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

11. Mai 2016

Datenstrukturen & Algorithmen Lösungen zu Blatt 10 FS 16

Lösung 10.1 *Pfadplanung in Labyrinthen.*

a) Wir definieren einen gerichteten Graphen, der einen Knoten für jeden möglichen Zustand des Systems enthält. Ein Zustand wird vollständig durch drei Parameter festgelegt:

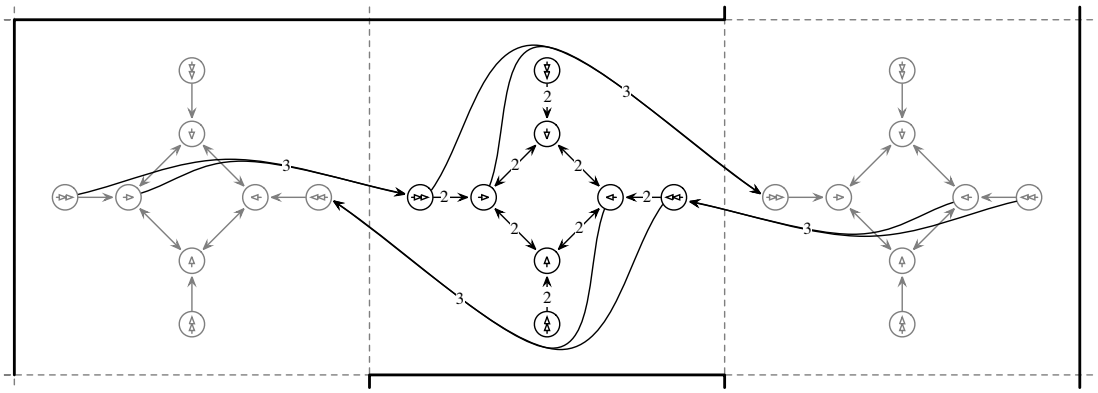
1) Das Feld auf dem der Roboter steht,

2) die Blickrichtung des Roboters und

3) ob der Roboter stillsteht oder in Bewegung ist.

Für jedes Feld, auf dem der Roboter stehen kann, haben wir also 8 Zustände (4 Blickrichtungen multipliziert mit 2 Möglichkeiten ob der Roboter steht oder sich bewegt). Wir konstruieren unseren Graphen entsprechend mit 8 Knoten pro Feld des Labyrinths. Ausserdem brauchen wir noch einen Knoten, der dem Zielzustand entspricht, in dem der Roboter entkommen ist. Wir fügen Kanten für Bewegung, Rotation und Stehenbleiben ein und gewichten diese entsprechend der Zeit, die die entsprechende Operation benötigt. Der kürzeste Pfad im Graphen vom Startzustand des Roboters zum Zielzustand entspricht einer Folge von Operationen des Roboters, die ihn schnellstmöglich entkommen lässt.

Das folgende Bild illustriert unsere Konstruktion an einem kleinen Ausschnitt eines Labyrinths. Die Beschriftung der Knoten zeigt die Blickrichtung des Roboters; Doppelpfeile bedeuten, dass er in Bewegung ist. Wir könnten natürlich Zustände entfernen, die gar nicht vorkommen können.



b) Da der konstruierte Graph keine negativen Kantengewichte enthält, kann ein kürzester Pfad mithilfe des Algorithmus von Dijkstra gefunden werden.

c) Für jedes Feld des Labyrinths wird eine konstante Anzahl Knoten (nämlich 8) und eine konstante Anzahl Kanten (höchstens 20) verwendet. Ist n die Anzahl der Felder im Labyrinth, dann sind $|V| \in \mathcal{O}(n)$ und $|E| \in \mathcal{O}(n)$. Die Laufzeit des Algorithmus von Dijkstra ist also durch $\mathcal{O}(n \log n)$ beschränkt.

