PT&SCHE

Introduction of part-time and short cycle studies in Serbia

Funded by the Erasmus+ Programme of the European Union

This project has been funded with support from the European Commission. This publication [communication] reflects the views only of the author, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

UNIVERZITET METROPOLITAN BEOGRAD

KI202 - JAVA 5: STRUKTURE PODATAKA I ALGORITMI – DEO B

Težinski grafovi

Lekcija 06

PRIRUČNIK ZA STUDENTE

KI202 - JAVA 5: STRUKTURE PODATAKA I ALGORITMI – DEO B

Lekcija 06

TEŽINSKI GRAFOVI

- Težinski grafovi
- Poglavlje 1: Osnovi težinskih grafova
- Poglavlje 2: Klasa WeightedGraph
- Poglavlje 3: Minimalno obuhvatno stablo
- Poglavlje 4: Problem najkraćih puteva
- Poglavlje 5: Problem devet novčića sa težinama
- Zaključak

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

UVOD

Ova lekcija treba da ostvari sledeće ciljeve:

U okviru ove lekcije studenti se upoznaju sa:

- Težinskim grafovima i predstavljanjem težinskih grafova
- Određivanjem minimalnog obuhvatnog tabla korišćenjem Primovog i Krusakalovog algoritma
- Problemom najkraćih puteva, i određivanjem najkraćih puteva korišćenjem Flojd-Varšal i Belman-Ford algoritma
- Određivanjem najkraćih puteva korišćenjem Dijkstra algoritma

Problem određivanja minimalnog obuhvatnog stabla ima široku primenu. Uzmimo u obzir kompaniju koja ima filijale u velikom broju gradova. Cilj kompanije je da umreži telefonske centrale svake od filijala. Telefonska kompanija naplaćuje uslugu povezivanja različito za različite parove gradova. Postoji mnogo načina da se gradovi povežu granama međusobno. Najjeftiniji način je da se formira obuhvatno stablo grafa sa namanjim ukupnim troškovima.

Još jedan karakterističan problem je mreža puteva. Mreža puteva može da se posmatra kao graf sa pozitivnim težinama. Čvorovi predstavljaju raskrsnice a svaka grana grafa predstavlja ulicu između dve raskrsnice. Težina grane može da predstavlja dužinu ulice, vreme potrebno da se pređe ta ulica ili cena prelaska te ulice. Koristeći usmerene grafove možemo predstaviti i jednosmerne ulice. Ovakvi grafovi su posebni zato što su neke grane bitnije od drugih za dugačka putovanja (npr. autoputevi).

Poglavlje 1

Osnovi težinskih grafova

UVODNA RAZMATRANJA

Težinski graf (engl. weighted graph) je onaj u kome svaka grana ima težinu putovanja između čvorova. Težina može predstavljati rastojanje, potrošnju energije, troškove, itd...

U prethodnoj lekciji o bestežinskim grafovima smo u okviru sekcije *Uvod u grafove*, na Slici-3, imali primer grafa koji predstavlja letove između gradova.

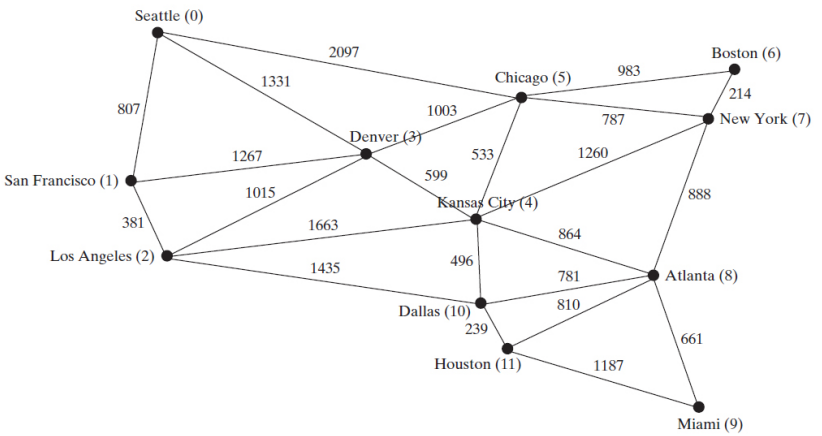
Primenom *BFS* algoritma mogli smo pronaći najmanji broj među-letova između dva grada.

