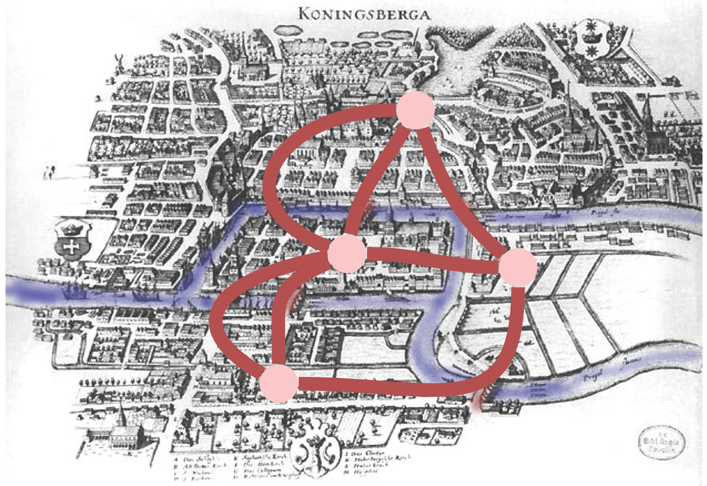
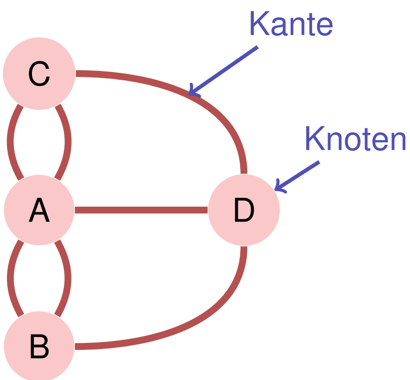
20. Graphen

Königsberg 1736



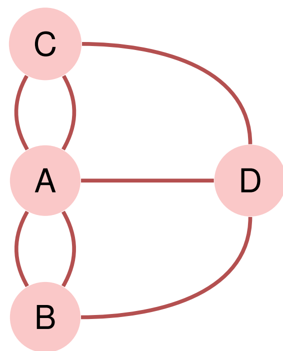
Graph



Zyklen

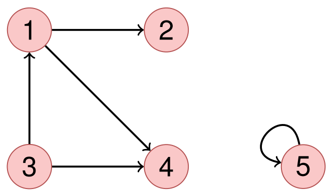
- Gibt es einen Rundweg durch die Stadt (den Graphen), welcher jede Brücke (jede Kante) genau einmal benutzt?
- Euler (1736): nein.
- Solcher Rundweg (*Zyklus*) heisst *Eulerscher Kreis*.
- Eulerzyklus $\Leftrightarrow$ jeder Knoten hat gerade Anzahl Kanten (jeder Knoten hat einen *geraden Grad*).

  "$\Rightarrow$" ist klar, "$\Leftarrow$" ist etwas schwieriger



Notation

Ein *gerichteter Graph* besteht aus einer Menge $V = \{v_1, \dots, v_n\}$ von Knoten (*Vertices*) und einer Menge $E \subseteq V \times V$ von Kanten (*Edges*). Gleiche Kanten dürfen nicht mehrfach enthalten sein.



Schleife

Notation

Ein *gewichteter Graph* $G = (V, E, c)$ ist ein Graph $G = (V, E)$ mit einer *Kantengewichtsfunktion* $c : E \to \mathbb{R}$. $c(e)$ heisst *Gewicht* der Kante e.

Notation

Für gerichtete Graphen $G = (V, E)$

- $w \in V$ heisst *adjazent* zu $v \in V$, falls $(v, w) \in E$
- *Vorgängermenge* von $v \in V$: $N^-(v) := \{u \in V | (u, v) \in E\}$. *Nachfolgermenge*: $N^+(v) := \{u \in V | (v, u) \in E\}$
- *Eingangsgrad*: $\deg^-(v) = |N^-(v)|$, *Ausgangsgrad*: $\deg^+(v) = |N^+(v)|$

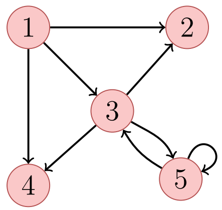
$\deg^-(v) = 3, \deg^+(v) = 2$ $\quad$ $\deg^-(w) = 1, \deg^+(w) = 1$

Wege

- *Weg*: Sequenz von Knoten $\langle v_1, \dots, v_{k+1} \rangle$ so dass für jedes $i \in \{1 \dots k\}$ eine Kante von v_i nach v_{i+1} existiert.
- *Länge* des Weges: Anzahl enthaltene Kanten k.
- *Gewicht* des Weges (in gewichteten Graphen): $\sum_{i=1}^{k} c((v_i, v_{i+1}))$ (bzw. $\sum_{i=1}^{k} c(\{v_i, v_{i+1}\})$)

Repräsentation mit Matrix

Graph $G = (V, E)$ mit Knotenmenge $v_1, \dots, v_n$ gespeichert als *Adjazenzmatrix* $A_G = (a_{ij})_{1 \le i,j \le n}$ mit Einträgen aus $\{0, 1\}$. $a_{ij} = 1$ genau dann wenn Kante von v_i nach v_j.

