TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

André Stender
IADB185489
IT süsteemide arendus

INDIVIDUAALTÖÖ AINES „ALGORITMID JA ANDMESTRUKTUURID“

Juhendaja: Jaanus Pöial

Tallinn 2019

Sisukord

1 Ülesande püstitus

Töö eesmärk on tutvuda lähemalt graafidega aines „algoritmid ja andmestruktuurid.“ Selle jaoks on valitud probleem, mis lahendadakse programmikoodiga kasutades Java programmeerimiskeelt. Probleem tuleb lahendada kasutades juhendaja poolt määratud malli [1]. Käesoleva töö ülesanne on valitud ülesannete kogust [2]. Valitud ülesande püstitus kõlab järgnevalt: „kirjutada algoritm, mis etteantud orienteeritud graafi korral kontrollib, kas graaf on tugevalt sidus. Graafi töö käigus muuta pole lubatud.“ [3] Lisaselgitusena - Graaf on tugevalt sidus, kui graafi igast tipust leidub tee graafi igasse teise tippu.

2 Lahenduse kirjeldus

Probleemi lahendamiseks on kasutatud Kosaraju algoritmiga sarnanevat loogikat. Kosaraju algoritmiga on võimalik leida graafi tugevalt sidusad komponendid. Selleks, et kontrollida kas graaf on tugevalt sidus, tuleb rakendada järgnevat loogikat:

1) Rakenda graafi igale tipule sügavutti läbimise (DFS) algoritm. Kui leidub mõni selline tipp, mis ei läbi kõiki tippe, siis graaf ei ole tugevalt sidus ning programm tagastagu false.

2) Transponeeri graaf. Rakenda transponeeritud graafi igale tipule samuti sügavutti läbimise (DFS) algoritm. Kui leidub mõni selline tipp, mis ei läbi kõiki tippe, siis graaf ei ole tugevalt sidus ning programm tagastagu false.

3) Kui programm läbib eelnevalt nimetatud punktid edukalt, siis graaf on tugevalt sidus ning program tagastagu true.

2.1 Sügavutti läbimine

Lahenduse ühe olulise osana on kasutatud sügavutti läbimise algoritmi, mis kasutab LIFO (Last In First Out) saavutamiseks magasini (Stack) ning tagastab lõpuks listi läbitud tippudest õiges järjekorras. Selle algoritmi pseudokood on järgnev [4]:

```
1  function DFS(Vertex v):
2      let D be a list of discovered vertices
3      let S be a stack
4      S.push(v)
5      while S is not empty:
6          v = S.pop()
7          if v is not discovered:
8              label v as discovered
9              for each edge in v:
10                 let W be a target vertex of edge
11                 S.push(w)
12     return D
```

Sügavutti läbimise algoritmi ajaline keerukus on lineaarne O(V+E) [5], kus V tähistab tippude arvu ning E tähistab kaarte arvu. Seda algoritmi rakendatakse lahenduses 2V korda – graafi igale tipule ning transponeeritud graafi igale tipule.

2.2 Graafi transponeerimine

Lahenduse saamiseks on tarvis leida lahendatava graafi transponeeritud graaf. Lahenduses on graaf transponeeritud järgnevate sammudega:

1) Leida graafi külgnevusmaatriks. Selle jaoks on juhendaja poolt määratud mallis juba meetod olemas.

2) Leida graafi transponeeritud külgnevusmaatriks. Seda saab teha kasutades esimesel sammul leitud külgnevusmaatriksit.

3) Luua uus transponeeritud graaf. Seda saab teha kasutades teisel sammul leitud transponeeritud külgnevusmaatriksit.

Graafi transponeerimise ajaline keerukus läbi külgnevusmaatriksi on $O(V^2)$, kus V tähistab tippude arvu.

3 Programmi kasutamisjuhend

Programmis on olemas meetod nimega run(), mis on mõeldud erinevate juhtude ja näidete koostamiseks. Selles meetodis tuleks esmalt luua graafi objekt, tehes seda näiteks järgnevalt: *Graph g = new Graph("G");*.