Pretpostavimo da grane predstavljaju rastojanja između gradova, Slika-1. Cilj je odgovoriti na sledeća pitanja:

- Kako pronaći najmanje ukupno rastojanje tako da se povežu svi gradovi?
- Kako pronaći najkraće rastojanje između dva grada?

Prvi problem je poznat kao minimalno obuhvatno stablo (minimum spanning tree - MST)

Drugi problem je poznat kao Problem najkraćih puteva (eng. shortest path problem).

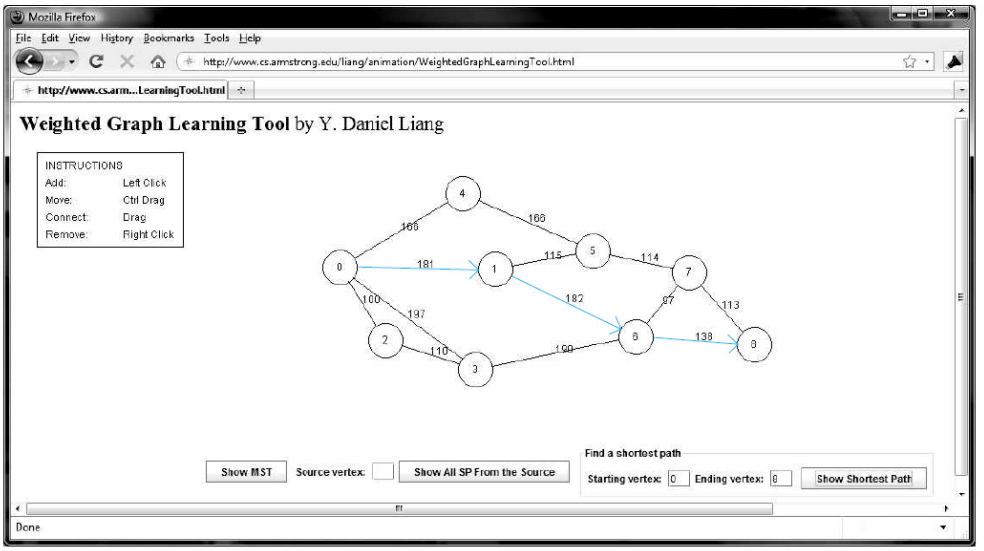


Slika 1.1 Graf koji uključuje i rastojanja između gradova.

ANIMACIJA RADA SA TEZINSKIM GRAFOVIMA

Sticanje boljeg uvida u rad težinskih grafova uz jednu animaciju

Pre nego što prikažemo algoritme u vezi težinskih grafova i primere primene, korisno je da pogledate interaktivni grafički interfejs i da se upoznate sa principima rada težinskih grafova, Interaktivni GUI se nalazi na adresi http://www.cs.armstrong.edu/liang/animation/web/WeightedGraphLearningTool.html, i izgleda kao na Slici-2. Ovaj alat vam omogućava da unesete temena, zadate grane i težine grana, prikažete graf, odredite minimalno obuhvatno stablo (MST) ili pronađete najkraći put između dva čvora grafa, kao što je prikazano na Slici-2.



Slika 1.2 Možete da koristite alat i da upotrebom miša kreirate težinski graf, i prikažete MST i najkraće putanje.

PREDSTAVLJANJE TEŽINSKOG GRAFA

Težinski grafovi mogu biti predstavljeni na sličan način kao bestežinski grafovi, osim što treba imati i podatak o težini grane

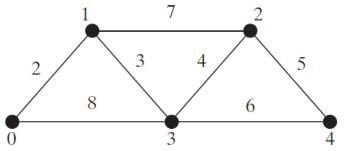
Postoje dva tipa težinskih grafova:

- grafovi sa težinskim čvorovima i
- grafovi sa težinskim granama.

U grafu sa težinskim čvorovima (eng. vertex weighted) je svakom čvoru dodeljena težina, dok je kod grafova sa težinskim granama (eng. edge-weighted) svakoj grani dodeljena određena težina. Od prethodno pomenutih, češće se koriste grafovi sa težinskim granama.

Težinski grafovi mogu biti predstavljeni na sličan način kao bestežinski grafovi, osim što treba imati i podatak o težini grane. Kao i kod bestežinskog grafa, čvorovi mogu biti smešteni u običnom nizu.