Anmerkung: Für einen gerichteten, ungewichteten Graphen kann ein kürzester Pfad auch mit einer Breitensuche gefunden werden. Da unser Graph nur ganzzahlige Kantengewichte hat, die alle größer als 0 und durch eine Konstante nach oben beschränkt sind, können wir einen erweiterten, ungewichteten Graphen konstruieren, indem wir eine Kante mit Gewicht $w > 1$ durch w Kanten mit je Gewicht 1 ersetzen. Dazu fügen wir $w-1$ Hilfsknoten ein. Hat diese Kante in unserem ursprünglichen Graphen einen Knoten x, der mit einem Knoten y verbunden ist, so sind diese Knoten im erweiterten Graphen durch einen Pfad bestehend aus w Kanten verbunden. Somit haben im erweiterten Graphen alle Kanten Gewicht 1 und wir können die Breitensuche verwenden, um einen kürzesten Pfad zu finden. Da die Kantengewichte durch eine Konstante nach oben beschränkt sind, werden insgesamt nur $\mathcal{O}(n)$ Knoten und Kanten hinzugefügt und die Laufzeit ist in $\mathcal{O}(n)$.

Lösung 10.2 *Varianten zu Kürzeste-Wege-Problemen.*

a) Zur Lösung kann eine Modifikation des Algorithmus von Bellman und Ford benutzt werden. Für jeden Knoten $v \in V$ und jedes $i \in \mathbb{N}$ speichere die Länge $d_{i,v}$ eines kürzesten Weges von s nach v, der aus höchstens i Kanten besteht. Weiterhin sei $\pi_{i,v}$ der Vorgängerknoten von v auf einem kürzesten (s, v)-Weg mit höchstens i Kanten. Wir erhalten den folgenden Pseudocode:

BELLMAN-FORD(V, E, w, s)

Eingabe: Gerichteter, gewichteter Graph (V, E, w), Startknoten $s \in V$

Ausgabe: Vorgängerknoten $\pi_{i,v}$ für alle $v \in V$ und alle $i \in \{0, \ldots, k\}$

```
1 for each v ∈ V do d_{0,v} ← ∞; π_{0,v} ← null
2 d_{0,s} ← 0
3 for i ← 1, ..., k + 1 do
4     for each v ∈ V do d_{i,v} ← d_{i−1,v}; π_{i,v} ← π_{i−1,v}
5     for each (u, v) ∈ E do
6         if d_{i−1,u} + w((u, v)) < d_{i,v} then
7             d_{i,v} ← d_{i−1,u} + w((u, v))
8             π_{i,v} ← u
```

Nach i Durchläufen der Schleife im dritten Schritt wurden alle Pfade vom Startknoten aus berücksichtigt, die aus höchstens i Kanten bestehen. Nach $k + 1$ Durchläufen haben wir also alle Pfade betrachtet, die aus höchstens $k + 1$ Kanten bestehen, d.h., die höchstens k Zwischenknoten besitzen (der normale Algorithmus durchliefe die Schleife $|V| - 1$ Mal, um *alle* möglichen kürzesten Wege zu berücksichtigen). Ein kürzester (s, t)-Weg mit höchstens $k+1$ Kanten kann dann von t ausgehend rückwärts rekonstruiert werden, indem wir initial $v \leftarrow t$ setzen und dem jeweiligen Vorgängerknoten von v so lange folgen, bis ein Knoten ohne Vorgänger (also der Startknoten s) erreicht wird. Diese Aufgabe kann wie folgt gelöst werden:

RECONSTRUCT-SHORTEST-PATH($\pi_{i,v}$, s, t)

Eingabe: Vorgängerknoten $\pi_{i,v}$ für alle $v \in V$ und alle $i \in \{0, \ldots, k\}$, Knoten s und t

Ausgabe: Ein kürzester (s, t)-Weg $P = \langle v_0, v_1, \ldots, v_l \rangle$ mit $v_0 = s$, $v_l = t$ und $l \leq k + 1$

1 **if** $d_{k+1,t} = \infty$ **then** Melde, dass kein (s,t)-Weg mit höchstens $k+1$ Kanten existiert.
2 $v \leftarrow t$; $P \leftarrow \langle\rangle$; $i \leftarrow k+1$
3 **while** $v \neq$ **null do** $P \leftarrow v \oplus P$; $v \leftarrow \pi_{i,v}$; $i \leftarrow i-1$
4 **return** P

Dabei sei P als sortierte Liste gespeichert. Der Operator $\oplus$ beschreibt die Konkatenation (Aneinanderhängung) zweier Listen, und $v \oplus P$ sei die Liste, die entsteht, wenn v an den Anfang von P eingefügt wird.

