TALLINNA TEHNIKAÜLIKOOL

Individuaaltöö aines

"Algoritmid ja andmestruktuurid"

Liivia Vanaisak

IADB31

Juhendaja: Jaanus Pöial

Tallinn 2020

Autori deklaratsioon

Kinnitan, et käesolev töö on minu iseseisva töö tulemus ning antud tööga ei ole varem individuaaltöö arvestust aines “Algoritmid ja Andmestruktuurid” taodeldud.

SISUKORD

ÜLESANDE PÜSTITUS

MÕISTED

Graaf

Graaf on abstraktne andmetüüp, millega on võimalik esitada objektipaaride vahelisi suhteid. [1] Lühidalt võib graafi kirjeldada kui hulka *G*, mis koosneb tippude hulgast *V* ja servade hulgast *E* (*G*=(*V*, *E*)). Kusjuures servade hulk *E* on tippu hulga *V* alamhulk ($E \in V \times V$). [2]

Orienteerimata graaf

Graafi servad võivad graafidel olla suunatud või suunamata.

Öeldakse, et serv (*u*, *v*) on suunatud *u*-st *v*-ni, kui paar (*u*, *v*) on järjestatud nii, et *u*-le järgneb *v*.

Suunatud serva nimetatakse kaareks. [1]

Öeldakse, et serv (*u*, *v*) on suunamata, kui paar (*u*, *v*) ei ole järjestatud.[1]

Kui graafi *G* kõik servad on suunamata, siis on tegemist orienteerimata graafiga.

Orienteerimata sidus graaf

Graaf on sidus, kui iga kahe tipu vahel on ühendus.[1]

Teisiti sõnastades, graaf on sidus, kui igast graafi tipust leidub tee teise graafi tippu.

Seega on orienteerimata sidus graaf graaf, kus igast graafi tipust leidub tee teise graafi tippu ning kõikideks tippude vahelisteks teedeks on suunamata servad.

Puustruktuuriga graaf

Puu on abstraktne andmetüüp, mis hoiab endas elemente hierarhiliselt. [1]

Matemaatiliselt võib puud kirjeldada kui lõpliku tippude hulka *T*, mis on kas tühi või milles on üks tipp – juur ehk juurtipp – välja eraldatud ning ülejäänud tipud on jaotatud $m \geq 0$ mittelõikuvaks alamhulgaks T, T_2, ..., T_m, millest igaüks on omakorda puu. [2]

Graaf on puustruktuuriga, kui ta on sidus ning graafi tippude vahel puuduvad tsüklid (kinnised ahelad, silmused). [3]

