Datenstrukturen und effiziente Algorithmen
Blatt 12

Markus Vieth, David Klopp, Christian Stricker

26. Januar 2016

Aufgabe 1

a)

Definiere zwei Farben (hier rot und blau)
Wähle einen beliebigen Knoten als "CurrentVertex" und setze ihn auf blau.

```
Wiederhole {
  Füge alle Nachbarknoten in eine Warteschlange.
  Falls CurrentVertex blau und mindestens einer der Nachbarknoten auch blau {
    Rückgabe: FALSE
  } ansonsten {
    setze alle Nachbarknoten auf rot
  }

  Falls CurrentVertex rot und mindestens einer der Nachbarknoten auch rot {
    Rückgabe: FALSE
  } ansonsten {
    setze alle Nachbarknoten auf blau
  }
  Wähle den nächsten Knoten aus der Warteschlange als CurrentVertex
}

Falls die Liste leer ist {
  Rückgabe: TRUE
}
```

a)

```
public interface UndirectedGraph {

  public void insertVertex(int value);
  public void insertEdge(int value1, int value2);
  public boolean isBipartite();

}
```

```
import java.util.HashMap;
import java.util.LinkedList;
import java.util.Vector;

public class BipartiterGraph implements UndirectedGraph{
  HashMap<Integer,Vertex> map = new HashMap<Integer, Vertex>(); //Speichert alle
      Knoten
  LinkedList<Vertex> list = new LinkedList<Vertex>();    //Warteschlange für die
      noch zu untersuchenden Knoten
  Vertex beginVertex;                  //Ein Knoten, mit dem der Algo beginnt

  class Edge{
```

```
11      Vertex start;  //Speichert seine beiden Knoten
12      Vertex end;

14      public Edge(Vertex start, Vertex end){
15        this.start = start;
16        this.end = end;
17      }
18    }

20    //Klasse Vertex, Speichert all seine Kanten in edge, speichert seine Prüffarbe
          und ein value
21    class Vertex{

23      Vector<Edge> edge = new Vector<Edge>();

25      String colour;
26      int value;

28      public Vertex(int value){ //Konstruktor mit eindeutigem Key/Value
29        this.value = value;
30      }

32      public void addEdge(Edge edge){ //Fügt dem Knoten eine seiner Kanten hinzu
33        this.edge.add(edge);
34      }
35    }

37    public BipartiterGraph(){

39      }

41    @Override
42    public void insertVertex(int value) {
43      Vertex vertex = new Vertex(value);  //Erzeugt einen Knoten
44      this.map.put(value, vertex);   //Fügt einen Knoten der Hashmap hinzu
45      if(this.beginVertex == null){   //Falls noch kein Knoten existiert, wird dieser
            als Startknoten definiert
46        this.beginVertex = vertex;
47      }
48    }

50    @Override
51    public void insertEdge(int value1, int value2) {
52      Vertex start = this.map.get(value1); //Sucht einen Endknoten der Kante
53      Vertex end = this.map.get(value2);  //Sucht den anderen Endknoten der Kante
54      if(start != null && end != null){  //Wenn beide existieren, füge beiden Knoten
            die Kante hinzu
55        Edge edge = new Edge(start, end);
56        start.addEdge(edge);
```

```
57      end.addEdge(edge);
58    }
59    else                //Ansonsten gibt Hinweis aus, das die Kante nicht im Algo
          berücksichtigt wird
60      System.out.println(" Mindestens einer der Knotenenden der Kante \" "+ value1
          +"---" + value2 + " \" existiert nicht! \n Diese Kante wird nicht dem
          Graphen hinzugefügt!");
61  }

63  @Override
64  public boolean isBipartite() {
65    this.beginVertex.colour = "blue"; //Setzt den Anfangsknoten auf eine bestimmte
          Farbe (blue)
66    list.add(this.beginVertex);   //Fügt den ersten Knoten der Warteschlange hinzu
67    Vertex next;          //Hilfsvariablen
68    Vertex currentVertex = beginVertex;
69    if(!this.map.isEmpty()){   //Wenn ein Graph existiert, starte
70      do{
71        currentVertex = list.getFirst(); //Der erste Knoten wird aus der Liste
              genommen

73        String colour = currentVertex.colour;
74        for(Edge edge : currentVertex.edge){ //Iteriert über alle Kanten des
              derzeitigen Knotens
75          if(!edge.start.equals(currentVertex) || !edge.end.equals(currentVertex)){
                //

77            if(edge.start.equals(currentVertex)){ //Bestimme den Nachbarknoten,
                  schließe den derzeitigenn Knoten als Nachbarknoten aus
78              next = edge.end;
79            }else
80              next = edge.start;

82            if(next.colour == null)  //Wenn der Knoten bisher noch nciht in der
                  Warteschlange gewesen ist, d.h. die Farbe ist noch null, füge den
                  Knoten dort ein
83              list.add(next);

85            if(next.colour == colour) //Wenn die Farbe zu dem derzeitigen Knoten
                  gleich ist, haben zwei Nachbarknoten die gleihce Farbe => kien
                  Bipartiter Graph
86              return false;
87            else if(currentVertex.colour.equals("red")) //Ansonsten färbe den Knoten
                   in der zum currentVertex verschiedenen Farbe
88              next.colour = "blue";
89            else
90              next.colour = "red";
91          }
92        }
```

```
93          list.remove(currentVertex); //Wenn alle Kanten abgehackt sind, lösche den
                derzeitigen Knoten aus der Liste(Listenanfang)
94        }while(!list.isEmpty());  //Wenn die Warteschlange leer ist und bisher kein
              Konflikt enstanden ist, gebe true zurück.
95      }
96      return true;
97    }

99    public static void main(String[] args) {
100     BipartiterGraph i = new BipartiterGraph();  //Der Konstruktor baut den Graphen
            auf und überprüft ihn. Funktionen müssen nicht static sein
101   }

103 }
```

Aufgabe 2

Aufgabe 3

a)

Knoten	Input	Output	Differenz
0	12	5 + 7	0
1	5	5	0
2	9	6 + 3	0
3	6 +7	13	0
4	0	0	0
5	0 + 3	3	0

Der Input ist niemals größer als die maximale Kapazität. Für alle Knoten bis auf s und t stimmt die Anzahl des Inputs mit der des Outputs überein. Es handelt sich daher um einen gültigen Fluss.