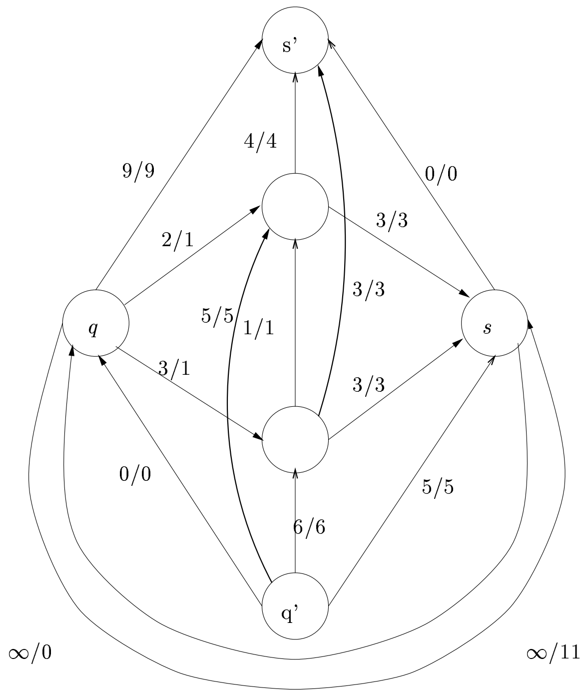


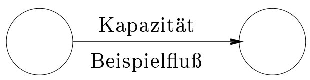

Um dieses Problem zu lösen, betrachten wir einen modifizierten Graphen $G' = (V', E')$ mit $V' = V \cup \{q', s'\}$ und $E' = E \cup \{q'\} \times V \cup V \times \{s'\} \cup \{(q,s),(s,q)\}$.

Den Kanten aus $E'$ ordnen wir dabei Kapazitäten zu, und zwar:

- $\forall x, y \in V : c(x, s') :=$ Summe der Mindestflüsse aus $x$ heraus
- $\forall x \in V : c(q', x) :=$ Summe der Mindestflüsse aus $x$ hinein
- $\forall x, y \in V, x \neq y : c(x, y) := o(x, y) - u(x, y)$
- $c(q, s) = c(s, q) := \infty$

Es ergibt sich in unserem Beispiel folgender Graph $G'$:




In diesem Graphen $G'$ suchen wir nun den maximalen Fluß von $q'$ nach $s'$. Ein solcher Fluß ist in dem obigen Schaubild eingezeichnet.

Für diesen Fluß gilt dann der folgende Satz:

$$G' \text{ hat einen Fluß } f' \text{ der Größe } S \iff G \text{ hat einen zulässigen Fluß } f$$

(wobei $S :=$ Summe der Mindestreinflüsse in $s'$ = Summe der Mindestrausflüsse aus $q'$). Somit können wir jetzt also entscheiden, ob $G$ einen zulässigen Fluß besitzt oder nicht. Aber wie können wir diesen zulässigen Fluß $f$ aus $f'$ berechnen? Die Antwort gibt folgende Formel:

$$f(x, y) = u(x, y) + f'(x, y).$$

Die Idee dieses Verfahrens sei nun noch einmal zusammengefaßt:

Die ausgelasteten Zusatzkanten von $q'$ bzw. nach $s'$ erzwingen in jedem Knoten ein "Fluß-gefälle", welches durch die ursprünglichen Mindestflüsse diktiert wurde und welches durch die Restkapazitäten kompensiert wird.

# Kapitel 10

# Matching

Gegeben: $G = (V, E)$ ungerichtet. Die Kanten symbolisieren hier mögliche Zuordnungen.

Gesucht: eine Zuordnung $M$, d.h. eine unabhängige Kantenmenge $M$. Unabhängig bedeutet, daß gilt: $(i, j), (i', j') \in M \Rightarrow i \neq i', j \neq j', i \neq j', j \neq i'$

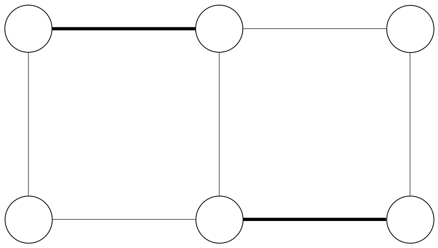
## 10.1 Maximum/maximales Matching

Ein **maximales Matching** ist ein Matching, welches nicht mehr erweitert werden kann. Ein **Maximum Matching** ist ein Matching mit maximaler Kantenkardinalität.

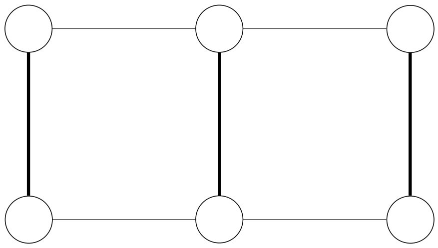
Dabei gilt: $M$ ist ein Maximum Matching. $\Rightarrow$ $M$ ist ein maximales Matching. (Die Umkehrung ist i. a. falsch !)

**Beispiel:**

maximales Matching:



Maximum Matching:



## 10.2 Bipartiter Graph

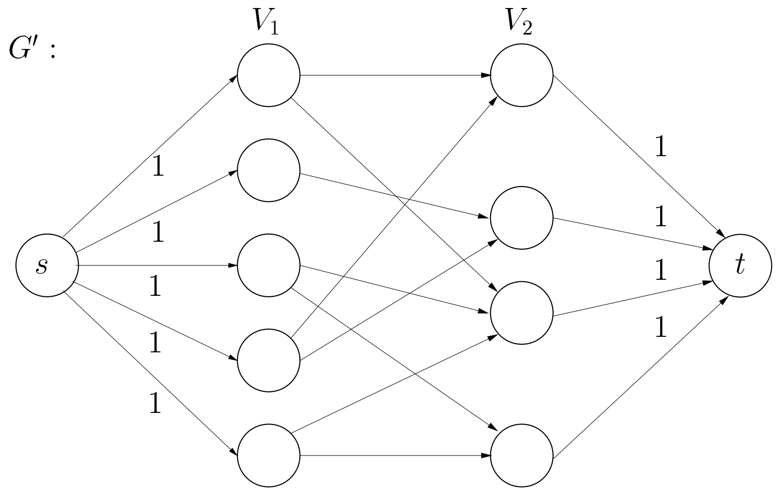
Am häufigsten werden Matchingprobleme in **bipartiten Graphen** betrachtet. Das sind Graphen $G = (V, E)$, für die gilt: $V = V_1 \cup V_2$ (disjunkt) und $E \subseteq V_1 \times V_2$. Wir wollen uns daher in den nächsten Ausführungen zunächst auf solche Graphen konzentrieren.