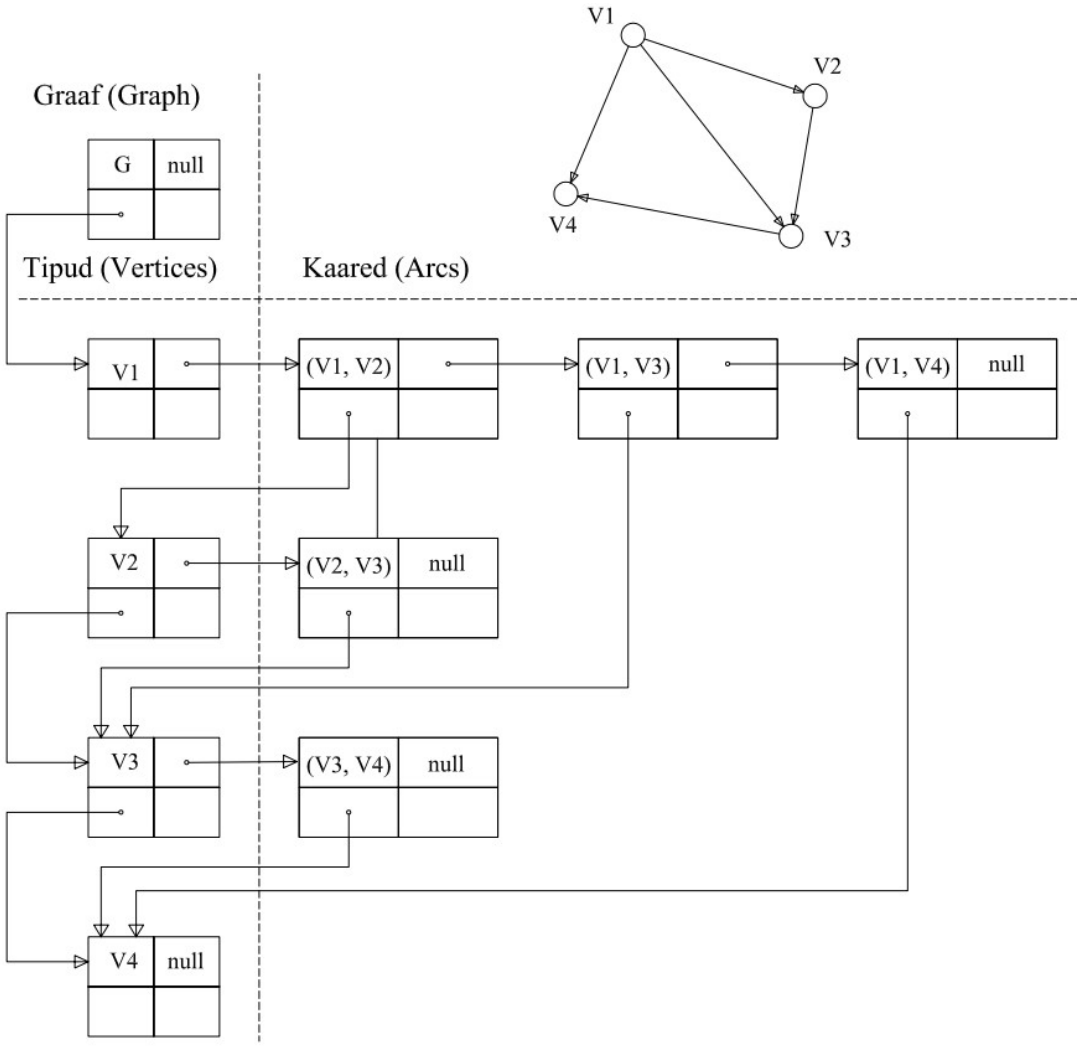
ÜLESANNE

Koostada meetod, mis kontrollib, kas etteantud orienteerimata sidus graaf on puustruktuuriga. Antud ülesande püstitus on pärit raamatust "Algoritmid ja andmestruktuurid. Ülesannete kogu".

LAHENDUSE KIRJELDUS

GRAAFI KUJUTAMINE PROGRAMMIS

Graafi kujutamiseks kasutatakse antud ülesande raames külgnevusstruktuuri.
Külgnevusstruktuuri skemaatiline seletus on toodud joonisel 1.



Joonis 1. Graafi kujutamine külgnevusstruktuur abil.

Graafi kujutamiseks kasutatakse kolme klassi:

- *Graph*
- *Vertex*
- *Arc*

Graph klass sisaldab endas:

- *graphName* – graafi nime
- f*irstVertex* – viita esimesele tipule

Vertex klass sisaldab endas:

- *vertexName* – tipu nime
- *nextVertex* – viita järgmisele tipule
- *firstArc* – viita esimesele väljuvale kaarele

Arc klass sisaldab endas:

- *arcName* – kaare nime
- *targetVertex*– suubuva tipu viita
- *nextArc* – lähtetipu järgmise kaare viita

Kuna ülesande raames tegeletakse orienteerimata graafiga, kus tippe ühendavad omavahel served, siis kujutatakse graafi serva *E* kahe kaare kaudu. Las olla *V1* ja *V2* graafi *G* kuuluvad tipud. Seega graafi serv *E* koosneb kahes kaares (*V1*, *V2*) ja (*V2*, *V1*).

Lähtudes asjaolust, et graaf on loodud eelnevalt kirjeldatud kujutamisviisi kasutades, loome meetodi, mis kontrollib kas loodud orienteerimata graaf on puustruktuuriga.

MEETODI VÄLJATÖÖTAMISEL KASUTATUD TEOORIA

Meetodi väljatöötamisel on lähtutud kahest teoreemist:

1. *n* tipuga puul on *n-1*serva
2. iga *n* tipu ja *n-1* servaga sidus graaf on puustruktuuriga

Teoreem 1: *n* tipuga puul on *n-1* serva

Tõestus: Tähistagu *n* puu tippude arvu

Kui *n=1*, siis on puu servade arv *0.*

Kui *n=2*, sii on puu servade arv *1.*

Kui *n=3*, siis on puu servade arv *2.*

Seega on väide tõene *n=1,2,3* korral.

