Datenstrukturen & Algorithmen 29. April, 2015

Handout 9

Sebastian Millius, Sandro Feuz, Daniel Graf

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

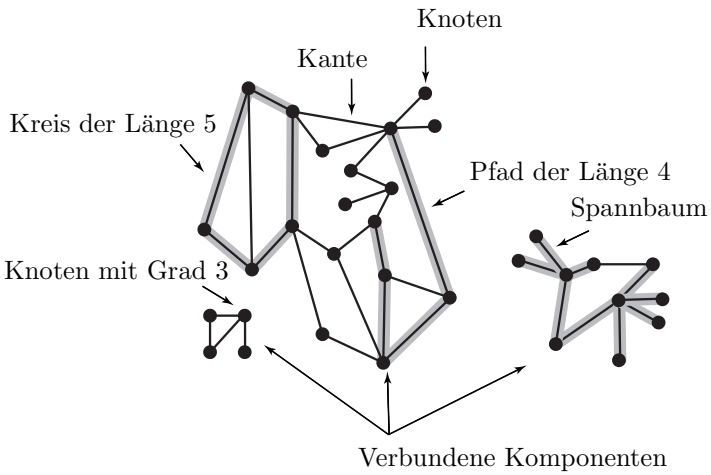
Links

- 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 2015: http://goo.gl/Wi1LiA

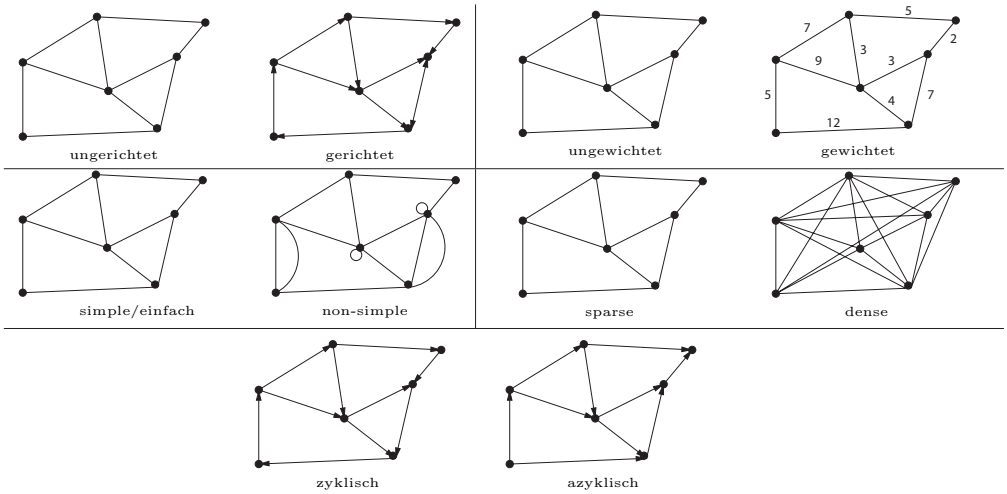
Graphen

Paarweise Verbindungen zwischen einzelnen Elementen spielen eine zentrale Rolle in vielen Gebieten der Informatik. Der Begriff des Graphen, der dies modelliert, ist fundamental in der Informatik. Viele grundlegende Probleme lassen sich als Graphenprobleme modellieren.

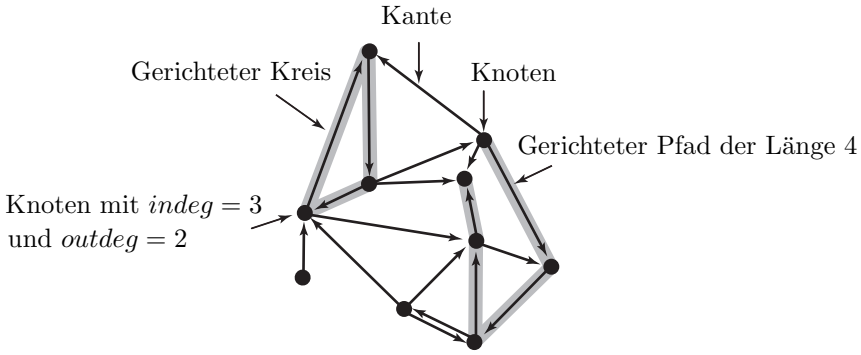
Ein Graph $G = (V, E)$ besteht aus einer (endlichen) *Knoten*menge V (engl. vertices) und einer Menge $E \subseteq \binom{V}{2}$ (engl. edges) von geordneten (bei einem *gerichteten* Graphen) oder ungeordneten (bei einem *ungerichteten* Graphen) Paaren von Konten aus V, genannt *Kanten*. Die Kanten repräsentieren die Beziehungen zwischen den Knoten.



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 können durch ganze, aufeinanderfolgende Zahlen beginnend bei 0 oder 1 bezeichnet werden. 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 häufig gebrauchte 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 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 für jede Kante benötigt.

Der Vorteil von Adjazenzlisten ist, dass sie insgesamt nur $\mathcal{O}(n + m)$ Speicherplatz brauchen und dass alle Nachbarn eines Knotens schnell aufgezählt werden können. Dafür ist es aufwändiger zu überprüfen, ob zwei Knoten benachbart sind.