Die Frage ist nun, wie man in bipartiten Graphen ein Maximum Matching bestimmen kann.

In diesem ersten Ansatz lösen wir dieses Problem unter Mithilfe von Flußalgorithmen, genauer gesagt mit der Berechnung von maximalen Flüssen:

Transformiere den Graphen $G = (V, E)$ in ein gerichtetes Netzwerk $G' = (V', E')$ mit $V' = V \cup \{s, t\}$ und $E' = E \cup \{(s, x) | x \in V_1\} \cup \{(y, t) | y \in V_2\}$. Die Kanten aus $E$ werden in $G'$ als gerichtet betrachtet und zwar von $V_1$ nach $V_2$ . Den Kanten von $s$ aus und nach $t$ hinein wird die Kapazität 1 zugeordnet.

**Beispiel:**




Durch die Kapazitäten der $(s, .)$- und $(., t)$-Kanten wird die Unabhängigkeit der Kantenmenge erreicht, die unser Matching darstellen soll. Führt man nämlich in $G'$ eine Max-Fluß-Betrachtung durch, so entspricht ein Fluß der Größe $k$ in $G'$ einem Matching $M$ der Kardinalität $k$.

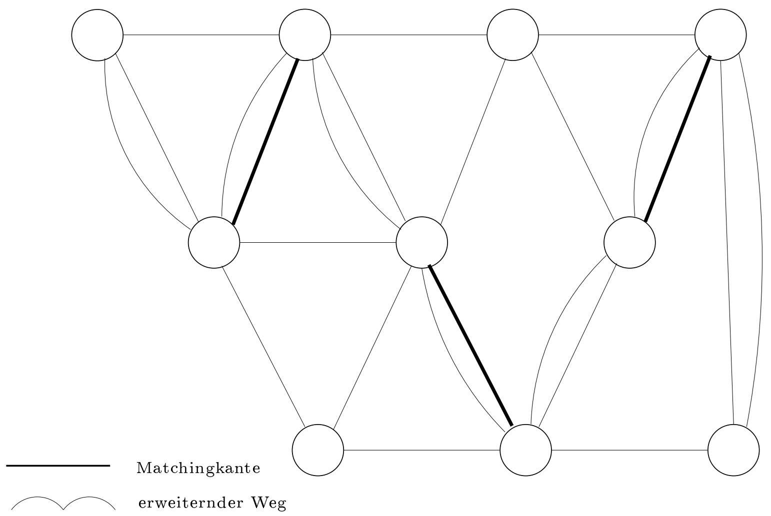
Sei $M$ nun ein Matching in $G = (V, E)$.

Ein Knoten $v$ heißt **belegt**, wenn er von einer Matchingkante berührt wird. Ist dies nicht der Fall, so heißt er **frei**.

## 10.3 Erweiternder Weg

Ein **erweiternder** Weg in $G$ bzgl. $M$ ist eine Folge von Kanten, die bei einem freien Knoten beginnt, bei einem anderen freien Knoten endet und abwechselnd Matching- und Nichtmatchingkanten benutzt.

### Beispiel:




Es ist einleuchtend, daß sich ein Matching $M$ entlang eines solchen erweiternden Weges um eine Kante erweitern läßt, indem man jede Matchingkante zu einer Nichtmatchingkante macht und umgekehrt. Dabei gilt der folgende Satz:

$$M \text{ ist Maximum Matching} \iff \text{Es gibt keinen erweiternden Weg bzgl. } M.$$

### Beweis:

$\Rightarrow$: trivial!

$\Leftarrow$: Es gibt keinen erweiternden Weg bzgl. $M$.

*Annahme:* Es gibt $M'$ mit $|M'| > |M|$.

Betracht nun $M$ und $M'$ in $G$ und entferne alle Kanten, die sowohl in $M$ als auch in $M'$ enthalten sind. Da $|M'| > |M|$, gibt es eine Folge von Kanten aus dem Rest von $G$, die abwechselnd in $M'$ und $M$ liegen, wobei diese Folge mit einer Kante aus $M'$ beginnt und aufhört.
Die Endpunkte dieser Folge von Kanten sind also frei bzgl. $M$. Somit ist diese Folge von Kanten ein erweiternder Weg bzgl. $M$. Dies ist ein Widerspruch zur Voraussetzung

Aus dieser Eigenschaft folgt ein neuer Ansatz zur Lösung des Problems, ein Maximum Matching zu bestimmen:

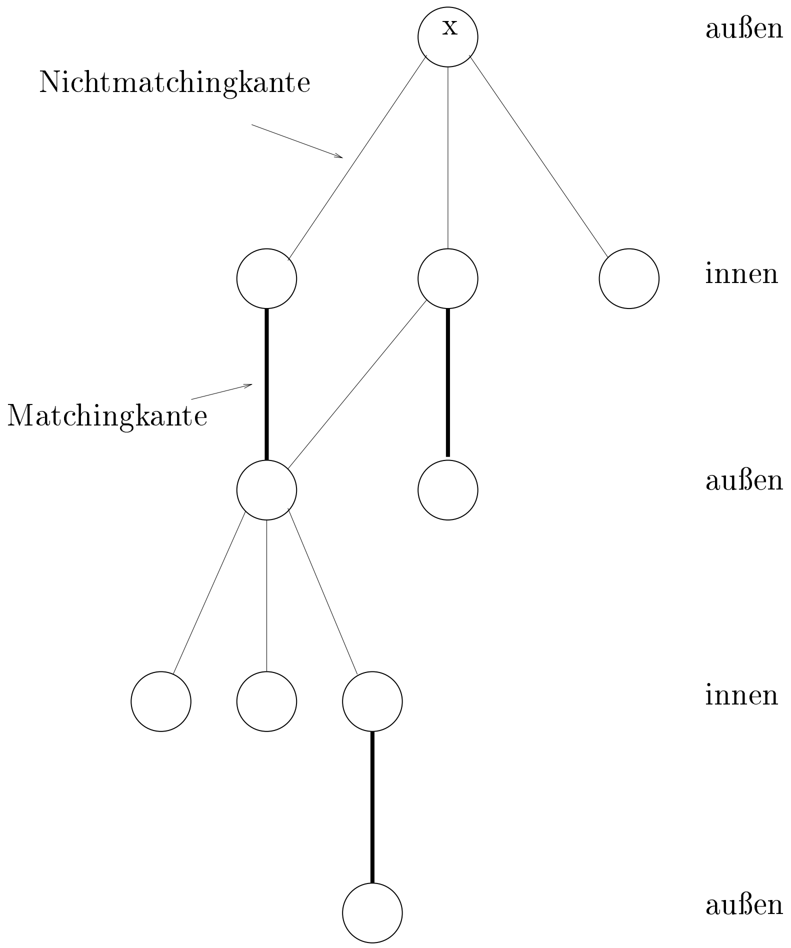
```
M := ∅;
REPEAT
   Finde einen erweiternden Weg bzgl. M;
   Falls gefunden, erweitere M längs dieses Weges;
UNTIL kein erweiternder Weg vorhanden;
```

