Handout 9

Sebastian Millius, Sandro Feuz, Daniel Graf

Thema: Graphen Repräsentation, Breiten- und Tiefensuche, Spannende Bäume, Kürzeste Wege

Links

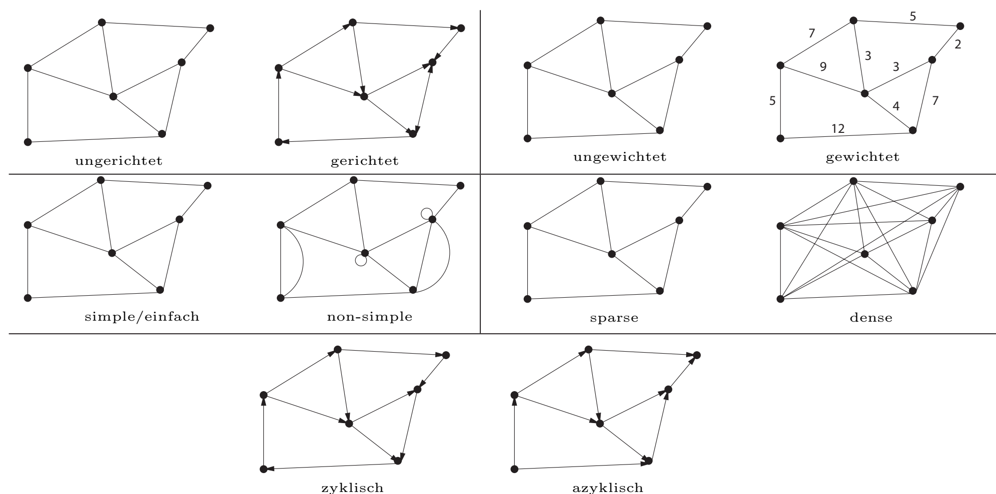
- Algorithms, S. Dasgupta et al. Chapter 3 & 4
  - http://www.cs.berkeley.edu/~vazirani/algorithms/chap3.pdf (http://goo.gl/Vx7Vy)
  - http://www.cs.berkeley.edu/~vazirani/algorithms/chap4.pdf (http://goo.gl/WiAx5)
- Visualisierungs Applets http://www.cs.princeton.edu/courses/archive/spr02/cs226/lectures.html (http://goo.gl/qZPT9)
- http://www.cs.washington.edu/homes/rahul/data/Dijkstra/index.html (http://goo.gl/jUyzp)
- TSort http://en.wikipedia.org/wiki/Tsort_(Unix) (http://goo.gl/Sdkj0)
- Sneak Peek: Basisprüfung 2014: http://goo.gl/Wi1LiA

Graphen

Der Begriff des Graphen ist fundamental in der Informatik. Viele Probleme lassen sich als Graphenprobleme modellieren.

Ein Graph $G = (V, E)$ besteht aus einer (endlichen) *Knoten*menge V und einer Menge $E \subseteq \binom{V}{2}$ von geordneten oder ungeordneten Paaren von Konten aus V, genannt *Kanten*.

Graphen können in unterschiedlichen Arten auftreten die insbesondere auch die Wahl der Repräsentation eines Graphen beeinflussen.



Beispiel: Gerichtete azyklische Graphen heissen *DAG*s (directed acyclic graph). Sie tauchen oft in Scheduling Problemen auf, wo beispielsweise eine Kante (x, y) bedeutet, dass eine Aktivität/Event x vor y eintreten muss. *Topologische Sortierung* ist ein Verfahren, dass die Knoten eines DAGs bezüglich dieser Vorgabe sortiert (siehe Vorlesung). (Topologische Sortierung ist oft ein erster Schritt eines Algorithmus der auf einem DAG arbeitet).

Repräsentation / Datenstrukturen für Graphen

Die Auswahl der Datenstruktur für die Repräsentation eines Graphen kann einen erheblichen Einfluss auf die Performance haben. Für die folgenden Überlegungen nehmen wir an, dass der Graph $G = (V, E)$, $n = |V|$ Knoten und $m = |E|$ Kanten enthält.