Olgu väide tõene *n=m* korral. Tõestame, et väide on tõene ka *n=m+1* korral.

Olgu *E* tippe *V1* ja *V2* ühendav serv. Kuna *T* on puu, siis eksisteerib tippude *V1* ja *V2* vahel vaid üks ühendus. Seega, kui kustutame serva *E*, lahutatakse puu kaheks komponendiks *T1* ja *T2*.

Nendel uutel puudele *T1* ja *T2* on vähem kui *m+1* tippu ning puudub ühendus, seega on puude tippudeks vastavalt m_1 ja m_2.

Servi on kokku $E=(m_1-1) + (m_2-1) + 1=(m_1+m_2)-1=m+1-1 = m.$ [4]

Seega *n=m+1* tipu korral on puus *T* kokku m serva. Matemaatilise induktsiooni järgi on *n* tipulisel puul täpselt *n-1* serva.

Teoreem 2: Iga *n* tipu ja *n-1* servaga sidus graaf on puustruktuuriga

Tõestus: On teada, et minimaalseks servade arvuks, mis on vajalik *n* tipust koosneva sidusa graafi loomiseks on *n-1* serva. Kui eemaldame graafilt ühe serva muutub graaf mittesidusaks.

Seega ei saa *n* tipu ja *n-1* servaga graafil olla tsükleid (kinniseid ahelaid). Eelnevat arvestades on tõestatud, et *n* tipu ja *n-1* servaga sidus graaf on puustruktuuriga.[4]

MEETODI KIRJELDUS

Meetod kuulub *graph* klassi meetotite hulka ning tagastab tõeväärtuse.

Meetodi alguses kontrollitakse tingimust kas graafil on olemas esimene tipp. Kui graafil puudub esimene tipp, visatakse erind tekstiga *„Null graph“.*

Pärast esmaste tingimuste kontrollimist luuakse kaks *List* tüüpi tühja loendit, milles ühte salvestatakse kõik graafi tipud ja teise kõik graafi tippupaarid ehk servad. Tipupaarid salvestatakse eraldi *HashSet* tüüpi loendisse, et hiljem saaks võrrelda kahest tipust koosnevaid loendeid ilma, et peaks arvestama tippude järjekorda tipupaarides.

Meetodis kasutatakse kahte *while* tsükklit.

Esimene *while* tsükkel käib läbi kõik graafi tipud ning teine *while* tsükkel, mis on esimese tsükkli sees, käib läbi kõik tipuga ühendatud kaared.

Enne *while* tsüklit luuakse uus *Vertex* objekt, mille väärtuseks määratakse *graph* objekti atribuut *firstVertex*.

Luuakse *while* tsükkel, mis käib nii kaua, kuni loodud *Vertex* objekt ei võrdu *null*-iga. Tipu läbimisel lisatakse antud tipp tippude loendisse. Seejärel luuakse uus *Arc* objekt, mille väärtuseks määratake *Vertex* objekti atribuut *firstArc.*

Luuakse uus *while* tsükkel, mis käib nii kaua, kuni loodud *Arc* objekt ei võrdu *null*-iga.

Tsükli sees tehakse kaks kontrolltingimust:

a) kas kaare tipud on võrdsed, kui jah, siis on tegu tsükliga(silmusega) ning tagastatakse tõeväärtus *false*.
b) kas tipupaar on juba tipupaaride loetelus ning olemasolu puudumisel lisatakse see tipupaar. *Arc* objekti väärtuseks määratakse *Arc* objekti atribuut *nextArc.*

Vertex objekti väärtuseks määratakse *Vertex* objekti atribuut *nextVertex.*

While tsüklite lõppedes kontrollitakse, kas tipude loendi suurus on ühe võrra suurem tipupaaride loendi suurusest ning tagastatakse vastav tõeväärtus.

```java
/**
 * Method that checks if given graph has tree structure
 * Throws exception when graph is null.
 * @return boolean
 */

public boolean isTree() {

    if(firstVertex==null){
        throw new RuntimeException("Null graph");
    }

    List<Vertex> visitedVertices = new ArrayList<Vertex>();
    List<HashSet<Vertex>> visitedEdges = new ArrayList<HashSet<Vertex>>();
    Vertex v = firstVertex;
    while (v !=null) {
        visitedVertices.add(v);
        Arc a = v.firstArc;
        while (a != null) {
            if(v.equals(a.targetVertex)){
                return false;
            }
            HashSet<Vertex> vertexPair = new HashSet<Vertex>();
            vertexPair.add(v);
            vertexPair.add(a.targetVertex);
            if(!visitedEdges.contains(vertexPair)){
                visitedEdges.add(vertexPair);
            }
            a = a.nextArc;
        }
        v = v.nextVertex;
    }
    return visitedVertices.size() - 1 == visitedEdges.size();
}
```

Joonis 2. Meetodi *isTree()* kuvatõmmis.