Ein noch ungeklärtes Problem bei diesem Algorithmus ist die Frage, wie man einen erweiternden Weg bzgl. eines Matchings findet. Die Antwort liegt in der Konstruktion eines **(erweiternden) alternierenden Baumes**:

## 10.4 (Erweiternder) alternierender Baum

Die Wurzel dieses alternierenden Baumes bildet einer der freien Knoten $x$ aus $G$. Alle Wege von $x$ ausgehend sind alternierend, d.h. sie benutzen abwechselnd Nichtmatching- bzw. Matchingkanten. Die Wurzel heißt außen, die Nachfolger von außen nennen wir innen, deren Nachfolger wiederum außen usw.

**Beispiel:**




Zu Beginn gilt nun:

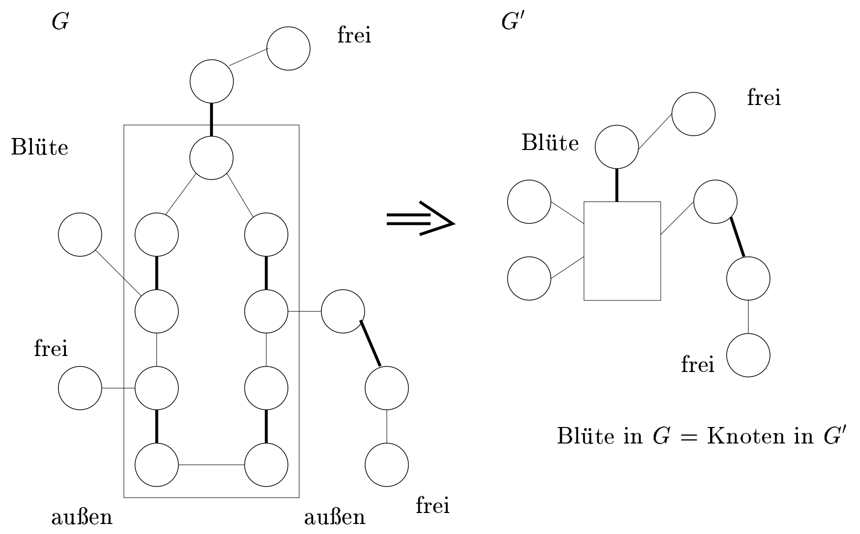
- Alle Kanten sind ungefärbt.
- Alle Knoten sind unmarkiert.
- Die Wurzel sei als außen markiert.

Wähle nun eine ungefärbte Kante $(x, y)$ inzident zu einem "außen"-Knoten $x$. Falls das nicht möglich ist, so gibt es keinen erweiternden Weg mehr zu finden. Gibt es jedoch eine solche Kante, so unterscheiden wir in zwei Fälle:

- $y$ ist markiert: Färbe $(x, y)$ blau, d.h. als nicht im Baum enthalten
- $y$ ist markiert: Färbe $(x, y)$ rot, d.h. als im Baum enthalten.
  - Falls $y$ frei ist, haben wir einen erweiternden Weg gefunden.
  - Falls $y$ belegt ist, färbe die eindeutig best. Matchingkante $(y, z)$ rot, markiere $y$ "innen" und $z$ "außen".

Durch diese Konstruktion ist es also möglich, erweiternde Wege in einem Graph $G$ zu finden.

Sei $G$ nun **nicht bipartit**.




Blüte (blossom) = Kreis ungerader Länge

Kante von "außen" nach "außen" ergibt **Blüte**. (Im bipartiten Graph nicht möglich.)

Es gilt folgender Satz:

$G'$ hat erweiternden Weg $\iff$ $G$ hat erweiternden Weg.

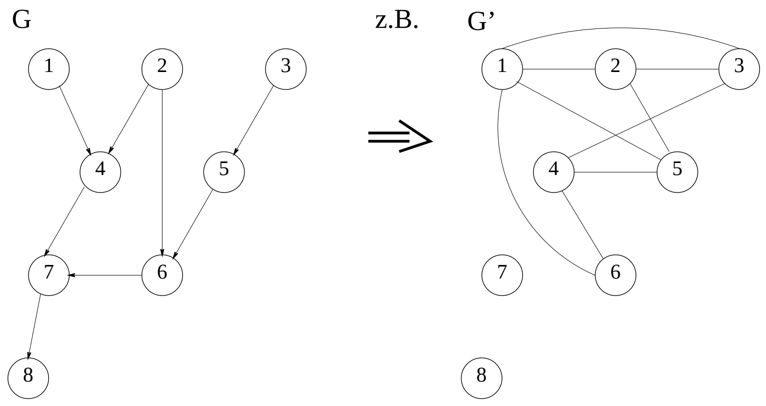
## 10.5 2-Prozessor Scheduling

Der Maximum-Matching-Algorithmus kann beim **2-Prozessor Scheduling** zur Anwendung kommen:

Gegeben: $G = (V, E)$ gerichtet $G$ beschreibt Vorrang-Relation zwischen Tasks

Gesucht: Schedule für 2 Prozessoren

z.B.




bedeutet: $i$ muß vor $j$ bearbeitet werden.

Kante $(x, y)$ in $G'$ besagt, daß $x$ und $y$ gleichzeitig bearbeitet werden können.

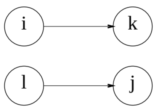
Bilde $G' = (V, E')$ ungerichtet wie folgt:

$(x, y) \in E' \iff x$ ist nicht Vorgänger von $y$ und $y$ ist nicht Vorgänger von $x$.

**Satz:**

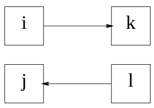
Kardinalität eines Maximum-Matchings in $G'$ = Zahl der gleichzeitig bearbeitbaren Taskpaare.