Nüüd on meil olemas uus tühi graaf, millele saab lisada tippe ja kaari. Tippe ja kaari on võimalik lisada randomiseeritud viisil, mille jaoks on olemas kaks meetodit. Meetod *createRandomSimpleGraph(int n, int m)* tekitab juhusliku orienteerimata graafi n tipu ja m kaarega. Meetod *createRandomDirectedGraph(int n, int m)* tekitab juhusliku orienteeritud graafi n tipu ja m kaarega.

Tippe ja kaari on võimalik luua ka käsitsi. Meetod *createVertex(String vid)* loob tipu mille id on vid. Meetod *createArc(String vid, Vertex from, Vertex to)* loob kaare mille id on vid, algustipp on from ning sihttipp on to.

Programmis on olemas meetod *isStronglyConnected()*, mis kontrollib, kas graaf on tugevalt sidus või mitte. See meetod tagastab boolean tüüpi väärtuse – true, kui graaf on tugevalt sidus ning false, kui graaf ei ole tugevalt sidus. Näiteks võime meetodit rakendada üleval loodud graafile ning meile tagastatakse vastavat tüüpi boolean väärtus: *g.isStronglyConnected();*.

4 Testimiskava

Testimiseks on loodud kuus erinevat juhtu. Lisaks kuuele juhule on testimiseks loodud ka kaks suuremat juhuslikku graafi (tippude arv >= 2000, kaarte arv >= 2000), et kontrollida programmi läbimiseks kuluvat aega. Lahendusnäited on lisatud koos jooniste ning programmi väljunditega käesolevas töös Lisa 2 all.

5 Viited

[1] J. Põial. [Võrgumaterjal]. Available: https://bitbucket.org/itc_algorithms/k6.git.

[2] A. Peder, J. Kiho ja H. Nestra, Algoritmid ja andmestruktuurid Ülesannete kogu, Tartu, 2017.

[3] A. Peder, J. Kiho ja H. Nestra, „Graafi läbimine," %1 *Algoritmid ja andmestruktuurid Ülesannete kogu*, Tartu, 2017, p. 90.

[4] J. M. Kleinberg ja E. Tardos, „Implementing Graph Traversal Using Queues and Stacks," %1 *Algorithm Design*, p. 93.

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest ja C. S. Stein, „Depth-first search," %1 *Introduction to Algorithms*, MIT Press, 2009, p. 606.