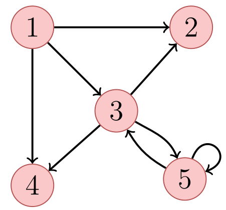


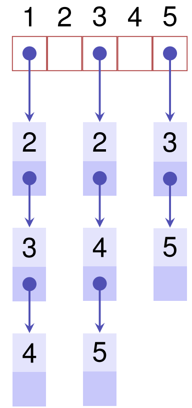
$$\begin{pmatrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

Speicherbedarf $\Theta(|V|^2)$. A_G ist symmetrisch, wenn G ungerichtet.

Repräsentation mit Liste

Viele Graphen $G = (V, E)$ mit Knotenmenge $v_1, \dots, v_n$ haben deutlich weniger als n^2 Kanten. Repräsentation mit *Adjazenzliste*: Array $A[1], \dots, A[n]$, A_i enthält verkettete Liste aller Knoten in $N^+(v_i)$.





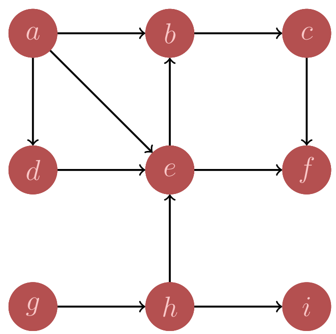
Speicherbedarf $\Theta(|V| + |E|)$.

Laufzeiten einfacher Operationen

Operation	Matrix	Liste
Nachbarn von $v \in V$ finden	$\Theta(n)$	$\Theta(\deg^+ v)$
$v \in V$ ohne Nachbar finden	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$
$(u, v) \in E$?	$\mathcal{O}(1)$	$\mathcal{O}(\deg^+ v)$
Kante einfügen	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Kante löschen	$\mathcal{O}(1)$	$\mathcal{O}(\deg^+ v)$

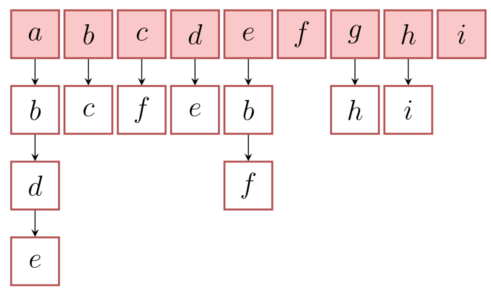
Graphen Traversieren: Tiefensuche

Verfolge zuerst Pfad in die Tiefe, bis nichts mehr besucht werden kann.



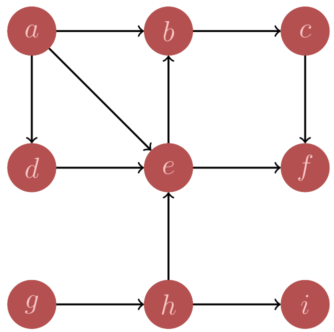
Reihenfolge
$a, b, c, f, d, e, g, h, i$

Adjazenzliste



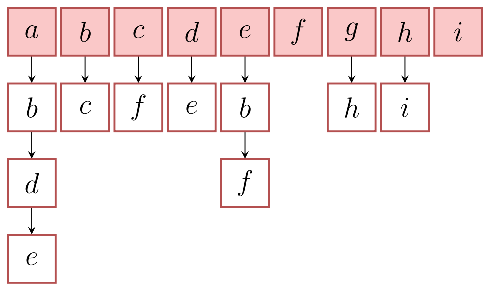
Graphen Traversieren: Breitensuche

Verfolge zuerst Pfad in die Breite, gehe dann in die Tiefe.



