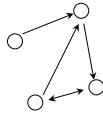
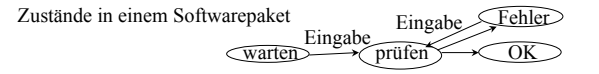
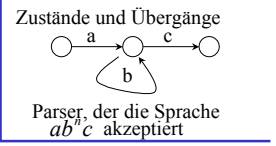
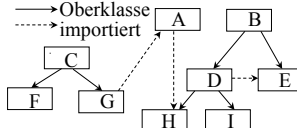
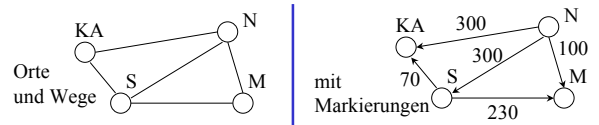


Über Graphen



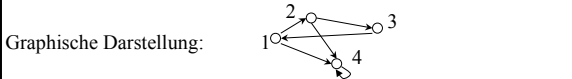
- Graphen haben:
 - Knoten (vertex)
 - Kanten (edge)
- Graphen drücken Beziehungen (Kanten) zwischen Dingen (Knoten) aus
- Kanten oft mit Markierungen versehen (beschreiben Beziehung)
- Kanten sind oft mit Richtungen versehen (gerichtete Graphen)
- Graphen sind meist nur eine von vielen sinnvollen Darstellungen
- Darstellung als Graph ist oft „menschfreundlicher“
- Graphen sind gut erforscht und bieten viele interessante Probleme

Beispiele für Graphen

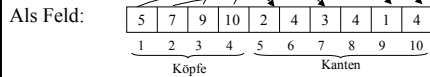
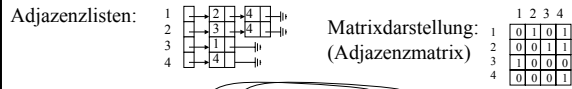


Darstellungen für Graphen

Graphen können auf verschiedene Weise im Rechner dargestellt werden:



Mengenschreibweise: $x \rightarrow y \Rightarrow (x,y) \in E$
 $V = \{1,2,3,4\}$ $E = \{(1,2), (1,4), (2,3), (2,4), (3,1), (4,4)\}$



Darstellungen für Graphen

Vergleich Adjazenzliste / Adjazenzmatrix:

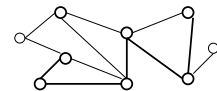
- Matrix: + sehr schnell feststellbar, ob eine Kante existiert
 + manche Graphenoperationen lassen sich als Matrizenoperationen darstellen
 - Speicherverschwendung bei dünnen Graphen
- Liste: + sehr kompakte Darstellung (wenig Speicher)
 - kostspielige Suche nach Kanten

Eigenschaften von Graphen

- gerichtet
- reflexive Kanten
- mehrfache Kanten (multigraph/simple)
- Spezialfälle:
 - Baum / Wald
 - zusammenhängend
 - endlich / unendlich

Definitionen

Untergraph:
oder Teilgraph:

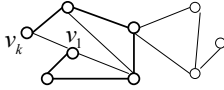


gegeben Graph mit Knotenmenge V , Kantenmenge E

ein Graph mit Knotenmenge V' und Kantenmenge E' ist Teilgraph wenn $V' \subseteq V$ und $E' \subseteq E$

Definitionen

Pfad:
oder *Weg:*
oder *Kantenzug:*



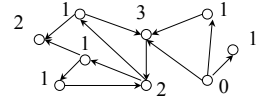
Ein Pfad von Knoten v_1 zu Knoten v_k ist eine Folge von Knoten v_1, v_2, \dots, v_k wobei $(v_1, v_2), \dots, (v_{k-1}, v_k)$ Kanten sind

auch Pfade können gerichtet oder ungerichtet sein

Definitionen

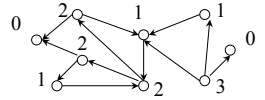
Eingangsgrad:

= Zahl ankommender Kanten



Ausgangsgrad:

= Zahl abgehender Kanten



bei ungerichteten Graphen ist

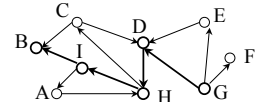
Ausgangsgrad = *Eingangsgrad* = *Grad*

Spezifikation von Graphen

- oft sind Graphen nicht explizit vorgegeben (insbesondere, wenn sie unendlich sind)
- oft werden Graphen definiert, durch Regeln, welche Kanten von einem gegebenen Knoten wohin führen
- z.B. bei Spielen:
Spielkonfiguration = Knoten, Spielzug = Kante,
Spielregeln geben an von wo welche Knoten erreichbar sind

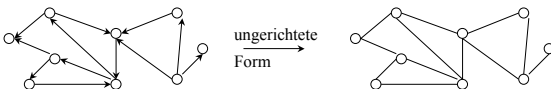
Definitionen

Erreichbarkeit:

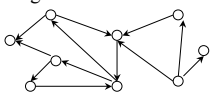


Knoten B ist *erreichbar* von Knoten G, wenn es einen Pfad von G nach B gibt

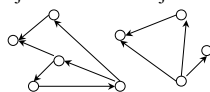
Definitionen



Ein Graph heißt *zusammenhängend*, wenn in seiner ungerichteten Form ein Pfad von jedem Knoten zu jedem existiert



zusammenhängend

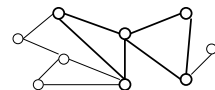


nicht zusammenhängend

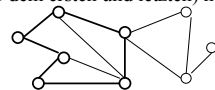
Problem: Herausfinden der einzelnen Komponenten

Definitionen

Ein Pfad heißt *Zyklus* (manchmal *geschlossener Pfad*) (*circuit*), wenn sein erster und letzter Knoten derselbe sind.



Ein Zyklus ist ein *einfacher Zyklus* (*cycle*, simple circuit), wenn jeder Knoten (außer dem ersten und letzten) nur einmal vorkommt:

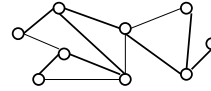


Bäume sind auch Graphen

- Bäume sind Graphen, die keine Zyklen enthalten.
- Graphen, die keine Zyklen enthalten heißen *Wald*
- zusammenhängende Graphen, die keine Zyklen enthalten heißen *Bäume*
- wenn ein gerichteter Graph ein Baum ist und genau einen Knoten (Wurzel) mit Eingangsgrad 0 hat, heißt er Baum mit *Wurzel*

Definitionen

Ein *Spannbaum* (*spannender Baum*) eines ungerichteten Graphen ist ein Untergraph dessen, und ist ein Baum, der alle seine Knoten enthält.

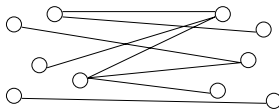


Problem 1: Wie findet man einen Spannbaum?