Die Knoten eines Graphen werden üblicherweise durch ganze, aufeinanderfolgende Zahlen beginnend bei 0 oder 1 bezeichnet. Falls für die Knoten noch weitere Informationen benötigt werden, können diese in einem oder mehreren Arrays abgespeichert werden. Um die Kanten zu speichern gibt es grundsätzlich zwei Möglichkeiten: Adjazenzlisten oder eine Adjazenzmatrix.

Adjazenzlisten

Die Adjazenzlisten-Darstellung eines Graphen $G = (V, E)$ ist im Wesentlichen ein Array von $|V|$ Listen, eine für jeden Knoten in V. Für jeden Knoten $v \in V$ enthält die Adjazenzliste $Adj[v]$ alle Knoten u für die eine Kante $(v, u) \in E$ existiert. Das heisst eine Adjazenzliste ist ganz einfach eine verkettete Liste von Knoten mit dem ein bestimmter Knoten verbunden ist. Man kann also also alle Kanten als Array von n Adjazenzlisten speichern. Bei gewichteten Graphen wird zusätzlich noch ein Gewicht in jedes Listenelement (Kante) gespeichert.

Der Vorteil von Adjazenzlisten ist, dass sie insgesamt nur $\mathcal{O}(n + m)$ Speicherplatz brauchen und dass alle Nachbarn eines Knotens sofort aufgezählt werden können.

In C++ kann ein Graph oft folgendermassen beschrieben werden

```
struct edge { int /* from, */ to; double weight; /* weitere Felder */ };
typedef vector<vector<edge> > graph;
```

Manche Algorithmen lassen sich leichter mit einer Kantenlisten-Repräsentation implementieren

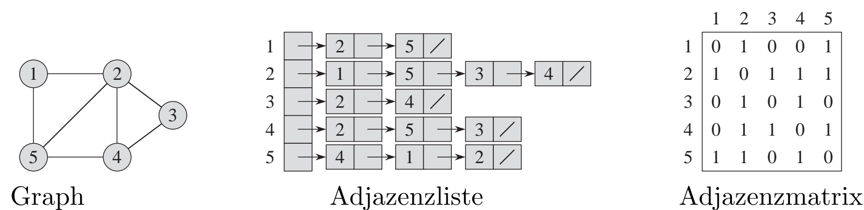
```
struct edgelist {
  vector<vector<int> > V; vector<edge> E;
  edgelist (int n=0) { V.resize (n); }
  void add (int v, edge e) { V[v].push_back (E.size()); E.push_back (e); }
};
```

Adjazenzmatrix

Die Adjazenzmatrix ist eine $n \times n$ Tabelle. Im Feld (i, j) ist bestimmt, ob eine Kante von Knoten i nach Knoten j existiert. Dies kann z.B. durch ein Boolean Wert erfolgen. Falls der Graph gewichtet ist, kann man hier auch das Gewicht (als Integer oder Floating Point Wert) der Kante reinschreiben. Bei ungerichteten Graphen ist diese Matrix symmetrisch. Die Adjazenzmatrix ist dann von Vorteil, wenn es sehr viele Kanten gibt, oder wenn schnell entschieden werden muss, ob eine Kante von i nach j existiert bzw. welches Gewicht diese Kante hat.

Es gibt natürlich auch noch andere Möglichkeiten Graphen zu repräsentieren (z.B. implizit durch eine verlinkte Objektstruktur, ein Adjazenz-Baum, ...). Die oben beschriebenen Varianten sind jedoch sehr praktisch um viele verschiedene Graphenprobleme für allgemeine Graphen zu lösen.

Beispiel:



Vergleich

Vergleich	Gewinner
Testen ob Kante $(x, y) \in E$, d.h. ob x und y verbunden sind	Adjazenzmatrix $\Theta(1)$ vs. $\Theta(d)$
Bestimme Grad eines Knotens	Adjazenzlisten
Weniger Speicher auf kleinen/sparse Graphen	Adjazenzlisten $(m+n)$ vs. (n^2)
Weniger Speicher auf grossen/dichten Graphen	Adjazenzmatrix (ein kleiner Gewinn)
Kante löschen	Adjazenzmatrix $\mathcal{O}(1)$ vs. $\mathcal{O}(d)$
Schneller zum Traversieren des Graphen	Adjazenzlisten $\Theta(n+m)$ vs. $\Theta(n^2)$
Besser für viele Probleme	Adjazenzlisten

Adjazenzlisten sind oft die bessere Wahl für viele Anwendungen von Graphen.

