Woche 12 – Übersicht & Tricks

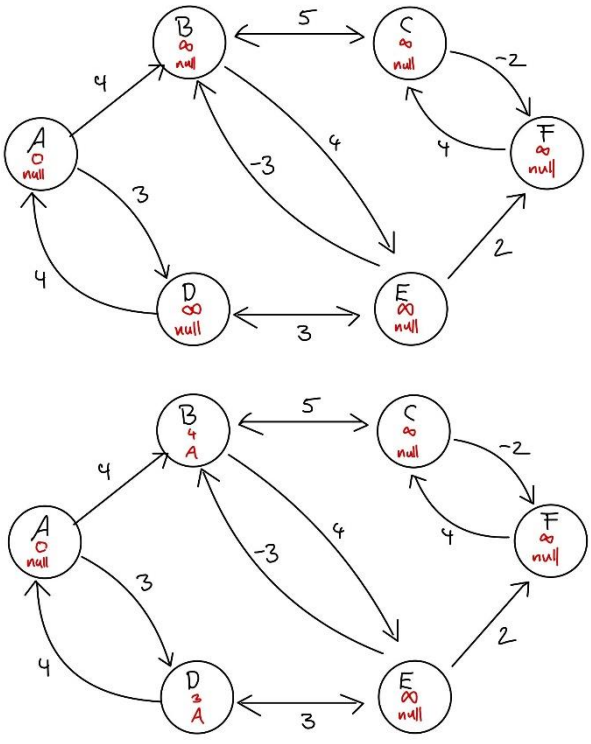
(Disclaimer: Die hier vorzufindenden Notizen haben keinerlei Anspruch auf Korrektheit oder Vollständigkeit und sind nicht Teil des offiziellen Vorlesungsmaterials. Alle Angaben sind ohne Gewähr.)

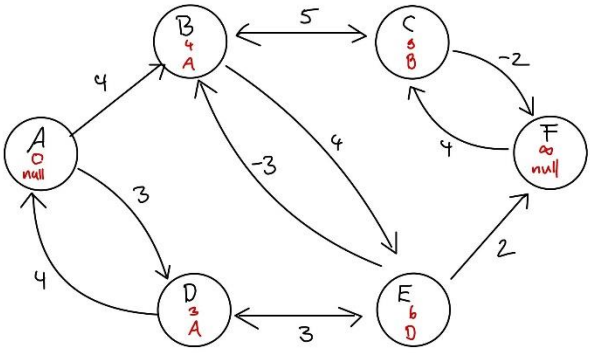
Anmerkungen zu Serie 10: Allgemeines

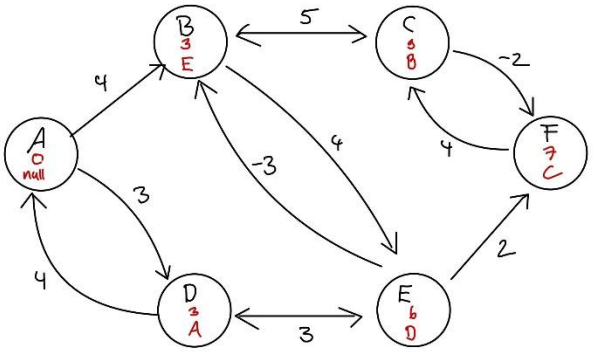
- Die Aufgaben wurden sehr gut gelöst
- Wenn ihr Laufzeiten von Graphenalgorithmen angebt, dürft ihr nicht schreiben, dass euer Algorithmus in (z.B.) $O(n+m)$ läuft, ohne vorher definiert zu haben, was n oder m bedeuten. Das heisst: Entweder, ihr definiert zu Beginn der Aufgabe $n = |V|$ und $m = |E|$ oder (und das ist absolut in Ordnung so) ihr schreibt die Laufzeit direkt mit den Mengen-Grössen, bspw. $O(|V| + |E|)$. Bei einer strengen Korrektur könnte es sonst passieren, dass man euch dafür Punkte abzieht, da nicht klar ist, was ihr mit n, m meint – **ausser natürlich, es wird schon in der Aufgabenstellung definiert, was n und m sind.**