PROGRAMMI KASUTUSJUHEND

Programmi kasutaja saab kontrollida kas graaf on puustruktuuriga, kui ta pöördub meetodi poole *isTree()*. Antud meetod on graafi klassi kuuluv, seega meetodi välja kutsumine toimub alljärgnevalt.

```
g.isTree();
```

Kus *g* on klassi *Graph* instants.

Antud meetod tagastab tõeväärtuse (*true, false*).

Kui tegemist on tühja graafiga, siis visatakse meetodi poolt erind tekstiga *„Null Graph“*.
Kui antud graaf on puustruktuuriga tagastab meetod väärtuse *true*.

TESTIMISKAVA

TESTID

Meetodi *isTree()* testimiseks on programmis abimeetod *createRandomSimpleGraph()*, mis genereerib juhusliku sidusa graafi, mis ei pea olema tingimata puustruktuuriga.

Meetodit *isTree()* testitakse 9 olukorra puhul.

testIsTreeSmall()

Testitakse, kas meetodi tagastatavaks tõeväärtuseks on *true*, kui sisendiks on väiksemahuline (*n=10*) puustruktuuriga graaf.

testIsNotTreeSmall()

Testitakse, kas meetodi tagastatavaks tõeväärtuseks on *false*, kui sisendiks on väiksemahuline (*n=10, m=11*) graaf, mis ei ole puustruktuuriga.

testIsTreeBig()

Testitakse, kas meetodi tagastatavaks tõeväärtuseks on *true*, kui sisendiks on suuremahuline (*n=2000*) puustruktuuriga graaf.

testIsNotTreeBig()

Testitakse, kas meetodi tagastatavaks tõeväärtuseks on *false*, kui sisendiks on suuremahuline (*n=2000, m=2210*) graaf, mis ei ole puustruktuuriga.

testDisconnectedGraph()

Testitakse, kas meetodi tagastatavaks tõeväärtuseks on *false*, kui sisendiks on mittesidus graaf kahe tipuga.

testGraphWithLoop()

Testitakse, kas meetodi tagastatavaks tõeväärtuseks on *false*, kui sisendiks on graaf, millel on kaks tippu ja üks kaar, kus kaar on silmus.

testWithOneVertexAndLoop()

Testitakse, kas meetodi tagastatavaks tõeväärtuseks on *false*, kui sisendiks on graaf, millel on üks tipp ja üks kaar, kus kaar on silmus.

testOneVertex()

Testitakse, kas meetodi tagastatavaks tõeväärtuseks on *true*, kui sisendiks graaf, mis koosneb ühest tipust.

testNullGraph()

Testitakse, kas meetod viskab erindi, kui sisendiks on tühi graaf.

KUVATÕMMISED MEETODI KASUTAMISEST JA TULEMUSTEST

```
Graph g = new Graph( s: "G");
g.createRandomSimpleGraph( n: 5,  m: 5);
System.out.println(g.toString());
System.out.println("Graph 'G' is tree: " + g.isTree());

Graph t = new Graph( s: "T");
t.createRandomSimpleGraph( n: 5,  m: 4);
System.out.println(t.toString());
System.out.println("Graph 'T' is tree: " + t.isTree());
```

Joonis 3. *isTree()* meetodi kasutamine.

```
G
v1 --  av1_v4 (v1-v4) av1_v2 (v1-v2)
v2 --  av2_v1 (v2-v1) av2_v5 (v2-v5)
v3 --  av3_v5 (v3-v5)
v4 --  av4_v1 (v4-v1) av4_v5 (v4-v5)
v5 --  av5_v2 (v5-v2) av5_v3 (v5-v3) av5_v4 (v5-v4)

Graph 'G' is tree: false

T
v1 --  av1_v5 (v1-v5)
v2 --  av2_v4 (v2-v4)
v3 --  av3_v5 (v3-v5)
v4 --  av4_v2 (v4-v2) av4_v5 (v4-v5)
v5 --  av5_v1 (v5-v1) av5_v3 (v5-v3) av5_v4 (v5-v4)

Graph 'T' is tree: true
```