Problem 2: Randbedingungen: geg.: Graph mit gewichteten Kanten
ges.: Spannbaum mit minimaler Summe der Kantengewichte (minimal spannender Baum)

Definitionen

Ein *bipartiter* Graph ist ein Graph, dessen Knoten so in zwei Mengen zerteilt werden können, daß jede Kante je einen Knoten aus beiden Mengen verbindet:



Problem 1: Herausfinden ob ein Graph bipartit ist.
Problem 2: Welches sind die Partitionen?

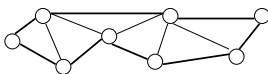
Typische Graphenalgorithmen

- Feststellen bestimmter Eigenschaften (z.B. zusammenhängend)
- Finden von Pfaden zwischen zwei Knoten
- Finden von Pfaden, die zusätzlichen Randbedingungen genügen
- in implizit definierten Pfaden: Finden von Knoten, die bestimmte Randbedingungen erfüllen
- Finden von Untergraphen, die bestimmte Randbedingungen erfüllen (z.B. minimal spannender Baum)

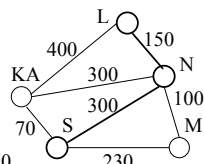
Also: Suchprobleme und Optimierungsprobleme

Typische Pfadsuchprobleme

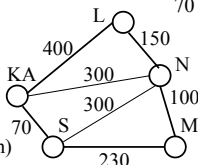
• Suche nach einem geschlossenen Pfad durch gegebene Knoten:



• Suche nach einem „billigen“ Pfad:

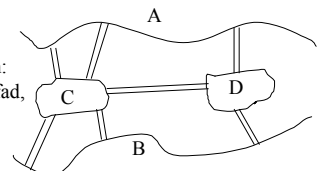


• oder beides:
(Problem des Handlungsreisenden)

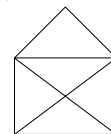


Finden eines Eulerschen Graphen

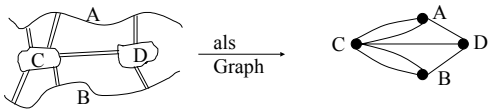
Königsberger Brückenproblem:
Gibt es einen geschlossenen Pfad, der über alle 7 Brücken führt?



Zeichenproblem:
Kann das Häuschen mit einem Strich gezeichnet werden?



Eulersche Pfade



Die Frage, ob ein Eulerscher Pfad existiert ist leicht zu beantworten.

Feststellung: Wenn man in einen Knoten kommt, muß man auf anderem Weg wieder herauskommen.

Also ist Bedingung für Existenz von Euler-Pfad:
Grad jedes Knotens muß durch 2 teilbar sein

Ist diese Bedingung auch hinreichend? Ja.

Der Ariadne-Algorithmus

Problem: gegeben ein Labyrinth,
gesucht ein Weg zum Ziel (und wieder zurück)

Lösung:

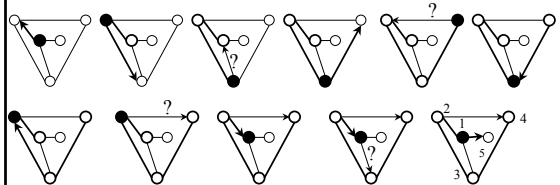
- Ariadne gibt Theseus ein Wollknäuel
- Theseus wickelt Knäuel ab und läuft durchs Labyrinth
- Theseus markiert jede Abzweigung die er nimmt
- Sackgasse oder Knäuel zu Ende
=> Knäuel aufwickeln, soweit bis unmarkierte Abzweigung
- wenn unmarkierte Abzweigung, dann hinein

Tiefensuche

- Ariadne-Algorithmus war die erste Realisierung einer Tiefensuche.
- Prinzip der Tiefensuche ist es, einen eingeschlagenen Weg so lange weiter zu verfolgen bis es nicht mehr geht / angebracht ist
- Erst wenn eine Richtung komplett durchsucht ist, wir die nächste Alternative in Angriff genommen

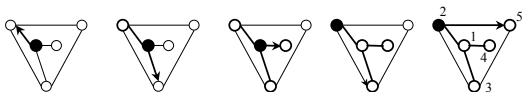
Algorithmus für Tiefensuche

PROZEDUR Tiefensuche (V)
markiere aktuellen Knoten V als besucht
für jede von V abgehende Kante (V,W)
wenn der Knoten W noch nicht markiert ist
Tiefensuche (W)



Algorithmus für Breitensuche

PROZEDUR Breitensuche(V)
markiere V als besucht
initialisiere Liste mit einem Element V
solange die Liste nicht leer ist
entferne erstes Element W aus der Liste
für alle Kanten (W,X)
falls X noch nicht markiert
hänge X hinten an die Liste an



Eigentliche Arbeit in der Tiefensuche

PROZEDUR Tiefensuche (V)
markiere aktuellen Knoten V als besucht
erledige Arbeit vor dem Abstieg
für jede von V abgehende Kante (V,W)
wenn der Knoten W noch nicht markiert ist
Tiefensuche (W)
erledige Arbeit während des Abstiegs
erledige Arbeit nach dem Abstieg

Welche Arbeiten zu erledigen sind,
hängt von der Anwendung ab.

Bemerkungen zur Tiefen-/Breitensuche

Wenn ein ungerichteter Graph zusammenhängend ist, wird sowohl mit Tiefen als auch mit Breitensuche garantiert jeder Knoten besucht (egal wo man anfängt).

Wenn ein Graph nicht zusammenhängend ist, muß der Suchalgorithmus erweitert werden zu:

PROZEDUR SucheAlles
solange ein nicht markierter Knoten V existiert
Suche(V)

Aufwand der Tiefen-/Breitensuche

Bei der Tiefensuche wird jede Kante genau einmal *in jeder Richtung* durchlaufen. Das heißt, der Aufwand beträgt mindestens $O(|E|)$.

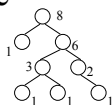
Da bei nicht zusammenhängenden Graphen u.U. jeder Knoten eine Komponente des Graphen darstellt, beträgt in diesem Fall der Aufwand mindestens $O(|V|)$.

Im allgemeinen also: $O(|E| + |V|)$

Breitensuche: Jeder Knoten wird einmal in die Schlange eingefügt, jede Kante einmal besucht => auch hier $O(|E| + |V|)$

Beispiel für Tiefensuche

Aufgabe: gegeben: ein Baum,
gesucht für jeden Knoten V : Anzahl der
Knoten des Unterbaums der V als Wurzel hat



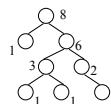
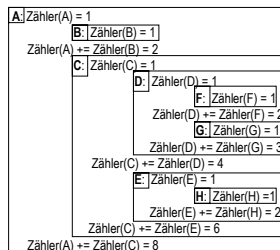
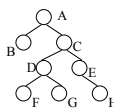
PROZEDUR Tiefensuche (V)

markiere aktuellen Knoten V als besucht
Arbeit vor dem Abstieg: **Zähler(V) = 1**
für jede von V abgehende Kante (V, W)
wenn der Knoten W noch nicht markiert ist
Tiefensuche (W)
Arbeit während des Abstiegs: **Zähler(V) += Zähler(W)**
Arbeit nach dem Abstieg: **nichts mehr**

Beispiel für Tiefensuche

PROZEDUR Tiefensuche (V)

markiere aktuellen Knoten V als besucht
Arbeit vor dem Abstieg: **Zähler(V) = 1**
für jede von V abgehende Kante (V, W)
wenn der Knoten W noch nicht markiert ist Tiefensuche (W)
Arbeit während des Abstiegs: **Zähler(V) += Zähler(W)**



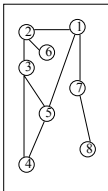
Knotennumerierung mit Tiefensuche

Die Tiefensuche (DFS) durchläuft die Knoten eines Graphen in einer bestimmten Reihenfolge.