In C++ kann ein Graph oft folgendermassen beschrieben werden

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

Für einen Graphen G gibt hier $G[i]$ die Adjazenzliste des $i$ten Knoten an. Eine Kante (v, u) kann eingefügt werden mit

```
G[v].push_back((edge){u});
G[u].push_back((edge){v});  // nur bei ungerichteten Graphen
```

Bei ungerichteten Graphen repräsentieren wir jede Kante als ein Paar zweier gerichteter Kanten.

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 addEdge (int v, int u) {
    V[v].push_back(E.size());
    V[u].push_back(E.size());  // nur bei ungerichteten Graphen
    E.push_back ((edge){v, u});
  }
};
```

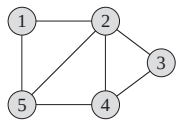
Der Vorteil mit dieser Darstellung ist, dass alle Kanten einfach enumeriert werden können.

Adjazenzmatrix

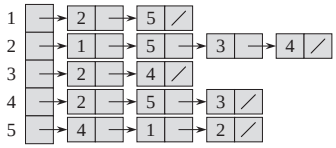
Die Adjazenzmatrix ist eine $n \times n$ Tabelle. Im Feld (i, j) ist bestimmt, ob eine Kante von Knoten i nach Knoten j existiert (z.B. $A[i, j] := [(i, j) \in E]$). Dies kann z.B. durch ein Boolean Wert erfolgen. Falls der Graph gewichtet ist, kann man hier auch das Gewicht (als Integer oder Floating 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:



Graph



Adjazenzliste

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Adjazenzmatrix

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.

Die Laufzeit (think about it!) sowohl von Breiten- als auch Tiefensuche ist $\mathcal{O}(m)$.

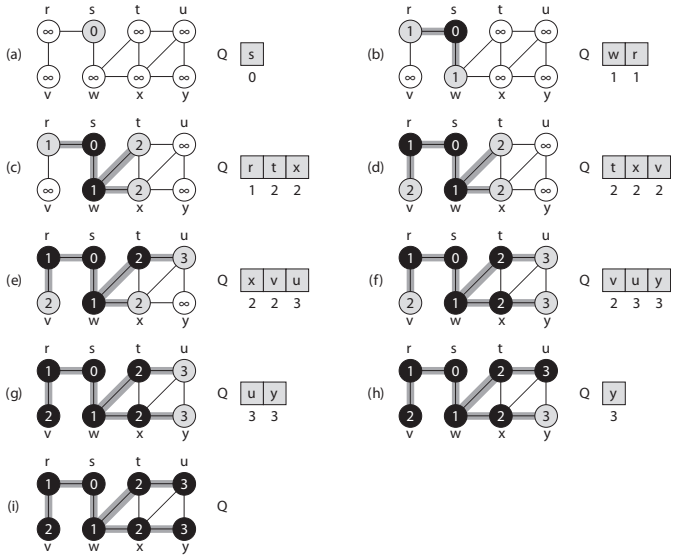
BFS(G, s)

// Initialisierung: alle Knoten unbesucht
initialize for each $u \in V[G] \setminus \{s\}$
    $visited[u]$ = FALSE // unbesucht
    $(d[u] = \infty)$ // Distanz zum Startknoten (falls benötigt)
    $(p[u] =$ NIL$)$ // Vorgänger/Vater (falls benötigt)
$visited[s]$ = TRUE // Fange mit Startknoten an
$(d[s] = 0)$
$(p[s] =$ NIL$)$
$Q = \emptyset$ // Queue, der bisher entdeckte Rand des Graphen
(Menge von Knoten welche noch unbesuchte Nachbarn haben)

INSERT(Q, s)

while $Q \neq \emptyset$
    $u =$ POP(Q)
    process vertex u as desired
    for each $v \in Adj[u]$ // Betrachte jeden Nachbarn
        if $visited[v]$ = FALSE // Falls noch nicht besucht
            $visited[v]$ = TRUE
            $(d[v] = d[u] + 1)$
            $(p[v] = u)$
            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.



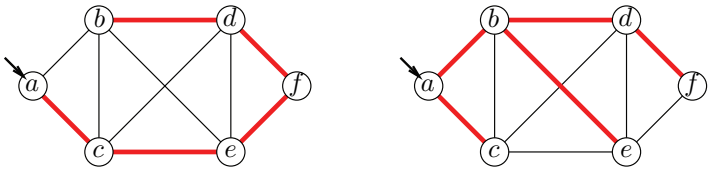
In C++ kann dieses Verfahren formuliert werden als

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

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

    // Initialization.
    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 (auto const &e : G[u])
```

```
        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.



Ein DFS (links) und BFS (rechts) Spannbaum, startend bei Knoten a.

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 $u \in V[G]$
    $visited[v]$ = FALSE
    ($p[u]$ = NIL)
  for each $u \in V[G]$
    if $visited[v]$ = FALSE
      DFS-VISIT(G, u) // Starte Tiefensuche von diesem Knoten

DFS-VISIT(G, u)
  $visited[u]$ = TRUE
  for each $v \in Adj[u]$ // Betrachte Kante (u, v)
    if $visited[v] = false$
      ($p[v] = u$)
      DFS-VISIT(G, v)

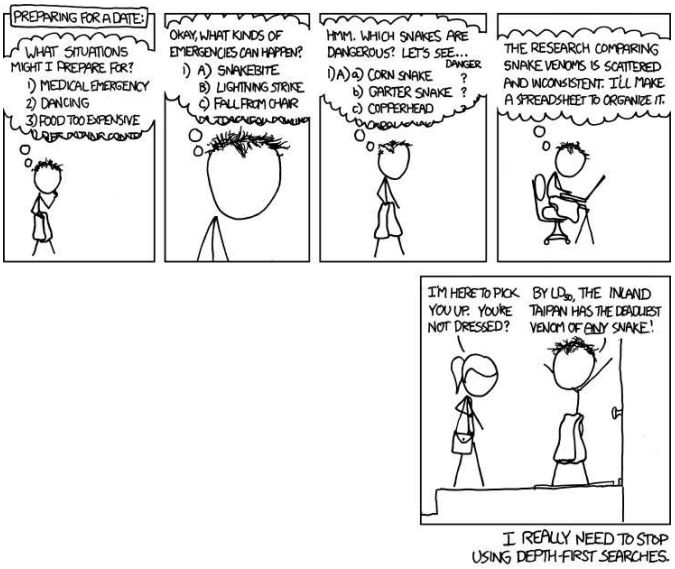
Dies kann in C++ formuliert werden als:

```
struct DFS {
  vector<bool> visited;
  vector<int> p;

  DFS(graph& G, int s) {
    visited.resize(G.size(), false);
    p.resize(G.size(), -1);
    dfs(G, s);
  }

  void dfs(graph& G, int v) {
    visited[v] = true;
    for (auto const& e : G[v])
      if (!visited[e.to]) {
        p[v] = e.to;
        dfs(G, e.to);
```

```
        }
    }
};
```

Beispiele, Anwendungen der Tiefensuche, Breitensuche

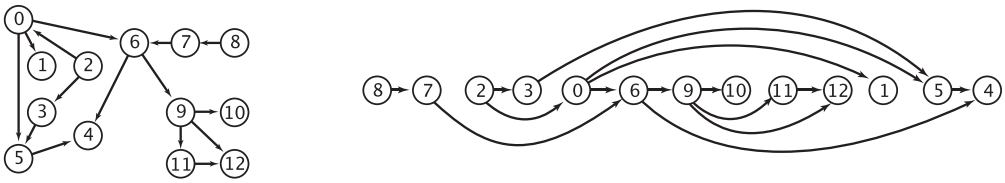
- (Anzahl) Zusammenhangskomponenten
- Bestimmen ob ein Graph zweifärbbar (bipartit) ist / Gibt es einen Zyklus ungerader Länge? Verwende zwei Farben. Färbe jeden Nachbarn mit der anderen Farbe als Vorgänger. Falls es zu einem Konflikt kommt $\rightsquigarrow$ Graph ist nicht zweifärbbar
- Breitensuche: 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

Topologische Sortierung

Eine *Topologische Sortierung* eines gerichteten Graphen ist eine Linearisierung seiner Knoten, so dass jeder Knoten vor seinen Kindern kommt. Z.B taucht dies oft in Scheduling Problemen auf, wo beispielsweise eine Kante (x, y) bedeutet, dass eine Aktivität/Event x vor y eintreten muss.



Ein Graph (links) und eine seiner möglichen typologische Sortierungen.

Ein Graph kann topologisch sortiert werden, wenn er keinen Zykus/Kreis enthält - der Graph ist ein DAG.

Wir können Tiefensuche verwenden (Tarjan 1976), um einen DAG zu linearisieren, da die *Post-Order* Durchlaufung jeden Knoten nach seinen Kindern aufzählt. Dies kann in C++ formuliert werden als:

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

  TopologicalSort(graph& G) {
    // Initialisierung.
    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 (auto const& e : G[v])
      if (!visited[e.to]) dfs(G, e.to);
    order[G.size()-(++k)] = v;
  }
};
```

Ein anderer Algorithmus (*Entfernung von Elementen ohne Vorgänger*), zuerst beschrieben durch Kahn (1962), beginnt mit einer Liste von *Startknoten* (Knoten, die keine einkommenden Kanten haben, also keine Vorgänger haben). Mindestens ein solcher Knoten existiert zu Beginn wenn der Graph azyklisch ist. Dann werden jeweils Knoten ohne Vorgänger entfernt, bis keine Knoten mehr übrig sind.

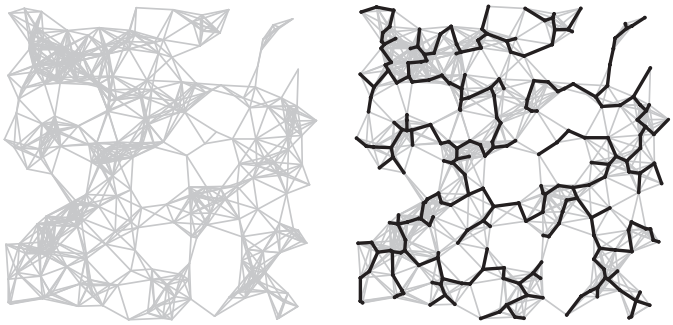
Dies kann in C++ formuliert werden als:

```
struct TopologicalSort {
  vector<int> order;

  TopologicalSort(graph& G) {
    // Berechne Eingangsgrad.
    vector<int> indeg = vector<int>(G.size(), 0);
    for (int v=0; v<G.size(); v++) for (auto const& e : G[v]) ++indeg[e.to];

    // Sources.
    queue<int> S;
    for (int v=0; v<G.size(); v++) if (indeg[v] == 0) S.push(v);

    while (!S.empty()) {
      int v = S.front(); S.pop();
      order.push_back(v);

      // Entferne Knoten.
      for (auto const &e : G[v]) {
        --indeg[e.to];

        // Neuer Startknoten?
        if (indeg[e.to] == 0)
          S.push(e.to);
      }
    }
  }
};
```