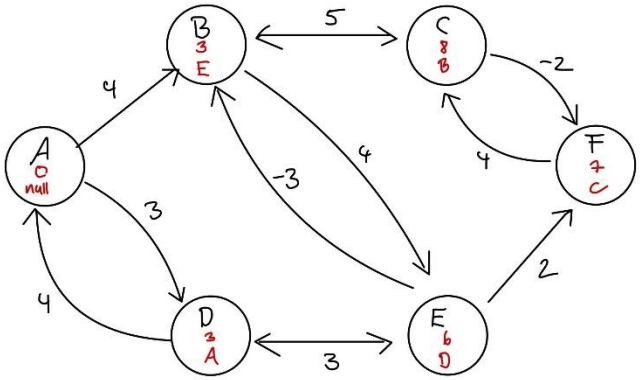
Bellman-Ford Algorithmus:

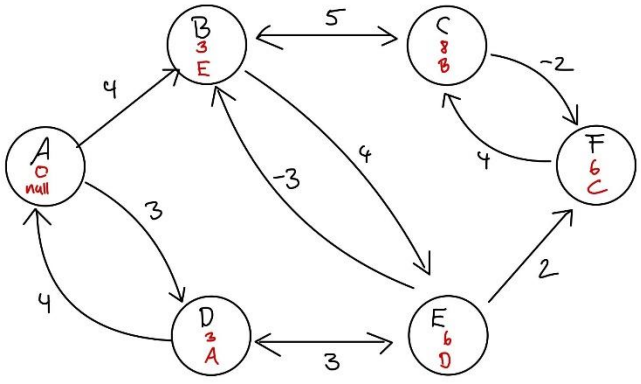
Da wir ihn in der letzen Woche nichtmehr vollständig besprechen konnten, werden wir uns den Algorithmus in dieser Woche noch einmal angucken. Hier ist ein ausführliches Beispiel, das wir in der Übungsstunde im Detail betrachten werden:




A 0 null
B 4 A
C 5 B
D 3 A
E 6 D
F ∞ null
5
-2
4
4
3
-3
4
4
2
3


A 0 null
B 3 E
C 5 B
D 3 A
E 6 D
F 7 C
5
-2
4
4
3
-3
4
4
2
3


A 0 null
B 3 E
C 8 B
D 3 A
E 6 D
F 7 C
5
-2
4
4
3
-3
4
4
2
3


A 0 null
B 3 E
C 8 B
D 3 A
E 6 D
F 6 C
5
-2
4
4
3
-3
4
4
2
3

Wir sehen, dass der Algorithmus auch mit negativen Kantengewichten funktioniert. In jeder jeder der $|V| - 1$ Iterationen betrachten wir alle Kanten und versuchen die min. Distanz zu einem Knoten von A aus mit dieser Kante noch weiter zu verringern.

BELLMAN-FORD$(G = (V, E), s)$

```
1  for each v ∈ V\{s} do                          ▷ Initialisiere für alle Knoten die
2      d[v] ← ∞; p[v] ← null                      ▷ Distanz zu s sowie Vorgänger
3  d[s] ← 0; p[s] ← null                          ▷ Initialisierung des Startknotens
4  for i ← 1, 2, ..., |V| − 1 do                  ▷ Wiederhole |V| − 1 Mal
5      for each (u, v) ∈ E do                     ▷ Iteriere über alle Kanten (u, v)
6          if d[v] > d[u] + w((u, v)) then        ▷ Relaxiere Kante (u, v)
7              d[v] ← d[u] + w((u, v))            ▷ Berechne obere Schranke
8              p[v] ← u                           ▷ Speichere u als Vorgänger von v
9  for each (u, v) ∈ E do                         ▷ Prüfe, ob eine weitere Kante
10     if d[u] + w((u, v)) < d[v] then            ▷ relaxiert werden kann
11         Melde Kreis mit negativem Gewicht
```

(Quelle: Skript der Vorlesung) Laufzeit: (Trivial) $O(|V| * |E|)$

Minimum Spanning Trees (MSTs):

Ein Spannbaum ist eine Menge von Kanten, die jeden Knoten des gegebenen Graphen überdeckt. Also suchen wir einen zusammenhängenden Teilgraphen. Da wir MSTs betrachten, ist für uns dabei noch etwas wichtig: Dieser zusammenhängende Teilgraph soll ausserdem auch minimal sein, das heisst konkret: Die Summe der Kantengewichte soll minimal sein.