Wenn jedem Knoten die Position in dieser Reihenfolge zugeordnet wird ist das eine DFS-Numerierung.

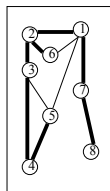
Der Algorithmus dazu: (Der Zähler Z wird vorher mit 1 initialisiert)

PROZEDUR Tiefensuche-Numerierung (V)
markiere aktuellen Knoten V als besucht
Arbeit vor dem Abstieg: **DFS-Numer(V) = Z++**
für jede von V abgehende Kante (V, W)
wenn der Knoten W noch nicht markiert ist Tiefensuche (W)



Erzeugung des Tiefensuchenbaums

Die durchlaufenen Kanten bei der Tiefensuche (DFS) bilden einen Baum (DFS-Baum):



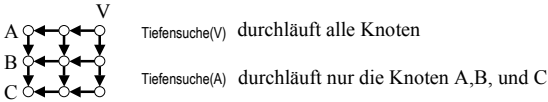
Der Algorithmus dazu:

PROZEDUR TiefensuchenBaum(V)
markiere aktuellen Knoten V als besucht
für jede von V abgehende Kante (V, W)
wenn der Knoten W noch nicht markiert ist
dann nimm Kante (V, W) in DFS-Baum auf
und Tiefensuche (W)

Tiefensuche in gerichteten Graphen

Prinzip ist gleich wie bei ungerichteten Graphen,

aber Problem: Welcher Teil des Graphen abgesucht wird, hängt vom Startknoten ab:

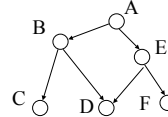


deshalb: **PROZEDUR Tiefensuche-gerichtet**
solange ein nicht markierter Knoten V existiert
Tiefensuche(V)

Über Knotenverwandtschaften

Bei gerichteten Graphen spricht man von verwandtschaftlichen Beziehungen zwischen Knoten (vgl. Stammbäume)

x ist *Kind* (Sohn, Nachfolger) von y , wenn $(y,x) \in E$
 x ist *Elter* (Vater, Vorgänger) von y , wenn $(x,y) \in E$
 x ist *Nachkomme* von y , wenn \exists Pfad von y nach x
 x ist *Vorfahr* von y , wenn \exists Pfad von x nach y

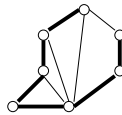
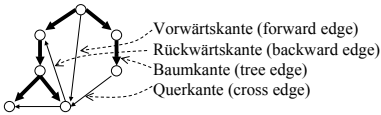


A ist Vorfahr von allen anderen Knoten
 C und D sind Kinder von B
 B und E sind Eltern von D
 F ist Nachkomme von A und von E

Verschiedene Kantenarten in Tiefensuchebäumen

DFS-Baum im gerichteten Graphen:

ungerichteter Fall:



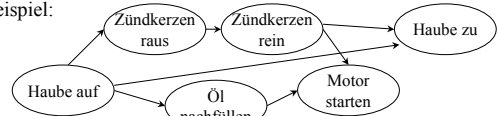
Baumkanten sind Teil des Tiefensuchensbaums,
Vorwärtskanten verbinden Knoten mit einem Nachkommen
Rückwärtskanten verbinden mit einem Vorfahr
Querkanten verbinden „nicht direkt verwandte“ Knoten

Bemerkung: keine Querkanten im ungerichteten Fall

Topologisches Sortieren

Problem: Mehrere Aufgaben, die nur in einer bestimmten Reihenfolge erledigt werden können (einzelne Abhängigkeiten)

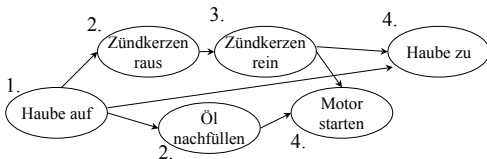
Beispiel:



Aufgabe: Erstelle Arbeitsplan, was wird wann gemacht?

Topologisches Sortieren

Eine mögliche Lösung:



gesucht ist also: Numerierung der Arbeitsschritte, so daß jeder Schritt eine Nummer bekommt, die größer ist als alle Schritte, die vorher ausgeführt werden sollten

Topologisches Sortieren

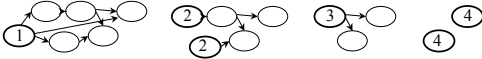
Offensichtlich: nur sinnvoll für azyklische endliche Graphen

Frage: Gibt es dann immer mindestens einen Anfangsknoten?

Antwort: Ja, jeder Knoten mit Eingangsgrad Null. Gäbe es keinen solchen, dann könnte man unendlich lange „rückwärts“ durch den Graphen gehen.

Topologisches Sortieren Der Algorithmus

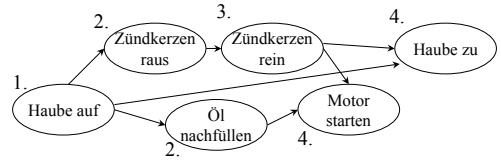
Zähler = 1
solange der Graph Knoten enthält
Markiere alle Knoten mit Eingangsgrad 0 mit Zähler
entferne diese Knoten
erhöhe Zähler um 1



Bemerkung: Entfernen von Knoten und Kanten aus einem azyklischen Graphen läßt den Graphen azyklisch

Topologisches Sortieren

Eine mögliche Lösung:



Gesucht ist also Numerierung der Arbeitsschritte, so daß jeder Schritt eine Nummer bekommt, die größer ist als alle Schritte, die vorher ausgeführt werden müssen.

Topologisches Sortieren

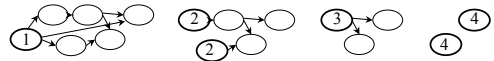
Offensichtlich nur sinnvoll für azyklische endliche Graphen.

Frage: Gibt es dann immer mindestens einen Anfangsknoten?

Antwort: Ja, jeder Knoten mit Eingangsgrad Null. Gäbe es keinen solchen, dann könnte man unendlich lange „rückwärts“ durch den Graphen gehen.

Topologisches Sortieren Der Algorithmus

Zähler = 1
solange der Graph Knoten enthält
markiere alle Knoten mit Eingangsgrad 0 mit Zähler
entferne diese Knoten
erhöhe Zähler um 1



Bemerkung: Entfernen von Knoten und Kanten aus einem azyklischen Graphen läßt den Graphen azyklisch.