Traversierung eines Graphen

Eines der fundamentalen Graphenprobleme ist es, jede Kante / Knoten in einer systematischen Art zu besuchen, d.h. den Graphen zu traversieren. Die wesentliche Idee hinter Traversierungs-Algorithmen ist es, jeden Knoten zu markieren zum Zeitpunkt wenn er zum ersten Mal besucht wird und sich zu merken, welche Teile des Graphen noch zu explorieren sind.

Breitensuche / Breadth-First-Search (BFS)

Die Breitensuche (engl. breadth-first-search) ist ein grundlegendes Verfahren, um alle Knoten eines Graphen zu durchlaufen und der Archetypus für eine Vielzahl von wichtigen Graphenalgorithmen. Beispielsweise funktionieren Prim's Minimum-Spanning Tree Algorithmus und Dijkstra's Single-Source Shortest-Paths Algorithmus auf ähnliche Weise.

Gegeben sei ein Graph $G = (V, E)$ und ein ausgezeichneter Startknoten s. Breitensuche durchsucht systematisch alle Kanten von G um jeden Knoten zu finden, der von s aus erreichbar ist.

Das Verfahren erzeugt (evtl. implizit) einen "Breadth-First Baum" mit Wurzel s der alle Knoten enthält, die von s aus erreichbar sind. Für jeden Knoten v in diesem Baum, entspricht der Pfad im Breadth-first Baum dem "kürzesten Pfad" (d.h. mit den wenigsten Kanten) von s nach v in G.

Breadth-first Suche heisst so, da das Verfahren in die Breite sucht, d.h. zuerst alle Knoten mit "Abstand" k von s besucht, bevor es irgendeinen Knoten mit "Abstand" $k+1$ besucht.

BFS(G, s)

// Initialisierung: alle Knoten unbesucht

initialize for each $u \in V[G] \setminus \{s\}$

$\quad visited[u] = \text{FALSE}$ // unbesucht

$\quad (d[u] = \infty)$ // Distanz zum Startknoten (falls benötigt)

$\quad (p[u] = \text{NIL})$ // Vorgänger/Vater (falls benötigt)

$visited[s] = \text{TRUE}$ // Fange mit Startknoten an

$(d[s] = 0)$

$(p[s] = \text{NIL})$

$Q = \emptyset$ // Queue, der bisher entdeckte Rand des Graphen (Menge von Knoten welche noch unbesuchte Nachbarn haben)

Insert(Q, s)

while $Q \neq \emptyset$

$\quad u = \text{Pop}(Q)$

$\quad$ process vertex u as desired

$\quad$ **for** each $v \in Adj[u]$ // Betrachte jeden Nachbarn

$\quad\quad$ **if** $visited[v] = \text{FALSE}$ // Falls noch nicht besucht

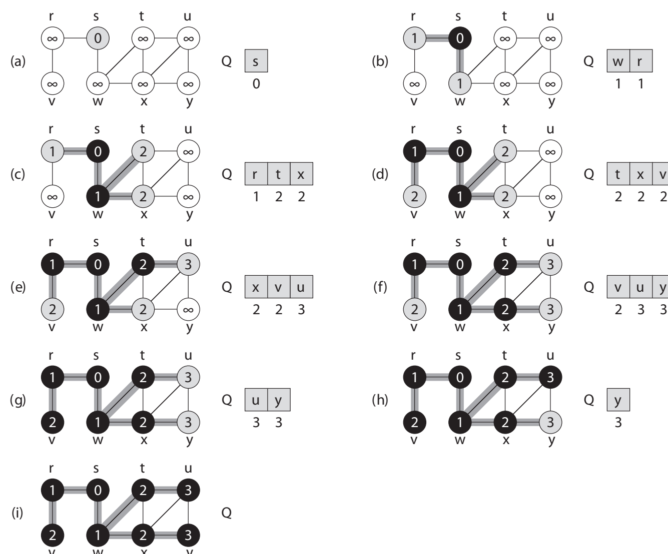
$\quad\quad\quad visited[v] = \text{TRUE}$

$\quad\quad\quad (d[v] = d[u] + 1)$

$\quad\quad\quad (p[v] = u)$

$\quad\quad\quad$ Insert(Q, s)

Beispiel für Breitensuche. In der folgenden Abbildung sind die weissen Knoten die unbesuchten, die grauen die Knoten in der Queue (die also noch ev. unbesuchte Nachbarn haben) und die schwarzen die bisher besuchten Knoten.