Im ersten Algorithmus benötigt jeder Schleifendurchlauf Laufzeit $\mathcal{O}(|V|+|E|)$. Zusätzlich kann mithilfe des zweiten Algorithmus ein kürzester Pfad in Zeit $\mathcal{O}(|V|)$ rekonstruiert werden. Die Gesamtlaufzeit unserer Lösung ist also in $\mathcal{O}(k \cdot (|V|+|E|))$, d.h. $\mathcal{O}(k \cdot |E|)$, wenn der Graph zusammenhängend ist.

b) Wir berechnen zuerst den kürzesten Weg von s nach t. Sei l die Anzahl der Kanten, die von diesem Weg benutzt werden.

Wir beschränken uns zuerst auf *einfache* Wege (kein Knoten wird mehrfach besucht): Der zweitkürzeste, einfache (s,t)-Weg muss sich vom kürzesten (s,t)-Weg in mindestens einer Kante unterscheiden, sonst wären die Wege ja gleich. Wir können deshalb jeweils eine der l Kanten des kürzesten Weges aus dem Graphen entfernen, und nochmals den kürzesten Weg berechnen. Es werden also l kürzeste Wege in einem Graphen mit einer (in jedem Schritt unterschiedlichen) Kante weniger berechnet. Von diesen l kürzesten Wegen nehmen wir denjenigen mit kürzester Länge, und erhalten somit den zweitkürzesten, einfachen Weg. Man beachte, dass sich die Länge dieses Weges nicht unbedingt von der Länge des kürzesten Weges unterscheiden muss. Offenbar ist die Laufzeit durch $\mathcal{O}(l \cdot$ (Laufzeit zur Berechnung eines kürzesten Weges)) beschränkt. Im schlimmsten Fall besucht der kürzeste Weg alle Knoten und benutzt $|V|-1$ Kanten. Wird der Algorithmus von Dijkstra zur Berechnung der kürzesten Wege benutzt, dann ist die Gesamtlaufzeit $\mathcal{O}(|V| \cdot |E| + |V|^2 \cdot \log(|V|))$.

Um nicht-einfache Wege ebenfalls zu berücksichtigen, berechnen wir zusätzlich die kürzeste Distanz zwischen jedem Knoten v und sich selbst (d.h. die Länge des kürzesten Weges von v zu v der mindestens eine Kante benutzt). Falls der zweitkürzeste Weg nicht einfach ist, aber alle Kanten des kürzesten Weges benutzt, so wird ein Knoten mehrfach besucht. Wir addieren also zur Länge des kürzesten Weges die minimale Distanz eines Knoten v, der von dem kürzesten Weg benutzt wird, zu sich selbst. Dies ist die Länge des zweitkürzesten, nicht-einfachen Weges von s nach t, sofern der zweitkürzeste einfache Weg länger ist.

Bemerkung: Es gibt einen Algorithmus, der für ein gegebenes $k \in \mathbb{N}$ die k besten nicht-einfachen Pfade (bei denen Knoten/Kanten mehrfach benutzt werden dürfen) zwischen zwei Knoten in Zeit $\mathcal{O}(|E|+|V|\log|V|+k)$ berechnet (Details finden sich in "Finding the k Shortest Paths", D. Eppstein, FOCS, 1994).