Falls folgende Situation

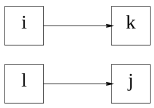


auftritt und zwischen $i$ und $j$ bzw. $k$ und $l$ keine Nachfolgebeziehungen bestehen, dann kann auch zwischen $i$ und $l$ bzw. $k$ und $j$ keine Nachfolgebeziehung bestehen, d.h., der Konflikt der auftritt, wenn $i$ parallel zu $j$ und $k$ parallel zu $l$ bearbeitet wird, kann dadurch beseitigt werden, indem man $i$ parallel zu $l$ und $k$ parallel zu $j$ bearbeitet.

D.h. ersetze



durch



.

## 10.6 Vertexcover

### Definition:

Sei $G = (V, E)$ ungerichtet.

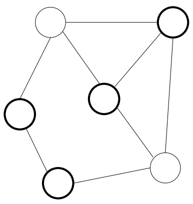
Für ein **Vertexcover** $K \subseteq V$ gilt: $(i, j) \in E \Rightarrow i \in K$ oder $j \in K$.

Ein Vertexcover heißt **optimal** $\iff |K|$ minimal.

(Vertexcover = Knotenüberdeckung)

### Beispiele:

i) $K = V$

ii)  hier $|K| = 4$ (optimal).

In bipartiten Graphen läßt sich aus einem Maximum-Matching ein minimales Vertexcover konstruieren.

Es gilt: Sei $M$ ein Matching, $K$ ein Vertexcover $\Rightarrow |M| \leq |K|$.

Konstruktion: Siehe unten.

## 10.7 Independent Set

### Definition:

Sei $G = (V, E)$ ungerichtet.

Für einen **Independent Set** $I \subseteq V$ gilt: $i, j \in I \Rightarrow (i, j) \notin E$.

### Behauptung:

Sei $K \subseteq V$ ein Vertexcover $\iff \overline{K}$ ist ein Independent Set.

### Beweis:

*Annahme:* $\exists\, x, y \in \overline{K}$ mit $(x, y) \in E \Rightarrow x$ oder $y$ im Vertexcover. Widerspruch!

**Analog zeigt man:** Sei $I \subseteq V$ ein Independent Set $\Rightarrow \overline{I}$ ist ein Vertexcover.

$\Rightarrow$ Kleinstes Vertexcover liefert größtes Independent Set (und umgekehrt).

### Beispiel:

Eine Gruppe von Jungen und Mädchen möchte eine Party feiern. Einige der Jungen haben mit einigen der Mädchen Streit. Wie muß die Gruppe zusammengestellt werden, damit möglichst viele an der Party teilnehmen, es aber zu keinem Streit kommt? Lösung: Suche max. Independent Set $\Rightarrow$ Ruhe auf Party.

Lösungsweg: Bestimme max. Independent Set mittels minimalem Vertexcover. Bestimme minimales Vertexcover mittels Maximum Matching.

**Konstruktion** eines minimalen Vertexcovers (in einem bipartiten Graphen):

Sei $G = (V, E)$ bipartit ($V = V_1 \cup V_2$ (disjunkt), wobei gilt: $(x, y) \in E \Rightarrow x \in V_1, y \in V_2$.) $M \subseteq E$ sei Maximum Matching.

Von jeder Matchingkante aus führt maximal ein alternierender Weg zu einem freien Knoten.

(i) Falls von einer Matchingkante aus ein alternierender Weg zu einem freien Knoten führt, dann bewerte den Endknoten der Matchingkante, der näher am Ende des alternierenden Weges (Ende = freier Knoten) liegt, mit '+' und den anderen Endpunkt mit '-'. Alle freien Knoten werden mit '-' bewertet. Auf einem alternierenden Weg, beginnend beim freien Knoten, folgt also auf ein '-' immer ein '+' und auf ein '+' immer ein '-'.

(ii) Bei allen Matchingkanten $m = (x, y)$, von denen kein alternierender Weg zu einem freien Knoten führt, bewerte $x$ mit '+' und $y$ mit '-'.

### Beobachtung:

Jede Matchingkante hat genau ein '+'.

### Behauptung:

Jede Nichtmatchingkante $k = (x, y)$ hat mindestens ein '+'.

### Beweis:

(i) $k$ liegt innerhalb eines alternierenden Weges zu einem freien Knoten. (trivial)

(ii) $k$ liegt nicht innerhalb eines alternierenden Weges zu einem freien Knoten.

Beh.: Beide zugehörigen Knoten $x$ und $y$ sind gematcht.

Bew.: *Annahme:* Die Behauptung ist falsch.

O.B.d.A. sei $x$ nicht gematcht, d.h. $x$ ist ein freier Knoten. Falls $y$ ebenfalls nicht gematcht ist, dann ist $M$ erweiterbar. Also muß $y$ gematcht sein und $k$ liegt somit innerhalb eines alternierenden Weges zu einem freien Knoten ($x$). (Widerspruch !)

$(x, w)$ und $(z, y)$ seien die beiden Matchingkanten. Von keiner dieser beiden Matchingkanten führt ein alternierende $r$ Weg zu einem freien Knoten, da sonst $k$ in einem alternierenden Weg zu einem freien Knoten liegen würde.

$x$ und $z$ werden also mit '+', $w$ und $y$ mit '-' bewertet.

Jede Kante in $G$ hat also mindestens ein '+'.

Wähle als Vertexcover $K$ alle mit '+' bewerteten Knoten.

Es gilt dann: $|M| = |K|$ und wegen $|M| \leq |K'|$ für alle Vertexcover $K'$ folgt: $K$ ist minimal!

## 10.8 Perfektes Matching

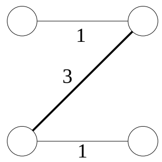
### Definition:

Ein Matching heißt **perfekt** $\iff |M| = \frac{|V|}{2}$ (d.h., jeder Knoten ist gematcht).

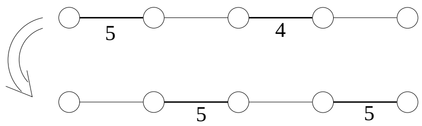
Sei $G = (V, E)$ ungerichtet, bipartit, gewichtet mit $c : E \to \mathbb{Z}$.

(z.B. $c(i, j) =$ Glücksgefühl bei Paarung Mann $i$ / Frau $j$)

Gesucht: Maximum cost matching z.B.:




Idee: Suche alternierende Wege, die die Gesamtkosten erhöhen.