In C++ kann dieses Verfahren formuliert werden als

```
struct BFS {
  // Result:
  vector<double> d;
  vector<int> p;

  BFS (graph& G, int s) {
    queue<int> Q;

    d.resize (G.size(), -1); p.resize (G.size(), -1);
    d[s] = 0;
    Q.push (s);

    while (!Q.empty()) {
      int u = Q.front(); Q.pop();
      for (vector<int>::iterator e = G[u].begin(); e != G[u].end(); e++)
        if (d[e->to] == -1) {
```

```
            d[e->to] = d[u]+1;
            p[e->to] = u;
            Q.push (e->to);
        }
    }
  }
};
```

Tiefensuche / Depth-First-Search (DFS)

Der Unterschied zwischen BFS und DFS ist die Ordnung in welcher die Knoten besucht werden.

Neben der Breitensuche ist die Tiefensuche die am häufigsten verwendete Methode, einen Graphen zu durchlaufen. Anstatt wie bei der Breitensuche alle Nachbarn eines Knotens v zu markieren, läuft man zunächst möglichst "tief" in den Graphen hinein.

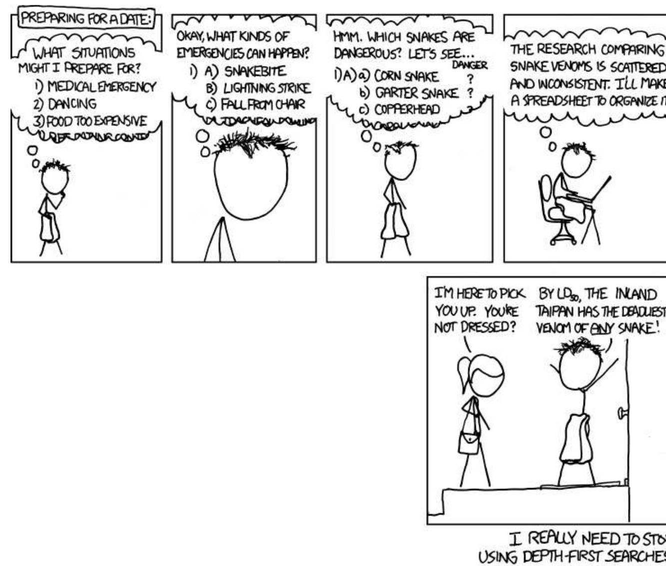
Tiefensuche kann entweder mit Hilfe eines Stacks oder vollständig rekursiv implementiert werden.

```
DFS(G)
    // Initialisierung
    for each u ∈ V[G]
        visited[v] = FALSE
        (p[u] = NIL)
    for each u ∈ V[G]
        if visited[v] = FALSE
            DFS-Visit(G, u)        // Starte Tiefensuche von diesem Knoten
```

```
DFS-Visit(G, u)
    visited[u] = TRUE
    for each v ∈ Adj[u]        // Betrachte Kante (u, v)
        if visited[v] = false
            (p[v] = u)
            DFS-Visit(G, v)
```

Beispiele, Anwendungen der Tiefensuche, Breitensuche

- (Anzahl) Zusammenhangskomponenten
- Bestimmen ob ein Graph zweifärbbar (bipartit) ist / Gibt es einen Zyklus ungerader Länge? Verwende zwei Farben. Färbe jeden Nachbarn mit der anderen Farbe als Vorgänger. Falls es zu einem Konflikt kommt $\rightsquigarrow$ Graph ist nicht zweifärbbar
- Breitensuche: Kürzesten Pfad bestimmen (falls einer existiert) in einem *ungewichteten* Graphen
- Cycle detection: Hat ein Graph einen Zyklus?
  Ein Graph hat einen Zyklus, wenn man bei der Tiefensuche zu einem Knoten gelangt, der schon besucht wurde
- Gibt es eine Brücke im Graphen? (Eine Brücke in einem Graphen ist eine Kante für die gilt: wird sie entfernt, so würde ein zusammenhängender Graph in zwei (disjunkte) Zusammenhangskomponenten zerfallen)