Reihenfolge
$a, b, d, e, c, f, g, h, i$

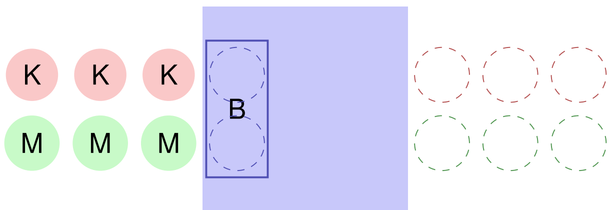
Adjazenzliste



21. Kürzeste Wege

Flussüberquerung (Missionare und Kannibalen)

Problem: Drei Kannibalen und drei Missionare stehen an einem Ufer eines Flusses. Ein dort bereitstehendes Boot fasst maximal zwei Personen. Zu keiner Zeit dürfen an einem Ort (Ufer oder Boot) mehr Kannibalen als Missionare sein. Wie kommen die Missionare und Kannibalen möglichst schnell über den Fluss? [12]



[12] Es gibt leichte Variationen dieses Problems, es ist auch äquivalent zum Problem der eifersüchtigen Ehemänner

Formulierung als Graph

Zähle alle erlaubten Konfigurationen als Knoten auf und verbinde diese mit einer Kante, wenn Überfahrt möglich ist. Das Problem ist dann ein Problem des kürzesten Pfades

Beispiel

	links	rechts
Missionare	3	0
Kannibalen	3	0
Boot	x	

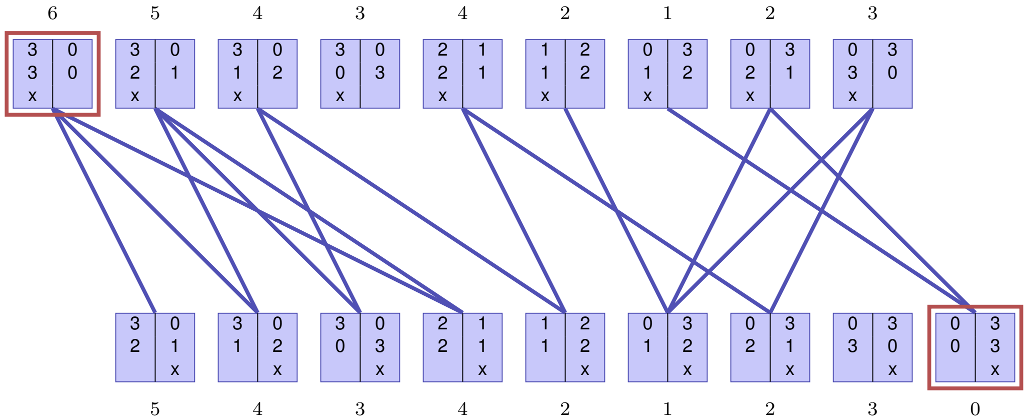
6 Personen am linken Ufer

Überfahrt möglich

	links	rechts
Missionare	2	1
Kannibalen	2	1
Boot		x

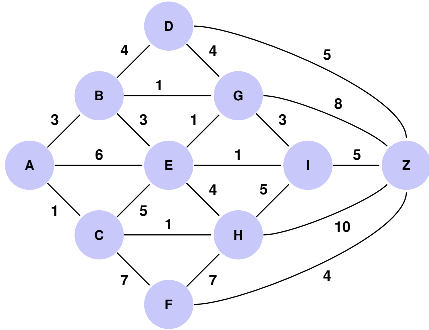
4 Personen am linken Ufer

Das ganze Problem als Graph



Routenfinder

Gegeben Städte A - Z und Distanzen zwischen den Städten.

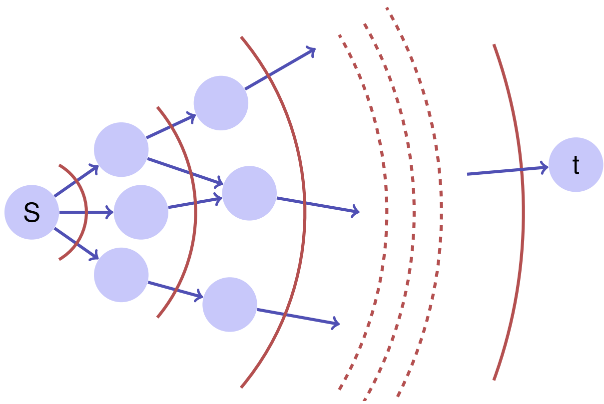


Was ist der kürzeste Weg von A nach Z?