Predstavljanje težinskih grana: Niz grana



(a)

```
                                   teme  težina
int[][] edges = {{0, 1, 2}, {0, 3, 8},
  {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
  {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
  {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
  {4, 2, 5}, {4, 3, 6}
};
```

(b)

Slika 1.3 Svakoj grani u težinskom grafu se dodeljuje određena težina

Težinske grane mogu biti predstavljene korišćenjem 2D niza, Na primer, moguće je u grafu sa Slike-3a koristiti niz sa Slike-3b za predstavljanje grana.

Matrica susedstva sa težinskim koeficijentima

Pretpostavimo da graf ima *N* čvorova. Moguće je koristiti 2D matricu pod nazivom *adjacencyMatrix* da se predstave težine svake grane u grafu. Član *adjacencyMatrix[i][j]* predstavlja težinu grane *(i,j)*. Ukoliko čvorovi *i* i *j* nisu povezani, član *adjacencyMatrix[i][j]* je jednak ***null***.

Na primer, težine grana grafa sa Slike-3a mogu biti predstavljenje korišćenjem matrice susedstva prikazane na Slici-4:

```
Integer[][] adjacencyMatrix = {
  {null, 2, null, 8, null},
  {2, null, 7, 3, null},
  {null, 7, null, 4, 5},
  {8, 3, 4, null, 6},
  {null, null, 5, 6, null}
};
```

	0	1	2	3	4
0	null	2	null	8	null
1	2	null	7	3	null
2	null	7	null	4	5
3	8	3	4	null	6
4	null	null	5	6	null

Slika 1.4 Predstavljanje grana korišćenjem matrice susedstva sa težinskim koeficijentima

KLASA WEIGHTEDEDGE

Jedan od načina da se predstave težinske grane je da se definišu grane kao objekti. Za težinske grane, možemo da definišemo klasu WeightedEdge

Drugi način da se predstave težinske grane je da se definišu grane kao objekti. U primeru sa bestežinsim grafom imali smo klasu *AbstractGraph.Edge* u cilju predstavljanja bestežinske grane. Za težinske grane možemo da definišemo klasu *WeightedEdge* na sledeći način:

```
public class WeightedEdge extends AbstractGraph.Edge
    implements Comparable<WeightedEdge> {
  public double weight; // The weight on edge (u, v)

  /** Create a weighted edge on (u, v) */
  public WeightedEdge(int u, int v, double weight) {
    super(u, v);
    this.weight = weight;
```

```
  }

  /** Compare two edges on weights */
  public int compareTo(WeightedEdge edge) {
    if (weight > edge.weight) {
    return 1;
    }
    else if (weight == edge.weight) {
    return 0;
    }
    else {
    return -1;
    }
  }
}
```

Klasa *AbstractGraph.Edge* je unutrašnja klasa definisina u okviru klase *AbstractGraph*. Ona predstavlja granu između čvorova *u* i *v*.

Klasa *WeightedEdge* proširuje *AbstractGraph.Edge* sa novim poljem za težinu *weight*.

Da bi kreirali novi objekat tipa *WeightedEdge*, treba upotrebiti konstrukciju:

new *WeightedEdge(i, j, w)*

gde je *w* težina grane *(i,j)*. Uvek je korisno čuvati grane koje izlaze iz čvora u nekom redu sa prioritetom da bi mogli da pristupate granama u rastućem poretku u odnosu na vrednost težine. Iz ovog razloga klasa *WeightedEdge* implemetira interfejs *Comparable*.

PREDSTAVLJANJE LISTE SUSEDSTVA

Uvek je korisno čuvati grane koje izlaze iz čvora u nekom redu sa prioritetom da bi mogli da pristupate granama u rastućem poretku u odnosu na vrednost težine