6 Lisad

6.1 Lisa 1 (programmi täielik tekst)

```
import java.util.*;

/** Container class to different classes, that makes the whole
 * set of classes one class formally.
 */
public class GraphTask {

   /** Main method. */
   public static void main (String[] args) {
      GraphTask a = new GraphTask();
      a.run();
   }

   /** Actual main method to run examples and everything. */
   public void run() {
      // Graph G0
      Graph g0 = new Graph("G0");
      g0.createVertex("v0");

      System.out.println(g0);
      System.out.println("Graph " + g0.id + " is strongly connected: " +
g0.isStronglyConnected());

      // Graph G1.
      Graph g1 = new Graph("G1");
      int n = 3;
      Vertex[] varray = new Vertex[n];

      for (int i = 0; i < n; i++) {
         Vertex v = g1.createVertex("v" + i);
```

```
            varray[i] = v;
        }

        g1.createArc("a" + varray[0].toString() + "_" + varray[1].toString(),
varray[0], varray[1]);
        g1.createArc("a" + varray[1].toString() + "_" + varray[2].toString(),
varray[1], varray[2]);
        g1.createArc("a" + varray[2].toString() + "_" + varray[0].toString(),
varray[2], varray[0]);

        System.out.println(g1);
        System.out.println("Graph " + g1.id + " is strongly connected: " +
g1.isStronglyConnected());

        // Graph G2
        Graph g2 = new Graph("G2");
        n = 6;
        varray = new Vertex[n];

        for (int i = 0; i < n; i++) {
            Vertex v = g2.createVertex("v" + i);
            varray[i] = v;
        }

        g2.createArc("a" + varray[0].toString() + "_" + varray[1].toString(),
varray[0], varray[1]);
        g2.createArc("a" + varray[0].toString() + "_" + varray[2].toString(),
varray[0], varray[2]);
        g2.createArc("a" + varray[2].toString() + "_" + varray[3].toString(),
varray[2], varray[3]);
        g2.createArc("a" + varray[2].toString() + "_" + varray[4].toString(),
varray[2], varray[4]);
        g2.createArc("a" + varray[4].toString() + "_" + varray[5].toString(),
varray[4], varray[5]);

        System.out.println(g2);
        System.out.println("Graph " + g2.id + " is strongly connected: " +
g2.isStronglyConnected());

        // Graph G3
        Graph g3 = new Graph("G3");
        n = 6;
        varray = new Vertex[n];

        for (int i = 0; i < n; i++) {
            Vertex v = g3.createVertex("v" + i);
            varray[i] = v;
        }

        g3.createArc("a" + varray[0].toString() + "_" + varray[2].toString(),
varray[0], varray[2]);
        g3.createArc("a" + varray[2].toString() + "_" + varray[1].toString(),
varray[2], varray[1]);
        g3.createArc("a" + varray[1].toString() + "_" + varray[0].toString(),
varray[1], varray[0]);
        g3.createArc("a" + varray[1].toString() + "_" + varray[3].toString(),
varray[1], varray[3]);
        g3.createArc("a" + varray[3].toString() + "_" + varray[1].toString(),
varray[3], varray[1]);
        g3.createArc("a" + varray[3].toString() + "_" + varray[4].toString(),
varray[3], varray[4]);
        g3.createArc("a" + varray[4].toString() + "_" + varray[5].toString(),
varray[4], varray[5]);
        g3.createArc("a" + varray[5].toString() + "_" + varray[3].toString(),
varray[5], varray[3]);
```

```
      System.out.println(g3);
      System.out.println("Graph " + g3.id + " is strongly connected: " +
g3.isStronglyConnected());

      // Graph G4
      Graph g4 = new Graph("G4");
      n = 5;
      varray = new Vertex[n];

      for (int i = 0; i < n; i++) {
         Vertex v = g4.createVertex("v" + i);
         varray[i] = v;
      }

      g4.createArc("a" + varray[0].toString() + "_" + varray[1].toString(),
varray[0], varray[1]);
      g4.createArc("a" + varray[1].toString() + "_" + varray[2].toString(),
varray[1], varray[2]);
      g4.createArc("a" + varray[2].toString() + "_" + varray[3].toString(),
varray[2], varray[3]);
      g4.createArc("a" + varray[2].toString() + "_" + varray[4].toString(),
varray[2], varray[4]);
      g4.createArc("a" + varray[3].toString() + "_" + varray[0].toString(),
varray[3], varray[0]);
      g4.createArc("a" + varray[4].toString() + "_" + varray[2].toString(),
varray[4], varray[2]);

      System.out.println(g4);
      System.out.println("Graph " + g4.id + " is strongly connected: " +
g4.isStronglyConnected());

      // Graph G5
      Graph g5 = new Graph("G5");
      n = 5;
      varray = new Vertex[n];

      for (int i = 0; i < n; i++) {
         Vertex v = g5.createVertex("v" + i);
         varray[i] = v;
      }

      g5.createArc("a" + varray[0].toString() + "_" + varray[1].toString(),
varray[0], varray[1]);
      g5.createArc("a" + varray[1].toString() + "_" + varray[2].toString(),
varray[1], varray[2]);
      g5.createArc("a" + varray[2].toString() + "_" + varray[3].toString(),
varray[2], varray[3]);
      g5.createArc("a" + varray[3].toString() + "_" + varray[0].toString(),
varray[3], varray[0]);