Variante: Suche perfektes Matching mit minimum cost.

Problem: Zuordnungsgewicht zwischen $i$ und $j$ wird von $i$ und $j$ unterschiedlich gesehen.

Antwort: Ranking (d.h. statt absoluter Werte eine Reihenfolge $1, 2, \ldots$ festlegen, also eine Beliebtheitsskala bzw. Präferenzliste erstellen)

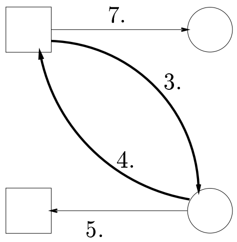
## 10.9 Heiratsproblem

Gegeben: $n$ Männer, $n$ Frauen
Jeder Mann hat Präferenzliste mit Frauen.
Jede Frau hat Präferenzliste mit Männern.

### Definition:

Eine Heirat (= Paarung aller) heißt **instabil**, falls es zwei Leute gibt, die nicht miteinander verheiratet sind, sich aber jeweils ihrem Partner vorziehen.

z.B.:




Eine Heirat heißt **stabil**, falls sie nicht instabil ist.

Gesucht: Stabile Heirat.

### Algorithmus

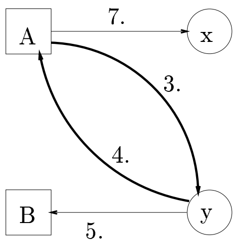
Ein unverheirateter Mann $x$ arbeitet seine Präferenzliste der Reihe nach ab, bis er zu einer unverheirateten Frau kommt (und sie heiratet) oder zu einer verheirateten Frau $y$ kommt, die $x$ ihrem jetzigen Partner $z$ vorzieht (Ehe $(y, z)$ wird geschieden, $x$ heiratet $y$ und $z$ arbeitet seine Liste weiter ab).

1. Alle werden verheiratet,

   denn für jeden Freier ex. mindestens eine unverheiratete Frau im Rest seiner Liste.

2. Instabile Heirat nicht möglich.

   *Annahme:* Doch! z.B.:

   Als $A$ um $y$ buhlte, wurde er entweder sofort oder später abgewiesen, da $y$ eine aus ihrer Sicht bessere Wahl hatte. Widerspruch!

3. Laufzeit:

   Offenbar gilt: Jeder Mann arbeitet seine Liste ab.
   Vorhanden: Matrix $prior[x, j] = p$ (Person $p$ ist $j$-te Wahl von $x$)
   Überführen in: $rang[x, p] = j$
   $O(m)$ bei Eingabelänge $m$
   (Aber quadratisch gegenüber $n =$ Anzahl der Männer bzw. Frauen)

Der Algorithmus ist “Mann-optimal”, d.h., es ex. keine andere stabile Heirat, bei der die Männer besser abschneiden.

# Kapitel 11

# Traveling Salesman

Gegeben: $G = (V, E)$ bewertet mit $c : E \to \mathbb{Z}$.

Gesucht: Permutation der Knoten $x_0, x_1, \ldots, x_{n-1} \in V$, so daß $c(x_i, x_{(i+1) \bmod n})$ minimal. (Billigster Rundweg (Hamiltonkreis))

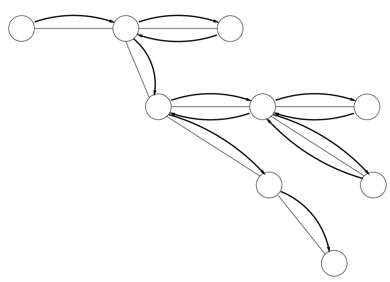
### Hamiltonkreis

Gibt es eine Permutation (wie oben) mit endlichen Kantenkosten. ($c(x_i, x_j) = \infty$, falls $(i, j) \notin E$)

### Exakte Lösungen für das TSP

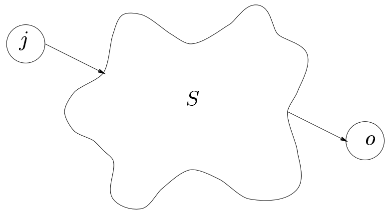
1. Alle Permutationen testen. (Aufwand: O (n!) )

2. **Backtracking** Skizze:



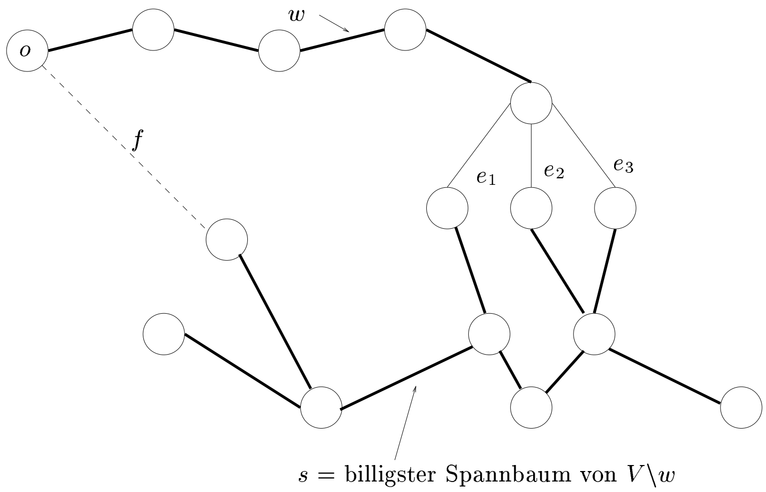
   Datenstruktur: Keller (Aufwand bei vollbesetztem Graph: $O(n!)$)

3. **Dynamic Programming** Sei $S \subset V$, $i \in V$. $g(i, S)$ = Länge des kürzesten Weges, der von $i$ aus über genau **alle** Knoten von $S$ läuft und beim Knoten $o$ endet.




$g(i, \emptyset) := c(i, o)$
$g(i, S) := \min_{j \in S}\{g(j, S\backslash\{j\}) + c(i, j)\}, (i \notin S)$
Lösung des TSP: $g(o, V\backslash\{o\})$
Platzbedarf (Speicher): $\binom{n}{1} + \binom{n}{2} + \ldots + \binom{n}{n} = O(2^n)$
Laufzeit: $O(2^n)$ (gute Verbesserung gegenüber $O(n!)$)

4. **Branch & Bound** Teillösung (Fixierung einiger Variablen) = Teilproblem $T =< w, b >$
   ($w$ = fixierter Weg, $b$ = untere Schranke)
   Jede Lösung, die $w$ enthält, kostet mindestens $b$.
   Aus T entstehen Teilprobleme durch Expansion. ($w_i = w \cup \{e_i\}, T_i =< w_i, b_i >$)
   Wir verwalten alle Teilprobleme in einem Heap.