Einfachster Fall

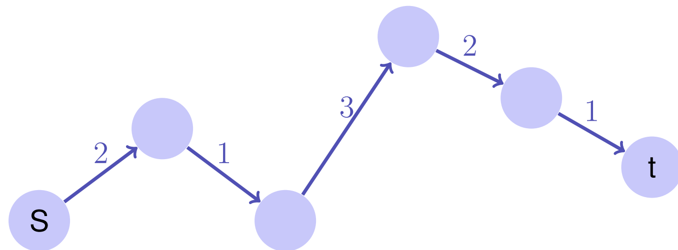
Konstantes Kantengewicht 1 (oBdA)

Lösung: Breitensuche



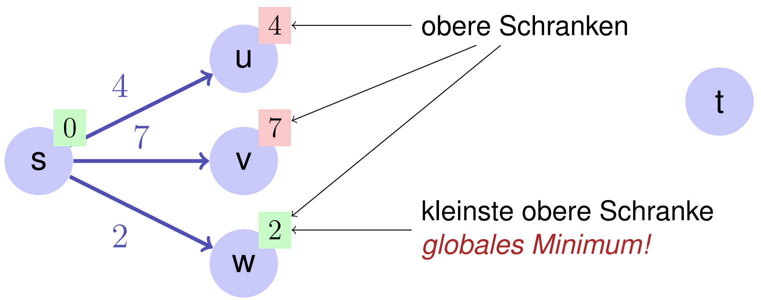
Positiv gewichtete Graphen

Gegeben: $G = (V, E, c)$, $c : E \to \mathbb{R}^+$, $s, t \in V$.
Gesucht: Länge eines kürzesten Weges (Gewicht) von s nach t.
Weg: $\langle s = v_0, v_1, \ldots, v_k = t \rangle$, $(v_i, v_{i+1}) \in E$ $(0 \leq i < k)$
Gewicht: $\sum_{i=0}^{k-1} c((v_i, v_{i+1}))$.



Weg mit Gewicht 9

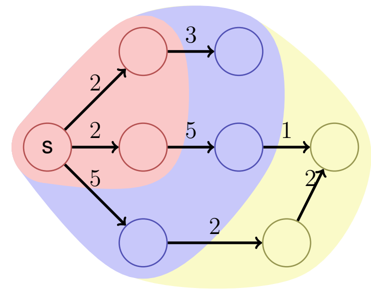
Beobachtung



Grundidee

Menge V aller Knoten wird unterteilt in

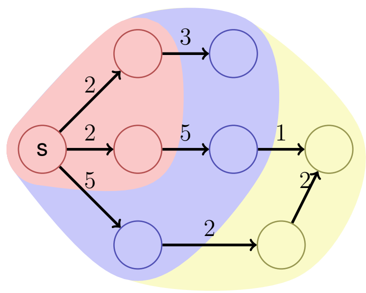
- die Menge M von Knoten, für die schon ein kürzester Weg von s bekannt ist
- die Menge $R = \bigcup_{v \in M} N^+(v) \setminus M$ von Knoten, für die kein kürzester Weg bekannt ist, die jedoch von M direkt erreichbar sind.
- die Menge $U = V \setminus (M \cup R)$ von Knoten, die noch nicht berücksichtigt wurden.



Induktion

Induktion über $|M|$: Wähle Knoten aus R mit kleinster oberer Schranke. Nimm r zu M hinzu, und update R und U.

Korrektheit: Ist innerhalb einer “Wellenfront” einmal ein Knoten mit minimalem Pfadgewicht gefunden, kann kein Pfad grösseren Gewichts über andere Knoten zu einer Verbesserung führen.

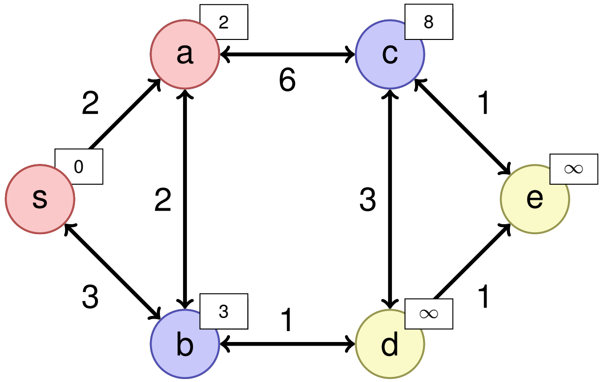


Algorithmus

Initial: $\mathrm{PL}(n) \leftarrow \infty$ für alle Knoten.

- Setze $\mathrm{PL}(s) \leftarrow 0$
- Starte mit $M = \{s\}$. Setze $k \leftarrow s$.
- Solange ein neuer Knoten k hinzukommt und dieser nicht der Zielknoten ist
  1. Für jeden Nachbarknoten n von k:
     - Berechne Pfadlänge x nach n über k
     - Wenn $\mathrm{PL}(n) = \infty$, so nimm n zu R hinzu
     - Ist $x < \mathrm{PL}(n) < \infty$, so setze $\mathrm{PL}(n) \leftarrow x$ und passe R an.
  2. Wähle als neuen Knoten k den mit kleinster Pfadlänge in R.

