Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Federal Institute of Technology at Zurich

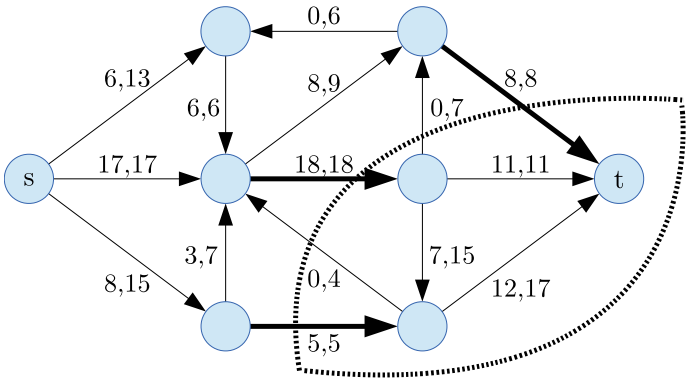
Institut für Theoretische Informatik
Peter Widmayer
Thomas Tschager
Antonis Thomas

25. Mai 2016

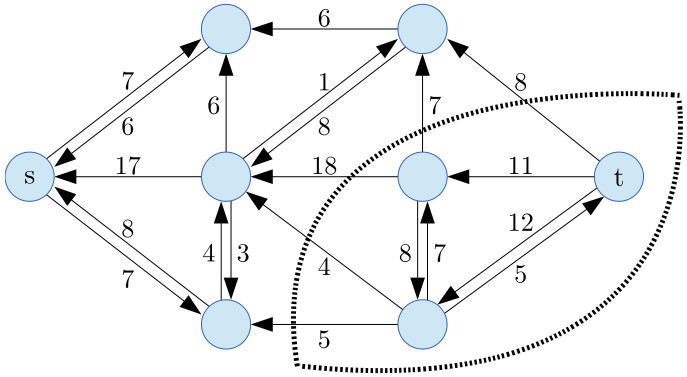
Datenstrukturen & Algorithmen Lösungen zu Blatt 12 FS 16

Lösung 12.1 *Max-Flow von Hand.*

Der maximale Flusswert beträgt 31. In der folgenden Abbildung ist ein maximaler Fluss dargestellt. Neben jeder Kante e sind die Kapazität c_e und der Flusswert x_e in der Reihenfolge x_e, c_e angegeben. Der minimale Schnitt ist gestrichelt eingezeichnet, und die zugehörigen Kanten sind fett dargestellt.

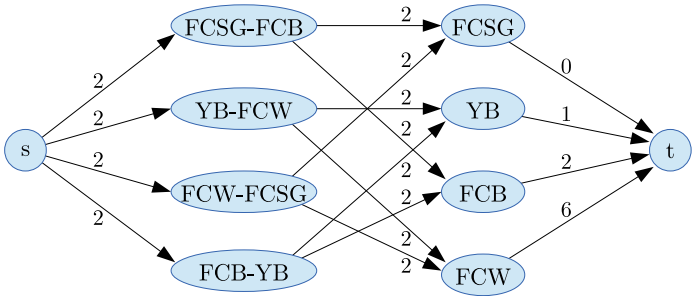


Die folgende Abbildung zeigt den Restgraphen für den oben dargestellten Fluss. An den Kanten ist die jeweilige Restkapazität ($c_e - x_e$ oder x_e) notiert.

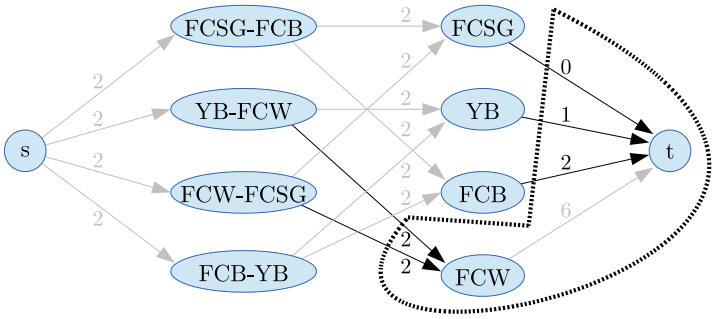


Lösung 12.2 *Meisterschaftsproblem.*

Angenommen, FCL gewinnt beide verbleibenden Spiele gegen FCZ und GCZ mit ausreichend vielen Toren. Dann hat FCL 37 Punkte. Zum Gewinn der Meisterschaft darf FCSG keine weiteren Punkte erhalten, YB maximal einen, FCB maximal zwei und FCW maximal sechs. Zentral für die Konstruktion des Netzes ist die Beobachtung, dass in jedem der verbleibenden Spiele genau zwei Punkte aufgeteilt werden: Der Gewinner erhält beide, oder jeder Teilnehmer bekommt genau einen im Fall von Unentschieden. Wir erzeugen also für jedes der Spiele FCSG-FCB, YB-FCW, FCW-FCSG und FCB-YB einen Knoten im Netz mit einer eingehenden Kante von einer Quelle s und je zwei ausgehenden Kanten zu den Teilnehmern des entsprechenden Spiels. Diese haben jeweils eine Kapazität von 2. Von jedem Teilnehmer erzeugen wir eine Kante zu einer Senke t und wählen als Kapazität die maximal erlaubte Punktzahl, sodass FCL noch Meister wird. Dies führt zu folgendem Netz.