Unix Betriebssysteme besitzen oft ein Programm namens *tsort*, das eine topologische Sortierung durchführt. Es war früher nötig, um übersetzte Objektdateien, die voneinander abh ängen, in korrekter Reihenfolge in eine Programmbibliothek einzufügen

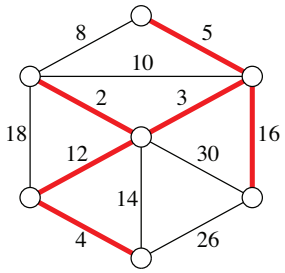
Spannende Bäume

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].



Ein Graph und ein Spannbaum.

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).

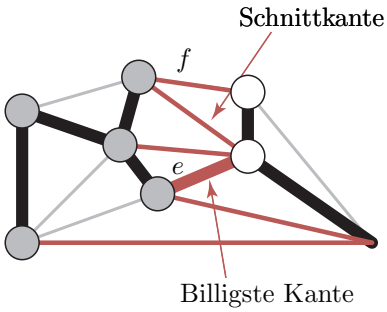


Ein gewichteter Graph und sein minimaler Spannbaum.

Es gibt verschiedene Algorithmen, um einen MST für einen Graphen zu berechnen; die bekanntesten sind Kruskals und Prims Algorithmus. Beide folgen dem gleichen Ansatz: sie bauen einen azyklischen Teilgraphen (nennen wir in F) des Eingabegraphen G auf. F ist dabei ein Teilgraph eines minimalen Spannbaumes von G und jede verbundene Komponente von F ist ein minimaler Spannbaum seiner Knoten (think about this!).

Zu Beginn besteht F aus n verschiedenen 1-Knoten Bäumen. Sukzessive werden Kanten zwischen diesen Knoten eingefügt. Am Ende ist F ein einziger n-Knoten Baum - ein minimaler Spannbaum von G. Natürlich muss aufgepasst werden, *welche* Kante in jedem Schritt hinzugefügt wird.

Die beiden Algorithmen nutzen dabei die sogenannte *Schnitt-Eigenschaft (cut property)* aus (vgl. Vorlesung): Ein *Schnitt* (engl. cut) eines Graphen ist eine Aufteilung seiner Knoten in zwei disjunkte nichtleere Mengen. Eine *Schnittkante* (engl. crossing edge) eines Schnittes ist eine Kante die einen Knoten einer Seite mit einem Knoten der anderen Seite verbindet. Dabei gilt nun, dass für jeden Schnitt von G und jede billigste Schnittkante, es einen minimalen Spannbaum gibt, der diese enthält. Eine solche Kante heisst *sicher* (engl. safe).



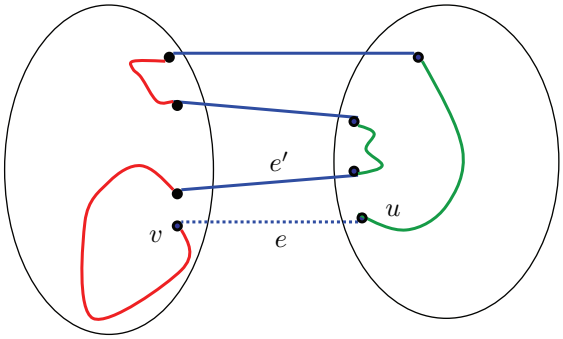
Prim's und Kruskals Algorithmus bauen einen minimalen Spannbaum jeweils aus sicheren Kanten auf, unterscheiden sich aber darin, welche Schnitte sie betrachten.

Prim's Algorithmus folgt der Schnitt-Eigenschaft in dem er den Spannbaum von einen Startknoten s aufbaut. In jedem Schritt wird jeweils die billigste Kante gewählt, den bisherigen aufgebauten Teilbaum F verlässt (d.h. eine Seite des Schnittes ist der bisherige Teilbaum, die andere Seite der Rest des Graphen).

Kruskal's Algorithmus wählt dagegen immer die billigste Kante, die irgendeine Komponente des bisherigen Teilgraphen F verlässt.

Beweis der Korrektheit der Schnitt-Eigenschaft:

Betrachte einen Schnitt des Graphen G. Sei $e = \{v, u\}$ eine billigste Kante. Nehme an, dass kein minimaler Spannbaum eine billigste Schnittkante enthält, insbesondere also e nicht.



Sei T ein MST von G. Die Endknoten von e sind verbunden in T (da T ein Spannbaum ist) durch einen Pfad (sagen wir P). Sei e' die erste Schnitt-Kante in P (d.h. die Kante, mit der erstmals in P der Schnitt überquert wird). Dann ist $T' := (T \setminus \{e'\}) \cup \{e\}$ ein Spannbaum geringeren Gewichts:

- $w(e) < w(e')$ per Annahme.
- T' ist verbunden: Fall ein Pfad $e' = (v', u')$ braucht, dann gehe von v' nach v, dann nehme e und gehe dann von u nach u'.
- T' ist azyklisch: $T \cup \{e\}$ hat nur einen Kreis (mit e und e'). Mit e' fällt der Kreis auseinander.