Za bestežinski graf je korišćena lista susedstva da se prestave grane. U težinskom grafu se i dalje koristi lista susedstva, ali je lista smeštena u prioritetnom redu. Na primer, lista susedstva za čvorove sa Slike-3a može biti predstavljena na sledeći način:

```
java.util.PriorityQueue<WeightedEdge>[] queues =
    new java.util.PriorityQueue<WeightedEdge>[5];
```

pri čemu su vrednosti prikazane na Slici-5:

queues[0]	WeightedEdge(0, 1, 2)	WeightedEdge(0, 3, 8)		
queues[1]	WeightedEdge(1, 0, 2)	WeightedEdge(1, 3, 3)	WeightedEdge(1, 2, 7)	
queues[2]	WeightedEdge(2, 3, 4)	WeightedEdge(2, 4, 5)	WeightedEdge(2, 1, 7)	
queues[3]	WeightedEdge(3, 1, 3)	WeightedEdge(3, 2, 4)	WeightedEdge(3, 4, 6)	WeightedEdge(3, 0, 8)
queues[4]	WeightedEdge(4, 2, 5)	WeightedEdge(4, 3, 6)		

Slika 1.5 Primer upotrebe prioritetnog reda za listu susedstva grafa.

Član niza *queues[i]* čuva sve grane koje polaze iz čvora *i*.

Radi fleksiblnosti, koristićemo listu baziranu na nizu (array list) radije nego niz fiksne veličine, da bi predstavili listu *queues*.

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. U liniji: ***WeightedEdge edge = new WeightedEdge(1, 2, 3.5),*** šta je **edge.u**, **edge.v**, a šta **edge.weight**?

Zadatak 2. Šta je izlaz sledećeg programa?

```
List<WeightedEdge> list = new ArrayList<WeightedEdge>();
list.add(new WeightedEdge(1, 2, 3.5));
list.add(new WeightedEdge(2, 3, 4.5));
WeightedEdge e = java.util.Collections.max(list);
System.out.println(e.u);
System.out.println(e.v);
System.out.println(e.weight);
```

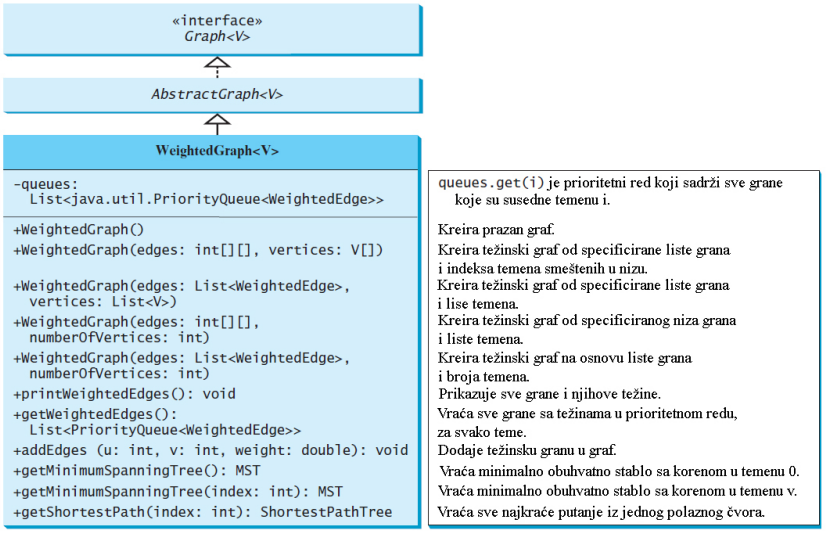
Poglavlje 2

Klasa WeightedGraph

UVODNA RAZMATRANJA

Klasa WeightedGraph proširuje klasu AbstractGraph i uvodi nove metode za dobijanje minimalnog obuhvatnog stabla i za određivanje najkraćeg puta od izvornog čvora

U prethodnoj lekciji o grafovima smo kreirali interfejs *Graph*, klasu *AbstractGraph*, dok smo u materijalima za vežbe predstavili implementaciju klase *UnweightedGraph* za modeliranje bestežinskih grafova. U cilju modeliranja težinskih grafova, neophodno je definisati klasu pod nazivom, npr. *WeightedGraph* , kao podklasu klase *AbstractGraph*, na način nako je to predstavljeno na Slici-1.



Slika 2.1 Klasa WeightedGraph proširuje klasu AbstractGraph.

Klasa *WeightedGraph* jednostavno proširuje klasu *AbstractGraph* sa 5 konstruktora za kreiranje konkretne instance klase *WeightedGraph*.

WeightedGraph nasleđuje sve metode *AbstractGraph*, predefiniše metode *clear* i *addVertex*, implementira novi *addEdge* metod za dodavanje težinskih grana, i takođe uvodi nove metode za dobijanje minimalnog obuhvatnog stable (minimum spanning trees), kao i za određivanje najkraće putanje do svih čvorova od jednog izvornog čvora (all single-source shortest paths).