Topologisches Sortieren Der Algorithmus

Zähler = 1
solange der Graph Knoten enthält
markiere **alle Knoten mit Eingangsgrad 0** mit Zähler
entferne diese Knoten
erhöhe Zähler um 1

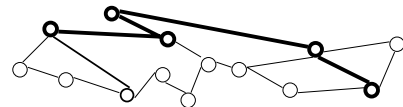
← welche sind das?
← was heißt das?

Zähler = 1
berechne **Eingangsgrad** für alle Knoten (z.B. mit Tiefensuche)
hänge alle Knoten mit **Eingangsgrad 0** an eine Schlange an
solange die Liste nicht leer ist
entferne ersten Knoten V aus der Schlange
markiere V mit Zähler und erhöhe Zähler um 1
für alle Kanten (V, W)
erniedrige **Eingangsgrad** von Knoten W um 1
falls **Eingangsgrad** == 0, hänge W an Schlange an

← hier allerdings jeder Knoten eigene Nummer

Ein Suchproblem

Gegeben: ein Graph, zwei Knoten V und W
Gesucht: der kürzeste (Anzahl Kanten) Pfad von V nach W




Verallgemeinerung:
Pfadlänge nicht durch Kantenzahl bestimmt,
sondern durch Summe der Kantenmarkierungen
(die alle positiv sein sollen)

Die Suche nach dem kürzesten Pfad

Verschiedene Variationen

- Gegeben: 2 Knoten v und w , gesucht: Pfad von v nach w
- Gegeben: Knoten v , gesucht: Pfade zu allen anderen Knoten
- Gegeben: Knoten w , gesucht: Pfade von allen anderen zu w
- Gegeben: Graph, gesucht alle kürzesten Pfade

Bemerkungen

- In gerichteten Graphen gilt natürlich nicht, daß der kürzest Pfad von v nach w der gleiche ist wie der von w nach v 
- Natürlich lassen sie die Variationen 2 bis 4 von oben durch mehrfaches Ausführen der ersten Variation lösen.

Suche nach dem kürzesten Pfad

Wir betrachten den allgemeinen Fall:

- gerichtete Graphen (falls ungerichtet, ersetze jede Kante durch zwei gerichtete)
- gewichtete Kanten (falls ungewichtet, betrachte jedes Kantengewicht als 1.0)

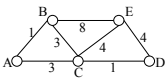
Zunächst Unterproblem:

Gesucht nur die Länge des kürzesten Pfades von V nach W

Dann:

Gesucht der Pfad selbst (die Folge von Knoten)

Länge des kürzesten Pfades



Gesucht: kürzester Weg von A nach E

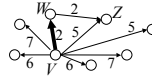
Überlegung:

- Geht Tiefensuche?
- Geht Breitensuche?
- Geht Tiefensuche mit Kantensortierung nach Länge?
- Geht Breitensuche mit Kantensortierung nach Länge?

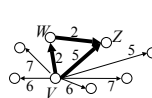
Länge des kürzesten Pfades

Überlegung:

Betrachten wir Knoten V und alle seine abgehenden Kanten.



Offensichtlich: Wenn (V,W) die kürzeste Kante ist, die von V abgeht, ist (V,W) der kürzeste Pfad von V nach W



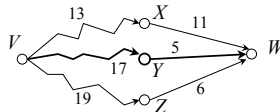
Betrachte zweitkürzeste Kante von V : (V,Z)
Entweder ist (V,Z) der kürzeste Pfad von V nach Z oder (V,W,Z) ist noch kürzer, andere Pfade sind garantiert nicht kürzer.

Länge des kürzesten Pfades

Induktionsannahme:

Wir kennen alle kürzesten Pfade von V zu allen Knoten, von denen man direkt zu W kommen kann

Dann ist der kürzeste Weg von V nach W gegeben durch die kürzeste Summe der Wege von V nach Y und der Länge der Kante (Y,W) .



Algorithmus an der Stelle:

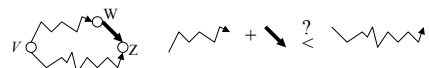
- $V.. X + (X,W) = 24$
- $V.. Y + (Y,W) = 22$
- $V.. Z + (Z,W) = 25$

also merken

Länge des kürzesten Pfades Der Algorithmus

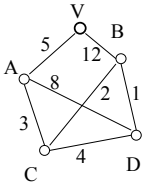
markiere alle Knoten außer V als unbesucht
setze $Länge(V) = 0$
setze $Länge(\text{alle anderen Knoten}) = \text{unendlich}$

solange nicht besuchte Knoten existieren
wähle darunter denjenigen Knoten W , für den $Länge(W)$ minimal
markiere W als besucht
für alle Kanten (W,Z) zu unmarkierten Knoten Z
falls $Länge(W) + Gewicht(W,Z) < Länge(Z)$
dann setze $Länge(Z) = Länge(W) + Gewicht(W,Z)$



Länge des kürzesten Pfades Der Algorithmus

Beispielgraph:

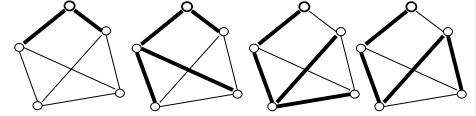
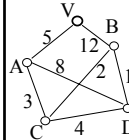


Länge(V)=0, Länge(A)=unendl., ... Länge(D)=unendl.
 Länge(A)=5, Länge(B)=12
 minimal ist Länge(A)=5 => V,A markiert
 Länge(A)+(A,C) = 5+3 < unendl. => Länge(C)=8
 Länge(A)+(A,D) = 5+3 < unendl. => Länge(D)=13
 minimal ist Länge(C)=8 => V,A,C markiert
 Länge(C)+(C,B) = 8+2 < 12 => Länge(B)=10
 Länge(C)+(C,D) = 8+4 < 13 => Länge(D)=12
 minimal ist Länge(B)=10 => V,A,C,B markiert
 Länge(B)+(B,D) = 10+1 < 12 => Länge(D)=11
 minimal ist Länge(D)=11 => alles markiert

Suche nach dem kürzesten Pfad

Gelöst: Länge des kürzesten Pfades,
Zweiter Teil des Problems: Was ist nun der kürzeste Pfad?

Feststellung: Der Algorithmus sucht immer nur Pfade zu noch nicht markierten Knoten
=> die abgesuchten Wege bilden einen Baum mit eindeutigem Weg von V zu jedem Knoten:



Baum der kürzesten Pfade

Minimale spannende Bäume

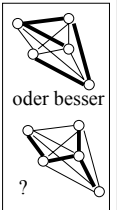
Problem:

- Gegeben ist eine Landkarte als Graph G .
- Die Post muß alle Ortschaften versorgen.
- Die Briefe werden hierarchisch zusammengefaßt.
- In jedem Ort kommt ein großes Paket an und mehrere kleine werden in die Nachbarorte verschickt.
- Die Gesamtkosten sind proportional zur Summe der Längen aller vorkommenden Verbindungswege.
- Welche Transportwege (Teilgraph von G) sollten gewählt, werden um die Kosten zu minimieren?