Dies ist ein Widerspruch zur Annahme.

Kruskal

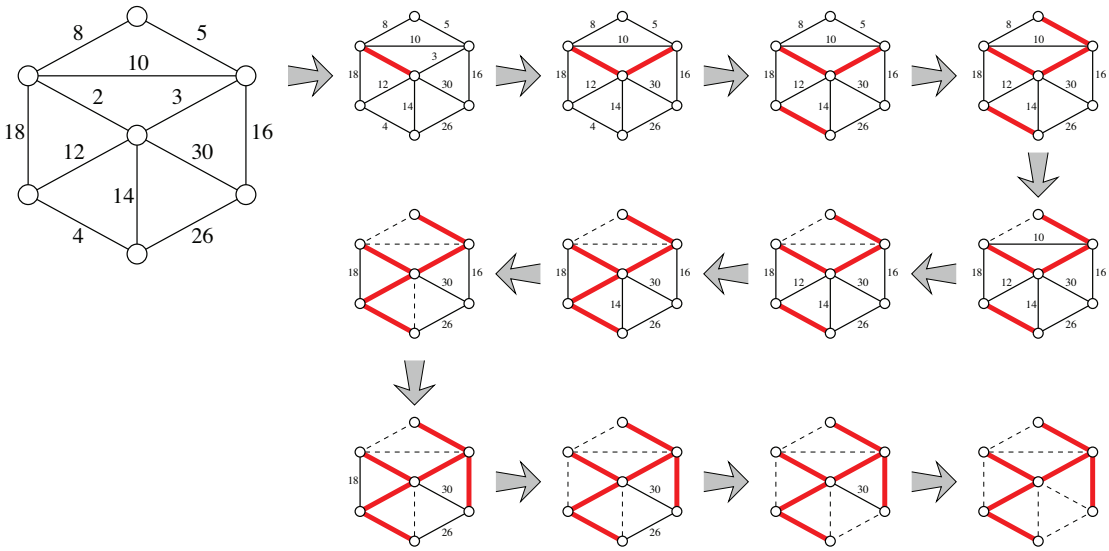
Starte mit dem Graph T, der keine Kanten hat. Solange T kein Spannbaum ist, füge die billigste Kante hinzu, die zwei verschiedene Zusammenhangskomponenten von T verbindet. Dazu wird eine Union-Find Struktur benötigt, welche es erlaubt effizient testen, ob zwei Knoten in der selben Zusammenhangskomponente 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.FIND$(u) \neq U$.FIND(v)
    $E' = E' \cup \{e\}$
    U.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.



Kruskal Algorithmus auf einem Beispielsgraphen. Die fett markierten Kanten sind im minimalen Spannbaum, die gestrichten Kanten wurden verworfen.

Mit Hilfe einer Union-Find Struktur (siehe nächster Abschnitt) kann Kruskal in C++ implementiert werden als:

```
struct Kruskal {
  // Der MST.
  vector<edge> T;
  double weight;

  Kruskal(edgelist& G) {
    // Initialisierung.
    UnionFind U(G.size());
    vector<edge> E = G.E;
    weight = 0;

    // Sortiere Kanten.
    sort(E.begin(), E.end(), [](const edge& a, const edge& w) {
        return a.weight < b.weight; });

    for (auto e : E) {
      // Nicht in gleicher Komponente?
      if (U.unite(e.from, e.to)) {
        T.push_back(e);
        weight += e.weight;
```

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

```
        }
      }
    }
};
```

Union-Find Struktur

Wir können mit einer *Union-Find-Datenstruktur* eine Partition einer Menge modelieren. Sie eignen sich z.B. gut, um Zusammenhangskomponenten eines Graphen zu verwalten - beispielsweise in Kruskal's Algorithmus.

Jede Menge hat dabei einen *Repräsentanten* der die Menge identifiziert (da wir eine Partition, also disjunkte Mengen, betrachten, kann kein Element der Repräsentant von mehr als einer Menge sein).

Unterstützte Operationen

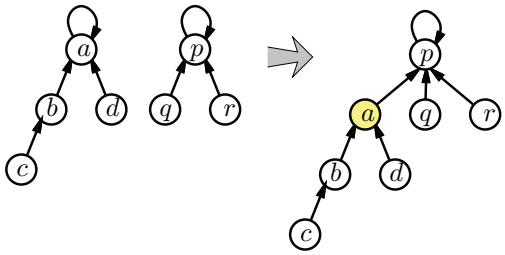
- MakeSet(x): Erzeuge eine neue Menge $\{x\}$. Der Repräsentant dieser Menge ist x.
- Find(x): Finde den Repräsentanten der Menge die x enthält.
- Union(x, y): Vereinige die Mengen die x und y enthalten.

Um zu entscheiden ob zwei Elemente x, y in der gleichen Menge sind, können die Repräsentanten verglichen werden: Find(x) = Find(y)?

Eine einfache Möglichkeit ist es die Mengen durch Bäume zu repräsentieren, bei welchen jeder Knoten ein Element der Menge repräsentiert. Jeder Knoten zeigt auf seinen Vorgänger (engl. parent); ausser dem Repräsentanten der Menge der auf sich selbst zeigt (also die Wurzel des Baumes ist).

MakeSet erzeugt einen einzelnen Knoten der auf sich selbst zeigt. In Find wandern wir den Baum hoch bis zur Wurzel um den Repräsentanten der Menge zu finden.