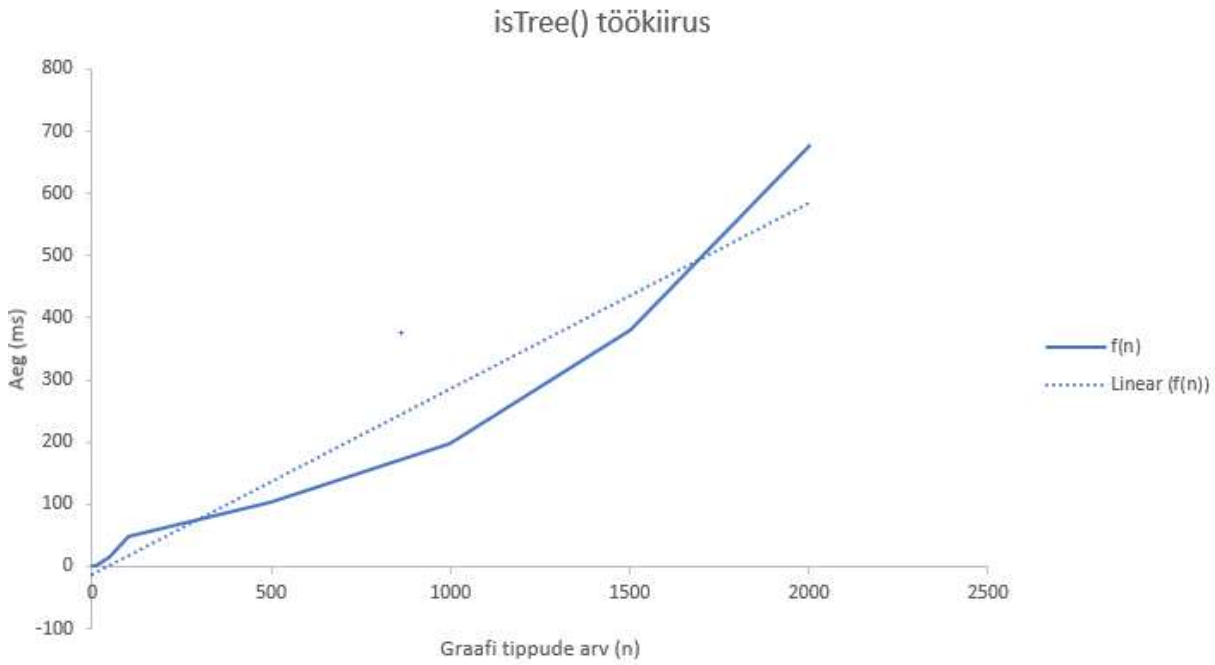
Joonis 4. Tulemused kahe erineva graafi kontrollimisel.

MEETODI TÖÖKIIRUS

Meetod *isTree()* on lineaarse keerukusega *O(n)*, läbides kõik graafi tipud ja tippudest väljuvad kaared ühe korra.

Meetodi *isTree()* töökiirust mõõdeti üheksa graafi korral, kus tippude arvuks oli vastavalt: 1, 5, 10, 50 ,100, 500, 1000, 1500, 2000.

Tulemused on välja toodud Joonisel 5.



Joonis 5. Meetodi *isTree()* töökiiruse muutumine andmemahtude suurenemisel.

KASUTATUD KIRJANDUS

1. Data Structures and Algorithms in Java / Michael T. Goodrich. 4.trükk

2. Algoritmid ja andmestruktuurid. (2003). / Jüri Kiho. 3. trükk. Tartu: Tartu Ülikooli Kirjastus

3. Võistlusprogrameerimine 2.osa. (2018). / T. Tennisberg, K. Gabrel. Tartu: Tartu Ülikooli Kirjastus

4. Discrete Mathematics: An Open Introduction. (2019). / O. Levin. 3. trükk. (http://discrete.openmathbooks.org/dmoi3/dmoi.html)

