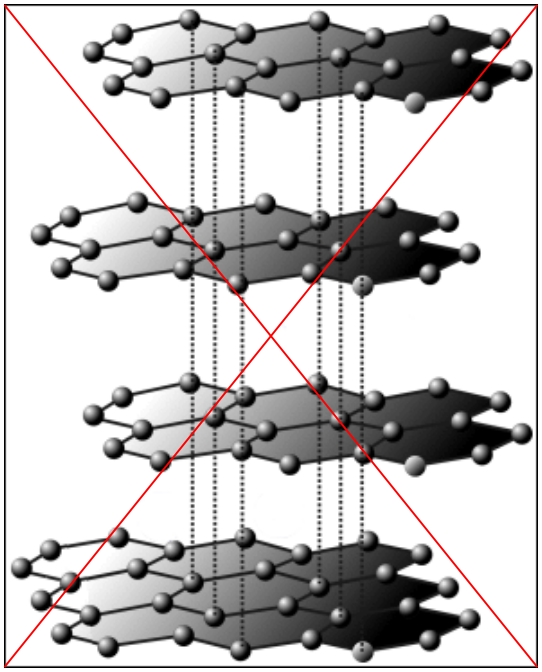
Alexander Nimmervoll - (A/B)
Eugen Kaltenegger - (C/D)

Algorithmen und Datenstrukturen
Sommersemester 2019

Agenda

- Graphen (Einführung)

Graphen



Graphen - Motivation

Graphen sind überall:

- Netzpläne
- Flugpläne
- Aufgabenlisten
- Fertigungsstraßen
- Elektronische Schaltungen

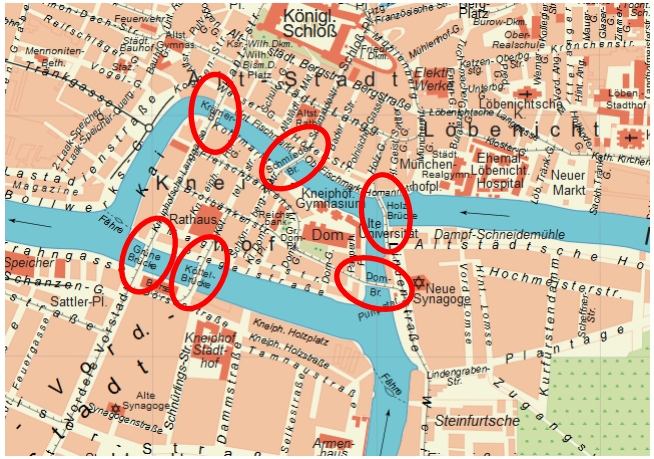
Diese Dinge können mathematisch als Graphen modelliert werden!

Graphen - Anwendungsbeispiel

Das Königsberger* Brückenproblem:

Gibt es einen Rundweg durch Königsberg, der jede der sieben Brücken genau einmal überquert?

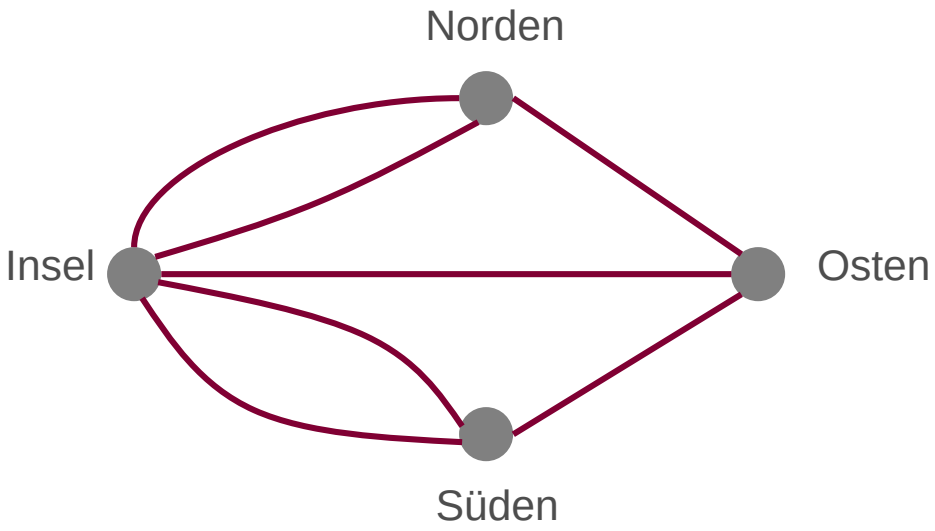
Formuliert durch den Mathematiker und Physiker Leonard Euler im achtzehnten Jahrhundert



* Königsberg war zu diesem Zeitpunkt die Hauptstadt von Preußen

Graphen - Anwendungsbeispiel

Keyword der Informatik: **Abstraktion!**

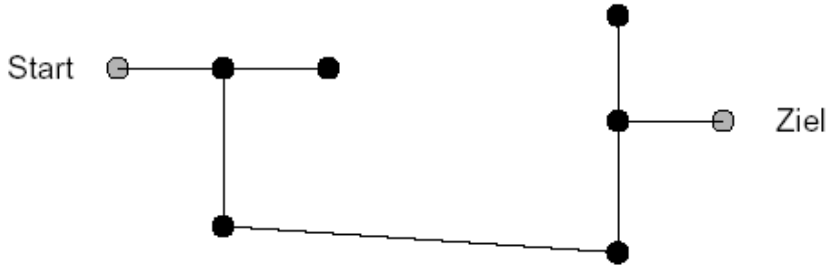


Gibt es einen Rundweg der jede Linie (Kante) genau einmal enthält?

Graphen - Anwendungsbeispiel

Labyrinth:

Gesucht ist ein Weg vom Start zum Ziel des Labyrinthes.



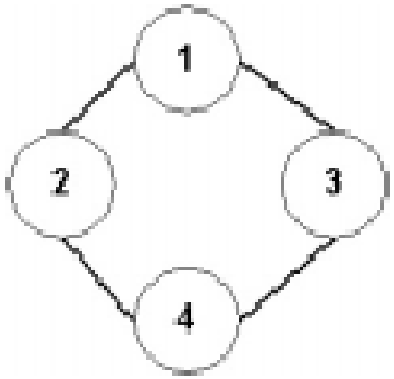
Graphen - Definition

Definition eines Graphen:

Ein **Graph** ist eine Menge von **Knoten** und **Kanten**. Knoten sind Objekte, die einen Namen und Eigenschaften besitzen können. Kanten sind Verbindungen zwischen Knoten.