Für Union lassen wir einfach den Repräsentanten der einen Menge auf den Repräsentanten der anderen Menge zeigen.

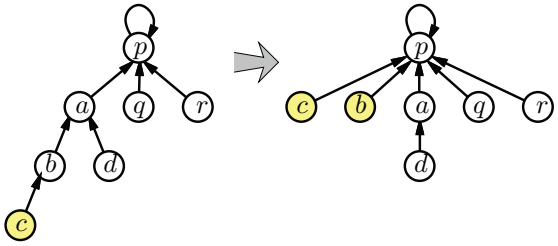


Eine Union (Vereinigungs-) - Operation.

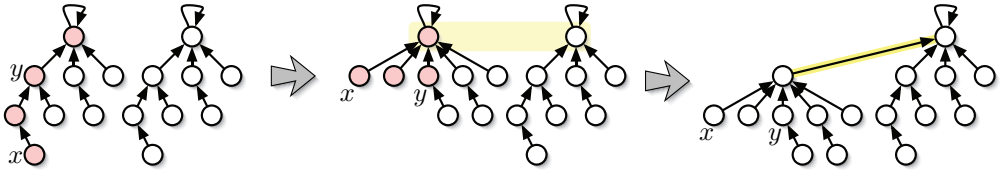
MakeSet ist damit $\Theta(1)$, Union ist $\mathcal{O}(1)$ plus zwei Find Operationen. Da die Bäume zu linearen Listen degenerieren können (think about it! Was wäre eine Beispielsequenz von Operationen?) ist Find in $\Theta(n)$.

Eine Idee ist es *Vereinigung nach Grösse* (engl. *union by size*) oder *Vereinigung nach Höhe* (engl. *union by height*) anzuwenden, bei denen jeweils der kleinere (bzw. weniger hohe) and den grösseren (bzw. tieferen) Baum angehängt wird. Man kann zeigen (vgl. Übungsstunde, Vorlesung) dass die Bäume dadurch logarithmische Höhe haben. Damit sind Find als auch Union in $\Theta(\log n)$.

Eine weitere Idee ist *Pfad - Komprimierung* (engl. *path compression*), um sowohl Find als auch Union *fast* linear zu machen. Dabei werden bei einer Find Operation, alle Knoten auf dem Pfad zur Wurzel, direkt an die Wurzel umgehänkt.



Pfad - Komprimierung.



Pfad - Komprimierung und anschliessende Union Operation.

Ein Verfahren das beide Ideen anwendet heisst *Union by Rank* (Vereinigung nach Rang). Dies ist im wesentlichen *union by height* mit *path compression*, Der Grund, warum dies Rang und nicht Höhe heisst, ist, da der Rang nur vor der Pfadkomprimierung mit der wahren Höhe übereinstimmt. Für jede Menge merken wir uns einen Rang (der zu Beginn wie die Höhe 0 ist). Wir hängen jeweils den Baum mit dem kleineren Rang an den anderen und erhöhen den Rang des Gesamtbaumes, falls beide Bäume den gleichen Rang haben. Man kann zeigen, dass damit FIND in $\mathcal{O}(\alpha(n))$ ist; wobei α die *Inverse Ackermann Funktion* bezeichnet, die extrem langsam wächst und daher für alle praktischen Belange konstant ist.

Dies kann in C++ implementiert werden als:

```
struct UnionFind {
  vector<int> u, rank;

  UnionFind(int n) {
    // Initialisierung.
    rank.resize(n, 0);
    for (int i=0; i<n: i++) u[i] = i;  // Jeder Knoten zeigt auf sich.
  }

  int find(int i) {
    // Path compression.
    if (u[i] != i) u[i] = find(u[i]);
    return u[i];
  }

  // false: Knoten sind in der gleichen Menge, true: Knoten sind neu verbunden.
  bool unite(int i, int j) {
    i = find(i); j = find(j);
    if (i != j) {
      // Union by rank.
      if (rank[i] == rank[j]) rank[i]++;
      if (rank[i] >  rank[j]) u[j] = i;
      else                    u[i] = j;

      return true;
    }
    return false;
  }
};
```

Prim

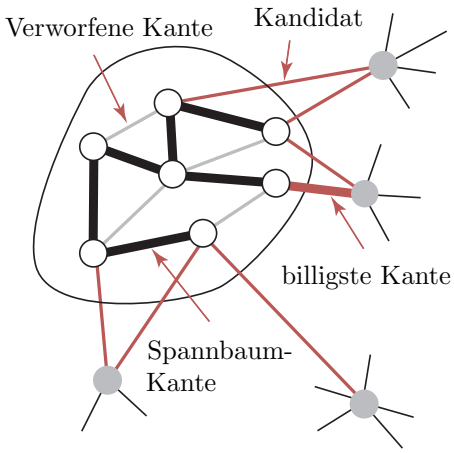
Statt wie bei Kruskal immer mehr Kanten hinzuzunehmen, machen wir nun etwas ähnliches mit den Knoten. Wir bauen den MST also iterativ auf und halten uns eine Menge V' von Knoten, die

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!).

