Datenstrukturen und effiziente Algorithmen
Blatt 10

Markus Vieth, David Klopp, Christian Stricker

14. Januar 2016

Markus Vieth, David Klopp, Christian Stricker

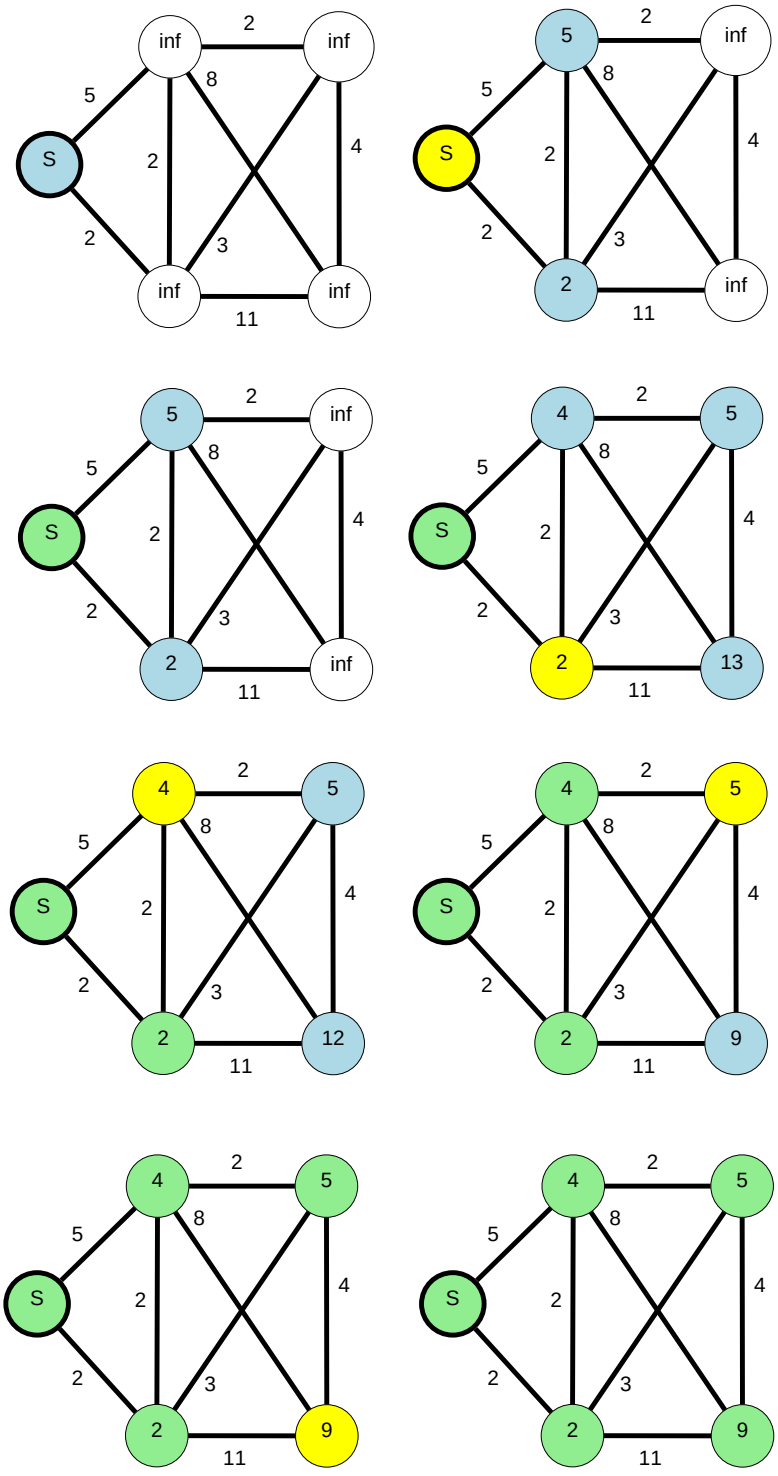
Aufgabe 1

Die Breitensuche kann das Single-Source-Shortest-Path-Problem schneller lösen. Die Breitensuche hat eine Laufzeit von O(|V|+|E|), wogegen der Dijkstra-Algorithmus eine Laufzeit von O(|V | log |V | + |E|) hat. Dies kommt durch die Suche des kleinsten Elements in der Warteschlange/Einsortieren des nächsten gefundenen Knotens in die Warteschlange. Die Breitensuche nimmt immer das nächste Element aus der Warteschlange und spart dadurch die Zeit zum Suchen. Sowohl die Breitensuche als auch der Dijkstra-Algorithmus findet immer die kürzesten Wege von einem Startknoten aus. Wenn man den kürzesten Weg mithilfe der Tiefensuche sucht, findet man meistens schneller eine Lösung. Da die Tiefensuche nach einem gefundenen Weg abbricht, ist dieser Weg mit einer großen Wahrscheinlich nicht der kürzester Weg. Je größer der Graph und je mehr Wege es gibt, desto unwahrscheinlicher ist es, dass die Tiefensuche die korrekte Lösung für das Problem findet. (Einzige Ausnahme ist, es gibt nur einen Weg und somit ist dies der kürzester Weg. Dies ist aber nur ein Spezialfall, der selten vorkommt.)