Minimale spannende Bäume

Problem:

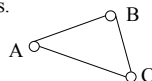
- Gegeben ist eine Landkarte als Graph G .
- Es soll ein Wasserleitungsnetz aufgebaut werden, das alle Ortschaften versorgt.
- Die Gesamtkosten sind proportional zur Summe der Längen aller vorkommenden Leitungen.
- Welche Ortschaften sollten verbunden werden (Teilgraph von G), so daß die Kosten minimal werden.



Ähnliche Probleme gibt es beim Entwurf von Rechnernetzen

Minimale spannende Bäume

Offensichtlich: Der gesuchte Teilgraph der zu wählenden Wege ist azyklisch, denn gäbe es einen Zyklus.



dann könnte man zumindest eine Kante entfernen, (also Kosten verringern) und trotzdem alles erreichen.

Also ist der gesuchte Teilgraph ein Baum.
Da er alle Knoten enthält, ist er ein Spannbaum.

Berechnung eines minimalen Spannbaums

1. Überlegung: Wir wissen, daß der gesuchte Baum die kürzeste Kante des Graphen beinhalten muß.

(Wenn sie (—) nicht dabei wäre und hinzugefügt würde, entstünde ein Zyklus, der durch Entfernen einer längeren Kante (---) aufgelöst werden könnte.)



Also Induktion: Beginne Spannbaum mit kürzester Kante, dann füge zweitkürzeste hinzu etc.

Geht das?

Berechnung eines minimalen Spannbaums

1. Überlegung geht nicht, Gegenbeispiel:



Problem ist, daß die Kante mit Länge 4 einen Zyklus erzeugt.

Frage also: Wie muß die Induktion verändert werden, damit ein korrekter Spannbaum entsteht?

Wir erinnern uns an die Suche nach dem kürzesten Pfad.

Berechnung eines minimalen Spannbaums

- Der Algorithmus für die Suche nach dem kürzesten Pfad lieferte einen Spannbaum.
- In jedem Schritt wurde eine Kante so hinzugefügt, daß der entstehende Pfad minimale Länge hatte.
- Jetzt interessiert nicht die Länge des gesamten Pfades, sondern nur die der neu hinzugenommenen Kante.

Minimale spannende Bäume: Der Algorithmus

wähle V angrenzend an kürzeste Kante
markiere alle Knoten außer V als unbesucht;
setze $Kosten(\text{alle Knoten}) = \text{unendlich}$
für alle Kanten (V, W)

setze $Kosten(W) = \text{Gewicht}(V, W)$

setze $Kante(W) = (V, W)$

solange nicht besuchte Knoten existieren

wähle darunter denjenigen Knoten W , für den $Kosten(W)$ minimal
markiere W als besucht

füge Kante(W) zum Spannbaum hinzu

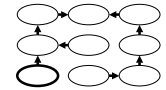
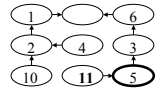
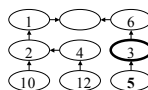
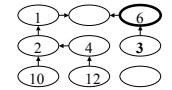
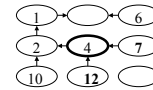
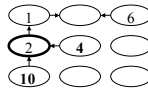
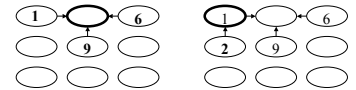
für alle Kanten (W, Z) zu unmarkierten Knoten Z

falls $\text{Gewicht}(W, Z) < \text{Kosten}(Z)$

dann setze $Kosten(Z) = \text{Gewicht}(W, Z)$

und setze $Kante(Z) = (W, Z)$

Beispiel für minimalen Spannbaum



Kruskal's Algorithmus

Für ungerichtete Graphen bietet sich der Kruskal-Algorithmus an:

Idee: In jedem Schritt wird eine Kante hinzugefügt

Induktion: Zu jedem Zeitpunkt haben wir einen Wald minimal spannender Bäume

Induktionsanfang: Jeder Knoten ist für sich ohne Kanten ein minimal spannender Baum

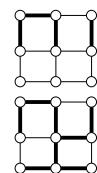
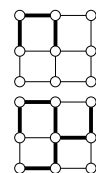
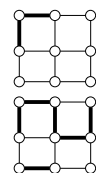
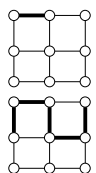
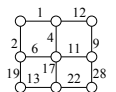
Induktionsschritt: Vereinige zwei Bäume zu einem, so daß der seine Knoten auch minimal aufspannt



Kruskal's Algorithmus

der Graph habe n Knoten
für i von 1 bis $n-1$

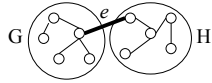
füge die kürzeste noch nicht hinzugefügte Kante zum Spannbaum hinzu, die keinen Zyklus entstehen läßt



Beweis für Kruskal's Algorithmus

Induktionsanfang: trivial

Induktionsschritt:



Nach Induktion gilt, daß der Teilgraph G ein minimaler Spannbaum ist und ebenso Teilgraph H

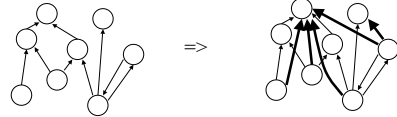
e ist die kürzeste Kante, die G und H verbindet
 $G \cup H \cup e$ ist auch minimal, weil $G \cup H \cup f$ nicht besser sein kann, und eine „Umordnung“ von G oder H diese „länger“ macht

Würde man G und H mit mehr als einer Kante verbinden, würde ein Zyklus entstehen, außer man entfernt dafür andere (kürzere) Kanten => Summe wieder länger

Die transitive Hülle

Problem: Auf einem Rechner gibt es viele Benutzer. Einige Benutzer erlauben anderen Benutzern den Zugriff auf ihr Account. Wer Zugriff auf ein Account X hat, hat auch Zugriff auf alle Accounts, auf die X Zugriff hat.

Sei die direkte Zugriffrelation als Graph G gegeben. Gesucht ist ein anderer Graph (transitive Hülle von G), in dem genau dann eine Kante von X nach Y enthalten ist, wenn X Zugriff auf Y hat (auch indirekt).

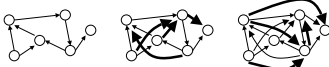


Die transitive Hülle

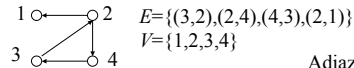
Ein erster einfacher Algorithmus:

PROZEDUR TransitiveHülle
 wenn der Graph n Knoten hat, dann mache $n-1$ mal
 für jede Kante (x,y)
 für jede Kante (y,z)
 füge Kante (x,z) hinzu

Der Algorithmus berechnet „Abkürzungen“ von x nach z .
 Da der längste mögliche Pfad höchstens $\#$ Knoten-1 Kanten hat, genügt es, $n-1$ mal alle Abkürzungen von zwei Schritten zu einem zu berechnen.



