Eidgenössische
Technische Hochschule
Zürich

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

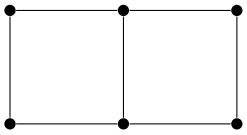
Institut für Theoretische Informatik
Peter Widmayer
Thomas Tschager
Antonis Thomas

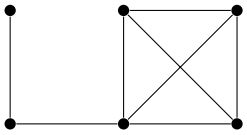
1. Juni 2016

Datenstrukturen & Algorithmen Lösungen zu Blatt 13 FS 16

Lösung 13.1 *Matchings.*

Es sind verschiedene Lösungen möglich, z.B.





Lösung 13.2 *Nicht-dominierte Punkte.*

Haltepunkte: Wir lassen die Scanline von links nach rechts laufen und halten genau dann an, wenn wir auf einen Punkt treffen. Falls mehr als ein Punkt auf der Scanline liegt, dann betrachten wir sie in absteigender Reihenfolge ihrer y-Koordinaten.

Scanline-Datenstruktur: Wir benutzen lediglich eine Variable zur Speicherung der y-Koordinate, die minimal unter den y-Koordinaten der bisher betrachteten Punkte ist. Initial wird diese Variable auf ∞ gesetzt.

Aktualisierung: Sobald ein neuer Punkt gefunden wird, dessen y-Koordinate kleiner oder gleich dem aktuell gespeicherten Minimum ist, wird er ausgegeben und das gespeicherte Minimum entsprechend aktualisiert.

Auslesen der Lösung: Offenbar werden alle nicht-dominierten Punkte direkt ausgegeben, sobald sie gefunden werden.

Laufzeit: Das Sortieren der n Punkte kann in Zeit $\Theta(n \log n)$ durchgeführt werden. Die Kosten zur Aktualisierung sind für jeden Haltepunkt konstant. Da es n Haltepunkte gibt, erhalten wir eine Gesamtlaufzeit von $\Theta(n \log n)$.

Lösung 13.3 *Kunstwerk.*

Haltepunkte: Wir verwenden eine horizontale Scanline, die von unten nach oben läuft. Diese stoppt jedes Mal, wenn an ihrer Position eine Platte startet oder endet. Die Haltepunkte sind also y_i und $y_i + b_i$ für alle $i \in \{1, \ldots, n\}$.

Scanline-Datenstruktur: Wir benutzen einen AVL-Baum, der als Schlüssel die x-Koordinaten aller Platten speichert, die die Scanline derzeit schneidet. Zusätzlich enthält jeder Knoten für die dort gespeicherte Platte den Betrag ihrer beleuchteten Breite unterhalb der Scanline (als Gesamtbetrag von links und rechts).

Aktualisierung: Sei y_{prev} die Position, die die Scanline zum Zeitpunkt der letzten Aktualisierung hatte, und sei $y_{current}$ die aktuelle Position der Scanline. Der Knoten mit minimalem (bzw. maximalem) Schlüssel repräsentiert die Platte, die von links (bzw. von rechts) her beleuchtet ist. Da diese im Intervall $[y_{prev}, y_{current})$ beleuchtet ist, erhöhen wir den im Knoten mit minimalem Schlüssel gespeicherten Beleuchtungswert um $y_{current} - y_{prev}$, und genauso verfahren wir für den im Knoten mit maximalem Schlüssel gespeicherten Beleuchtungswert. Nach dieser Aktualisierung verfahren wir wie folgt:

1. *Fall: Eine neue Platte $P_i = (x_i, y_i, b_i)$ startet.* Wir fügen den Schlüssel x_i in die Scanline-Datenstruktur ein und setzen den Betrag der beleuchteten Breite auf 0.

2. *Fall: Eine Platte $P_i = (x_i, y_i, b_i)$ endet.* Wir löschen den Knoten mit Schlüssel x_i aus der Scanline-Datenstruktur und geben den dort gespeicherten Beleuchtungswert für die Platte P_i aus.

Auslesen der Lösung: Die Lösung wird direkt während der Aktualisierung der Datenstruktur ausgegeben, wenn eine Platte endet.

Laufzeit: Es gibt $2n$ Haltepunkte, und das Sortieren der Haltepunkte dauert Zeit $\mathcal{O}(n \log n)$. Im Aktualisierungsschritt müssen wir den Knoten mit minimalem bzw. maximalem Schlüssel finden, neue Schlüssel einfügen und bestehende Schlüssel löschen. Ein AVL-Baum unterstützt die Ausführung all dieser Operationen in Zeit $\mathcal{O}(\log n)$. Da die Aktualisierung an $2n$ Haltepunkten vorgenommen wird, erhalten wir eine Gesamtlaufzeit von $\mathcal{O}(n \log n)$.