      g5.createArc("a" + varray[4].toString() + "_" + varray[2].toString(),
varray[4], varray[2]);

      System.out.println(g5);
      System.out.println("Graph " + g5.id + " is strongly connected: " +
g5.isStronglyConnected());

      // Random directed graph G6
      Graph g6 = new Graph("G6");
      n = 2000;
      int m = 2500;
      g6.createRandomDirectedGraph(n, m);
      System.out.println();
      System.out.println("Graph " + g6.id + " is with " + n + " vertices and "
+ m + " edges.");

      // Check execution time of G6
```

```
      long startTime = System.currentTimeMillis();
      g6.isStronglyConnected();
      long estimatedTime = System.currentTimeMillis() - startTime;
      System.out.println("Execution time of graph " + g6.id + ": " +
estimatedTime + "ms");

      // Random undirected graph G7
      Graph g7 = new Graph("G7");
      n = 2000;
      m = 2500;
      g7.createRandomSimpleGraph(n, m);
      System.out.println();
      System.out.println("Graph " + g7.id + " is with " + n + " vertices and "
+ m + " edges.");

      // Check execution time of G7
      startTime = System.currentTimeMillis();
      g7.isStronglyConnected();
      estimatedTime = System.currentTimeMillis() - startTime;
      System.out.println("Execution time of graph " + g7.id + ": " +
estimatedTime + "ms");
   }

   class Vertex {

      private String id;
      private Vertex next;
      private Arc first;
      private int info = 0;

      Vertex (String s, Vertex v, Arc e) {
         id = s;
         next = v;
         first = e;
      }

      Vertex (String s) {
         this (s, null, null);
      }

      @Override
      public String toString() {
         return id;
      }

   }


   /** Arc represents one arrow in the graph. Two-directional edges are
    * represented by two Arc objects (for both directions).
    */
   class Arc {

      private String id;
      private Vertex target;
      private Arc next;
      private int info = 0;

      Arc (String s, Vertex v, Arc a) {
         id = s;
         target = v;
         next = a;
      }

      Arc (String s) {
         this (s, null, null);
```

```
   }

   @Override
   public String toString() {
      return id;
   }
}


class Graph {

   private String id;
   private Vertex first;
   private int info = 0;

   Graph (String s, Vertex v) {
      id = s;
      first = v;
   }

   Graph (String s) {
      this (s, null);
   }

   @Override
   public String toString() {
      String nl = System.getProperty ("line.separator");
      StringBuffer sb = new StringBuffer (nl);
      sb.append (id);
      sb.append (nl);
      Vertex v = first;
      while (v != null) {
         sb.append (v.toString());
         sb.append (" -->");
         Arc a = v.first;
         while (a != null) {
            sb.append (" ");
            sb.append (a.toString());
            sb.append (" (");
            sb.append (v.toString());
            sb.append ("->");
            sb.append (a.target.toString());
            sb.append (")");
            a = a.next;
         }
         sb.append (nl);
         v = v.next;
      }
      return sb.toString();
   }

   public Vertex createVertex (String vid) {
      Vertex res = new Vertex (vid);
      res.next = first;
      first = res;
      return res;
   }

   public Arc createArc (String aid, Vertex from, Vertex to) {
      Arc res = new Arc (aid);
      res.next = from.first;
      from.first = res;
      res.target = to;
      return res;
   }
```

```
/**
 * Create a connected undirected random tree with n vertices.
 * Each new vertex is connected to some random existing vertex.
 * @param n number of vertices added to this graph
 */
public void createRandomTree (int n) {
   if (n <= 0)
      return;
   Vertex[] varray = new Vertex [n];
   for (int i = 0; i < n; i++) {
      varray [i] = createVertex ("v" + String.valueOf(n-i));
      if (i > 0) {
         int vnr = (int)(Math.random()*i);
         createArc ("a" + varray [vnr].toString() + "_"
            + varray [i].toString(), varray [vnr], varray [i]);
         createArc ("a" + varray [i].toString() + "_"
            + varray [vnr].toString(), varray [i], varray [vnr]);
      } else {}
   }
}

/**
 * Create an adjacency matrix of this graph.
 * Side effect: corrupts info fields in the graph
 * @return adjacency matrix
 */
public int[][] createAdjMatrix() {
   info = 0;
   Vertex v = first;
   while (v != null) {
      v.info = info++;
      v = v.next;
   }
   int[][] res = new int [info][info];
   v = first;
   while (v != null) {
      int i = v.info;
      Arc a = v.first;
      while (a != null) {
         int j = a.target.info;
         res [i][j]++;
         a = a.next;
      }
      v = v.next;
   }
   return res;
}

/**
 * Create a connected simple (undirected, no loops, no multiple
 * arcs) random graph with n vertices and m edges.
 * @param n number of vertices