Graph $G = (V, E)$ entspricht:

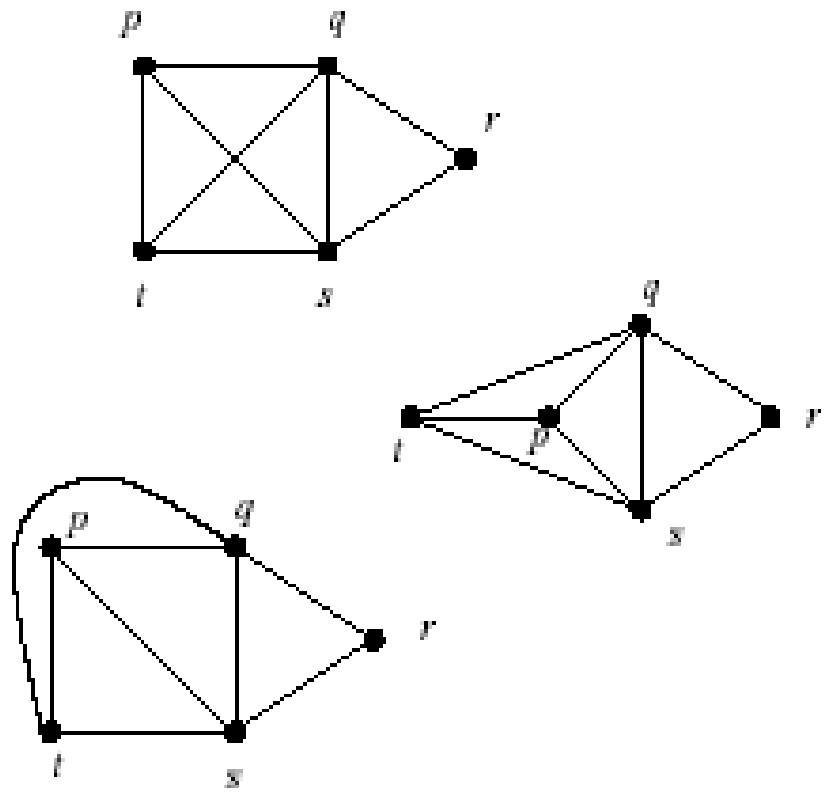
- Menge aller Knoten: $V = \{1,2, \dots, |V|\}$
- Menge aller Kanten: $E \subseteq V \times V$



Graphen - Darstellung

Die Visualisierung eines Graphen wird auch Diagramm genannt.

Ein Graph kann durch verschiedene Diagramme dargestellt werden.

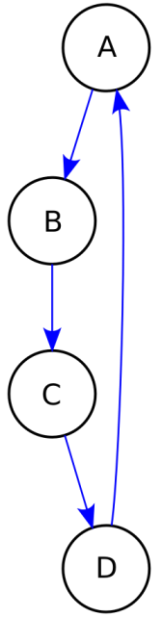


Graphen - Gerichtet

Wenn die Kante $\langle v_i, v_j \rangle$ ein geeordnetes Paar ist so werden die Knoten in eine Richtung verbunden (in diesem Beispiel von i zu j).

Beispiel:

- Produktionspläne (Abfolgepläne)
- Wegbeschreibung (Einbahnstraßen)

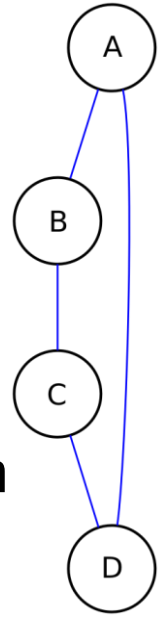


Graphen - Ungerichtet

Wenn die Kante (v_i, v_j) ein ungeeordnetes Paar ist so werden die Knoten ohne Richtung verbunden (in diesem Beispiel existiert damit eine Verbindung von i zu j und eine Verbindung von j zu i).

Beispiel:

- Wanderkarten
- Nachbarschaftsbeziehungen



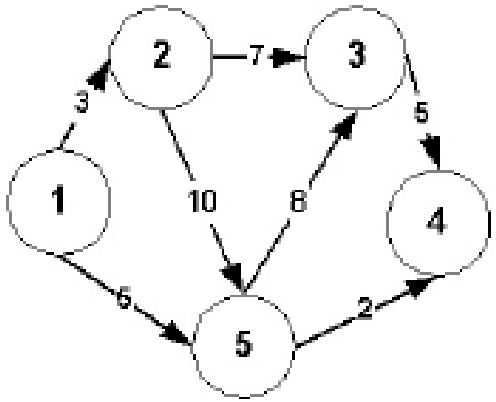
“Spezialfall”: der ungerichtete Graph ist ein gerichteter Graph bei dem Jede Kante in beiden Richtungen vertreten ist.

Graphen - Gewichtet

Wenn jeder Kante ein Wert zugeordnet ist so handelt es sich um einen gewichteten Graphen. Diese Werte können Kosten, Kapazität, Entfernung, oder ähnliches darstellen.

Beispiel:

- Straßenkarten
  - Knoten: Orte
  - Kanten: Straßen
  - Gewichte: Entfernung



Netzwerk

Graphen die **gewichtet** und **gerichtet** sind werden auch als Netzwerke bezeichnet.

Beispiel:

- Straßenkarten
- Schaltpläne
- Projektpläne
- Pipelines

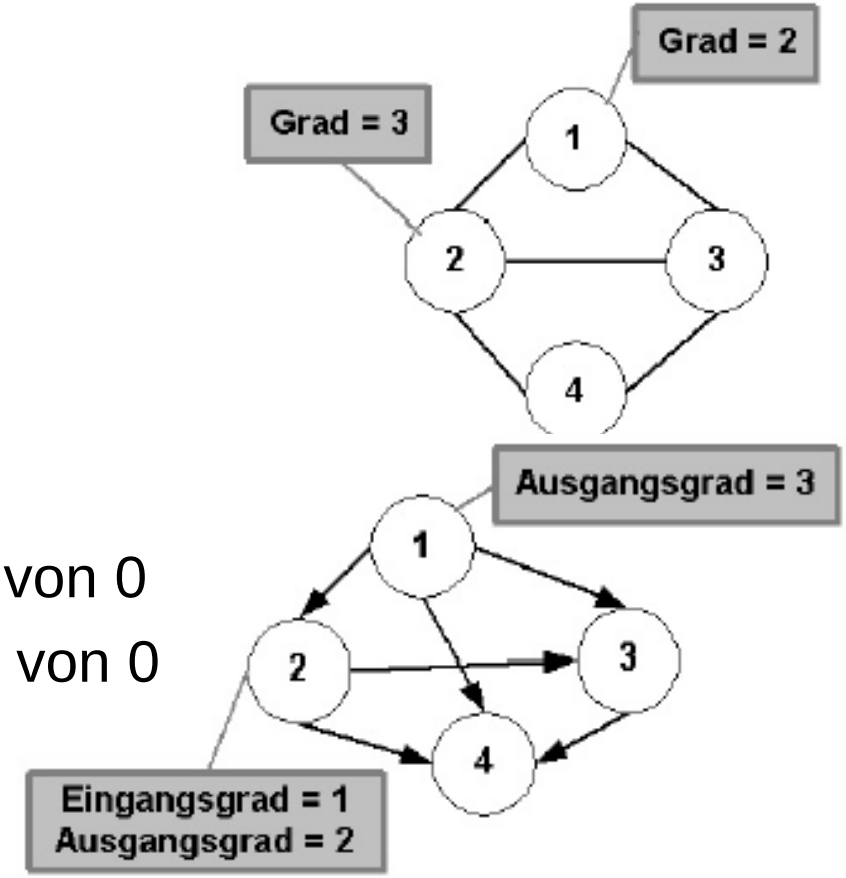
Graphen - Begriffsklärung

Grad (eines Knoten): Anzahl der adjazenten Kanten ($deg(v)$)

- Eingangsgrad: Anzahl der eingehenden Kanten
- Ausgangsgrad: Anzahl der ausgehenden Kanten

Quelle: Knoten mit Eingangsgrad von 0

Senke: Knoten mit Ausgangsgrad von 0



Graphen - Begriffsklärung

Isolierter Knoten:

Ein Knoten mit einem Grad von 0 heißt isolierter Knoten

!&/§&§“)§ :

Ein Knoten mit einem Grad von 1 heißt !&/§&§“)§

Graphen - Begriffsklärung

Isolierter Knoten:

Ein Knoten mit einem Grad von 0 heißt isolierter Knoten

Blatt:

Ein Knoten mit einem Grad von 1 heißt Blatt

Graphen - Handshaking Lemma

Bekannt ist das jeder Knoten $v \in V$ hat einen Grad $deg(v)$ hat.

Gesucht ist die Summe der Grade aller Knoten des Graphen $G = (V, E)$.

$$\sum_{v \in V} deg(v) = 2|E|$$

Diese Relation ist als Handschlaglemma (Handshaking Lemma) bekannt.

Graphen - Pfade

Pfad:

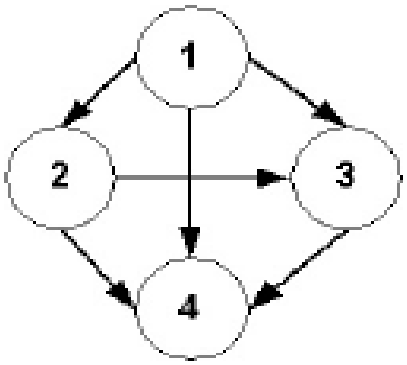
Liste von aufeinanderfolgenden verbundenen Knoten

Pfadlänge:

Anzahl der Kanten in einem Pfad

Beispiel:

- Pfad: (1,2,3,4)
- Länge: 3



Graphen - Pfade

Zusammenhängend:

Ein Graph ist zusammenhängend wenn von jedem Knoten zu jedem anderen Knoten im Graphen ein Weg existiert. Jeder zusammenhängende Graph mit n Knoten hat mindestens $n - 1$ Kanten.

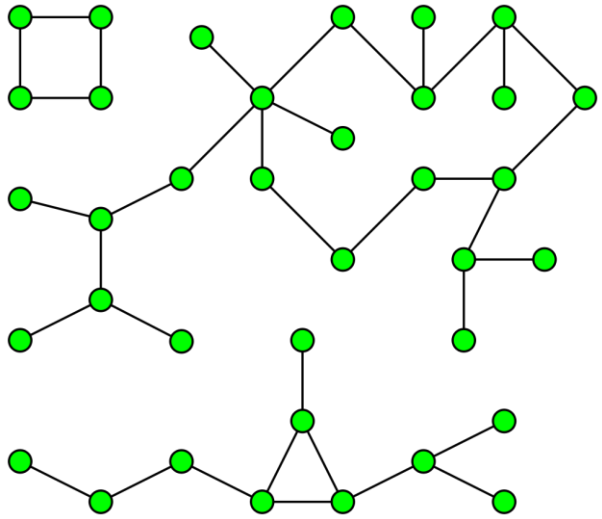
Stark Zusammenhängend:

Ein gerichteter Graph ist stark zusammenhängend wenn zwischen zwei Knoten i und j einen Weg von i nach j und einen Weg von j nach i gibt.

Graphen - Pfade

Komponente:

Eine Komponente (auch Connected Component) von einem Graphen ist ein Untergraph der zusammenhängend und bezüglich der Knotenzahl maximal ist.



Ein zusammenhängender Graph besteht aus genau einer Komponente

Graphen - Pfade

Einfacher Pfad:

Ein Pfad auf dem sich kein Knoten wiederholt heißt einfacher Pfad.

Zyklischer Pfad:

Ein einfacher Pfad mit identischem Anfangs und Endpunkt heißt zyklischer Pfad.

Ein Graph ohne Zyklen wird Baum genannt.

Eine Gruppe nicht zusamennhängender Bäume wird Wald genannt.

Spannbaum

Definition:

Ein **Spannbaum** eines Graphen ist ein Teilgraph, der alle Knoten enthält, doch nur so viele von den Kanten, dass er einen Baum bildet.

Hinzufügen einer Kante zu einem Spannbaum führt zu einem Zyklus!

Graph mit v Knoten und

- $< v - 1$ Kanten: Graph nicht zusammenhängend
- $> v - 1$ Kanten: Graph ist zyklisch
- $= v - 1$ Kanten: kann aber muss nicht notwendigerweise ein Baum sein

Graphen - Vollständigkeit

Gegeben ist ein Graph $G = (V, E)$

Vollständigkeit:

Der Graph G ist vollständig wenn für alle Knoten v und v' mit $v, v' \in V$ eine Kante (v, v') in E existiert. In diesem Fall gilt $E = |V| * \frac{(|V|-1)}{2}$

Dicht:

Ein Graph, dem nur wenige Kanten zu einem vollständigen Graph fehlen werden dicht genannt

Licht:

Graphen mit relativ wenigen Kanten (z.B.: $|E| < |V| * \log|V|$) werden licht genannt.