Damit FCL noch Meister wird, muss im obigen Netz ein Fluss mit Wert 8 existieren, denn in jedem der vier Spiele werden auf jeden Fall je zwei Punkte auf die entsprechenden Teilnehmer verteilt. Das Max-Flow-Min-Cut-Theorem besagt, dass der Wert eines maximalen Flusses durch den Wert eines minimalen Schnitts nach oben beschränkt ist. Im obigen Netz existiert nun der folgende Schnitt vom Wert 7:



Aus diesem Grund kann kein Fluss mit Wert 8 existieren, und daher kann FCL auch dann nicht Meister werden, wenn beide verbleibenden Spiele haushoch gewonnen werden.

Lösung 12.3 *Algorithmus von Edmonds und Karp (**Programmieraufgabe**).*

Die Aufgabe bestand in der Implementierung des Algorithmus von Edmonds und Karp zur Berechnung eines maximalen Flusses. Sei $G = (V, E, c)$ ein gerichteter Graph mit der Kapazitätsfunktion $c : E \to \mathbb{Q}^+$, der Quelle $s \in V$ sowie der Senke $t \in V$. Im Folgenden wollen wir vereinfachend annehmen, dass

$$\forall v, w \in V : (v, w) \in E \Rightarrow (w, v) \notin E \tag{1}$$

gilt. Für jede Kante $(v, w) \in E$ des Ursprungsgraphen ist die entsprechende Rückwärtskante (w, v) also *nicht* vorhanden.

Der Algorithmus startet mit dem *leeren Fluss* $f \equiv 0$, d.h. auf keiner Kante ist Fluss vorhanden. Danach suchen wir einen kürzesten flussvergrössernden Pfad $P = \langle s = v_0, v_1, ..., v_{k-1}, t = v_k \rangle$ mit $f((v_{i-1}, v_i)) < c((v_{i-1}, v_i))$ für alle $i = 1, ..., k$, berechnen die minimale Restkapazität $C = \min_{1 \le i \le k} c((v_{i-1}, v_i)) - f((v_{i-1}, v_i))$ auf P und erhöhen den Fluss auf den Kanten (v_{i-1}, v_i) für $i = 1, ..., k$ jeweils um C. Das Verfahren endet, wenn kein solcher flussvergrössernder Pfad mehr existiert. Zum Auffinden flussvergrössernder Pfade berechnen wir ausgehend vom Fluss f den *Restgraphen* $G_R(f) = (V, E_R, c_R)$ mit

$$E_R := \{(v, w) \in E \mid f((v, w)) < c((v, w))\} \cup \{(w, v) \mid (v, w) \in E \wedge f((v, w)) > 0\} \tag{2}$$

und der Kapazitätsfunktion

$$\begin{aligned} c_R &: E_R \to \mathbb{Q}^+, \\ c_R((v, w)) &:= \begin{cases} c((v, w)) - f((v, w)) & \text{falls } (v, w) \in E \wedge f((v, w)) < c((v, w)) \\ f((w, v)) & \text{falls } (w, v) \in E \wedge f((w, v)) > 0 \end{cases} \end{aligned} \tag{3}$$

Der Restgraph erhält also alle Kanten $(v, w) \in E$ des Ursprungsgraphen G, auf denen die Kapazität noch nicht ausgeschöpft ist, und die Rückwärtskante (w, v) zu jeder Kante $(v, w) \in E$ des Ursprungsgraphen, auf der ein Fluss $f((v, w)) > 0$ vorhanden ist.

Aufgrund von (1) können in der Implementierung Graph und Restgraph kombiniert im Graphen $G_J = (V, E_J, c_J)$ mit $E_J = \{(v, w), (w, v) \mid (v, w) \in E\}$ und

$$\begin{aligned} c_J &: E_J \to \mathbb{Q}_0^+, \\ c_J((v, w)) &:= \begin{cases} c((v, w)) & \text{falls } (v, w) \in E \\ 0 & \text{sonst} \end{cases} \end{aligned} \tag{4}$$

gespeichert werden. Für jede Kante $(v, w) \in E$ des Ursprungsgraphen wird also die Rückwärtskante (w, v) mit Kapazität 0 eingefügt. Wir ändern die Definition eines Flusses dahingehend ab, dass auch negative Flusswerte zugelassen werden. Sei nun erneut $P = \langle s = v_0, v_1, ..., v_{k-1}, t = v_k \rangle$ ein flussvergrössernder Pfad, d.h. es gilt $f((v_{i-1}, v_i)) < c((v_{i-1}, v_i))$ für alle $i = 1, ..., k$, und $C = \min_{1 \le i \le k} c((v_{i-1}, v_i)) - f((v_{i-1}, v_i)) > 0$ die minimale Restkapazität auf diesem Pfad. Wird der Fluss auf einer Kante (v, w) auf diesem Pfad um C erhöht, dann muss zeitgleich der Fluss auf der zugehörigen Rückwärtskante (w, v) um C reduziert werden. Insbesondere wird damit der Fluss auf den Rückwärtskanten (w, v) zu den Kanten $(v, w) \in E$ des Ursprungsgraphen nicht-positiv sein. Es ist nicht schwierig zu sehen, dass dieses modifizierte Verfahren noch immer einen maximalen Fluss berechnet.