xkcd.com/761

Wir können Tiefensuche verwenden, um einen DAG zu linearisieren (topologisch sortieren). Dies kann in C++ formuliert werden als:

```
struct TopologicalSort {
  vector<bool> visited;
  vector<int> order;
  int k;

  TopologicalSort (graph& G) {
    order.resize (G.size());
    visited.resize (G.size(), false);
    k=0;

    for (int i=0; i<G.size(); i++) if (!visited[i]) dfs (G, i);
  }

  void dfs (graph& G, int v) {
    visited[v]=true;
    for (vector<int>::iterator e=G[v].begin(); e != G[v].end(); e++)
      if (!visited[e->to]) dfs (G, e->to);
    order[G.size()-(++k)] = v;
  }
};
```

Spannende Bäume

Ein spannender Baum zu einem zusammenhängenden Graph $G = (V, E)$ ist ein Teilgraph G' von G, mit den gleichen Knoten aber nur einer Teilmenge der Kanten. G' muss dabei ein Baum sein, das heisst er muss verbunden sein und darf keine Zyklen haben. Es gibt im allgemeinen viele verschiedene Spannbäume für einen Graphen G, so hat beispielsweise der vollständige Graph K_n mit n Knoten ganze n^{n-2} Spannbäume [Cayley 1889]. Falls der Graph G gewichtet ist, können wir einen minimalen Spannbaum (MST, minimum spanning tree) definieren als einen Spannbaum, dessen Kantengewichtssumme minimal ist (im allgemeinen gibt es mehr als einen MST). Die Standardalgorithmen für MST sind Kruskals und Prims Algorithmus.

Kruskal

Starte mit dem Graph T, der keine Kanten hat. Solange T kein Spannbaum ist, füge die billigste Kante hinzu, die zwei verschiedene Zusammenhangskomponenten von T verbindet. Dazu wird eine Union-Find Struktur benötigt, welche es erlaubt effizient testen, ob zwei Knoten in der selben Zusammenhangskomponente sind.

KRUSKAL

Input: gewichteter Graph $G = (V, E, w)$
Output: MST $T = (V, E')$

Union Find U // Anfangs jeder Knoten ein eigenes Set.
// Die Sets sind die aktuellen Zusammenhangskomponenten.
$E' = \emptyset$
Sortiere Kanten aufsteigend nach Gewicht
for each $e = \{u, v\} \in E$
  if $U.\text{FIND}(u) \neq U.\text{FIND}(v)$
    $E' = E' \cup \{e\}$
    $U.\text{UNION}(u, v)$

Der Sortierschritt braucht $\Theta(|E| \cdot \log(|E|)) = \Theta(|E| \cdot \log(|V|))$ Zeit. Die Union-Find Abfragen brauchen insgesamt $O(|E| \cdot \alpha(|V|))$[1] Zeit, falls sie mit Union-by-Size und Path-Compression implementiert sind.

Prim

Statt wie bei Kruskal immer mehr Kanten hinzuzunehmen, machen wir nun etwas ähnliches mit den Knoten. Wir bauen den MST also iterativ auf und halten uns eine Menge V' von Knoten, die schon im MST drin sind. In jedem Schritt kommt ein neuer Knoten aus $V \setminus V'$ hinzu, beginnen können wir bei einem beliebigen Knoten (am Schluss müssen ja sowieso alle drin sein).

PRIM

Input: gewichteter Graph $G = (V, E, w)$
Output: MST $T = (V, E')$

$E' = \emptyset$
$V' = \{v_1\}$ // v_1 ist ein beliebiger Knoten aus V.
while $V' \neq V$
  $e =$ billigste Kante, die ein Ende in V' und das andere in $V \setminus V'$ hat.
  $E' = E' \cup \{e\}$
  $V = \{v\}$

Dabei ist der schwierige Schritt jeweils die billigste Kante zu finden um einen Knoten zu V' hinzuzufügen (Zeile 3). Im Idealfall wird dafür ein (Min-)Fibonacci-Heap mit allen Kandidaten verwaltet. Das heisst zu jedem Zeitpunkt sind im Fibonacci-Heap alle Knoten, die noch nicht in V' sind aber durch das Hinzufügen einer einzigen Kante dazugenommen werden können. Der Wert eines Knoten v im Fibonacci-Heap ist jeweils das Gewicht der (billigsten) Kante, die den Knoten mit V' verbindet.