Graphendarstellung

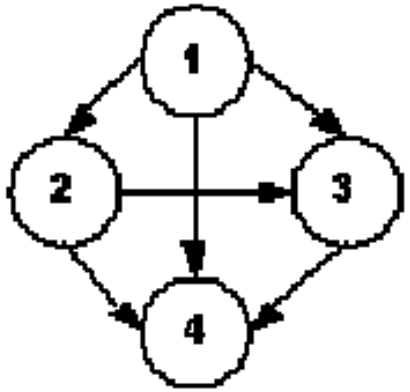
Graphendarstellung

Adjezenzmatrix:

Matrix $(|V| \times |V|)$ für den Graphen wird mit boolschen Werten gefüllt. Diese Geben an ob eine Kante im Graphen vorhanden ist oder nicht.

Speicherbedarf: $\mathcal{O}(|V|^2)$

- Unabhängig von Kantenanzahl
- Geeignet für dichte Graphen



A[i,j]	1	2	3	4
1	-	●	●	●
2	-	-	●	●
3	-	-	-	●
4	-	-	-	-

V= {1,2,3,4}

E = {<1,2>, <1,3>, <1,4>, <2,3>,<2,4>, <3,4>}

Graphendarstellung

Adjazenzmatrix:

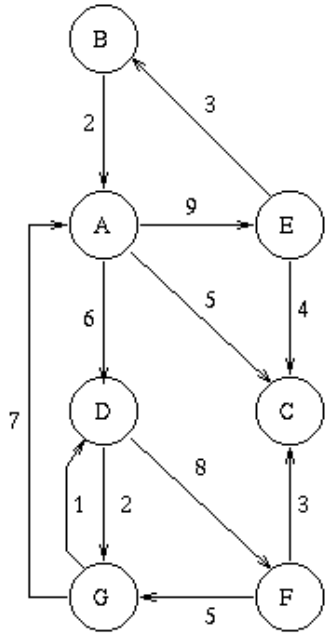
Anpassung der Adjazenzmatrix für gewichtete Graphen: in der Matrix warden die Gewichte der Kanten gespeichert.

Adjazenzmatrix ungewichtet (Graph siehe Adjazenzlisten)

	A	B	C	D	E	F	G
A	0	0	1	1	1	0	0
B	1	0	0	0	0	0	0
C	0	0	0	0	0	0	0
D	0	0	0	0	0	1	1
E	0	1	1	0	0	0	0
F	0	0	1	0	0	0	1
G	1	0	0	1	0	0	

Adjazenzmatrix gewichtet
(– bezeichnet z.B. 64536)

	A	B	C	D	E	F	G
A	0	–	5	6	9	–	–
B	2	0	–	–	–	–	–
C	–	–	0	–	–	–	–
D	–	–	–	0	–	8	2
E	–	3	4	–	0	–	–
F	–	–	3	–	–	0	5
G	7	–	–	1	–	–	0



Adjazenzmatrix und zugehöriger Graph

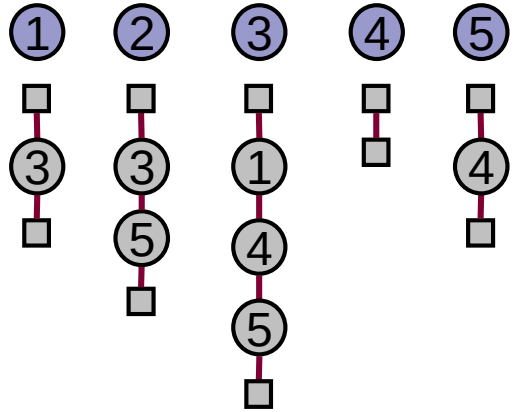
Graphendarstellung

Adjazenzliste:

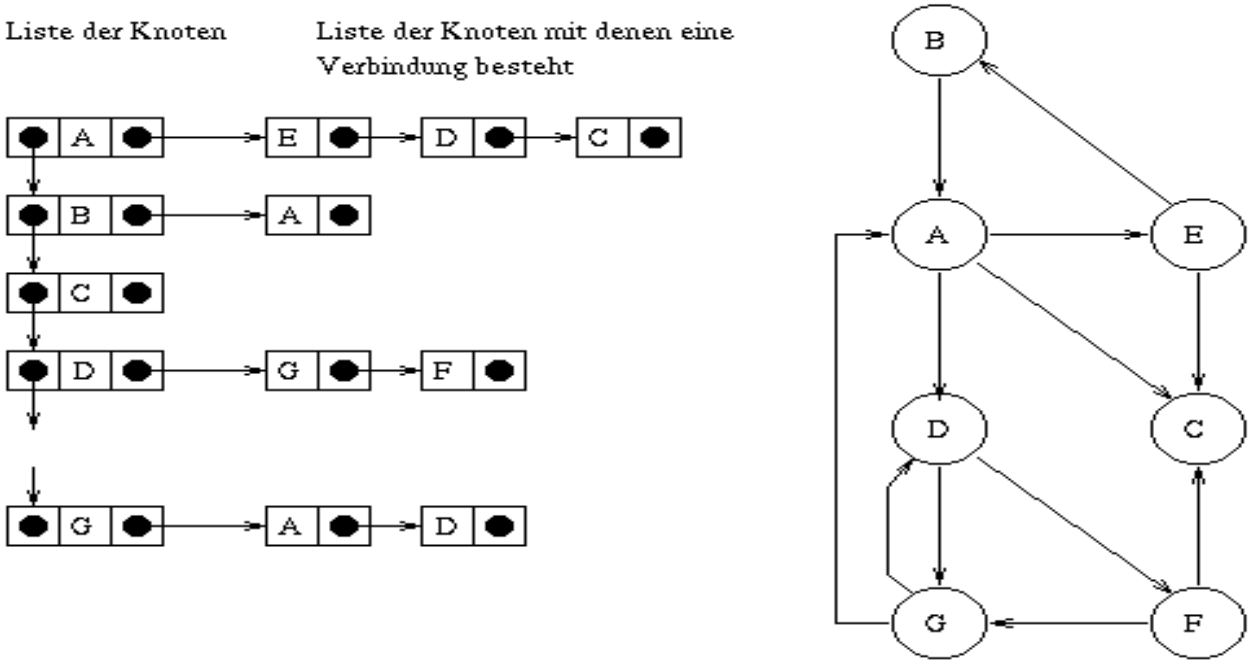
Knoten werden einem linearen Feld oder in einer linearen Liste gespeichert (länge $|V|$). Für jeden Knoten werden benachbarte Knoten, die durch eine Kante verbunden sind, als Verweise in einer linearen verketten Liste gespeichert. Dadurch werden die Knoten mehrmals in der Liste abgelegt.