Beispiel



$$M = \{s, a\}$$
$$R = \{b, c\}$$
$$U = \{d, e\}$$

Zur Implementation

Benötigte Operationen

- ExtractMin (über R)
- Insert (Hinzunehmen zu R)
- DecreaseKey (Update in R)

Datenstruktur: MinHeap.

DecreaseKey

- DecreaseKey: Aufsteigen im MinHeap in $\mathcal{O}(\log |V|)$
- Position im Heap?
  - Möglichkeit (a): Speichern am Knoten
  - Möglichkeit (b): Hashtabelle über Knoten

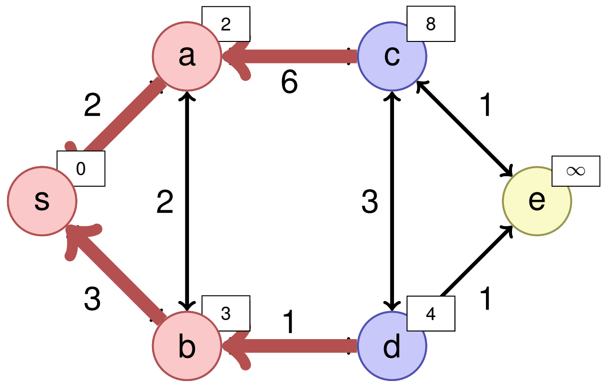
Laufzeit

- $|V| \times$ ExtractMin: $\mathcal{O}(|V| \log |V|)$
- $|E| \times$ Insert oder DecreaseKey: $\mathcal{O}(|E| \log |V|)$
- $1 \times$ Init: $\mathcal{O}(|V|)$
- Insgesamt: $\mathcal{O}(|E| \log |V|)$.

Kürzesten Weg Rekonstruieren

- Beim Updateschritt im obigen Algorithmus jeweils besten Vorgänger merken, an Knoten oder in separater Datenstruktur.
- Besten Pfad rekonstruieren durch Rückwärtslaufen der besten Kanten

Beispiel



$$M = \{s, a, b\}$$
$$R = \{c, d\}$$
$$U = \{e\}$$

Java: Kanten und Knoten

```
public class Edge {
  private Node to;
  private int length;

  Edge (Node t, int l) {
    to = t; length = l;
  }

  // Getters and Setters omitted for brevity
}
```

```
public class Node {
  private Vector<Edge> out;
  private String name;
  private int pathLen;
  private Node pathParent;

  Node(String n) {
    out = new Vector<Edge>();
    pathParent = null;
    pathLen = Integer.MAX_VALUE;
    name = n;
  }
  // Getters and Setters omitted for brevity
}
```


Java: Graph

```
public class Graph {
  private Vector<Node> nodes;
  private HashMap<String,Node> nodeByName;
  Graph(){
    nodes = new Vector<Node>();
    nodeByName = new HashMap<String,Node>();
  }
  public void AddNode(String s){}
  public Node FindNode(String s){}
  public void AddEdge(String from, String to, int length) {}

  Vector<Node> ShortestPath(Node S, Node E) { ... }
}
```

MinHeap

```
public class Heap {
  ...
  // Vergleich zweier Knoten
  // = Vergleich der aktuellen Pfadlaengen
  private boolean Smaller(Node l, Node r) {
    return l.GetPathLen() < r.GetPathLen();
  }
  public void Insert(Node n) { ... }
  public Node ExtractRoot() { ... }
  public void DecreaseKey(Node n){ ... }
}
```


Heap mit Find

```
public class Heap {
  ...
  HashMap <Node, Integer> map;

  // Damit die Hashtabelle immer konsistent bleibt, muss nun
  // an allen Stellen im Code, wo vorher data[x] = y stand,
  // Set(x,y) stehen:
  private void Set(int index, Node value){
    data[index] = value;
    map.put(value, index);
  }
}
```


DecreaseKey

```
public class Heap {
  ...
  public void DecreaseKey(Node n){
    int current = map.get(n); // hier brauchen wir die Hashtabelle
    int parent = (current−1)/2;
    // aufsteigen
    while (current > 0 && Smaller(n, data[parent])) {
      Set(current, data[parent]);
      current = parent;
      parent = (current−1)/2;
    }
    Set(current, n);
  }
}
```