ETH Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

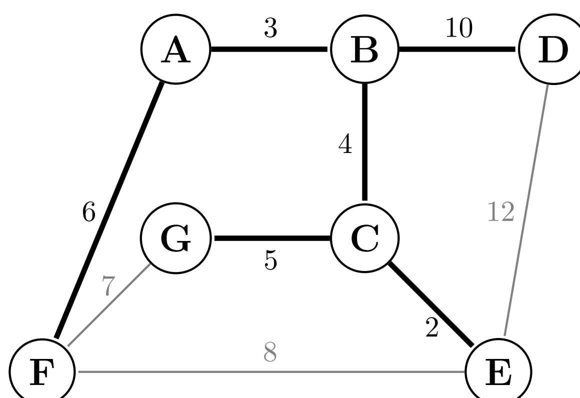
Institut für Theoretische Informatik
Peter Widmayer
Thomas Tschager
Antonis Thomas

4. Mai 2016

Datenstrukturen & Algorithmen   Lösungen zu Blatt 9   FS 16

Lösung 9.1 *Minimale Spannbäume.*

a) Das Verfahren von Kruskal berechnet den folgenden minimalen Spannbaum.



b) Wir beweisen die Aussage durch vollständige Induktion über die Knotenanzahl $|V|$.

Induktionsverankerung ($|V| = 1$): Ein Graph mit genau einem Knoten hat keine Kante, also muss $|E| = 0 \leq 1 - 1 = |V| - 1$ gelten.

Induktionsannahme: Angenommen, jeder ungerichtete kreisfreie Graph mit genau $|V| - 1$ Knoten hat höchstens $|V| - 2$ Kanten.

Induktionsschluss ($|V| - 1 \to |V|$): Betrachte nun einen ungerichteten kreisfreien Graphen $G = (V, E)$. Aufgrund der Kreisfreiheit gibt es mindestens einen Knoten w mit Grad 0 oder 1. Seien nun $V' := V \backslash \{w\}$ und $E' := \{\{u, v\} \in E \mid u, v \in V'\}$. Offenbar ist $|V'| = |V| - 1$ und, da von w höchstens eine einzige Kante ausgeht, $|E'| \geq |E| - 1$ (d.h., $|E| \leq |E'| + 1$). Nach Induktionsvoraussetzung ist $|E'| \leq |V'| - 1$, also erhalten wir

$$|E| \leq |E'| + 1 \leq |V'| - 1 + 1 = |V'| = |V| - 1. \quad (1)$$

■

c) Sei $G = (V, E, w)$ ein ungerichteter Graph, bei dem keine zwei Kanten $e, e' \in E$ das gleiche Gewicht haben. Angenommen, G hätte zwei minimale Spannbäume $T_1 = (V, E_1)$ und $T_2 = (V, E_2)$. Die Kanten aus $E_1 \cap E_2$ kommen sowohl in T_1 als auch in T_2 vor. Seien $E_1' = E_1 \backslash E_2$ und $E_2' = E_2 \backslash E_1$ die Kanten, die ausschliesslich in T_1 bzw. in T_2 enthalten sind. Da die Gewichte aller Kanten paarweise verschieden ist, enthält die Menge $E_1' \uplus E_2'$ eine eindeutig bestimmte Kante $e^{\perp}$ minimalen Gewichts. Sei diese o.B.d.A. in T_1 enthalten. Würde sie T_2 hinzugefügt, dann enthielte T_2 nach Aufgabenteil b) einen Kreis. Dieser enthält mindestens

eine Kante $e' \in E_2'$ (ansonsten hätte er neben $e^\perp$ ausschliesslich Kanten aus $E_1 \cap E_2$ und wäre auch in T_1 ein Kreis). Wird e' aus T_2 entfernt und stattdessen $e^\perp$ eingefügt, dann erhalten wir eine Teilmenge von Kanten, die noch immer kreisfrei und zusammenhängend (also ein Spannbaum) ist, wegen $w(e^\perp) < w(e')$ aber ein geringeres Gesamtgewicht hat. Also war T_2 kein minimaler Spannbaum, was der Annahme widerspricht. ■

Lösung 9.2 *Union-Find Strukturen.*

Wenn jeweils der Baum mit weniger Knoten an denjenigen mit mehr Knoten angehängt wird, dann gilt für die Union-Find Struktur für jeden Baum mit Höhe h und n Knoten die Invariante $n \geq 2^h$ (Lemma 6.3, Kapitel 6.2.2).