Bei ungerichteten Graphen wird jede Kante in beide richtungen eingetragen.

```
V = {1,2,3,4,5}
E = {<1,3>,<2,3>,<2,5>,
     <3,1>,<3,4>,<3,5>,<5,4>}
```



Graphendarstellung



Adjazenzliste und zugehöriger Graph

Graphendarstellung

Adjazenzliste:

Speicherbedarf gerichtet: $\mathcal{O}(|V| + |E|)$

Speicherbedarf ungerichtet: $\mathcal{O}(|V| + 2|E|)$

- Abhängig von Kantenzahl
- Geeignet für lichte Graphen

Graphenalgorithmen

Graphenalgorithmen - Suche

Tiefensuche:

Rekursive Lösung: es werden von einem Startpunkt aus alle Kinder eines Knoten besucht. Bei einem besuchten Kind werden wiederum alle Kinder besucht. Damit werden Kinder gegenüber Nachbarn bevorzugt.

Achtung: Zyklen können hierbei zu Problemen führen! Eine weitere Datenstruktur verwaltet die bereits besuchten Knoten.

```
depthFirst(Vertice v) {
    if (n not visited yet) {
        for (all (n,m) in E, n ==v) {
            depthFirst(m)
        }
    }
}
```

Graphenalgorithmen - Suche

Breitensuche:

Es werden von einem Startpunkt aus alle Kinder eines Knoten besucht. Wenn alle Kinder besucht wurden, werden die Kinder dieser bereits besuchten Kinder besucht. Damit werden Nachbarn gegenüber Kindern bevorzugt.

Achtung: die Breitensuche kann nicht rekursive durch die Verwendung einer Quelle realisiert werden.

Graphenalgorithmen - Suche

Breitensuche:

```
breadthFirst(Vertice v) {
        Queue.put(v);
        visited[v]=TRUE;
        do {
                Vertice n = Queue.get();
                for each (neighbour m of n){
                   if (not visited[m]) {
                       Queue.put(m);
                       visited[m]=TRUE;
                   }
                }
        } while (not Queue.empty());
}
```

Graphenalgorithmen – Kürzester Pfad

Kürzester Pfad Problem:

(auch bekannt asl Single-Source Shortest Path Problem)

- **Gegeben:**
  ein gewichteter Graph $G = (V, E)$, Ziel und Start Knoten
- **Gesucht:**
  der kürzeste Weg vom Start zum Ziel Knoten

Anwendung:

- Pfadplanung
- Routing im Internet (OSPF)

Graphenalgorithmen - Küzester Pfad

Definition:

Die Entfernung zwischen zwei Knoten x und y ist definiert als:

$d(x,y) = min\{w(p) | p\ ist\ ein\ Weg\ von\ x\ zu\ y\}$ falls ein Weg existiert

$d(x,y) = \infty$ sonst

Ein Weg p zwischen x und y istder kürzeste Weg wenn $w(p) = d(x,y)$ gilt. Es wird auch als Shortest Path $sp(x,y)$ bezeichent.

Die Entfernung eines Knotens von sich selbst ist null!

Graphenalgorithmen - Dijkstra

Algorithmus von Dijkstra:

Greedy Algorithmus zur Lösung des Single-Source Shortest Path Problems (nur für positive Kantengewichte). Dabei wird von einem Startknoten aus die kürzeste Entfernung zu allen anderen Knoten berechnet.

Initialisierung:

`S = {(v, d(x,v))|sp(x,v) bekannt}`	`Menge der besuchten Knoten`
	`kürzeste Distanz bekannt`
	`(enthält Startknoten)`

`D = {(v, d(x,v))|sp(x,v) noch nicht bekannt}`	`Menge der noch nicht besuchten Knoten`
	`Kürzeste Distanz noch nicht bekannt`
	`(enthält alle Knoten außer dem Startknoten)`

Graphenalgorithmen - Dijkstra

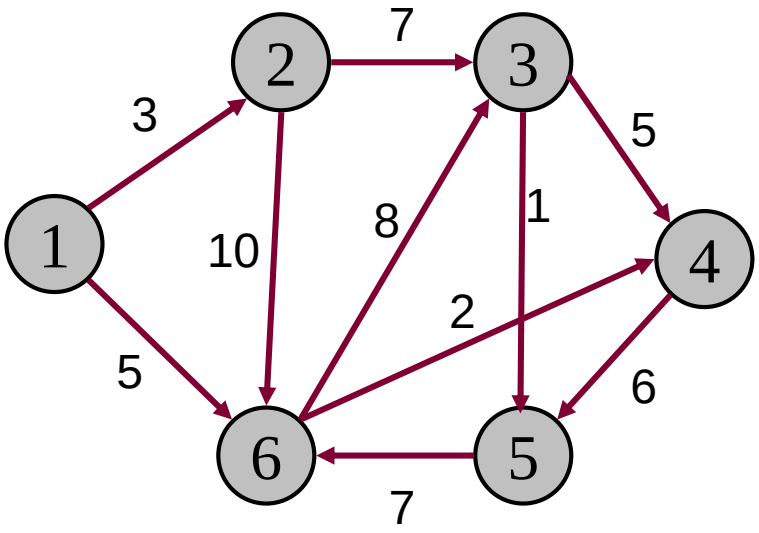
Algorithmus von Dijkstra:

Ablauf:

- Solange Knoten in D vorhanden sind:
  - Füge Knoten aus D mit dem kleinsten Abstand zu irgendeinem Knoten in S zur Menge S hinzu und entferne ihn aus D
  - Für diesen Knoten berechne den Abstand zu seinen Nachbarn in D. Falls der Weg zum betrachteten Knoten plus dem Abstand zum Nachbarn kürzer ist als der kürzeste Weg bisher übernimm den neuen Wert.

Graphenalgorithmen - Dijkstra

Aufgabe: Suche nach kürzestem Pfad von 1 zu allen anderen Knoten.