Lösung 13.4 *Partnervermittlung (**Programmieraufgabe**).*

```
1  import java.util.ArrayList;
2  import java.util.LinkedList;
3  import java.util.Scanner;
4
5  class Main {
6
7      // Adjacency list representation of the graph
8      static int n;
9      static ArrayList<Integer> graph[];
10     static int[][] capacity, flow;
11
12     // Compute a path with positive residual capacity using BFS
13     // Return true if such a path exists and false otherwise
14     public static boolean augmentingPathExists(int previousVertexOnPath[]) {
15         // Allocate space for auxilary data structures
16         LinkedList<Integer> queue = new LinkedList<Integer>();
17         boolean[] visited = new boolean[n];
18
19         // Initialization of auxilary data structures
20         for(int i = 0; i < n; i++) visited[i] = false;
21         queue.add(0);
22         visited[0] = true;
23
24         // BFS
25         while(!queue.isEmpty()) {
26             int v = queue.poll();
27             for(int w : graph[v])
28                 if(!visited[w] && capacity[v][w] > flow[v][w]) {
29                     visited[w] = true;
30                     previousVertexOnPath[w] = v;
```

```
31                      queue.add(w);
32                      if(w == n-1) return true;
33                  }
34          }
35
36          return false;
37      }
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
49              // Compute smallest remaining capacity on P
50              flowOnPath = Integer.MAX_VALUE;
51              for(i = n-1; i != 0; i = previousVertexOnPath[i]) {
52                  int p = previousVertexOnPath[i];
53                  flowOnPath = Math.min(flowOnPath, capacity[p][i] - flow[p][i]);
54              }
55
56              // Add the smallest remaining capacity to each edge of P
57              for(i = n-1; i != 0; i = previousVertexOnPath[i]) {
58                  int p = previousVertexOnPath[i];
59                  flow[p][i] += flowOnPath;
60                  flow[i][p] -= flowOnPath;
61              }
62              maxFlow += flowOnPath;
63          }
64
65          return maxFlow;
66      }
67
68      public static void main(String[] args) {
69      // write your code here
70          int test, ntest, m, u, v, c, i, M, W,  maxFlow;
71          Scanner sc = new Scanner(System.in);
72          ntest = sc.nextInt();
73          for(test = 0; test < ntest; test++) {
74              //Read the number of users
75              M = sc.nextInt(); W = sc.nextInt();
76              //numbers of vertices in the graph:
77              //this is a bipartite graph with two columns
78              //one column for men and one for women.
79              //We also have two extra vertices, one for source and one for sink.
80              //Note that the source will be at vertex 0
81              //and the sink will be at the last vertex (n-1).
82              //This is because we want to use our code from last week.
83              n = M+W + 2;
84
85              // Read the number of edges
86              m = sc.nextInt();
87
88              // Initialize the graph
89              capacity = new int[n][n];
90              flow = new int[n][n];
91              graph = (ArrayList<Integer>[])new ArrayList[n];
```

```
92              for(i = 0; i < n; i++)
93                  graph[i] = new ArrayList<Integer>();
94
95              // Add the input edges to the graph
96              for(i = 0; i < m; i++) {
97                  u = sc.nextInt();
98                  v = sc.nextInt();
99
100                 // Initialize capacity, flow and store adjacency list
101                 capacity[u][M + v] = 1;
102                 flow[u][M + v] = 0;
103                 graph[u].add(M + v);
104                 capacity[M + v][u] = 0;
105                 graph[M + v].add(u);
106             }
107
108             //add the source and sink of the graph
109             for (i = 1; i <= M; i++) {
110                 //add edges from the source to men
111                 graph[0].add(i);
112                 capacity[0][i] = 1;
113                 flow[0][i] = 0;
114                 capacity[i][0] = 0;
115                 graph[i].add(0);
116             }
117             for (i = 1; i <= W; i++) {
118                 //add edges from women to the sink
119                 graph[M + i].add(n - 1);
120                 capacity[M + i][n - 1] = 1;
121                 flow[M + i][n - 1] = 0;
122                 capacity[n - 1][M + i] = 0;
123                 graph[n - 1].add(M + i);
124             }
125
126             // Compute maximum flow which is equal to the maximum matching
127             maxFlow = computeMaximumFlow();
128             System.out.println(maxFlow);
129         }
130
131     }
132 }
```