

Datenstrukturen und effiziente Algorithmen

Blatt 8

Markus Vieth, David Klopp, Christian Stricker

5. Januar 2016

Aufgabe 1

a) Pseudo-Code

```

1 forall ( v in V \{ S }) {
2     col[v] = white ;
3     d[v] = infinity ;
4 }

6 List kanten = [];
7 col[s] = grey ;
8 d[s] = 0;
9 Queue Q;
10 Q.push ( s );
11 while (!Q.empty ()) {
12     u = Q.pop();
13     forall ((u , v ) in E) {
14         if (d[u] < d[v])
15             kanten.add((u,v));

17         if (col[v] == white) {
18             col[v] == grey ;
19             d[v] = d[u]+ 1 ;
20             Q.push(v);
21         }
22     }
23     col[u] = black ;
24 }
```

Der Algorithmus funktioniert analog zur Breitensuche, allerdings wird eine Kante nur dann hinzugefügt zum Levelgraph, wenn die Distanz des Nachbarknotens größer ist als die Distanz des aktuellen Knoten zum Ursprung (z. 14-15). Die Laufzeit verhält sich daher identisch zur Breitensuche und ist somit linear: $O(|V| + |E|)$.

b)

Der Levelgraph soll lediglich kürzeste Wege enthalten. Eine Kante von rechts nach links würde bedeuten, dass der Weg länger wird als der bisherige Weg. Dieser Weg sollte dementsprechend nicht im Levelgraph beinhaltet sein, da dieser nur kürzeste Wege enthält.

c)

Der Graph beinhaltet alle mögliche kürzesten Wegen zu einem Knoten, daher können auch bei erneuten Generierung keine neuen kürzesten Wege gefunden werden oder wegfallen, sofern sich der Graph nicht geändert hat. D.h der Graph ist eindeutig.

Aufgabe 2

ii)

```

1  package app;

3  // java default stuff
4  import java.util.Vector;
5  import java.util.ArrayList;
6  import java.util.HashMap;
7  import java.util.HashSet;
8  import java.util.LinkedList;

10 class Search{

12     public static Vector<Tuple> BFS (boolean [][][] cheese, Tuple origin){

14         HashMap<Tuple, ArrayList<Tuple>> edges = createHash(cheese.length, cheese);
15         HashSet<Tuple> visited = new HashSet<Tuple>();
16         LinkedList<Tuple> queue = new LinkedList<Tuple>(); //queue
17         HashMap<Tuple, Tuple> pi = new HashMap<Tuple, Tuple>();
18         HashMap<Tuple, Integer> d = new HashMap<Tuple, Integer>(); //Distanzfunktion
19         visited.add(origin);
20         pi.put(origin, null);
21         d.put(origin, 0);

23         queue.add(origin); //queue
24         while(queue.size() > 0) {
25             Tuple u = queue.poll(); //queue
26             ArrayList<Tuple> edge = edges.get(u);
27             for ( int i = 0; i < edge.size(); i++) {
28                 Tuple next = edge.get(i);
29                 if (!visited.contains(next)) {
30                     visited.add(next);
31                     pi.put(next, u);
32                     queue.add(next); //queue
33                     d.put(next, d.get(u)+1); //Länge des neuen Knoten ist Länge des Entdeckers +1
34                 }
35                 if (next.one == 0) {
36                     Vector<Tuple> temp = new Vector<Tuple>();
37                     temp.addElement(next);
38                     System.out.println("BFS: Laenge = "+d.get(next));
39                     while (pi.get(next) != null) {
40                         next = pi.get(next);
41                         temp.addElement(next);
42                     }
43                     return temp;
44                 }
45             }
46         }
47         return new Vector<Tuple>();
48     }

50     public static Vector<Tuple> DFS (boolean [][][] cheese, Tuple origin){

52         HashMap<Tuple, ArrayList<Tuple>> edges = createHash(cheese.length, cheese);
53         HashSet<Tuple> visited = new HashSet<Tuple>();
54         LinkedList<Tuple> stack = new LinkedList<Tuple>(); //stack
55         HashMap<Tuple, Tuple> pi = new HashMap<Tuple, Tuple>();
56         HashMap<Tuple, Integer> d = new HashMap<Tuple, Integer>(); //Distanzfunktion
57         visited.add(origin);
58         pi.put(origin, null);
59         d.put(origin, 0);

61         stack.push(origin); //stack
62         while(stack.size() > 0) {
63             Tuple u = stack.pop(); //stack
64             ArrayList<Tuple> edge = edges.get(u);
65             for ( int i = 0; i < edge.size(); i++) {

```

```

66     Tuple next = edge.get(i);
67     if (!visited.contains(next)) {
68         visited.add(next);
69         pi.put(next, u);
70         stack.push(next); //stack
71         d.put(next, d.get(u)+1); //Länge des neuen Knoten ist Länge des Entdeckers +1
72     }
73     if (next.one == 0) {
74         Vector<Tuple> temp = new Vector<Tuple>();
75         temp.addElement(next);
76         System.out.println("DFS: Laenge = "+d.get(next));
77         while (pi.get(next) != null) {
78             next = pi.get(next);
79             temp.addElement(next);
80         }
81         return temp;
82     }
83 }
84 }
85     return new Vector<Tuple>();
86 }

89 //hilfsmethode zur Erzeugung der Mashmap der möglichen Züge in einem Puzzle
90 private static HashMap<Tuple, ArrayList<Tuple>> createHash(int n, boolean [][][] cheese) {

91     //Erstelle Hashmap
92     HashMap<Tuple, ArrayList<Tuple>> edges = new HashMap<Tuple, ArrayList<Tuple>>();
93     for (int x = 0; x < n; x++)
94         for (int y = 0; y < n; y++)
95             for (int z = 0; z < n; z++) {
96                 ArrayList<Tuple> temp = new ArrayList<Tuple>();
97                 edges.put(new Tuple(x,y,z), temp);
98                 if ( x > 0)
99                     if (!cheese[x-1][y][z])
100                         temp.add(new Tuple((x-1), y, z));
101                 if ( x < n-1)
102                     if (!cheese[x+1][y][z])
103                         temp.add(new Tuple((x+1), y, z));
104                 if ( y > 0)
105                     if (!cheese[x][y-1][z])
106                         temp.add(new Tuple( x, (y-1), z));
107                 if ( y < n-1)
108                     if (!cheese[x][y+1][z])
109                         temp.add(new Tuple( x, (y+1), z));
110                 if ( z > 0)
111                     if (!cheese[x][y][z-1])
112                         temp.add(new Tuple( x ,y, (z-1)));
113                 if ( z < n-1)
114                     if (!cheese[x][y][z+1])
115                         temp.add(new Tuple( x, y, (z+1)));
116             }
117     }
118     return edges;
119 }

121 }

```

iii)

Für die Breitensuche ergibt sich eine Weglänge von 56, für die Tiefensuche eine Länge von 2317. Laut Vorlesung liefert die Breitensuche stets den kürzeren Weg.