U nastavku lekcije ćemo opisati minimalna obuhvatna stabla kao i algoritam za određivanje najkraćih puteva.

IMPLEMENTACIJA KLASE WEIGHTEDGRAPH

Cilj programa je implementacija programa u Javi koji kreira klasu za težinski graf i implementira algoritam za MST stablo (Primov algoritam) i algoritam za određivanje najkraćeg

Celokupna implementacija klase *WeightedGraph* se nalazi u nastavku.

Kada kreirate objekat tipa WeightedGraph korišćenjem konstruktora bez argumeanta, posredstvom super se poziva odgovarajući konstruktor osnovne klase AbstractClass, koji takođe nema argumente.

Kada kreirate objekat tipa WeightedGraph korišćenjem ostala četiri konstruktora, pozivaju se odgovarajući konstruktori osnovne klase (lines 14, 20, 26, 33) koji inicijalizuju liste vertices i neighbors klase AbstractGraph. Kao dodatak, za instance klase WiehtedGraph se kreiraju prioritetni redovi. Metodi clear i addVertex klase AbstractGraph su nadglasani (overridden) u linijama 82–92 da bi mogli da prihvate težinske grane. Metod addEdge(u, v, weight) dodaje novu granu (u, v) odgovarajuće težine u graf (linije 95–99).

```
import java.util.*;

public class WeightedGraph<V> extends AbstractGraph<V> {
  // Priority adjacency lists
  private List<PriorityQueue<WeightedEdge>> queues
    = new ArrayList<PriorityQueue<WeightedEdge>>();

  /** Construct a WeightedGraph from edges and vertices in arrays */
  public WeightedGraph() {
  }

  /** Construct a WeightedGraph from edges and vertices in arrays */
  public WeightedGraph(int[][] edges, V[] vertices) {
    super(edges, vertices);
    createQueues(edges, vertices.length);
  }

  /** Construct a WeightedGraph from edges and vertices in List */
  public WeightedGraph(int[][] edges, int numberOfVertices) {
    super(edges, numberOfVertices);
    createQueues(edges, numberOfVertices);
  }

  /** Construct a WeightedGraph for vertices 0, 1, 2 and edge list */
  public WeightedGraph(List<WeightedEdge> edges, List<V> vertices) {
    super((List)edges, vertices);
    createQueues(edges, vertices.size());
  }

  /** Construct a WeightedGraph from vertices 0, 1, and edge array */
  public WeightedGraph(List<WeightedEdge> edges,
```

```
    int numberOfVertices) {
  super((List)edges, numberOfVertices);
  createQueues(edges, numberOfVertices);
}

/** Create priority adjacency lists from edge arrays */
private void createQueues(int[][] edges, int numberOfVertices) {
  for (int i = 0; i < numberOfVertices; i++) {
    queues.add(new PriorityQueue<WeightedEdge>()); // Create a queue
  }

  for (int i = 0; i < edges.length; i++) {
    int u = edges[i][0];
    int v = edges[i][1];
    int weight = edges[i][2];
    // Insert an edge into the queue
    queues.get(u).offer(new WeightedEdge(u, v, weight));
  }
}

/** Create priority adjacency lists from edge lists */
private void createQueues(List<WeightedEdge> edges,
    int numberOfVertices) {
  for (int i = 0; i < numberOfVertices; i++) {
    queues.add(new PriorityQueue<WeightedEdge>()); // Create a queue
  }

  for (WeightedEdge edge: edges) {
    queues.get(edge.u).offer(edge); // Insert an edge into the queue
  }
}

/** Display edges with weights */
public void printWeightedEdges() {
  for (int i = 0; i < queues.size(); i++) {
    System.out.print(getVertex(i) + " (" + i + "): ");
    for (WeightedEdge edge : queues.get(i)) {
      System.out.print("(" + edge.u +
        ", " + edge.v + ", " + edge.weight + ") ");
    }
    System.out.println();
  }
}

/** Get the edges from the weighted graph */
public List<PriorityQueue<WeightedEdge>> getWeightedEdges() {
  return queues;
}

/** Clears the weighted graph */
public void clear() {
  vertices.clear();
  neighbors.clear();
```

```
  queues.clear();
}

/** Add vertices to the weighted graph */
public void addVertex(V vertex) {
  super.addVertex(vertex);
  queues.add(new PriorityQueue<WeightedEdge>());
}

/** Add edges to the weighted graph */
public void addEdge(int u, int v, double weight) {
  super.addEdge(u, v);
  queues.get(u).add(new WeightedEdge(u, v, weight));
  queues.get(v).add(new WeightedEdge(v, u, weight));
}

/** Get a minimum spanning tree rooted at vertex 0 */
public MST getMinimumSpanningTree() {
  return getMinimumSpanningTree(0);
}

/** Get a minimum spanning tree rooted at a specified vertex */
public MST getMinimumSpanningTree(int startingVertex) {
  List<Integer> T = new ArrayList<Integer>();
  // T initially contains the startingVertex;
  T.add(startingVertex);

  int numberOfVertices = vertices.size(); // Number of vertices
  int[] parent = new int[numberOfVertices]; // Parent of a vertex
  // Initially set the parent of all vertices to -1
  for (int i = 0; i < parent.length; i++)
    parent[i] = -1;
  double totalWeight = 0; // Total weight of the tree thus far

  // Clone the priority queue, so to keep the original queue intact