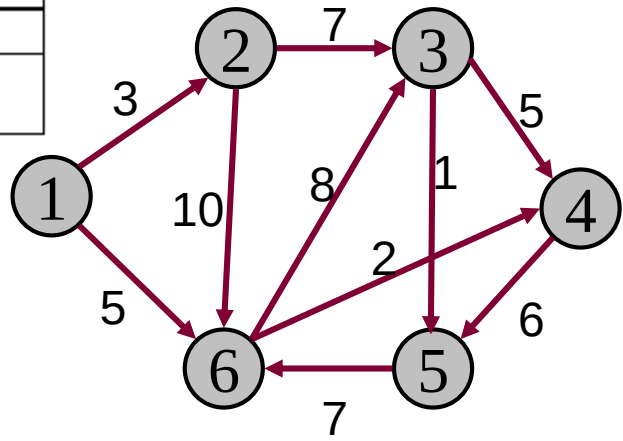


Adjazenzmatrix

	1	2	3	4	5	6
1	0	3	-	-	-	5
2	-	0	7	-	-	10
3	-	-	0	5	1	-
4	-	-	-	0	6	-
5	-	-	-	-	0	7
6	-	-	8	2	-	0

Graphenalgorithmen - Dijkstra

	neuer Knoten	S	D
1.	*Mengen initialisieren*		
		{(1,0)}	{(2,3) (3, ?) (4, ?) (5, ?) (6,5)}
2.	*Knoten 2 in Menge S aufnehmen (kleinste Distanz vom Startknoten 1)*		
	u = 2; dist(1,2) = 3	{(1,0) (2,3)}	{(3,10) (4, ?) (5, ?) (6,5)} Knoten 3: min(? , 3+7) = 10 Knoten 6: min(5, 3+10) = 5
3.	*Knoten 6 in Menge S aufnehmen*		
	u = 6; dist(1,6)=5	{(1,0) (2,3) (6,5)}	{(3,10) (4,7) (5, ?)}
4.	*Knoten 4 in Menge S aufnehmen*		
	u = 4; dist(1,4) = 7	{(1,0) (2,3) (6,5) (4,7)}	{(3,10) (5,13)}
5.	Knoten 3 in Menge S aufnehmen		
	u = 3; dist(1,3) = 10	{(1,0) (2,3) (6,5) (4,7) (3,10)}	{(5,11)}
6.	Fertig !		
		{(1,0) (2,3) (6,5) (4,7) (3,10) (5,11)}	-



Graphenalgorithmen - Dijkstra

Priority-First-Search:

Implementierung des Dijkstra Algorithmus zur Suche des kürzesten Pfades zwischen 2 Knoten i und j

Datenstruktur:

- Adjazenzliste zur Speicherung der Kanten pro Knoten
- boolesches Hilfsfeld `visited[]` zur Markierung bereits besuchter Knoten, zu Beginn mit FALSE initialisiert
- Heap zur Speicherung der besuchten Kanten und Pfade mit 2 Grundfunktionen
  - `Heap.put(Node n,int gewicht)` ordnet den Knoten n mit entsprechenden Gewicht ein
  - `Node n=Heap.get()` holt und entfernt den Knoten mit geringstem Gewicht

Graphenalgorithmen - Dijkstra

```
int MinWeg(Node i, Node j) {
    int minweg=0;
    do {
        visited[i]=TRUE; // Knoten markieren
        for each (neighbour m of i){
            if ( not visited[m]) {
                // Nachbarknoten in Heap einordnen
                Heap.put(m,m.gewicht+minweg);
            }
        }
        do {
            // Knoten mit minimalem Gewicht aus
            // Heap entfernen, minweg aktualisieren
            i=Heap.get();
         } while (visited[i]);
    } while (i!=j); // Solange Zielknoten nicht erreicht
    return minweg;
}
```

Graphenalgorithmen - Minimaler Spannbaum

Gegeben:

ungerichteter zusammenhängender und gewichteter Graph $G = (V, E)$

Gesucht:

Minimaler Spannbaum (MST):

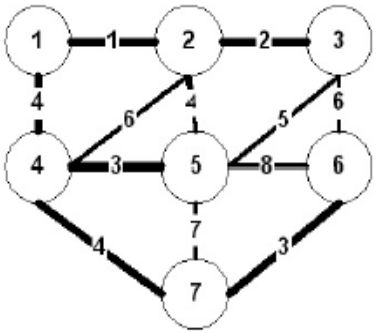
- MST ist Teilgraph von G
- Für den MST gilt $E_{MST} = V_{MST} - 1$ wobei $V = V_{MST}$
- MST ist ein Baum, und somit zusammenhängend sowie frei von Zyklen
- Die Summe der Kantenlängen des MST ist minimal

Graphenalgorithmen - Minimaler Spannbaum

Beispiel: Verbindungsnetz

- Zwischen n Knotenpunkten P_1 ... P_n soll ein möglichst billiges Verbindungsnetz geschaltet werden, so dass jeder Knotenpunkt mit jedem anderen verbunden ist, gegebenenfalls auf einem Umweg über andere Knotenpunkte.
- Bekannt sind die Kosten $d_{i,j}$ für die direkte Verbindung zw. P_i und $P_j, 1 \leq i, j \leq n$.

Beispiel:



Semantik

V	Knoten entsprechen den Orten
w(i, j)	Kosten für den Bau einer Straße (Leitung, ...) von Ort *i* nach Ort *j*
(V, E_{MST})	billigste Trassenführung, die alle Orte $\in$ V verbindet.

Graphenalgorithmen - Kruskal

Algorithmus von Kruskal:

Greedy Algorithmus um MST zu finden.

Ablauf:

- Alle Kanten werden nach aufsteigender Länge geordnet.
- Der minimale Spannbaum wird sequentiell aus Teilbäumen aufgebaut.
- Begonnen wird mit Q Startbäumen, bestehend aus einem Knoten $\in V$

Graphenalgorithmen - Kruskal

Algorithmus von Kruskal:

Ablauf:

- In jedem Verbindungsschritt werden jeweils zwei Teilbäume durch eine noch nicht eingebaute kürzeste Kante (greedy!) zu einem neuen Teilbaum verbunden
  - wenn die Endknoten der Kante in verschiedenen Teilbäumen liegen.
  - sonst scheidet die Kante aus (führt zu Kreisfreiheit!).
- Diese Vorgehensweise wird erschöpfend angewandt.
- Ende, wenn alle Knoten $\in$ V zu einem Baum verbunden sind.
- **Ergebnis: der (ein) minimaler Spannbaum des gegebenen Graphen.**

Graphenalgorithmen - Prim

Algorithmus von Prim:

Minimaler Spannbaum wird beginnend mit willkürlich gewählter Wurzel zusammenhängend aufgebaut.