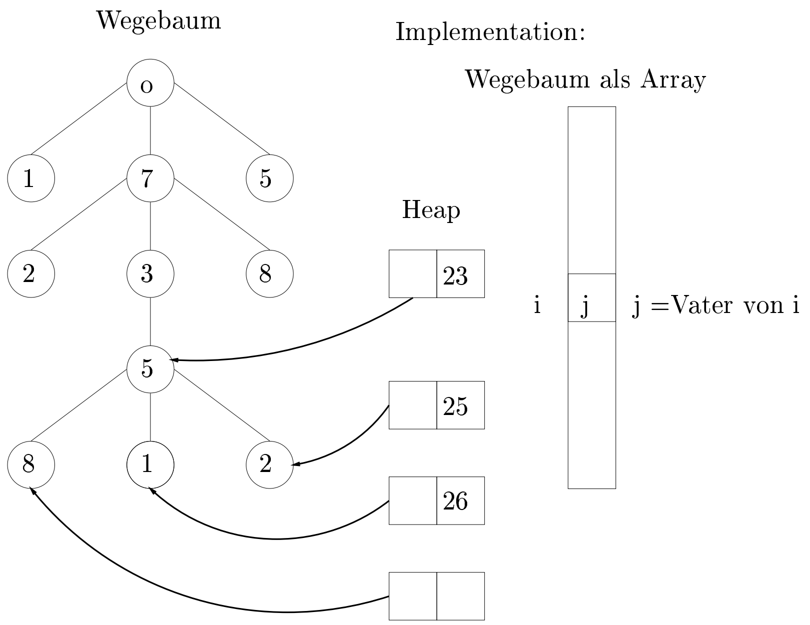



Eine Schranke $b_i$ für $T_i$ lautet: $c(w) + c(e_i) + c(S) + c(f)$ ($f$ = billigste Kante von $V\backslash w$ zum Knoten $o$).

**Algorithmus:**

```
Startprobleme <o, 0 > in Heap.
Sei L_0 eine Näherungslösung.
REPEAT
   Entferne billigste Teillösung T = <w,b > aus Heap.
   Expandiere w zu w_1, w_2, ... , w_k mit b_1, b_2, ... , b_k.
   Falls Lösung entsteht und besser als L_0, dann ersetzen.
   Sinnvolle Teilprobleme in den Heap tun.
UNTIL Heap enthält keine sinnvollen Probleme mehr.
L_0 ist Lösung.
```

(Teil-) Problem sinnvoll $\iff$ $b$-Wert ¡ bisher beste Lösung ($L_0$).




Laufzeit: exponentiell (z.B. $O((1,2)^n)$)

## Näherungslösungen für das TSP

Sei $P$ ein Minimierungsproblem und $A$ ein Näherungsalgorithmus für dieses Problem. Sei $I$ eine Instanz (z.B. Anzahl von Städten bei TSP). Sei $Apr(I)$ die (Näherungs-) Lösung, die $A$ für Instanz $I$ findet, $Opt(I)$ die optimale Lösung.

Gilt für alle $I$: $\frac{Apr(I)}{Opt(I)} \geq f \Rightarrow$ Fehlerschranke von $A$ ist $f$ ($f \leq 1$).

**Näherungslösungen für TSP:**

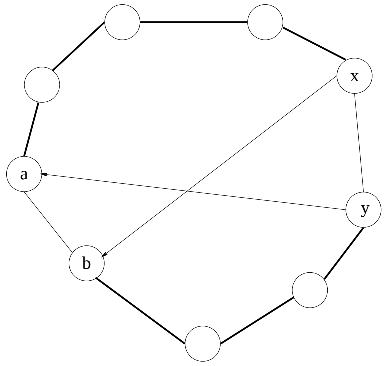
1. Nearest Neighbour (NN)

   Nimm jeweils billigsten Nachbarn. Für diesen Algorithmus ex. keine konstante Fehlerschranke, d.h., für jede Fehlerschranke ex. ein Graph, so daß NN diese Schranke nicht einhält.

2. $2 - opt$ für ungerichteten Graphen

   Bestimme Rundweg. Tausche Paare aus, falls

$$c(x,y) + c(a,b) > c(x,b) + c(a,y)$$



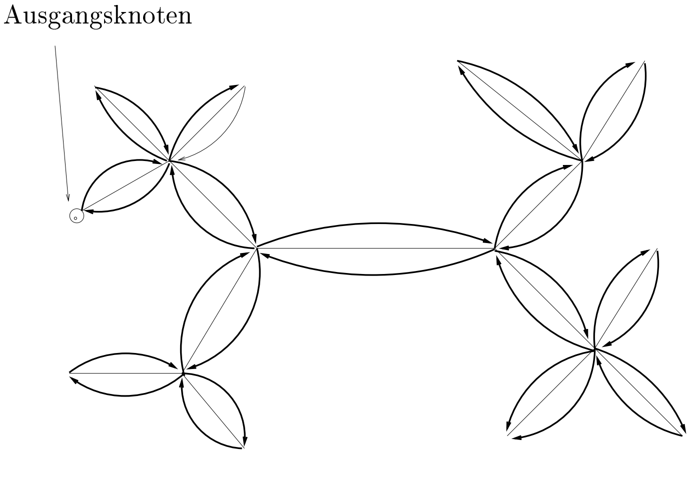

Keine konstante Fehlerschranke.

Im folgenden sei die Dreiecksungleichung erfüllt d.h. $c(x,z) \geq c(x,y) + c(y,z) \forall x,y,z \in V$. (Insbesondere gilt also: G ist voll besetzt.)