LISAD

PROGRAMMI KOOD

```
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() {

        List<Integer> nodes =
Arrays.asList(1,5,10,50,100,500,1000,1500,2000);

        List<Graph> graphs = new ArrayList<Graph>();

        for (Integer node : nodes) {
            Graph g = new Graph("G");
            g.createRandomSimpleGraph(node, node - 1);
            graphs.add(g);
        }

        for (int i = 0; i < graphs.size(); i++) {
            long stime, ftime, diff;
            stime = System.nanoTime();
            graphs.get(i).isTree();
            ftime = System.nanoTime();
            diff = ftime - stime;
            System.out.printf("%34s%11d%n", "Graph with " + nodes.get(i) + "
node(s) - time (ms): ", diff / 1000000);
        }

        Graph g = new Graph("G");
        g.createRandomSimpleGraph(5, 5);
        System.out.println(g.toString());
        System.out.println("Graph 'G' is tree: " + g.isTree());

        Graph t = new Graph("T");
        t.createRandomSimpleGraph(5, 4);
        System.out.println(t.toString());
        System.out.println("Graph 'T' is tree: " + t.isTree());

    }

    /** Vertex represents one vertex in the graph.
```

```
     *  Vertex has attributes:
     *      * vertexName, that holds vertex name.
     *      * nextVertex, that holds next vertex.
     *      * firstArc, that holds vertex first arc.
     */
    static class Vertex {

        private String vertexName;
        private Vertex nextVertex;
        private Arc firstArc;
        private int info = 0;


        Vertex (String s, Vertex v, Arc e) {
            vertexName = s;
            nextVertex = v;
            firstArc = e;
        }

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

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

    }

    /** Arc represents one arrow in the graph. Two-directional edges are
     * represented by two Arc objects (for both directions).
     * Arc has attributes:
     *      * arcName, that holds arc name.
     *      * targetVertex, that holds target vertex.
     *      * nextArc, that holds next arc.
     */
    static class Arc {

        private String arcName;
        private Vertex targetVertex;
        private Arc nextArc;

        Arc (String s, Vertex v, Arc a) {
            arcName = s;
            targetVertex = v;
            nextArc = a;
        }

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

        @Override
        public String toString() {
            return arcName;
```

```
        }

    }

    /** Graph represents one graph that has vertices and arcs.
     * Graph has attributes:
     *      * graphName, that holds graph name.
     *      * firstVertex, that holds graphs first vertex.
     */

    static class Graph {

        private String graphName;
        private Vertex firstVertex;
        private int info = 0;

        Graph (String s, Vertex v) {
            graphName = s;
            firstVertex = v;
        }

        Graph (String s) {
            this (s, null);
        }
        public Vertex getFirstVertex(){
            return firstVertex;
        }

        @Override
        public String toString() {
            String nl = System.getProperty ("line.separator");
            StringBuilder sb = new StringBuilder(nl);
            sb.append (graphName);
            sb.append (nl);
            Vertex v = firstVertex;
            while (v != null) {
                sb.append (v.toString());
                sb.append (" -- ");
                Arc a = v.firstArc;
                while (a !=null) {
                    sb.append (" ");
                    sb.append (a.toString());
                    sb.append (" (");
                    sb.append (v.toString());
                    sb.append ("-");
                    sb.append (a.targetVertex.toString());
                    sb.append (")");
                    a = a.nextArc;
                }
                sb.append (nl);
                v = v.nextVertex;
            }
            return sb.toString();
        }

        /**
```

```
         * Creates new vertex.
         * Assigns current graphs vertex to newly created vertex attribute
nextVertex.
         * Assigns newly created vertex to graphs first vertex.
         * @param vid vertex name
         * @return res created vertex
         */
        public Vertex createVertex (String vid) {
            Vertex res = new Vertex (vid);
            res.nextVertex = firstVertex;
            firstVertex = res;
            return res;
        }

        /**
         * Creates new arc
         * Assigns "from" vertex's first arc to newly created arcs attribute
nextArc
         * Assigns created arc to "from" vertex first arc
         * Assigns "to" vertex to created arcs attribute targetVertex
         * @param aid arc name
         * @param from vertex where arc started
         * @param to vertex where arc is going
         * @return res created arc
         */
        public Arc createArc (String aid, Vertex from, Vertex to) {
            Arc res = new Arc (aid);
            res.nextArc = from.firstArc;
            from.firstArc = res;
            res.targetVertex = 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 = firstVertex;
    while (v != null) {
        v.info = info++;
        v = v.nextVertex;
    }
    int[][] res = new int [info][info];
    v = firstVertex;
    while (v != null) {
        int i = v.info;
        Arc a = v.firstArc;
        while (a != null) {
            int j = a.targetVertex.info;
            res [i][j]++;
            a = a.nextArc;
        }
        v = v.nextVertex;
    }
    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);
    firstVertex = null;
    createRandomTree (n);       // n-1 edges created here
    Vertex[] vert = new Vertex [n];
    Vertex v = firstVertex;
    int c = 0;
    while (v != null) {
        vert[c++] = v;
        v = v.nextVertex;
    }
    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
            }
        }

        /**
         * Method that checks if given graph has tree structure
         * Throws exception when graph is null.
         * @return boolean
         */

        public boolean isTree() {

            if(firstVertex==null){
                throw new RuntimeException("Null graph");
            }

            List<Vertex> visitedVertices = new ArrayList<Vertex>();
            List<HashSet<Vertex>> visitedEdges = new
ArrayList<HashSet<Vertex>>();
            Vertex v = firstVertex;
            while (v !=null) {
                visitedVertices.add(v);
                Arc a = v.firstArc;
                while (a != null) {
                    if(v.equals(a.targetVertex)){
                        return false;
                    }
                    HashSet<Vertex> vertexPair = new HashSet<Vertex>();
                    vertexPair.add(v);
                    vertexPair.add(a.targetVertex);
                    if(!visitedEdges.contains(vertexPair)){
                        visitedEdges.add(vertexPair);
                    }
                    a = a.nextArc;
                }
                v = v.nextVertex;
            }
            return visitedVertices.size() - 1 == visitedEdges.size();
        }

    }

}
```