Ablauf:

- Initialisiere die Lösungsmenge B mit einem beliebigen Knoten aus V. Sei U die Menge aller noch nicht behandelten Knoten.
- In jedem Schritt wird die kürzeste verfügbare Kante, die einen Knoten $\in B$ mit einem beliebigen Knoten $\in U$ verbindet, eingebaut. Diese neu hinzugekommenen Knoten werden zu B hinzugefügt und aus U entfernt.
- Das Verfahren wird angewendet, bis die Lösungsmenge B alle Knoten des ursprünglichen Graph ($\in V$) enthält.
- **Ergebnis: der (ein) minimaler Spannbaum des gegebenen Graphen.**

P vs NP

Problemklassen von Algorithmen

- Problemklasse P
  - Menge aller Probleme, die mit Hilfe deterministischer Algorithmen in polynomialer Zeit gelöst werden können
    - effizient lösbar, Aufwand maximal $\mathcal{O}(n^k)$
- Problemklasse NP
  - Menge aller Probleme, die mit Hilfe nicht deterministischer Algorithmen in polynomialer Zeit gelöst werden können
    - nicht deterministisch bedeutet: Der Algorithmus hat die Fähigkeit, bei der Entscheidung zwischen mehreren Varianten die richtige zu „erraten“
    - momentan nicht effizient lösbar, Umsetzung als deterministischer Algorithmus hätte mindestens exponentiellen Aufwand $O(2^n)$

Problemklassen von Algorithmen

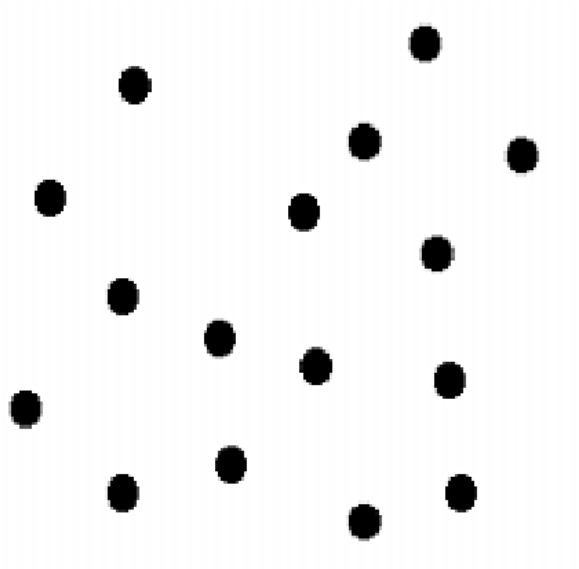
- Zusammenhang zwischen den Klassen P und NP
  - Es gibt eine Reihe von Problemen der Klasse NP, d.h. Probleme die sich momentan nicht mit polynomialen Aufwand lösen lassen
  - Für diese Probleme ist es noch nicht gelungen, einen deterministischen effizienten Algorithmus zu finden
  - Es konnte allerdings noch nicht bewiesen werden, ob diese Probleme nicht doch mit polynomialen Aufwand gelöst werden können, was jedoch sehr unwahrscheinlich wäre
  - Offenes Problem der theoretischen Informatik, ob P=NP gilt

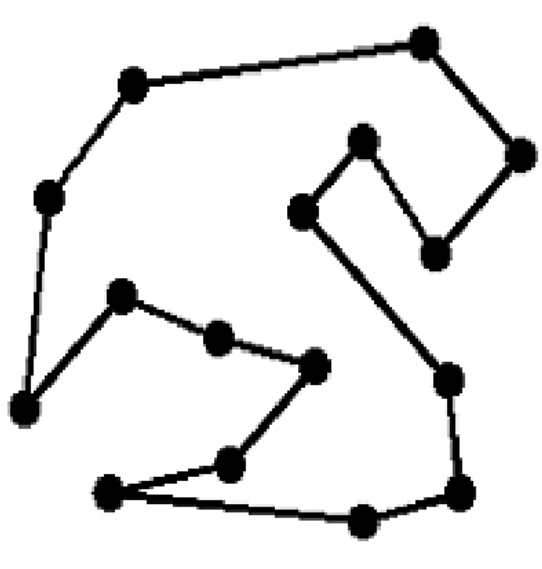
Problemklassen von Algorithmen

- NP vollständige Probleme
  - Eine Klasse von verwandten Problemen aus NP mit folgender Eigenschaft:
  - Falls ein deterministischer Algorithmus gefunden wird, der eines dieser Probleme in polynomialer Zeit löst, so ist das für alle Probleme aus NP möglich
  - Damit wäre bewiesen dass P=NP gilt

- Beispiele
  - Erfüllbarkeitsproblem in der mathematischen Logik
    - gesucht: Variablenbelegung die für eine geg. Formel true liefert
  - Rucksackproblem
  - TSP

Traveling Salesman Problem





TSP - Problemstellung

Gegeben:

Ungerichteter, vollständiger, zusammenhängender und gewichteter Graph; G = (V,E)

- Knoten = Städte
- Kanten = Verbindung zwischen den Städten mit Entfernung als Gewicht der Kanten

Gesucht:

Rundreise durch alle Städte, die jede Stadt genau einmal besucht mit minimalen Gesamtkosten

- Handlungsreisendenproblem bzw.
  Traveling Salesman Problem (TSP)

TSP - Problemstellung Formal

Formale Definitionen:

Graph $G(V, E)$

- Ein Weg, der jeden Knoten von G genau einmal enthält, heißt *hamiltonscher Weg*.
- Ein Kreis, der jeden Knoten von G genau einmal enthält, heißt *hamiltonscher Kreis* (HC)
- G heißt *hamiltonsch* genau dann wenn G einen hamiltonschen Kreis enthält.