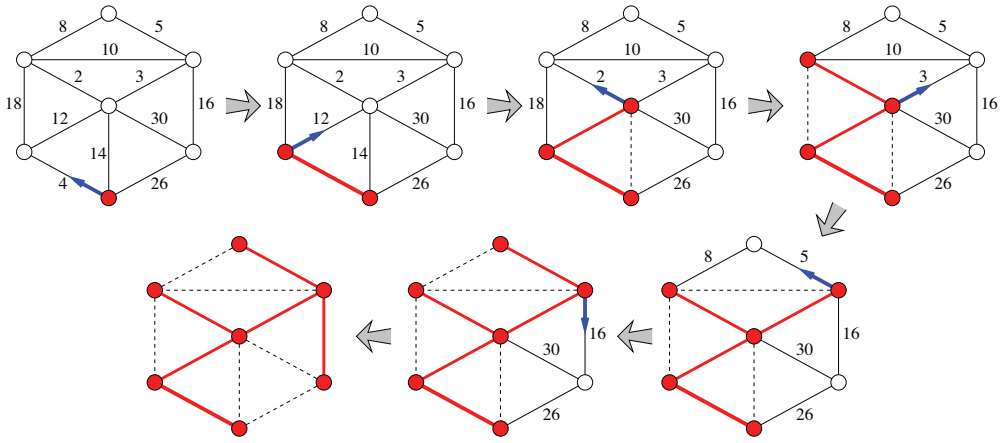
Eine einfachere Implementierung verwendet statt einem Fibonacci-Heap einen normalen Heap und fügt die Kandidaten-Knoten mehrmals in den Heap ein. Da der Heap sortiert ist, kommt automatisch die "billigste" Kopie der Knoten zuerst und die folgenden können ignoriert werden. Dabei vorgrössert sich der Heap 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]

```
struct Prim {
  // Der MST.
  vector<edge> T;
  double weight;

  Prim (graph& G) {
    // Initialisierung.
    vector<int> parent(G.size(), -1);
    vector<int> dist(G.size(), DOUBLE_MAX);   // Infinity.
```

[2] Benutzt Adjazenzliste (notwendig für $O((|E| + |V|) \cdot \log(|V|))$ Laufzeit! Mit Adjazenzmatrix nur $O(|V|^2)$ möglich.

```
    vector<bool> visited (G.size(), false);

    priority_queue<pair<double, int>,vector<pair<double, int> >,
      greater<pair<double, int> > > Q ; // Min-heap <gewicht , knoten>

    Q.push(make_pair (0, 0)); // Beliebiger Startknoten mit Gewicht 0
    weight = 0;

    while (!Q.empty()) {
      double w = Q.top().first; int u = Q.top().second; Q.pop();

      if (visited[u]) continue;
      visited[u]=true; weight += w;

      if (parent[u] != -1)
        T.push_back((edge){parent[u], u, w});

      for (auto const& e : G[u]) {
        if (e.weight < dist[e.to]) {
          dist[e.to] = e.weight;
          parent[e.to] = u;
          Q.push(make_pair(e.weight, e.to));
        }
      }
    }
  }
};
```



Prim's Algorithmus auf einem Beispielgraphen startend mit dem untersten Knoten. In jedem Schritt sind die fett markierten Kanten im minimalen Spannbaum, gestrichte Kanten sind verworfen und der Pfeil zeigt jeweils die nächste günstigste Kante an.