 * @param m number of edges
 */
public void createRandomSimpleGraph (int n, int m) {
   if (n <= 0)
      return;
   if (n > 2500)
      throw new IllegalArgumentException ("Too many vertices: " + n);
   if (m < n-1 || m > n*(n-1)/2)
      throw new IllegalArgumentException
         ("Impossible number of edges: " + m);
   first = null;
   createRandomTree (n);       // n-1 edges created here
   Vertex[] vert = new Vertex [n];
   Vertex v = first;
```

```
      int c = 0;
      while (v != null) {
         vert[c++] = v;
         v = v.next;
      }
      int[][] connected = createAdjMatrix();
      int edgeCount = m - n + 1;  // remaining edges
      while (edgeCount > 0) {
         int i = (int)(Math.random()*n);  // random source
         int j = (int)(Math.random()*n);  // random target
         if (i==j)
            continue;  // no loops
         if (connected [i][j] != 0 || connected [j][i] != 0)
            continue;  // no multiple edges
         Vertex vi = vert [i];
         Vertex vj = vert [j];
         createArc ("a" + vi.toString() + "_" + vj.toString(), vi, vj);
         connected [i][j] = 1;
         createArc ("a" + vj.toString() + "_" + vi.toString(), vj, vi);
         connected [j][i] = 1;
         edgeCount--;  // a new edge happily created
      }
   }

   /**
    * Create a random directed graph with n vertices and m edges.
    * @param n number of vertices
    * @param m number of edges
    */
   private void createRandomDirectedGraph(int n, int m) {
      Vertex[] varray = new Vertex[n];

      for (int i = 0; i < n; i++) {
         Vertex v = createVertex("v" + i);
         varray[i] = v;
      }

      int[][] adjMatrix = createAdjMatrix();

      while (m > 0) {
         int i = (int)(Math.random() * n);
         int j = (int)(Math.random() * n);
         if (i == j) continue;
         if (adjMatrix[i][j] != 0 || adjMatrix[j][i] != 0) continue;
         Vertex vi = varray[i];
         Vertex vj = varray[j];
         createArc ("a" + vi + "_" + vj, vi, vj);
         m--;
      }
   }

   private int getTotalVertices() {
      return createAdjMatrix().length;
   }

   /**
    * Transpose adjacency matrix of this graph.
    * @return transposed adjacency matrix
    */
   public int[][] createTransposedAdjMatrix() {
      int[][] matrix = createAdjMatrix();
      int[][] transposedMatrix = new int[matrix.length][matrix.length];
      for (int i = 0; i < matrix.length; i++) {
         for (int j = 0; j < matrix.length; j++) {
            transposedMatrix[i][j] = matrix[j][i];
         }
```

```
        }

        return transposedMatrix;
    }

    /**
     * Create transposed graph of this graph.
     * @return transposed graph
     */
    private Graph transposeGraph() {
        Graph transposedGraph = new Graph(id + " Transposed");
        Vertex[] vertices = new Vertex[getTotalVertices()];

        Vertex v = first;
        int c = 0;

        while (v != null) {
            vertices[c++] = transposedGraph.createVertex(v.id);
            v = v.next;
        }

        int[][] adjMatrix = createTransposedAdjMatrix();

        for (int i = 0; i < adjMatrix.length; i++) {
            for (int j = 0; j < adjMatrix.length; j++) {
                if (adjMatrix[i][j] == 1) {
                    Vertex vi = vertices[i];
                    Vertex vj = vertices[j];
                    transposedGraph.createArc("a" + vi + "_" + vj, vi, vj);
                }
            }
        }

        return transposedGraph;
    }

    /**
     * Create Depth-first search (DFS) algorithm.
     * @param v Vertex
     * @return list of visited vertices.
     */
    private ArrayList<Vertex> depthFirstSearch(Vertex v) {
        ArrayList<Vertex> visited = new ArrayList<>();
        Stack<Vertex> stack = new Stack<>();

        stack.push(v);

        while (!stack.isEmpty()) {
            v = stack.pop();

            if (!visited.contains(v)) {
                visited.add(v);
                Arc a = v.first;

                while (a != null) {
                    stack.push(a.target);
                    a = a.next;
                }
            }
        }

        return visited;
    }

    /**
     * Create method that checks whether graph is strongly connected or not.
```

```
         * @return boolean true or false
         */
        public boolean isStronglyConnected() {
           int totalVertices = getTotalVertices();
           Vertex v = first;

           while (v != null) {
              if (depthFirstSearch(v).size() != totalVertices) return false;
              v = v.next;
           }

           Graph transposedGraph = transposeGraph();
           v = transposedGraph.first;

           while (v != null) {
              if (depthFirstSearch(v).size() != totalVertices) return false;
              v = v.next;
           }

           return true;
        }
     }
}
```