- TSP:
  - Gesucht wird ein hamiltonscher Kreis mit minimalen Kosten
  - NP schwieriges Problem (kein exakter Algorithmus bekannt, der Problem in polynomieller Zeit lösen kann)
  - nur für kleine Städteanzahl exakt lösbar

TSP - Anwendungen

Das TSP findet vielseitige Anwendung:

- Vehicle Routing
- Leiterplattenherstellung
- Roboterbewegungen
- Scheduling von Aufgaben
- …

TSP - In Numbers

Primitive Lösung: Enumeration

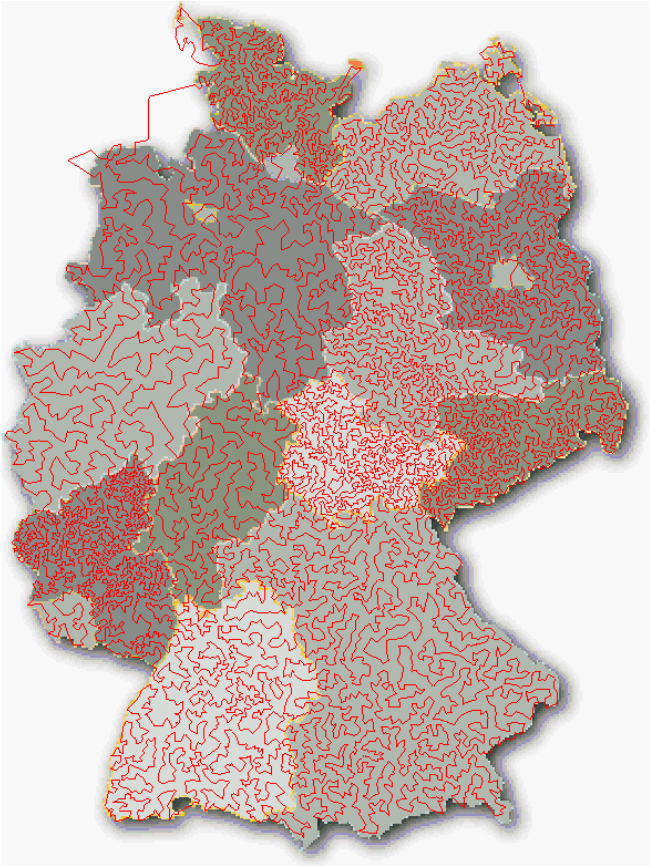
- Auflisten aller möglichen Rundreisen (Touren) und Bestimmung der Tour mit minimalen Kosten
- Wie viele Touren gibt es?
  - Vollständiger ungerichteter Graph mit n Knoten besitzt $n * \frac{n-1}{2}$ Kanten
  - Ausgangsstadt fixiert, Richtung der Tour egal
  - Insgesamt $\frac{(n-1)!}{2}$ Touren
- Angenommen ein Rechner kann 40 Millionen Hamiltonsche Kreise (HC) pro Sekunde berechnen:

N	Anzahl HC	Zeit
10	181440	0,00045 Sekunden
17	ca. 10^{13}	3 Tage
19	ca. 10^{15}	2,5 Jahre
20	ca. 10^{17}	48 Jahre
25	ca. 10^{23}	10^{8} Jahre
60	ca. 10^{80}	10^{64} Jahre

TSP - In Practice

TSP für 15112 Städte Deutschlands gelöst:

- Rice und Princeton University
- 110 Prozessoren
- 22.6 Jahre total computation time



TSP - Heuristik

Approximative Algorithmen

- Algorithmen, die die optimale Lösung nur annähern = Heuristiken

Heuristiken für das TSP

- Konstruktionsheuristiken
  - Erzeugen Näherungslösungen
  - Nearest Neighbour
  - Insert Heuristiken
  - Spanning Tree Heuristik
- Verbesserungsheuristiken
  - Ausgehend von einer zulässigen Lösung wird versucht, diese durch lokale Änderungen zu verbessern
  - Austauschverfahren, Lokale Suchverfahren
  - Genetische Algorithmen

TSP - Heuristik

Nearest Neighbour Heuristic:

- Setze aktuellen Knoten auf einen beliebigen Startknoten
- Wiederhole solange noch nicht jeder Knoten besucht wurde
  - Suche Kante mit minimalen Gewicht vom aktuellen Knoten zu einem beliebigen Nachbarknoten, der noch nicht besucht wurde und nimm sie in die Rundreise auf
  - Setze aktuellen Knoten auf neuen Nachbarknoten
- Letze Kante zurück zum Starknoten, fertig!
- Laufzeit $O(n^2)$, Greedy Algorithmus
- Problem: kann beliebig schlecht werden, letzte Kante meist sehr lang

TSP - Heuristik

Insertion Heuristic:

- Starte mit einer initialen Subtour
- Füge fehlende Knoten nacheinander in die Tour ein, bis alle Knoten inkludiert sind

Herausforderungen:

- Bestimmen der initialen Tour
  - z.B.: irgendein Kreis der Länge 3
  - größtes Dreieck
  - Konvexe Hülle aller Knoten (umschliessender Kreis)
- Auswahl des nächsten einzufügenden Knotens
- Auswahl der Einfügeposition des neuen Knotens
  - Ersetze eine Kreiskante durch 2 neue Kanten zu dem gefunden Knoten mit minimalen zusätzlichen Kosten

TSP - Heuristik

Insertion Heuristic:

- Starte mit einer initialen Subtour
- Füge fehlende Knoten nacheinander in die Tour ein, bis alle Knoten inkludiert sind

Herausforderungen:

- Bestimmen der initialen Tour
  - z.B.: irgendein Kreis der Länge 3
  - größtes Dreieck
  - Konvexe Hülle aller Knoten (umschliessender Kreis)
- Auswahl des nächsten einzufügenden Knotens
- Auswahl der Einfügeposition des neuen Knotens
  - Ersetze eine Kreiskante durch 2 neue Kanten zu dem gefunden Knoten mit minimalen zusätzlichen Kosten

TSP - Heuristik

- **Nearest Insert Heuristic**
  - Bestimme Knoten mit minimaler Distanz zu einem Kreisknoten
- **Farthest Insert Heuristic**
  - Bestimme Knoten dessen minimale Distanz zu einem Kreisknoten maximal ist
- **Cheapest Insert Heuristic**
  - Bestimme Knoten, dessen Einfügen in die Tour minimale neue Kosten verursacht (Greedy Algorithmus)
- **Spanning Tree Heuristic**
  - Verwendet MST zum Finden einer Tour, sehr gute Ergebnisse in der Praxis

APPLET für Heuristiken

TSP - Branch & Bound

Exaktes Verfahren zur Berechnung ist zum Beispiel: **Branch & Bound**

- Geschicktes Enumerationsverfahren
- Mittels einer Heuristik wird eine zulässige Lösung U berechnet
- Weiters wird mit einem Verfahren eine untere Schranke L berechnet
- Falls $U = L$ fertig, optimale Lösung gefunden!
- Ansonsten wird Lösungsmenge partitioniert (Branching) und die Heuristiken werden auf die Teilmengen angewandt (Bounding)
  - Ist die berechnete untere Schranke einer Teilmenge nicht kleiner als die beste gefundene Lösung, kann man die Teilmenge vernachlässigen
  - Ansonsten wird rekursiv weiter zerlegt