Nun wollen wir uns einige Algorithmen angucken, die das MST Problem für einen gegebenen Graphen effizient lösen. Dabei werde ich mich an dem Buch Algorithmen und Datenstrukturen, 5. Auflage von Thomas Ottmann und Peter Widmayer, orientieren.

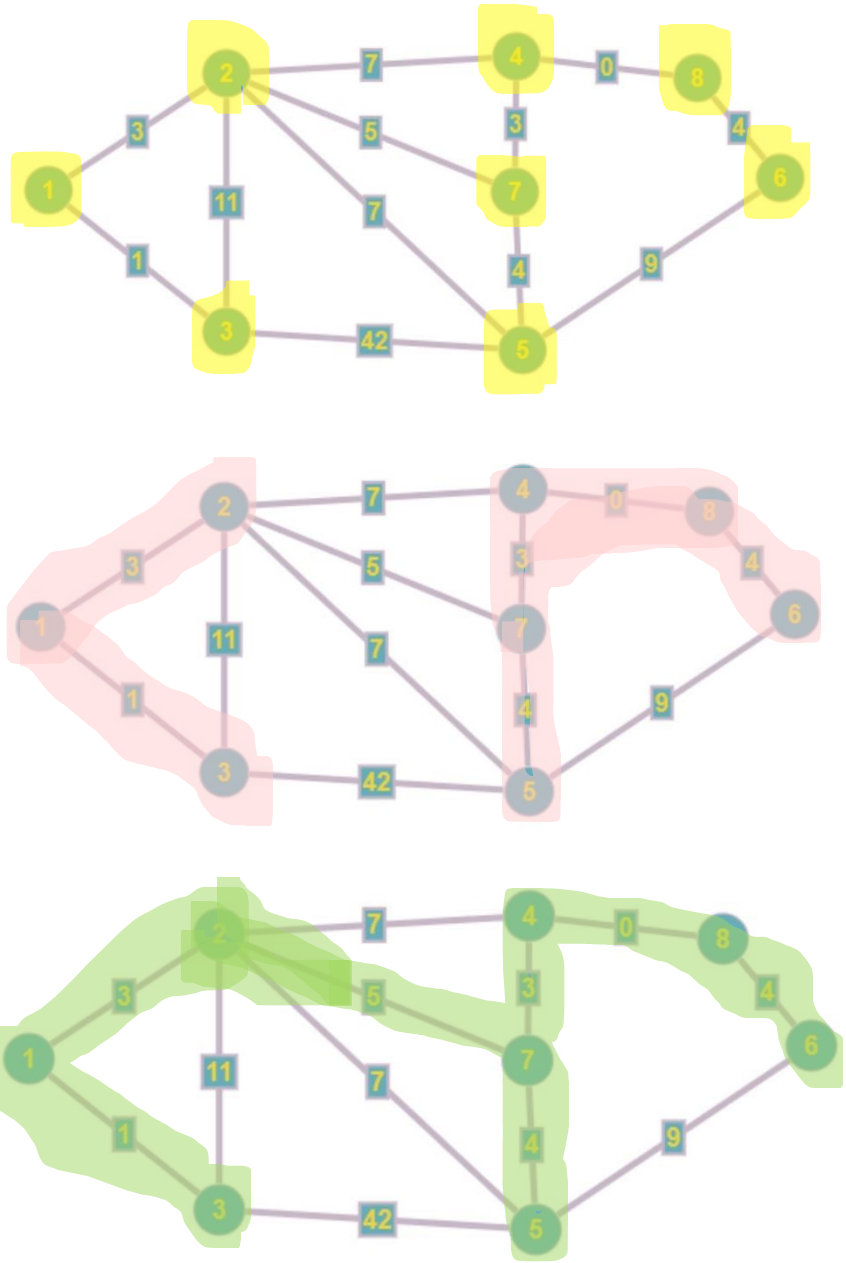
Das Grundgerüst eines Algorithmus für das MST-Problem muss diese Form haben:

```
Algorithmus-Gerüst Minimaler spannender Baum
{liefert zu einem zusammenhängenden, ungerichteten, bewerteten
 Graphen G = (V,E) mit c : E → ℝ einen minimalen spannenden
 Baum T' = (V,E') von G}
begin
  E' := ∅;
  while noch nicht fertig do
    begin
      wähle geeignete Kante e ∈ E;
      E' := E' ∪ {e}
    end
end {Minimaler spannender Baum}
```