Die transitive Hülle



1			
	1		1
		1	
			1

Adjazenzmatrix:

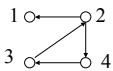
Nach Definition der Adjazenzmatrix $A[i,j] = 1 \iff (i,j) \in E$

$$A^2[i,j] = \sum_k A[i,k] \cdot A[k,j]$$

$A^2[i,j]$ ist genau dann nicht 0, wenn es einen Knoten k mit zwei Kanten (i,k) und (k,j) gibt. $A^2[i,j]$ ist also eine Abkürzung von i nach j in einem statt in zwei Schritten.

Entsprechend folgt $A^n[i,j]$ ist eine Abkürzung von i nach j in einem statt in n Schritten.

Die transitive Hülle



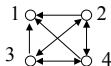
Adjazenzmatrix:

1			
	1		1
		1	
			1

$$A^1 = \begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & & 1 \end{bmatrix} \quad A^2 = \begin{bmatrix} 1 & & & \\ & 1 & & 1 \\ & & 1 & \\ & & & 1 \end{bmatrix} \quad A^3 = \begin{bmatrix} 1 & & & \\ & 1 & & 1 \\ & & 1 & \\ & & & 1 \end{bmatrix} \quad A^4 = \begin{bmatrix} 1 & & & 1 \\ & 1 & & 1 \\ & & 1 & \\ & & & 1 \end{bmatrix} = A^1$$

$$\sum_{i=1..3} A^i = \begin{bmatrix} 1 & & & \\ 1 & 1 & & 1 \\ 1 & 1 & 1 & \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

Also ist die transitive Hülle:



Transitive Hülle, verschiedene Algorithmen

PROZEDUR TransitiveHülle
 wenn der Graph n Knoten hat, dann mache $n-1$ mal
 für jede Kante (x,y)
 für jede Kante (y,z)
 füge Kante (x,z) hinzu

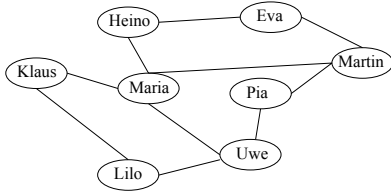
Oder, wenn A die Adjazenzmatrix und $H = \sum_{i=1..n} A^i$ ist, ist $H[i,j]$ genau dann nicht 0, wenn (i,j) in der transativen Hülle ist.

Ein weiterer Algorithmus ist in Manber S. 216

Matching

Motivation:

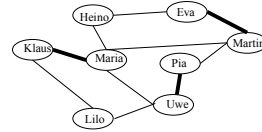
Wir befinden uns im Tanzkurs, jeder Teilnehmer ist ein Knoten, und jeder weiß, mit wem er gerne tanzt (Kante).



Aufgabe: Bestimme mögliche Paarungen.

Matching

Problem:



Drei Paare sind gefunden, aber nicht jeder Knoten hat einen Partner, und es sind keine weiteren Paarungen möglich.

Frage: Wie kriegt man eine optimale Paarbildung zustande?

Matching

Definitionen: Zwei Kanten (u,v) und (x,y) heißen *unabhängig*, wenn u,v,x,y vier verschiedene Knoten sind,

Wenn $u=x$, $u=y$, $v=x$ oder $v=y$, dann heißen die Kanten *benachbart* (oder verbunden oder adjazent)

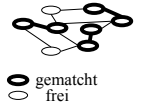
Definitionen: Eine Kantenmenge M heißt *unabhängig*, wenn alle ihre Elemente paarweise unabhängig sind. Solche Kantenmengen heißen auch *Matching*.

Bemerkung: Für ein Matching gilt, daß der Grad jedes Knotens kleiner 2 ist (gdw.).

Matching

Noch mehr Definitionen:

- Ein Knoten heißt *frei* bzgl. eines Matchings, wenn er Teil keine Kante des Matchings hat, sonst heißt er *gematcht*.



- Ein Matching heißt *perfekt*, wenn es alle Knoten des Graphen überdeckt.



- Ein Matching heißt *maximal* (nicht maximum) wenn es um keine Kante erweitert werden kann.



Matching

Noch mehr Definitionen:

- Ein Matching heißt *maximal* (nicht maximum) wenn es um keine Kante erweitert werden kann.



- Ein Matching heißt *maximum* Matching wenn es kein Matching mit mehr Kanten gibt:



ist nicht perfekt

- Ein Matching bei dem nur ein Knoten frei bleibt, heißt *fast perfekt*.

Matching

Ein paar Gedanken:

- Ein „gerader Kreis“ hat genau 2 perfekte Matchings:



- Nicht jeder Graph hat ein (fast) perfektes Matching:



Matching

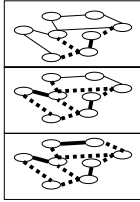
Ein erster Algorithmus: gegeben Graph G , gesucht Matching M

Solange eine unmarkierte Kante (u,v) in G existiert
 markiere (u,v)
 markiere alle benachbarten Kanten
 übernehme (u,v) nach M

Das ist ein „greedy“ Algorithmus, wie der von Kruskal.

Was liefert der Algorithmus?

- ein maximales Matching
- aber kein (fast) perfektes Matching



Matching (ein Spezialfall)

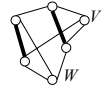
Betrachten wir einen Graphen mit $2n$ Knoten, von denen jeder einen Grad von mindestens n hat:



Können wir für so einen Graphen ein perfektes Matching finden?

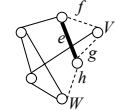
Algorithmus wie vorhin, erweitert um folgendes:

Wenn keine Kante mehr „geht“ und Matching nicht perfekt, wähle Knoten V, W nicht gematcht:



=> es gibt $2n$ Kanten von V oder W benachbart zu $<n$ gematchten Kanten

=> es gibt eine Kante e aus dem Matching, die benachbart ist zu drei Kanten (f,g,h) von V oder W :

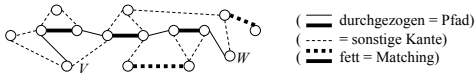


Dann ersetze e durch je eine Kante zu V und eine zu W :



Matching für beliebige Graphen

Definition: Ein bzgl. eines Matchings M alternierender Pfad beginnt im nicht gematchten Knoten V und endet im nicht gematchten Knoten W , wobei jede zweite Kante zum Matching gehört:

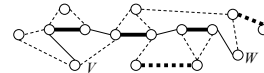


Satz von Berge:

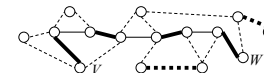
Ein Matching M in einem Graphen G ist maximum gdw. G keinen bzgl. M alternierenden Pfad enthält.

Matching, Satz von Berge

Beweis „=>“: Angenommen, M ist maximum Matching und es gibt einen alternierenden Pfad von V nach W , z.B.:



Dann vertausche Matchingzugehörigkeit aller Kanten auf dem Pfad.