Markus Vieth, David Klopp, Christian Stricker

Aufgabe 2

a)



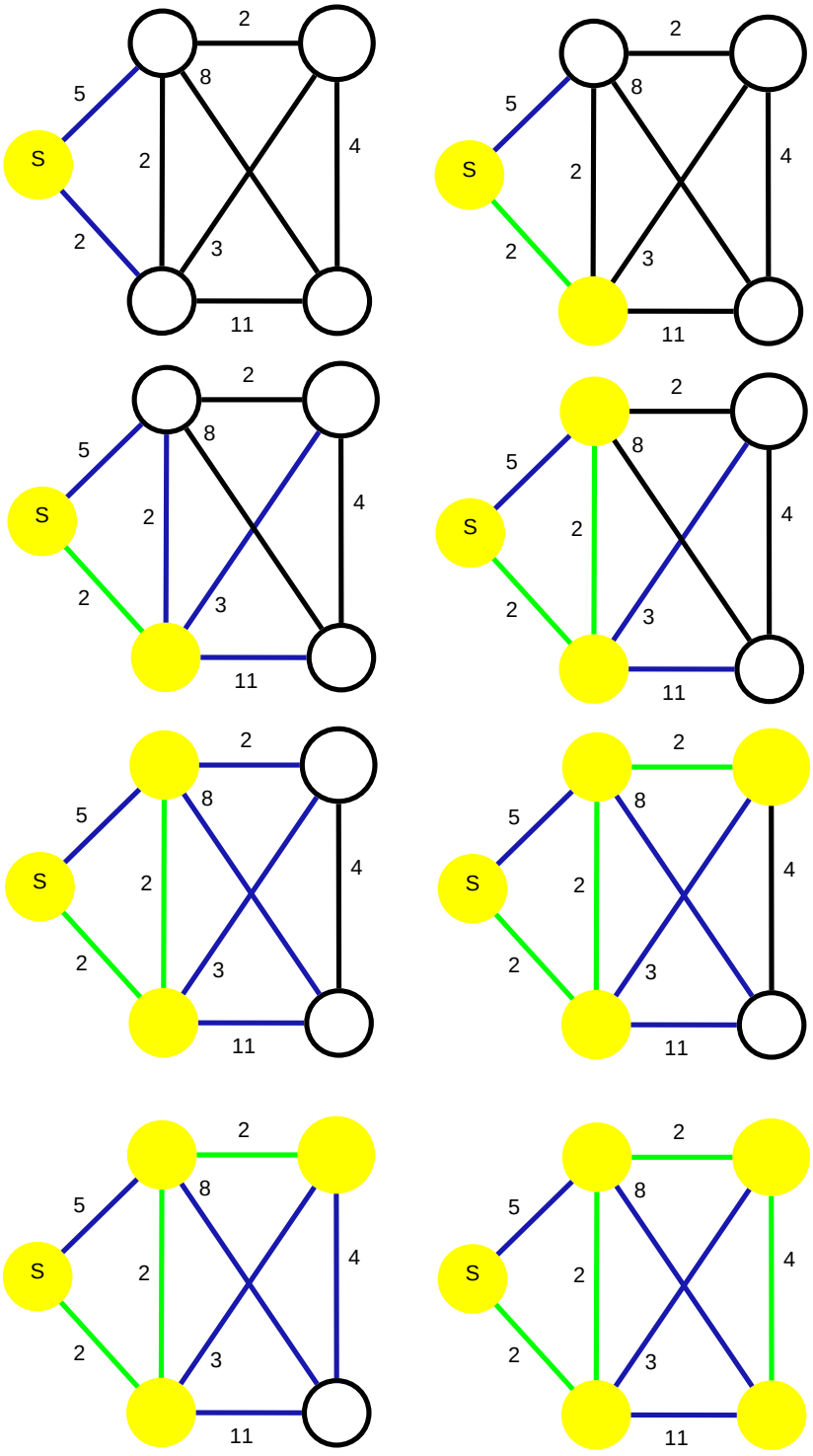
Legende:

Blau Knoten in der Warteschlange

Gelb Der derzeitige ausgewählte Knoten

Grün Abgearbeiteter Knoten

b)



Legende:

Blau Potenzielle Kante

Gelb Besuchte Knoten

Grün Gewählte Kante

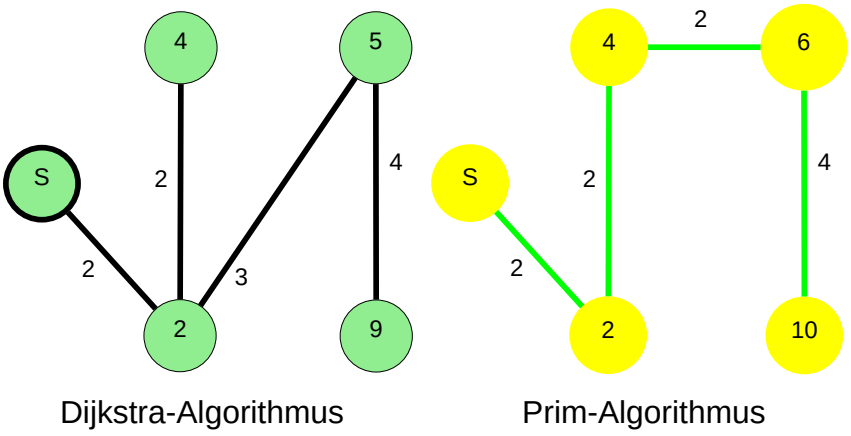
Spannbaum siehe d)

c)

Der Prim-Algorithmus sucht eine Vernetzung aller Knoten, sodass die Gesamtlänge des Netzes minimal wird. Der Startpunkt ist bei diesem Algorithmus egal, er kommt immer zu dem selben Ergebnis. Dies ist z.B. beim Aufstellen von Strommasten nützlich, um die Kosten zu minimieren und trotzdem alle Orte mit Strom zu versorgen. Der Dijkstra-Algorithmus hingegen sucht den kürzesten Weg von einem bestimmten Startknoten aus zu allen anderen Knoten. Hilfreich ist dies z.B. Bei Onlineversandhändlern. Diese haben ein lokales Lager (bestimmten Startknoten) und suchen den kürzesten Weg zu ggf. allen anderen Orten (Knoten). Ein „langer Weg“ über die Autobahn ist meist schneller als durch viele eng zusammen liegende Orte zu fahren. Deshalb ist die Gesamtlänge der Strecke im Spannbaum meist länger als beim Prim-Algorithmus.

d)

Siehe Teilaufgabe c)



Markus Vieth, David Klopp, Christian Stricker

Aufgabe 3

UnionFind.java

```
1  package app;
2
3  class UnionFind extends AbstractUnionFind {
4
5      private int[] ref;
6      private int[] size;
7      private int[] next;
8
9      UnionFind(int n){
10         ref = new int[n];
11         size = new int[n];
12         next = new int[n];
13
14         for (int i = 0; i < n; i++) {
15             ref[i] = i;
16             size[i] = 1;
17             next[i] = -1;
18         }
19 }
20
21     public int find(int repr){
22
23         // here should be something intelligent
24         return ref[repr];
25     }
26
27     //Entspricht dem Pseudo-Code aus der Vorlesung
28     public boolean union(int i, int j){
29
30         // here should also be something intelligent
31         int x = ref[i];
32         int y = ref[j];
33
34         if (size[x] > size[y]) {
35             x = ref[j];
36             y = ref[i];
37         }
38
39         int h = next[y];
40         next[y] = x;
41         int z = y;
42
43         while (next[z] >= 0) {
44             z = next[z];
45             ref[z] = y;
46         }
47
48         next[z] = h;
49         size[y] = size[y] + size[x];
50
51         return true;
52     }
53 }
```

SpanningTree.java

```
1  package app;
2
3  // default java stuff
4  import java.util.Vector;
5  import java.util.Collections;
6
7  class SpanningTree {
8
9      static Vector<Tuple> kruskal (Vector<Triple> cities, int cityCount){
10
11         // final mst and vector of all edges
12         Vector<Tuple> mst = new Vector<Tuple>();
13         Vector<WeightedEdge> edges = new Vector<WeightedEdge>();
14         UnionFind uf = new UnionFind(cityCount*2);
15         // here should be something intelligent
16         //Erstelle Kantenmengen für die Cities untereinander und jeweils zwischen Citiy
               unt virtual city
17         for (int i = 0; i < cityCount; i++) {
18             for (int j = 0; j < cityCount; j++) {
19                 if ( i != j) {
20                     edges.add(new WeightedEdge(euclid(cities.get(i), cities.get(j)), new
                           Tuple(i, j)));
21                 }
22             }
23             edges.add(new WeightedEdge(euclid(cities.get(i), cities.get(i + cityCount)),
                   new Tuple(i, i + cityCount)));
24         }
25         // sort all edges
26         Collections.sort(edges);
27         //stecke alle virtuellen Cities in eine Partition
28         for (int i = 0; i < cityCount-1; i++) {
29             uf.union(i + cityCount, i + 1 + cityCount);
30         }
31         // here should also be something intelligent
32         for (int i = 0; i < edges.size(); i++) {
33             int u = edges.get(i).edge.x;
34             int v = edges.get(i).edge.y;
35
36             //Wenn schon Verbunden (wenn auch nicht direkt) ist die Kante uninteressant
37             if (uf.find(u) == uf.find (v))
38                 continue;
39
40             mst.add(edges.get(i).edge);
41             uf.union(u,v);
42         }
43         return mst;
44     }
45     static double euclid(Triple a, Triple b){
46
47         return (a.x - b.x) * (a.x - b.x) +
48                (a.y - b.y) * (a.y - b.y) +
49                (a.z - b.z) * (a.z - b.z);
50     }
51 }
```