6.2 Lisa 2 (lahendusnäited)

Graaf G0 (tugevalt sidus)

joonis 1.0

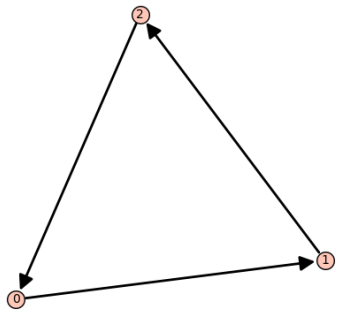
Programmi väljund:

G0

v0 -->

Graph G0 is strongly connected: true

Graaf G1 (tugevalt sidus)



joonis 1.1

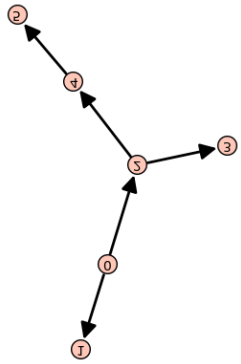
Programmi väljund:

G1

v2 --> av2_v0 (v2->v0)
v1 --> av1_v2 (v1->v2)
v0 --> av0_v1 (v0->v1)

Graph G1 is strongly connected: true

Graaf G2 (ei ole tugevalt sidus)



joonis 1.2

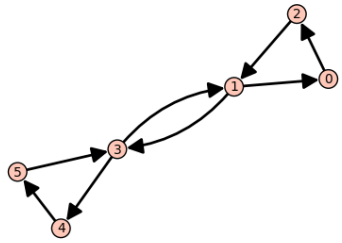
Programmi väljund:

G2

v5 -->
v4 --> av4_v5 (v4->v5)
v3 -->
v2 --> av2_v4 (v2->v4) av2_v3 (v2->v3)
v1 -->
v0 --> av0_v2 (v0->v2) av0_v1 (v0->v1)

Graph G2 is strongly connected: false

Graaf G3 (tugevalt sidus)



joonis 1.3

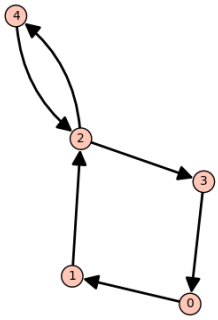
Programmi väljund:

G3

v5 --> av5_v3 (v5->v3)
v4 --> av4_v5 (v4->v5)
v3 --> av3_v4 (v3->v4) av3_v1 (v3->v1)
v2 --> av2_v1 (v2->v1)
v1 --> av1_v3 (v1->v3) av1_v0 (v1->v0)
v0 --> av0_v2 (v0->v2)

Graph G3 is strongly connected: true

Graaf G4 (tugevalt sidus)



joonis 1.4

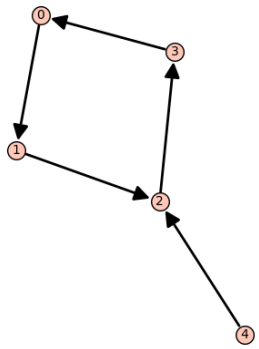
Programmi väljund:

G4

v4 --> av4_v2 (v4->v2)
v3 --> av3_v0 (v3->v0)
v2 --> av2_v4 (v2->v4) av2_v3 (v2->v3)
v1 --> av1_v2 (v1->v2)
v0 --> av0_v1 (v0->v1)

Graph G4 is strongly connected: true

Graaf G5 (ei ole tugevalt sidus)



joonis 1.5

Programmi väljund:

G5

v4 --> av4_v2 (v4->v2)
v3 --> av3_v0 (v3->v0)
v2 --> av2_v3 (v2->v3)
v1 --> av1_v2 (v1->v2)
v0 --> av0_v1 (v0->v1)

Graph G5 is strongly connected: false

Juhuslik orienteeritud graaf G6. Mõõdetakse meetodi .isStronglyConnected() kiirust.

Programmi väljund:

Graph G6 is with 2000 vertices and 2500 edges.

Execution time of graph G6: 10ms

Juhuslik orienteerimata graaf G7. Mõõdetakse meetodi .isStronglyConnected() kiirust.

Programmi väljund:

Graph G7 is with 2000 vertices and 2500 edges.

Execution time of graph G7: 7959ms