Die Frage, die sich nun stellt ist: Wie genau wählen wir in jedem Schritt eine geeignete Kante. Dafür präsentiere ich hier nun drei Vorschläge:

Boruvka's MST Algorithmus:

Die Idee ist folgende: Für einen Graphen $G = (V, E)$ tun wir anfangs so, als wäre jeder einzelne Knoten ein Baum. In einem **Auswahlschritt** wird dann für jeden Baum eine kürzeste/günstigste Kante zu einem anderen Baum gewählt. Gibt es mehrere kürzeste/günstigste Kanten, wählen wir nur eine (um Zyklen zu vermeiden). Jetzt haben wir weniger Bäume, da wir ja Bäume vereint haben. Im nächsten Auswahlschritt wiederholen wir das Prozedere, bis es nur noch einen Baum gibt.

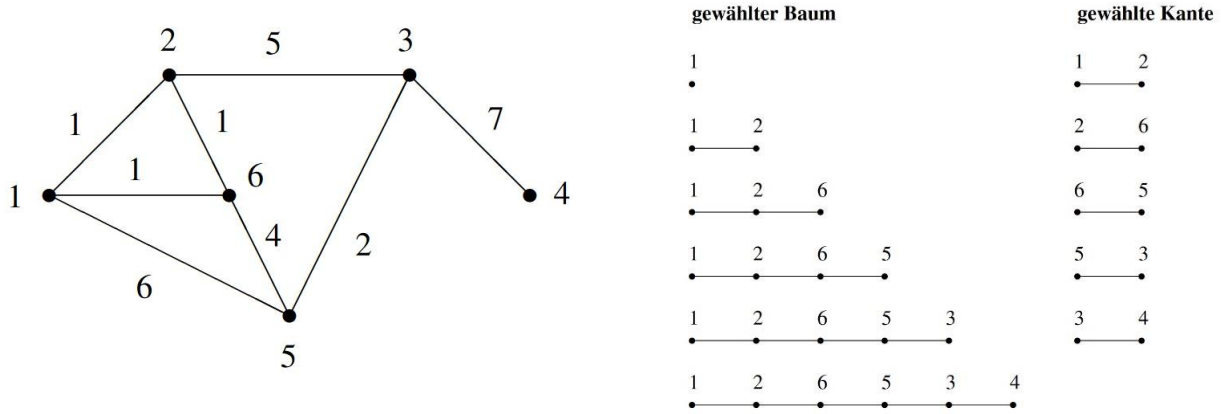


In jeder Iteration müssen wir die einzelnen Zusammenhangskomponenten unseres Teilgraphs (der nur aus den Knoten und den bis dahin gewählten Kanten besteht) verwalten und dabei immer die billigste Kante von einer ZHK zu irgendeiner anderen ZHK bestimmen. Das ist in $O(|E| + |V|)$ möglich. In jeder Iteration halbieren wir (mindestens!) die Anzahl der ZHKs, daher benötigen wir $O(\log(|V|)$ Auswahlschritte. Insgesamt also $O\big(\log|V| * (|E| + |V|)\big)$.

Prim's MST Algorithmus:

Es ist ein Greedy-algorithmus, der der Idee von Dijkstra stark ähnelt: Wir beginnen mit einem Startknoten und führen den folgenden **Auswahlschritt** $|V| - 1$ mal aus: Wir wählen eine Kante mit minimalen Kosten, für die genau ein Endknoten zum Baum gehört. Der andere Knoten der Kante ist nach dem Schritt nun auch Teil des Knotens. Da hier nur ein Baum wächst – und wir nicht mehrere ZHKs verwalten müssen, genügt eine simple Priority-Queue für eine optimale Implementation. Er läuft also in $O((|E| + |V|) * \log|V|)$ (wobei es mit Fibonacci-Heaps noch besser geht, aber das geht über den Vorlesungsstoff hinaus).

Hier ist ein Beispiel aus dem Buch:



Kruskal's MST Algorithmus:

Anfangs ist – wie bei Boruvka – jeder Knoten ein einzelner Baum. Dann wird für jede Kante $\{u, v\} \in E$ (nach Gewicht aufsteigend sortiert) folgender Auswahlschritt angewendet: Sind beide u, v in einer Zusammenhangskomponente (bzw. einem Baum), verwirf die Kante. Ansonsten: Verwende die Kante und verbinde die beiden unterschiedlichen Zusammenhangskomponenten.