c) Wir benutzen den Algorithmus von Dijkstra, speichern aber für jeden Knoten $v \in V$ nicht allein die Distanz d_v von s zu v, sondern zusätzlich noch die Anzahl N_v der Wege mit dieser Distanz. Für alle $v \in V \backslash \{s\}$ setzen wir initial $d_v \leftarrow \infty$ und $N_v \leftarrow 0$. Zusätzlich werden $d_s \leftarrow 0$ und $N_s \leftarrow 1$ gesetzt. Der Test, ob sich die Abkürzung über einen Knoten u lohnt, um v zu erreichen, wird wie folgt angepasst: Falls $d_u + w((u,v)) < d_v$ ist (d.h., der kürzeste Weg über u kürzer als der bisher kürzeste Weg zu v ist), dann werden $d_v \leftarrow d_u + w((u,v))$ und $N_v \leftarrow N_u$ gesetzt. Ist dagegen $d_u + w((u,v)) = d_v$ (d.h., die Weglänge der Abkürzung über u ist genauso lang wie die aktuelle Weglänge), dann wird N_v um N_u erhöht (denn alle kürzesten Wege, die durch u verlaufen, kommen noch zusätzlich hinzu).

Die Korrektheit dieses Vorgehens folgt aus der Tatsache, dass der Algorithmus von Dijkstra jede Kante nur einmal relaxiert. Dabei ändert sich die Laufzeit im Vergleich zum üblichen Algorithmus nicht, da pro Kante nur konstant viele zusätzliche Operationen anfallen.

Lösung 10.3 *Closeness centrality (Programmieraufgabe).*

```
1  import java.util.Scanner;
2
3  class Main {
4
5      // graph holds the distance between every pair of vertices
6      private int[][] graph;
7
8      public Main(int n) {
9          this.graph = new int[n][n];
10         initGraph();
11     }
12
13     // graph initialization.
14     // the distance between every pair of nodes is
15     // initialized to 100000 (arbitrary large enough value).
16     // for i==j it is initialized to 0.
17     private void initGraph() {
18         for (int i = 0; i < graph.length; i++) {
19             for (int j = 0; j < graph.length; j++) {
20                 if (i == j) {
21                     graph[i][j] = 0;
22                 } else {
23                     graph[i][j] = 100000;
24                 }
25             }
26         }
27     }
28
29     // utility function to add edges to the adjacencyList
30     public void addEdge(int from, int to) {
31         //sinlge edge has distance 1
32         graph[from][to] = 1;
33     }
34
35     // all-pairs shortest path
36     public void floydWarshall() {
37         int n = graph.length;
38
39         //this is the three nested loops for Floyd-Warshall
40         for (int k = 0; k < n; k++) {
41             for (int i = 0; i < n; i++) {
42                 for (int j = 0; j < n; j++) {
43                     if (graph[i][j] > graph[i][k] + graph[k][j]) {
44                         graph[i][j] = graph[i][k] + graph[k][j];
45                     }
46                 }
47             }
48         }
49
50         //computation of closeness centrality
51         //dk is the sum of the distances for vertex k
52         for (int k = 0; k < n; k++) {
53             int dk = 0;
54             for (int i = 0; i < n; i++) {
55                 if (graph[k][i] < 100000)
56                     dk += graph[k][i];
57             }
58             System.out.println(dk);
59         }
```

```
60      }
61
62      public static void main(String[] args) {
63          int n; //number of nodes
64          Scanner sc = new Scanner(System.in);
65          n = sc.nextInt();
66
67          Main fw = new Main(n);
68
69          int m; //number of edges of each node
70          int ignore; //an integer variable in which we store the input values
71                      //that we ignore for this exercise
72          for (int i=0; i < n; i++) {
73              ignore = sc.nextInt();
74              m = sc.nextInt();
75              ignore = sc.nextInt();
76              String name = sc.nextLine();
77
78              for (int j=0; j<m; j++) {
79                  int edgeto = sc.nextInt();
80                  //the -1 is because indices in the input file start from 1
81                  fw.addEdge(i,edgeto-1);
82              }
83          }
84
85          fw.floydWarshall();
86      }
87  }
```