Berechnung eines kürzesten flussvergrössernden Pfades

Es wird nun eine Breitensuche benutzt, um einen kürzesten flussvergrössernden Pfad P von $s = 1$ nach $t = n$ zu berechnen. Die Methode `augmentingPathExists` liefert genau dann "true" zurück, wenn ein solcher Pfad existiert. Ausserdem wird als Parameter ein Array `previousVertexOnPath` übergeben, in dem der jeweilige Vorgängerknoten eines Knotens $v \in P$ im Eintrag $v - 1$ gespeichert wird. Man beachte, dass der in der Position $v - 1$ gespeicherte Wert undefiniert ist, wenn v nicht auf P liegt oder überhaupt kein flussvergrössernder Pfad gefunden wurde.

Berechnung eines maximalen Flusses

Mit der vorher definierten Funktion `augmentingPathExists` können nun kürzeste flussvergrössernde Pfade sukzessive gefunden werden, bis keine solchen mehr existieren. Wie bereits in der Einleitung erläutert, muss bei einer Erhöhung des Flusswerts einer Kante (v, w) um C Einheiten der Fluss auf der jeweiligen Rückwärtskante (w, v) um C Einheiten reduziert werden.

Code

```java
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Scanner;

class Main {

    // Adjacency list representation of the graph
    static int n;
    static ArrayList<Integer> graph[];
    static int[][] capacity, flow;

    // Compute a path with positive residual capacity using BFS
    // Return true if such a path exists and false otherwise
    public static boolean augmentingPathExists(int previousVertexOnPath[]) {
        // Allocate space for auxilary data structures
        LinkedList<Integer> queue = new LinkedList<Integer>();
        boolean[] visited = new boolean[n];

        // Initialization of auxilary data structures
        for(int i = 0; i < n; i++) visited[i] = false;
        queue.add(0);
        visited[0] = true;

        // BFS
        while(!queue.isEmpty()) {
            int v = queue.poll();
            for(int w : graph[v])
                if(!visited[w] && capacity[v][w] > flow[v][w]) {
                    visited[w] = true;
                    previousVertexOnPath[w] = v;
                    queue.add(w);
                    if(w == n-1) return true;
                }
        }

        return false;
    }
```

```
38
39      // Computes the value of a maximum flow
40      public static int computeMaximumFlow() {
41          int i, flowOnPath;
42          // Find paths with BFS and return path in previousVertexOnPath array
43          int[] previousVertexOnPath = new int[n];
44          // Start with empty flow
45          int maxFlow = 0;
46          // Use augmenting path P as long as possible
47          while(augmentingPathExists(previousVertexOnPath)) {
48              // Compute smallest remaining capacity on P
49              flowOnPath = Integer.MAX_VALUE;
50              for(i = n-1; i != 0; i = previousVertexOnPath[i]) {
51                  int p = previousVertexOnPath[i];
52                  flowOnPath = Math.min(flowOnPath, capacity[p][i] - flow[p][i]);
53              }
54              // Add the smallest remaining capacity to each edge of P
55              for(i = n-1; i != 0; i = previousVertexOnPath[i]) {
56                  int p = previousVertexOnPath[i];
57                  flow[p][i] += flowOnPath;
58                  flow[i][p] -= flowOnPath;
59              }
60              maxFlow += flowOnPath;
61          }
62
63          return maxFlow;
64      }
65
66      public static void main(String[] args) {
67          int test, ntest, m, u, v, c, i, maxFlow;
68          Scanner sc = new Scanner(System.in);
69          ntest = sc.nextInt();
70          for(test = 1; test <= ntest; test++) {
71              // Read the number of vertices
72              n = sc.nextInt();
73              // Read the number of edges
74              m = sc.nextInt();
75
76              // Initialize the graph
77              capacity = new int[n][n];
78              flow = new int[n][n];
79              graph = (ArrayList<Integer>[])new ArrayList[n];
80              for(i = 0; i < n; i++)
81                  graph[i] = new ArrayList<Integer>();
82
83              // Add the input edges to the graph
84              for(i = 0; i < m; i++) {
85                  u = sc.nextInt();
86                  v = sc.nextInt();
87                  c = sc.nextInt();
88
89                  // Index vertices internally from 0 to n-1 (instead of 1..n)
90                  u--; v--;
91
92                  // Initialize capacity, flow and store adjacency list
93                  capacity[u][v] = c;
94                  flow[u][v] = 0;
95                  flow[v][u] = 0;
96                  graph[u].add(v);
97                  capacity[v][u] = 0;
98                  graph[v].add(u);
```

```
99                }
100
101               // Compute maximum flow
102               maxFlow = computeMaximumFlow();
103
104               // Output result
105               System.out.println(maxFlow);
106           }
107       }
108
109   }
```