Sobald dann ein Knoten v neu zu V' hinzukommt, müssen alle seine Nachbarn, die noch nicht in V' sind in den Fibonacci-Heap eingefügt werden (Insert-Operation) oder ihr Wert im Fibonacci-Heap verkleinert werden, falls via dem neuen Knoten v eine kürzere Verbindung zu V' existiert (Decrease-Key Operation). Das heisst im ganzen werden $|V|$ Insert-, $|V|$ Extract-Min- und bis zu $|E|$ Decrease-Key-Operationen ausgeführt. Die Kosten dafür sind in $O(|E| + |V| \cdot \log(|V|))$ (think about it!).

Eine einfachere Implementierung verwendet statt einem Fibonacci-Heap einen normalen Heap und fügt die Kandidaten-Knoten mehrmals in den Heap ein. Da der Heap sortiert ist, kommt automatisch die "billigste" Kopie der Knoten zuerst und die folgenden können ignoriert werden. Dabei vergrössert sich der Heap auf $|E|$ Elemente (mit Fibonacci-Heap waren es nur $|V|$) und statt den $|E|$ Decrease-Key-Operationen haben wir nun $|E|$ Insert-Operationen was auf eine Laufzeit von $O((|E| + |V|) \cdot \log(|V|))$ hinausläuft (think about it!).

Eine mögliche Implementierung in C++: [2]

[1] Dabei beschreibt α die Inverse Ackermannfunktion und ist für Werte unter 10^{80} kleiner als 5 (also praktisch konstant).

[2] Dabei wird nur das Gewicht eines MST gefunden (erweiterbar um einen MST auszugeben (think about it!)). Benutzt Adjazenzliste (notwendig für $O((|E|+|V|) \cdot \log(|V|))$ Laufzeit! Mit Adjazenzmatrix nur $O(|V|^2)$ möglich.

```
struct Prim {
  double weight;

  Prim (graph& G) {
    priority_queue<pair<double, int>,vector<pair<double, int> >,
      greater<pair<double, int> > > Q; //min-heap <gewicht, knoten>

    weight = 0;
    vector<bool> visited (G.size(), false);
    Q.push (make_pair (0, 0)); // beliebiger startknoten mit gewicht 0

    while (!Q.empty()) {
        double w = Q.top().first; int u = Q.top().second;
        if (visited[u]) continue;
        visited[u]=true; weight += w;
        for (vector<int>::iterator e = G[u].begin(); e != G[u].end(); e++)
          Q.push (make_pair (e->weight, e->to));
    }
  }
};
```

Kürzeste Wege

Das Problem einen kürzesten Weg in einem gewichteten Graphen zu finden ist ein sehr wichtiges (Routenplanner, etc.) und eng mit MST verbunden. Gegeben ist ein gewichteter Graph $G = (V, E, w)$ sowie ein Start- und ein Endknoten v_s, v_e und wir suchen den kürzesten Weg von v_s nach v_e. Zwei wichtige Algorithmen für kürzeste Wege sind Bellman-Ford und Dijkstra.

Dijkstra

Wir verwenden die gleiche Idee wie bei Prim's MST Algorithmus: wir halten ein Set V' von Knoten, für welche wir den kürzesten Weg startend bei v_s schon wissen (am Anfang gilt $V' = \{v_s\}$, der kürzeste Weg von v_s zu sich selbst hat Länge 0). In einem Heap halten wir die jeweiligen Kandidaten, die wir zu V' hinzufügen möchten, dabei ist ihr Gewicht die Länge des kürzesten Weges von v_s aus (bei Prim war es die Länge zu irgend einem Knoten in V'). Die Laufzeitanalyse ist völlig äquivalent zu derjenigen von Prim[3] und auch die Implementierung ist fast identisch:

```
struct Dijkstra {
  vector<double> d; // distanz von start aus

  Dijkstra (graph& G, int s, int t=-1) {
    priority_queue<pair<double, int>,vector<pair<double, int> >,
      greater<pair<double, int> > > Q; //min-heap <gewicht, knoten>

    d.resize (G.size(), -1);

    vector<bool> visited (G.size(), false);
    Q.push (make_pair (0, s));

    while (!Q.empty()) {
        double w = Q.top().first; int u = Q.top().second;
        if (visited[u]) continue;
        visited[u]=true;
        d[u]=w;
        for (vector<int>::iterator e = G[u].begin(); e != G[u].end(); e++)
          Q.push (make_pair (w+e->weight, e->to));
    }
  }
};
```