Das ist immer noch ein Matching, jetzt ein größeres als M , Widerspruch!

Matching, Satz von Berge:

Beweis „<=<“: Sei M' ein maximum Matching.

Betrachte Teilgraphen T mit Kanten, die entweder in M oder in M' sind (nicht aber in beiden).

Nach Definition von Matching ist der Grad jedes Knotens in T höchstens 2, d.h. T besteht

nur aus M - M' -alternierenden Pfaden

oder aus M - M' -alternierenden Zyklen

jeder alternierende Zyklus hat eine gerade Anzahl Knoten (also gleich viele aus M und M'), und wenn M nicht maximum, also $|M| < |M'|$, gibt es einen alternierenden Pfad mit mehr Knoten aus M' als aus M .

Matching, ein Algorithmus

Aus dem Satz von Berge kann man einen Algorithmus ableiten:

Benutze irgend einen einfachen Algorithmus um ein maximales Matching M zu finden
 solange ein M -alternierender Pfad vorhanden ist
 vertausche die M -Zugehörigkeit der Kanten auf diesem Pfad

Der Algorithmus fügt in jedem Schritt eine Kante zu M hinzu:

Da es nur endlich viele Kanten gibt, terminiert er.

Wenn er terminiert hat er ein maximum Matching gefunden.

Noch offen: Wie findet man M -alternierende Pfade?

Bipartites Matching

Warum bipartite Graphen?

- einfacheres Finden von alternierenden Pfaden
- praktische Relevanz:

viele Zuordnungsprobleme ordnen Dinge verschiedener Arten einander zu, z.B.

- Männer / Frauen im Tanzkurs
- Arbeiten / Arbeitskräfte
- Koffer / Schließfächer u.s.w.

Bipartite Graphen

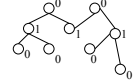
Satz von König: Graph bipartit \Leftrightarrow keine ungeraden Zyklen

Beweisskizze: Markiere V mit 0
solange es eine Kante (X, Y) gibt mit X markiert und Y nicht markiere Y mit 1-Markierung(X)

Interessant: Alle Hyperwürfel sind bipartit.



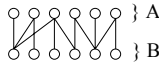
Alle Bäume sind bipartit.



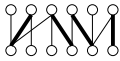
Bipartite Graphen

Finden eines alternierenden Pfades

Das geht einfach in bipartiten Graphen G mit zwei Klassen A und B .



Sei ein Matching gegeben:

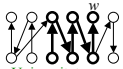


Definiere $G' =$



so daß alle Matching-Kanten von A nach B und alle anderen von B nach A gerichtet sind

Starte Tiefensuche (oder Breitensuche) in ungematchten Knoten v :

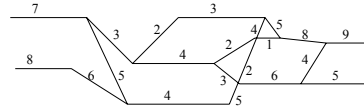


Jeder Pfad wechselt zwischen M' und $G \setminus M'$.
Fertig, sobald ein ungematchter Knoten w erreicht

Netzwerkflüsse

Motivation: Wir haben eine Kanalisation, die wie ein Graph aufgebaut ist.

Jeder Kanal hat eine Kapazität (m^3 / s), z.B.:



Frage: Wieviel kann pro Sekunde maximal von links nach rechts fließen?

Andere Frage: Wie ändert sich mein Duschwasser, wenn im Erdgeschoß jemand die Wanne volllaufen läßt?

Netzwerkflüsse

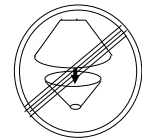
Definitionen

- Ein *Netzwerk* ist ein gerichteter Graph und eine Funktion, die jeder seiner Kanten eine *Kapazität* zuordnet, und zwei ausgezeichneten Knoten: „Quelle“ und „Senke“.
- Die *Quelle* eines Netzwerks ist ein Knoten mit Eingangsgrad 0.
- Die *Senke* eines Netzwerks ist ein Knoten mit Ausgangsgrad 0.
- Ein *Fluß* ist eine Funktion, die jeder Kante eines Netzwerks eine Last (lokaler Fluß) zuordnet, die einigen Bedingungen genügt.
- Der *Wert* eines Flusses ist die Summe aller lokalen Flüsse aus der Quelle (=Summe aller lokalen Flüsse in die Senke)
- Der Fluß mit maximalem Wert heißt *maximaler Fluß*

Netzwerkflüsse Bedingungen an einen Fluß

Sei $c(e)$ die Kapazität der Kante e , und $f: e \rightarrow \mathbb{R}^+$ ein Fluß, dann muß für f gelten:

1. (Zulässigkeit) $0 \leq f(e) \leq c(e)$



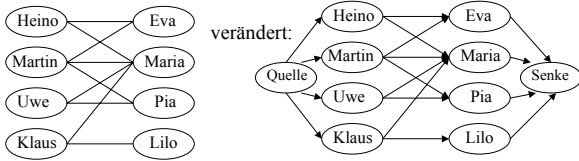
2. (Flußerhaltung) $\sum_u f(u, v) = \sum_w f(v, w)$



Netzwerkflüsse und bipartites Matching

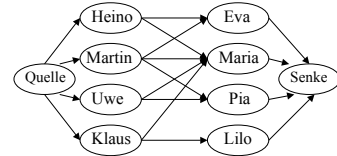
Was hat die Berechnung von Netzwerkflüssen mit Matching zu tun?

Hier nochmal der Tanzkursgraph:



Jetzt entspricht maximale Zahl von Matches einem maximalen Fluß von der Quelle zur Senke.

Netzwerkflüsse und bipartites Matching



Es gilt: Da alle Kanten die Kapazität 1.0 haben, kann durch jede Person nur der Fluß 1.0 fließen.

Der maximale Fluß ordnet genau den Kanten des maximum Matchings (und denen von Quelle und Senke) 1.0 zu, sonst 0.0.

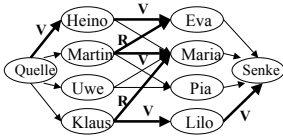
Netzwerkflüsse

Definitionen:

Ein *augmentierender* Pfad bzgl. eines Flusses f ist ein Pfad von der Quelle zur Senke ohne Berücksichtigung der Kantenrichtung, wobei für jede Kante (V, W) gilt:

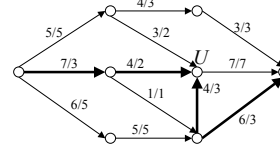
entweder $(V, W) \in E$ und $f(V, W) < c(V, W)$ (**Vorwärtskante**)

oder $(W, V) \in E$ und $f(V, W) > 0$ (**Rückwärtskante**)



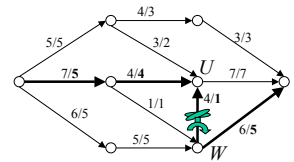
Netzwerkflüsse, augmentierende Pfade

Beispiel für Flußoptimierung mit einem augmentierenden Pfad:

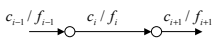


Auf dem augmentierenden Pfad wäre bis zum Knoten U noch Platz für 2 (Restkapazität).