TESTIDE KOOD

```
import static org.junit.Assert.*;
import org.junit.Test;

/** Tests.
 * @author Liivia Vanaisak
 */
public class GraphTaskTest {

   @Test (timeout=20000)
   public void test1() {
      GraphTask.main (null);
      assertTrue ("There are no tests", true);
   }

   @Test (timeout=1000)
   public void testIsTreeSmall(){
     GraphTask.Graph g = new GraphTask.Graph("G");
     g.createRandomSimpleGraph(10, 9);
     assertTrue(g.isTree());
   }

   @Test (timeout=1000)
   public void testIsNotTreeSmall(){
     GraphTask.Graph g = new GraphTask.Graph("G");
     g.createRandomSimpleGraph(10, 11);
     assertFalse(g.isTree());
   }

    @Test (timeout=2000)
    public void testIsTreeBig(){
        GraphTask.Graph g = new GraphTask.Graph("G");
        g.createRandomSimpleGraph(2000, 1999);
        assertTrue(g.isTree());
    }

    @Test (timeout=2000)
    public void testIsNotTreeBig(){
        GraphTask.Graph g = new GraphTask.Graph("G");
        g.createRandomSimpleGraph(2000, 2200);
        assertFalse(g.isTree());
    }

    @Test (timeout=2000)
    public void testDisconnectedGraph(){
        GraphTask.Graph g = new GraphTask.Graph("G");
        g.createVertex("v1");
        g.createVertex("v2");
        System.out.println(g.toString());
        assertFalse(g.isTree());
    }

    @Test (timeout=2000)
    public void testGraphWithLoop(){
```

```
        GraphTask.Graph g = new GraphTask.Graph("G");
        g.createVertex("v1");
        g.createVertex("v2");
        g.createArc("av2",g.getFirstVertex(), g.getFirstVertex());
        assertFalse(g.isTree());
    }


    @Test (timeout=2000)
    public void testWithOneVertexAndLoop(){
        GraphTask.Graph g = new GraphTask.Graph("G");
        g.createVertex("v1");
        g.createArc("av1",g.getFirstVertex(), g.getFirstVertex());
        assertFalse(g.isTree());
    }


    @Test (timeout=1000)
    public void testOneVertex(){
        GraphTask.Graph g = new GraphTask.Graph("G");
        g.createVertex("v1");
        assertTrue(g.isTree());
    }
    @Test(expected = RuntimeException.class)
    public void testNullGraph() {
        GraphTask.Graph g = new GraphTask.Graph("G");
        g.isTree();
    }

}
```