[3] Also $O(|E| + |V| \log(|V|))$ mit und $O((|E| + |V|) \log(|V|))$ ohne Fibonacci-Heap.

Eine Darstellung und kurze Beweise der Korrektheit von Dijkstras und den beiden MST Algorithmen finden sich in
http://www.cs.washington.edu/education/courses/cse521/10wi/Greedy.pdf
(http://goo.gl/fKQXa)

Bellman-Ford

Die Idee ist durch Dynamische Programmierung kürzeste Wege von dem Startknoten aus zu finden, welche immer mehr Kanten verwenden. Am Anfang ist die Distanz zu allen Knoten unendlich (ausser diejenige des Startknoten selbst, welche auf 0 gesetzte wird). Damit haben wir schon alle kürzesten Wege, die maximal null Kanten brauchen. Im nächsten Schritt suchen wir alle kürzesten Wege von dem Startknoten aus, die maximal eine Kante brauchen: das sind einfach die vom Startknoten ausgehenden Kanten.

Im Allgemeinen haben wir alle kürzesten Wege, die höchstens $i-1$ Kanten verwenden gefunden und wollen nun alle kürzesten Wege mit höchstens i Kanten berechnen. Dazu müssen wir lediglich alle Kanten $e = (u, v)$ durchgehen: Wenn wir zuvor mit höchstens $i-1$ Kanten und Gewicht x von dem Startknoten bis zum Knoten u gelangen konnten, ist es nun möglich durch die Kante e mit maximal i Kanten von dem Startknoten mit Gewicht $x+w(e)$ bis zum Knoten v zu gelangen.

Wir müssen höchstens $|V|-1$ solche Schritte ausführen, da kein kürzester Weg einen Knoten zwei mal besuchen wird. Ein Schritt braucht konstante Zeit pro Kante, also läuft Bellman-Ford in $O(|V| \cdot |E|)$.

Ein grosser Vorteil von Bellman-Ford ist, dass er auch bei negativen Kantengewichten funktioniert (im Gegensatz zu Dijkstra!). Jedoch darf der Graph keine negativen Zyklen haben.

Eine Implementierung in C++:

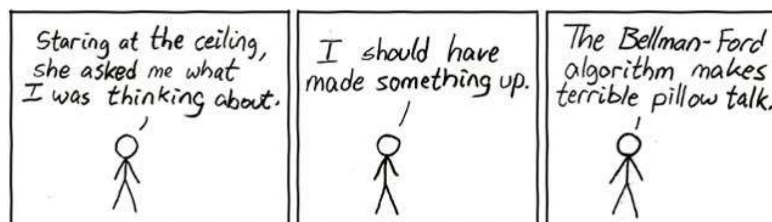
```cpp
struct BellmanFord {
  vector<double> d;
  vector<int> p;
  bool negativeCycle;

  BellmanFord (graph& G, int s) {
    p.resize (G.size(), -1); d.resize (G.size(), DINF); d[s]=0;

    for (int i=1; i<G.size(); i++) // paths with n-1 or fewer edges
      for (int j=0; j<G.size(); j++)
        for (vector<int>::iterator e = G[j].begin(); e != G[j].end(); e++)
        if (d[e->to] > d[j] + e->weight) {
          d[e->to] = d[j] + e->weight;
          p[e->to] = j;
        }

    // detect negative cycle
    for (int j=0; j<G.size(); j++)
      for (vector<int>::iterator e = G[j].begin(); e != G[j].end(); e++)
      if (d[e->to] > d[j] + e->weight) {
        negativeCycle = true;
        return;
      }

    negativeCycle = false;
  }
};
```



xkcd.com/69