Kürzeste Wege

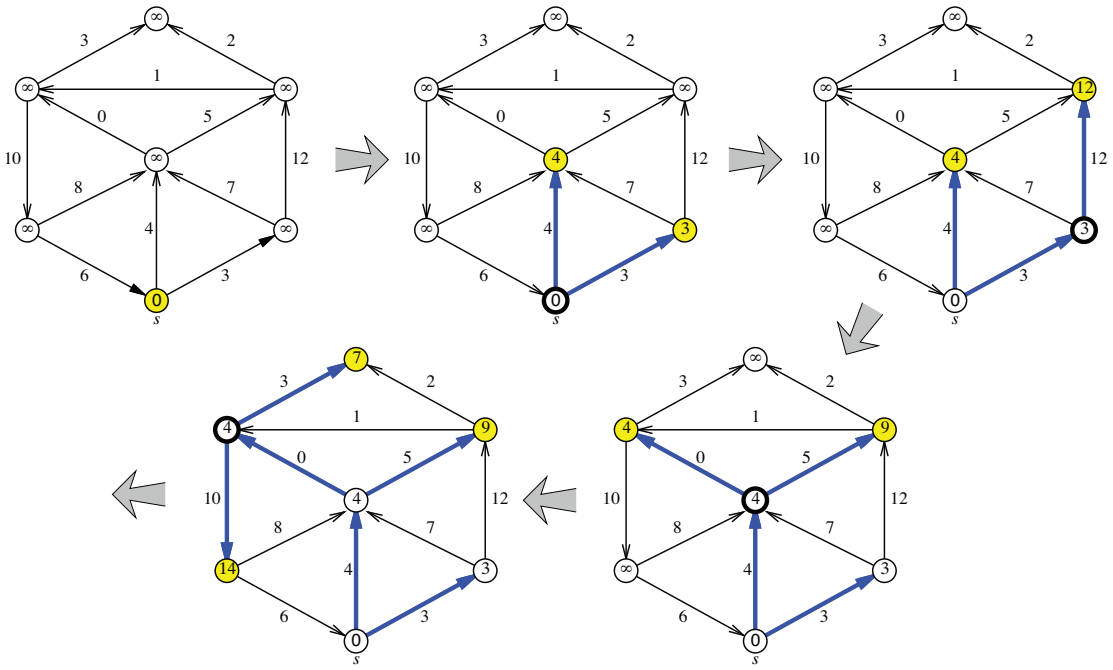
Das Problem einen kürzesten Weg in einem gewichteten Graphen zu finden ist ein sehr wichtiges (Routenplanner, etc.) und eng mit MST verbunden. Gegeben ist ein gewichteter Graph $G = (V, E, w)$ sowie ein Start- und ein Endknoten v_s, v_e und wir suchen den kürzesten Weg von v_s nach v_e. Zwei wichtige 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 (auto const& e : G[u])
        Q.push(make_pair(w+e.weight, e.to));
    }
  }
};
```

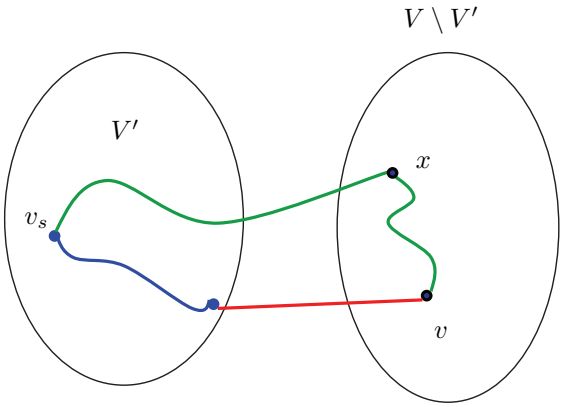


Die ersten vier Schritte von Dijkstra's Algorithmus auf einem Graphen ohne negative Kantengewichte.

Dijkstra behält eine Menge V' für Knoten, für welche der kürzeste Weg bekannt ist. Danach werden jeweils die kürzesten Pfade betrachtet, die einen Knoten des momentanen Randes erreichen (d.h. die in V' starten und mit einer letzten Kante einen Knoten aus $V \setminus V'$ erreichen). Sukzessive werden Knoten aus dem Rand in V' hinzugefügt und der Rand aktualisiert.

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

Beweis der Korrektheit: Induktionsbeweis: Nehme an dass alle Distanzen in V' korrekt sind (dies gilt zu Beginn). Der Knoten v hat den kleinsten Erreichbarkeits-Wert im Rand (dies ist die Länge des kürzesten Weges von v_s aus, wobei nur die letzte Kante V' verlässt).



Nehme an, dass v auf einem kürzeren Weg erreichbar ist. Betrachte, die erste Kante dieses Weges, die V' verlässt und sei x der erste Knoten in $V \setminus V'$. Da alle Kantengewichte positiv sind (hier wird diese Bedingung gebraucht!) ist die Distanz von x nach v nicht negativ. Da der Erreichbarkeits-Wert von x grösser oder gleich dem Erreichbarkeits-Wert von v ist, ist dieser Weg länger. Dies ist ein Widerspruch und deshalb bewahrt das Hinzufügen von v zu V' korrekte kürzeste Wege.

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.

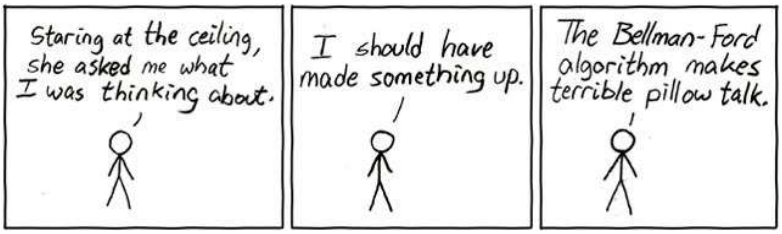
Wir müssen höchstens $|V|-1$ solche Schritte ausführen, da kein kürzester Weg einen Knoten zwei mal besuchen wird. Ein Schritt braucht konstante Zeit pro Kante, also läuft Bellman-Ford in $O(|V| \cdot |E|)$.

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++:

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

  BellmanFord (graph& G, int s) {
    p.resize (G.size(), -1); d.resize (G.size(), DOUBLE_MAX); 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 (auto const& e : G[j])
        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 (auto const& e: G[j])
      if (d[e.to] > d[j] + e.weight) {
        negativeCycle = true;
        return;
      }

    negativeCycle = false;
  }
};
```



xkcd.com/69

Brain Teaser: Zwei Gefangene sind den Händen eines Zauberers der sie freilässt wenn sie seine Aufgabe korrekt lösen. Beiden Gefangenen wird die Aufgabe erkärt und sie können sich im Vorherein absprechen, bis die Aufgabe startet. Danach gibt es nur noch die Kommunikation wie folgt:

- Der Zauberer hat ein Schachbrett mit 64 Münzen (eine auf jedem Spielfeld) in einem geschlossenen Raum hinterlegt. die Münzen sind entweder Kopf oder Zahl, in einer beliebigen Konfiguration.
- Gefangener 1 betritt den geschlossenen Raum mit dem Zauberer.
- Der Zauberer zeigt dem Gefangenen ein beliebig gewähltes Feld - das magische Feld.
- Der Gefangene darf nun genau 1 Münze umdrehen, er darf selbst wählen welche.
- Gefangener 1 geht und trifft nicht mehr auf Gefangenen 2.
- Gefangener 2 geht in den gleichen Raum, kann auf das Schachbrett schauen und muss nun wissen, welches das magische Feld ist.

Welche Strategie kann es den Gefangenen möglich machen, die Aufgabe zu lösen?