  List<PriorityQueue<WeightedEdge>> queues = deepClone(this.queues);

  // All vertices are found?
  while (T.size() < numberOfVertices) {
    // Search for the vertex with the smallest edge adjacent to
    // a vertex in T
    int v = -1;
    double smallestWeight = Double.MAX_VALUE;
    for (int u : T) {
      while (!queues.get(u).isEmpty() &amp;amp;&amp;amp;
          T.contains(queues.get(u).peek().v)) {
        // Remove the edge from queues[u] if the adjacent
        // vertex of u is already in T
        queues.get(u).remove();
      }

      if (queues.get(u).isEmpty()) {
        continue; // Consider the next vertex in T
```

```
      }

      // Current smallest weight on an edge adjacent to u
      WeightedEdge edge = queues.get(u).peek();
      if (edge.weight < smallestWeight) {
        v = edge.v;
        smallestWeight = edge.weight;
        // If v is added to the tree, u will be its parent
        parent[v] = u;
      }
    } // End of for

    if (v != -1)
      T.add(v); // Add a new vertex to the tree
    else
      break; // The tree is not connected, a partial MST is found

    totalWeight += smallestWeight;
  } // End of while

  return new MST(startingVertex, parent, T, totalWeight);
}

/** Clone an array of queues */
private List<PriorityQueue<WeightedEdge>> deepClone(
  List<PriorityQueue<WeightedEdge>> queues) {
  List<PriorityQueue<WeightedEdge>> copiedQueues =
    new ArrayList<PriorityQueue<WeightedEdge>>();

  for (int i = 0; i < queues.size(); i++) {
    copiedQueues.add(new PriorityQueue<WeightedEdge>());
    for (WeightedEdge e : queues.get(i)) {
      copiedQueues.get(i).add(e);
    }
  }

  return copiedQueues;
}

/** MST is an inner class in WeightedGraph */
public class MST extends Tree {
  private double totalWeight; // Total weight of all edges in the tree

  public MST(int root, int[] parent, List<Integer> searchOrder,
      double totalWeight) {
    super(root, parent, searchOrder);
    this.totalWeight = totalWeight;
  }

  public double getTotalWeight() {
    return totalWeight;
  }
}
```

```
/** Find single source shortest paths */
public ShortestPathTree getShortestPath(int sourceVertex) {
  // T stores the vertices whose path found so far
  List<Integer> T = new ArrayList<Integer>();
  // T initially contains the sourceVertex;
  T.add(sourceVertex);

  // vertices is defined in AbstractGraph
  int numberOfVertices = vertices.size();

  // parent[v] stores the previous vertex of v in the path
  int[] parent = new int[numberOfVertices];
  parent[sourceVertex] = -1; // The parent of source is set to -1

  // cost[v] stores the cost of the path from v to the source
  double[] cost = new double[numberOfVertices];
  for (int i = 0; i < cost.length; i++) {
    cost[i] = Double.MAX_VALUE; // Initial cost set to infinity
  }
  cost[sourceVertex] = 0; // Cost of source is 0

  // Get a copy of queues
  List<PriorityQueue<WeightedEdge>> queues = deepClone(this.queues);

  // Expand T
  while (T.size() < numberOfVertices) {
    int v = -1; // Vertex to be determined
    double smallestCost = Double.MAX_VALUE; // Set to infinity
    for (int u : T) {
      while (!queues.get(u).isEmpty() &amp;amp;&amp;amp;
          T.contains(queues.get(u).peek().v)) {
        queues.get(u).remove(); // Remove the vertex in queue for u
      }

      if (queues.get(u).isEmpty()) {
        // All vertices adjacent to u are in T
        continue;
      }

      WeightedEdge e = queues.get(u).peek();
      if (cost[u] + e.weight < smallestCost) {
        v = e.v;
        smallestCost = cost[u] + e.weight;
        // If v is added to the tree, u will be its parent
        parent[v] = u;
      }
    } // End of for

    T.add(v); // Add a new vertex to T
    cost[v] = smallestCost;
  } // End of while
```

```
    // Create a ShortestPathTree
    return new ShortestPathTree(sourceVertex, parent, T, cost);
  }

  /** ShortestPathTree is an inner class in WeightedGraph */
  public class ShortestPathTree extends Tree {
    private double[] cost; // cost[v] is the cost from v to source

    /** Construct a path */
    public ShortestPathTree(int source, int[] parent,
        List<Integer> searchOrder, double[] cost) {
      super(source, parent, searchOrder);
      this.cost = cost;
    }

    /** Return the cost for a path from the root to vertex v */
    public double getCost(int v) {
      return cost[v];
    }

    /** Print paths from all vertices to the source */
    public void printAllPaths() {
      System.out.println("All shortest paths from " +
        vertices.get(getRoot()) + " are:");
      for (int i = 0; i < cost.length; i++) {
        printPath(i); // Print a path from i to the source
        System.out.println("(cost: " + cost[i] + ")"); // Path cost
      }
    }
  }
}
```