3. Spannbaum umlaufen

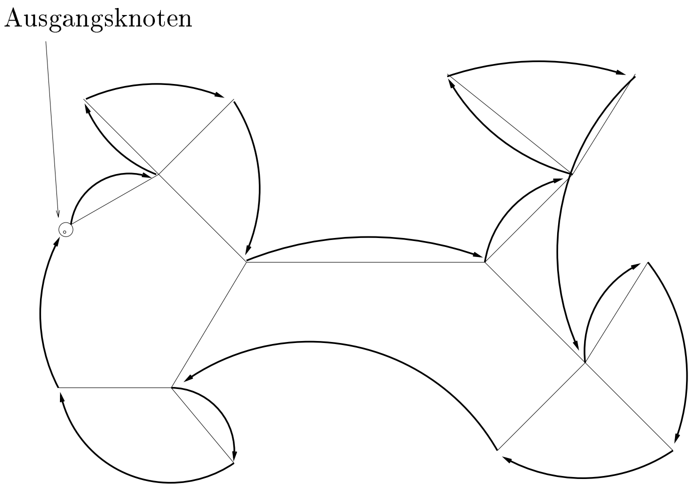
   (a) Minimum-Spanning-Tree (MST) bestimmen

   (b) Umlaufen des MST




$\Rightarrow$ Länge $= 2c(MST)$, da $c(MST) < c(TSP) \Rightarrow$ Umlauf $< 2c(TSP)$.

   (c) Transformiere Umlauf zu einer "TSP"-Rundreise, indem schon besuchte Knoten ausgelassen werden.




$\Rightarrow$ Fehlerschranke $f = 2$.

4. Verfahren nach Christofides

   (a) $MST$ bestimmen.

   (b) Sei $V' \subseteq V$.
   $V' = \{$Knoten aus MST mit ungeradem (Kanten-) Grad bzgl. $MST\}$.

   (c) Bestimme ein Minimum Cost Matching $M$ in $G' = (V', E)$.
   (Wegen $|V'|$ geradzahlig, ist das Matching perfekt.)

   (d) Betrachte $G'' = (V, MST \cup M)$
   Alle Knoten haben geraden Grad (in $G''$). $\Rightarrow$ Graph $G''$ ist eulersch (d.h. es ex. eine Eulertour in $G''$). Bestimme Eulertour $K$. Es gilt: $c(K) = c(MST) + c(M)$ $(c(MST) < c(TSP), c(M) < \frac{1}{2} \cdot c(TSP))$

   (e) Transformiere wie bei 3.) die Eulertour $K$ zu einer TSPRundreise. $\Rightarrow$ Fehlerschranke $f = \frac{3}{2}$.

Zur Begründung der weiter unten folgenden Vermutung benötigen wir den Begriff der NP-Vollständigkeit:

# Kapitel 12

# Begriff der NP-Vollständigkeit

**Definition:**

(Ja-Nein-) Problem $A$ ist in Polynomzeit reduzierbar auf Problem $B$ (kurz: $A \geq_P B$)

$$\Rightarrow \text{Es gibt } f \in \mathbf{P} : x \in A \iff f(x) \in B.$$

($\mathbf{P}$ = Menge aller in Polynomzeit berechenbaren Funktionen)

$NP := \{L$ : Zu gegebener Instanz $x$ und Beweisvorschlag $y$ kann in Polynomzeit überprüft werden, ob $y$ beweist: $x \in L\}$ ($L$ = Ja-Nein-Problem)

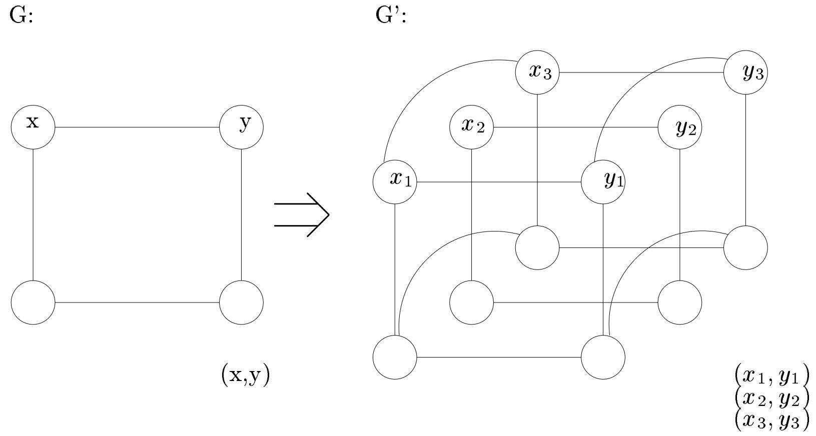
$L_0$ ist **NP-vollständig**, wenn

1. $L_0 \in \mathbf{NP}$
2. $\forall L \in +\mathbf{NP}$ gilt: $L \geq_P L_0$

Nachweis der **NP-Vollständigkeit**, über ein "erstes" **NP**-vollständigiges Problem $L_1$, für ein "neues" **NP**-vollständiges Problem $L_2$ durch Finden einer Reduktion $L_1 \geq_P L_2$. ("Erstes" **NP**-vollständigiges Problem $L_1$ fand Cook 1972: Satisfiability Problem) Beispiele für Reduktion:

**Beispiele für Reduktion:**

| | | | |
|---|---|---|---|
| 1.) | Independent Set | $\geq_P$ | Vertexcover |
| | | | (siehe Abschnitt Independent Set) |
| 2.) | 3-Färbbarkeit | $\geq_P$ | Independent Set |
| | | | |
| Als Ja-Nein-Problem formulieren: | Gibt es eine 3-Färbung? | $\geq_P$ | Gibt es ein Independent Set mit bestimmter Anzahl von Knoten? |




**Behauptung:**

$G = (V, E)$ ist 3-färbbar $\iff$ $G'$ hat Independent Set der Größe $|V|$.

**Beweis:**

$\Leftarrow$: Aus jeder "Dreier"-Gruppe gehört genau ein Knoten zum Independent Set $I$. Falls $i$-ter Knoten $\in I$, färbe entsprechenden Knoten in $G$ mit Farbe $i$ $(i = 1, 2, 3)$.

$\Rightarrow$: Füge jeweils $i$-ten Knoten aus jeder "Dreier"-Gruppe zu $I$ hinzu, wobei $i$ die Farbe des entsprechenden Knotens in $G$ ist.

**Vermutung:**

Es gibt keinen Polynomzeitalgorithmus mit konstanter Fehlerschranke $f$ für das $TSP$.

**Annahme:**

Doch, es existiert ein solcher Algorithmus. Sei $G = (V, E)$ mit $n$ Knoten, ungerichtet. Bilde $G' = (V, E')$ mit $c : E' \to \mathbb{N}$, $(E' = V \times V)$

$$c(i,j) := \begin{cases} 1, & \text{falls } (i,j) \in E \\ f \cdot n, & \text{falls } (i,j) \in E \end{cases}$$

Sei $A$ der Näherungsalgorithmus mit Schranke $f$. Wende $A$ auf $G'$ an.