Wenn wir den Hahn von W nach U etwas zudrehen, fließt insgesamt 2 mehr.

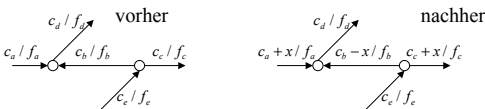


Optimierung am augmentierenden Pfad



nur Vorwärtskanten: erhöhe Fluß um minimale Restkapazität (slack) $\min(c_k - f_k)$ über alle Pfadkanten k

Wenn Rückwärtskanten vorkommen, dann:



Hier gilt $x = \min(c_v - f_v)$ über alle Vorwärtskanten v ,

oder $x = \min c_r$ über alle Rückwärtskanten r (das kleinere).

Finden eines augmentierenden Pfades

zum Beispiel so:

markiere Quelle s
wiederhole
wenn (v, w) existiert mit: v markiert und $f(v, w) < c(v, w)$
markiere w
wenn (w, v) existiert mit: v markiert und $f(w, v) > 0$
markiere w
solange in der Schleife neue Knoten markiert werden

Dieser Algorithmus markiert genau die Knoten, die von der Quelle aus mit augmentierenden Pfaden erreichbar sind. Ist am Ende auch die Senke markiert, haben wir einen augmentierenden Pfad durch das Netzwerk gefunden.

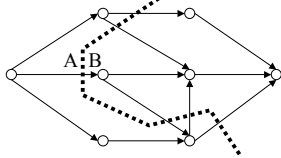
Also Algorithmen für optimalen Fluß:

setze $f(e) = 0$ für alle Kanten e
solange ein augmentierender Pfad von Quelle zu Senke existiert
optimiere diesen Pfad

Netzwerkflüsse: Schnitte

Definitionen:

Eine Unterteilung eines Netzwerks in eine Knotenmenge A und eine Knotenmenge B nennen wir einen **Schnitt**.



Die **Kapazität** $c(A,B)$ eines Schnitts A/B ist die Summe der Kapazitäten aller Kanten von A nach B

Bemerkung:

Der Wert eines Flusses (der Gesamtfluß) ist nie größer als die Kapazität eines beliebigen Schnittes (irgendwie muß es ja durch).

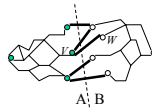
D.h. $w(f) \leq \min_{\text{Schnitt } AB} c(A,B)$ Beweis: \curvearrowright

Das Max-Flow-Min-Cut-Theorem

Satz: $w(f)$, der Wert von f , ist maximal
 \Leftrightarrow es gibt keinen augm. Pfad von Quelle zu Senke
 $\Leftrightarrow w(f) = \min_{\text{Schnitt } AB} c(A,B)$

Beweis „ \Rightarrow “: durch angeben der Optimierungsregel

Beweis „ \Leftarrow “: Definiere Schnitt A/B, so daß A = alle Knoten, die von Quelle aus mit augm. Pfad erreichbar sind.



Für alle $V \in A, W \in B$ gilt $f(V,W) = c(V,W)$, da sonst W auch mit augm. Pfad erreichbar.

Also ist $w(f) = \underbrace{c(A,B)}_{\text{max flow}} = \underbrace{c(A,B)}_{\text{min cut}}$

Das Integral-Flow-Theorem

Satz: Wenn in einem Netzwerk alle Kapazitäten ganzzahlige Werte sind, dann ist der maximale Fluß auch ganzzahlig.

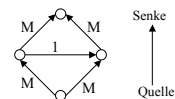
Beweis: Verwende Algorithmus von vorhin. Am Anfang ist der Fluß 0 (ganzzahlig)

In jedem Schritt wird er um die Restkapazität eines augmentierenden Pfades erhöht.

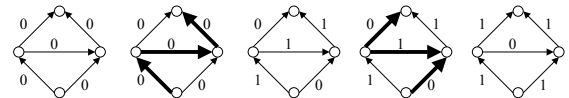
Da alter Fluß ganzzahlig und Kapazität ganzzahlig, ist auch Restkapazität ganzzahlig und neuer Fluß auch.

Ein paar Gedanken zum Aufwand

Betrachte folgendes Beispiel:
 offensichtlich: maximaler Fluß=2M

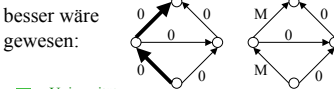


Was macht unser Algorithmus?



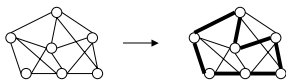
Der Gesamtfluß wird in jedem Schritt nur um 1 erhöht. Laufzeit also 2M.

besser wäre gewesen: Edmonds, Karp: „Nimm kürzesten augm. Pfad.“

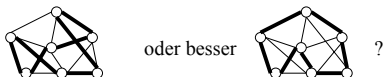


Hamiltonsche Pfade

Motivation: Gegeben ist eine Landkarte mit Orten und Verbindungen.
 Gesucht ist ein Rundgang einmal durch jeden Ort.



Verschärfung: Jede Verbindung ist mit Kosten gewichtet. Gesucht ist jetzt der billigste Rundgang.



Hamiltonsche Pfade

Ein Hamiltonscher Pfad ist ein einfacher Zyklus, der jeden Knoten eines Graphen enthält.

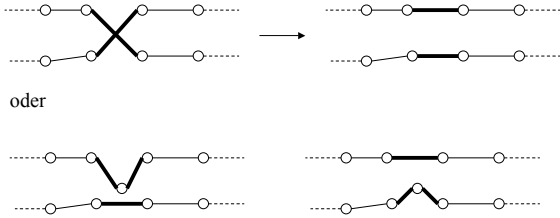
Ein Algorithmus für das Finden eines Hamiltonschen Graphen ist relativ einfach (Tip: Tiefensuche modifizieren) aber sehr aufwendig.

Bis heute kein Algorithmus bekannt, der eine Lösung in polynomialer Zeit findet.

Genauso schwierig: kürzester Hamiltonscher Pfad, auch Problem des Handlungsreisenden (Traveling Salesman Problem, TSP) genannt.

Das Problem des Handlungsreisenden

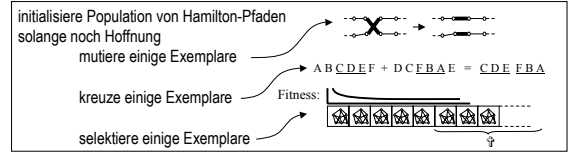
Mögliche Suche nach billigstem Pfad
z.B. durch sukzessives Optimieren:



Das Problem des Handlungsreisenden

Standardalgorithmen haben extrem hohe Laufzeit
(so daß für große TSP oft kein Minimum gefunden werden kann).

Daher verwendet man erfolgreicher alternative Algorithmen,
z.B. sog. genetische Algorithmen.



Der Einfachheit halber nehmen wir alle Knoten als miteinander
verbunden an aber manche Kanten haben extrem hohe Kosten (evtl. ∞).