Das Sortieren der Kanten ist in $O(|E| * \log|E|)$ möglich, pro Iteration müssen wir im worst case $O(|V|)$ Knoten betrachten (um zu prüfen, ob zwei Knoten in unterschiedlichen ZHKs liegen und sie anschliessend zu einer ZHK zu vereinen) und im worst case brauchen wir $O(|E|)$ Iterationen (betrachten jede Kante einmal). Also insgesamt: $O(|V| * |E| + \log|E| * |E|)$.

Union-Find Datenstruktur:

Die vorgestellten Algorithmen sind mit naiven Implementationen schon sehr effizient. Allerdings fällt auf, dass sehr viel Zeit verloren geht, weil wir verschiedene Zusammenhangskomponenten verwalten müssen und schnell bestimmen wollen, ob zwei Knoten zu einer Zusammenhangskomponente gehören – bzw. zwei Zusammenhangskomponenten effizient "verschmelzen lassen" wollen.

Für dieses Problem hat man die sogenannte Union-Find Datenstruktur entwickelt, die folgende 3 Methoden bereitstellen muss:

`Make-set(x,i)`: Erschafft neue Menge i mit x als einzigem Element. i ist also der Name der Menge.

`Find(x)`: Liefert den Namen der Menge, die das Element x enthält.

`Union(i,j,k)`: Vereinigt die Mengen i und j zu einer neuen Menge mit Namen k.

Wir sehen, dass diese 3 Methoden uns exakt das liefern, was wir für eine noch effizientere Implementation von Kruskal's Algorithmus benötigen.

Für Kruskal verwalten wir die entstehenden Teilbäume bzw. Zusammenhangskomponenten als Mengen in der Union-Find Datenstruktur und vereinen Zusammenhangskomponenten, wenn wir eine Kante wählen. Des Weiteren können wir mit der $Find(x)$ Funktion in $O(1)$ für zwei Knoten u, v bestimmen, ob sie in einer Zusammenhangskomponente liegen.

Wir implementieren Union-Find mit einem Array $rep[v]$, das für einen Knoten speichert, zu welcher ZHK er gehört. Anfangs initialisieren wir jeden entry mit einem anderen Integer (da es alles unterschiedliche ZHKs sind).

Nun verwenden wir eine sehr mächtige Technik namens Path-Compression, die eine effiziente Implementierung von Union erlaubt:

Hier ist eine sehr schöne Erklärung der Technik:

https://www.youtube.com/watch?v=VHRhJWacxis

Und hier eine Implementierung der Datenstruktur: NÄCHSTE SEITE

```java
class UnionFind {
    int[] labels;
    int[] sz;
    //
    // The constructor, just calls `create`
    //
    public UnionFind (int N) {
      sz = new int[N];
      create(N);
    }
    //
    // Initialize the union-data structure, by creating a
    // set for each element from [0, 1, ..., N - 1].
    //
    void create (int N) {
      labels = new int[N];
      for (int i = 0; i < N; i += 1) {
        labels[i] = i;
        sz[i] = 1;
      }
    }
    //
    // Determine which set a particular element belongs to.
    // Return the label or the 'id' of the set for a given x
    //
    int find (int x) {
      int root = x;

      while(root != labels[root])
        root = labels[root];

      //path compression:
      while(x != root){
        int parent = labels[x];
        labels[x] = root;
        x = parent;
      }
      return root;
    }

    boolean connected(int x, int y){
      return find(x) == find(y);
    }

    int componentSize(int p){
      return sz[find(p)];
    }
    //
    // Connect or join two sets. In other words, change
    // the family by replacing two sets, the one containing
    // x and the one containing y, by a single set that is
    // the union of these two sets.
    //
    void union (int x, int y) {
      int rootx = find(x);
      int rooty = find(y);
      if(rootx == rooty)
        return;
      if(sz[rootx] < sz[rooty]){
        sz[rooty] += sz[rootx];
        labels[rootx] = labels[rooty];
      }
      else{
        sz[rootx] += sz[rooty];
        labels[rooty] = labels[rootx];
      }
    }
```