Die Invariante besagt, dass ein Baum der Höhe h mindestens 2^h Knoten enthalten muss. Wir konstruieren einen Baum der Höhe h und genau 2^h Knoten wie folgt: Ein Baum der Höhe $h = 0$ besteht aus $n = 1 = 2^0$ Knoten. Um einen Baum der Höhe $h > 0$ zu konstruieren, verschmelzen wir zwei Bäume der Höhe $h-1$ mit je genau 2^{h-1} Knoten. Da der eine in den anderen eingehängt wird, wächst die Höhe um eins. Somit hat der neue Baum genau Höhe h und $2 \cdot 2^{h-1} = 2^h$ Knoten. Die Anzahl benötigter UNION-Operationen ist durch folgende rekursive Gleichung gegeben:

$$u(0) = 0, u(h) = 2 \cdot u(h-1) + 1. \quad (2)$$

Damit ist die Anzahl benötigter UNION-Operationen genau $u(h) = \sum_{i=1}^{h} 2^{i-1} = 2^h - 1$.

Mit weniger Operationen kann man unmöglich einen Baum mit 2^h Knoten erzeugen, da für jeden zusätzlichen Knoten im Baum mindestens eine UNION-Operation nötig ist. Da ein Baum der Höhe h auch mindestens 2^h Knoten enthalten muss, ist es unmöglich eine Höhe von h mit weniger als $2^h - 1$ Operationen zu erreichen.

Lösung 9.3 *Längste aufsteigende Teilfolge.*

```
1 static int binarySearch(int A[], int l, int r, int key) {
2     while (l < r) {
3         int m = l + (r - l + 1)/2;
4         if (A[m] >= key) r = m-1;
5         else l = m;
6     }
7     return l;
8 }
9
10 static int computeTable(int A[], int size) {
11     int[] T = new int[size+1];       // DP-Tableau
12     for (int i = 1; i <= size; i++)  // Initialisierung
13         T[i] = Integer.MAX_VALUE;
14     T[0] = Integer.MIN_VALUE;
15     int l = 0;
16
17     for (int i = 0; i < size; i = i+1) {
18         int j = binarySearch(T, 0, l, A[i]);
19         if ( A[i] < T[j+1] ) T[j+1] = A[i];
20         if (l < j+1) l = j+1;
21     }
22     return l;
23 }
```

Lösung 9.4 *Minimale Spannbäume (**Programmieraufgabe**).*

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.Scanner;
import java.util.Arrays;

class Edge implements Comparable<Edge>{
    public int u, v, c;
    Edge(int u, int v, int c)
    {
        this.u = u;
        this.v = v;
        this.c = c;
    }
    public int compareTo(Edge e) {
        return this.c - e.c;
    }
}

class UnionFind {
    private int[] parent;
    private int[] rank;

    public int find(int i) {
        int p = parent[i];
        if (i == p) return i;
        parent[i] = find(p);
        return parent[i];
    }

    public void union(int i, int j) {
        int rooti = find(i);
        int rootj = find(j);

        if (rooti == rootj) return;

        //i and j are not in the same set; we merge them.
        if (rank[rooti] < rank[rootj]) parent[rooti] = rootj;
        else if (rank[rooti] > rank[rootj]) parent[rootj] = rooti;
        else {
            parent[rootj] = rooti;
            rank[rooti]++;
        }
    }

    public UnionFind(int n) {
        parent = new int[n];
        rank = new int[n];

        for (int i=0; i<n; i++) parent[i] = i;

    }
}

class UnionFindSlow {
    private int[] parent;

    public int find(int i) {
        int p = parent[i];
        if (i == p) return i;
```

```
60          return find(p);
61      }
62
63      public void union(int i, int j) {
64          int rooti = find(i);
65          int rootj = find(j);
66
67          parent[rooti] = rootj;
68      }
69
70      public UnionFindSlow(int n) {
71          parent = new int[n];
72
73          for (int i=0; i<n; i++) parent[i] = i;
74
75      }
76  }
77
78  class Main {
79
80      public static void main(String[] args) {
81          int test, ntest, n, m, u, v, c;
82          Scanner sc = new Scanner(System.in);
83          ntest = sc.nextInt();
84          for(test = 1; test <= ntest; ++test)
85          {
86              // Read the number of vertices.
87              n = sc.nextInt();
88              // Read the number of edges.
89              m = sc.nextInt();
90              // Read the edges.
91              ArrayList<Edge> edgeList = new ArrayList<Edge>();
92              for(int i = 0; i < m; ++i) {
93                  u = sc.nextInt();
94                  v = sc.nextInt();
95                  c = sc.nextInt();
96                  // We store nodes as numbers from 0 to n-1
97                  edgeList.add(new Edge(u-1, v-1, c));
98              }
99
100             // Sort the edge list by cost.
101             Collections.sort(edgeList);
102
103             UnionFind UF = new UnionFind(n);
104
105             int costMST = 0;
106             for(int j = 0; j < m; ++j) {
107                 u = edgeList.get(j).u; v = edgeList.get(j).v; c = edgeList.get(j).c;
108                 if (UF.find(u) != UF.find(v))
109                 {
110                     UF.union(u,v);
111                     costMST += c;
112                 }
113             }
114             System.out.println(costMST);
115
116         //
117         }
118     }
119 }
```