1. $G$ hat einen Hamiltonkreis $\Rightarrow Opt(G') = n \Rightarrow Apr(G') \geq f \cdot n$
2. $G$ hat keinen Hamiltonkreis $\Rightarrow Opt(G') > f \cdot n \Rightarrow Apr(G') > f \cdot n$

   $\Rightarrow$ Ergebnis würde über die Existenz eines Hamiltonkreises in $G$ befinden.

Wäre die Annahme richtig, so wäre das Hamiltonkreisproblem in Polynomzeit lösbar. (*) (Transformation G $\to$ G' erfolgt in Polynomzeit) Das Hamiltonkreisproblem ist $NP$-vollständig. (*) $\Rightarrow P = NP$ (sehr unwahrscheinlich)

# Kapitel 13

# Näherungslösung für das Vertexcover Problem

Das Vertexcover-Problem (Gesucht: Minimales $VC$) läßt sich mit Fehlerschranke $f = 2$ lösen.

Finde maximales Matching.
Nimm Endpunkte jeder Matching-Kante.
$\Rightarrow f = 2$.

Sei $K$ ein Vertexcover für $G = (V, E) \Rightarrow V \backslash K$ ist Independent Set.

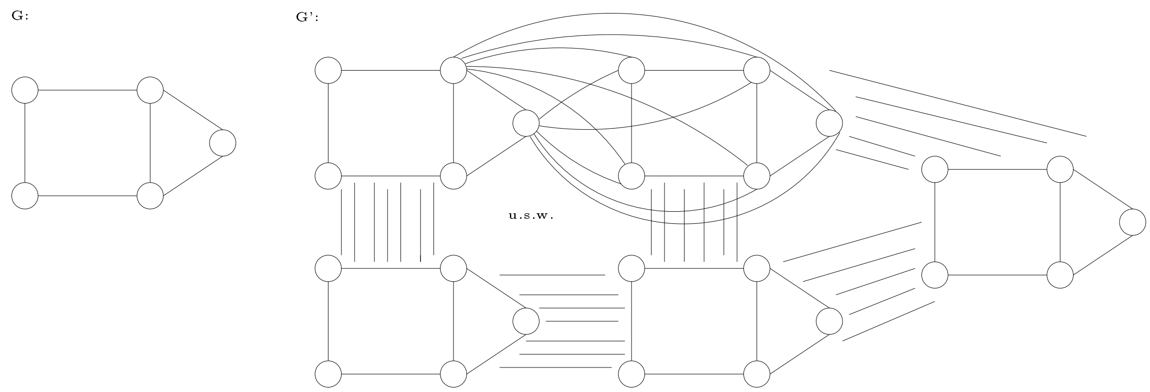
Der gleiche Näherungsalgorithmus wie oben liefert jedoch eine beliebig schlechte Fehlerschranke für das Independent Set-Problem.

z.B.: $n$ =100 Knoten, minimales Vertexcover sei 45 Knoten.
Näherung für Vertexcover ($VC$) liefert 90 Knoten.
$\Rightarrow$ Näherung für Independent Set hat 10 Knoten.
Opt. Independent Set = 55 Knoten $\Rightarrow$ Fehlerschranke $f > 5$!

**Satz:**

Falls das Independent Set Problem mit Fehlerschranke $f > 1$ gelöst werden kann, so kann es mit beliebiger Fehlerschranke $f' > 1$ gelöst werden.

Hierzu bilden wir aus einem Graphen $G$ einen Graphen $G'$, der $|V|$ Kopien von $G$ enthält, wobei Kopie $i$ mit Kopie $j$ genau dann paarweise verbunden ist, wenn es Kante $(i, j) \in E$ gibt.




$G$ hat ein Independent Set der Größe $k \iff G'$ hat ein Independent Set der Größe $k^2$. Näherungsalgorithmus findet Independent Set der Größe $\frac{k^2}{f}$ in $G'$. Daraus läßt sich in $G$ ein Independent Set der Größe $\sqrt{\frac{k^2}{f}} = \frac{k}{\sqrt{f}}$ finden. (Bisher wurde aber noch kein Näherungsalgorithmus mit konst. Fehlerschranke $f$ gefunden.)

# Kapitel 14

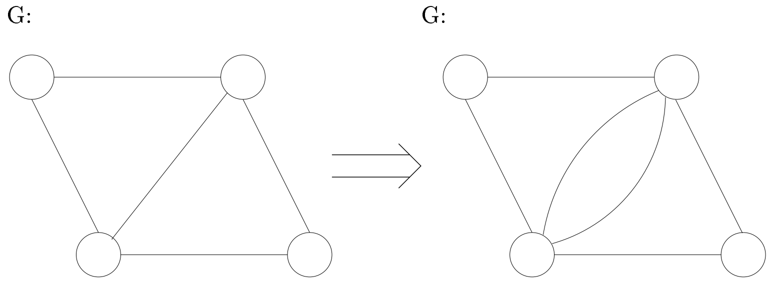
# Chinese Postman

Gegeben: $G = (V, E)$ ungerichtet, gewichtet mit $c : E \to \mathbb{N}$

Gesucht: Billigste Kantenrundreise, d.h. geschlossene Kantenfolge, in der jede Kante mindestens einmal auftritt.

i) Ist $G$ eulersch (d.h., jeder Knoten hat geraden Kantengrad), so liefert die Eulertour die billigste Postman-Tour.

ii) Ist $G$ nicht eulersch (d.h., es ex. Knoten mit ungeradem Kantengrad), so muß $G$ durch Verdoppelung einiger Kanten in einen Eulergraph $G'$ überführt werden. Die gesuchten Kanten bilden ein System von kantendisjunkten Wegen zwischen den Knoten ungeraden Grades in $V$.




Eine Kante in $G'$ wird maximal zweimal durchlaufen, sonst entferne zwei Kanten $\Rightarrow G'$ bleibt eulersch.

Seien $n_1, n2_, \ldots, n_k$ die Knoten mit ungeradem Kantengrad. ($k$ gerade, denn in einem Graph ist die Anzahl der Knoten mit ungeradem Kantengrad gerade)

Bilde Distanzmatrix $d_{ij}$ mit Floyd.
Bilde Clique mit $k$ Knoten und Kantengewicht $d_{ij}$.

In dieser Clique suchen wir nun das Minimum Cost Matching. Dieses Matching liefert uns die Knotenpaare, die gemäß Floyd verbunden werden. Wir erhalten also einen Eulergraph und somit auch die Postman-Tour.