TESTIRANJE KLASE ZA TEŽINSKI GRAF

Cilj programa je implementacija programa u Javi koji kreira klasu za težinski graf

Testiranje kreirane klase za težinski graf:

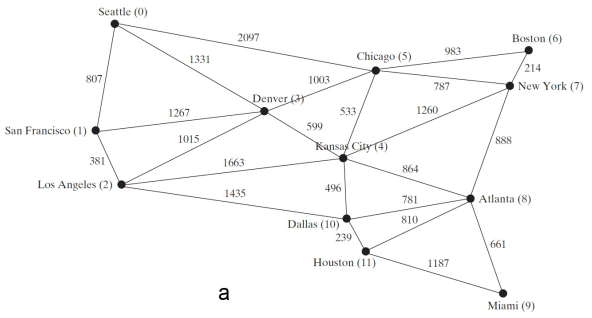
```
public class TestWeightedGraph {
  public static void main(String[] args) {
    String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
    "Denver", "Kansas City", "Chicago", "Boston", "New York",
    "Atlanta", "Miami", "Dallas", "Houston"};

    int[][] edges = {
    {0, 1, 807}, {0, 3, 1331}, {0, 5, 2097},
    {1, 0, 807}, {1, 2, 381}, {1, 3, 1267},
    {2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
    {3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599},
```

```
        {3, 5, 1003},
    {4, 2, 1663}, {4, 3, 599}, {4, 5, 533}, {4, 7, 1260},
        {4, 8, 864}, {4, 10, 496},
    {5, 0, 2097}, {5, 3, 1003}, {5, 4, 533},
        {5, 6, 983}, {5, 7, 787},
    {6, 5, 983}, {6, 7, 214},
    {7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},
    {8, 4, 864}, {8, 7, 888}, {8, 9, 661},
        {8, 10, 781}, {8, 11, 810},
    {9, 8, 661}, {9, 11, 1187},
    {10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},
    {11, 8, 810}, {11, 9, 1187}, {11, 10, 239}
    };

    WeightedGraph<String> graph1 =
    new WeightedGraph<String>(edges, vertices);
    System.out.println("The number of vertices in graph1: "
    + graph1.getSize());
    System.out.println("The vertex with index 1 is "
    + graph1.getVertex(1));
    System.out.println("The index for Miami is " +
    graph1.getIndex("Miami"));
    System.out.println("The edges for graph1:");
    graph1.printWeightedEdges();
  }
}
```

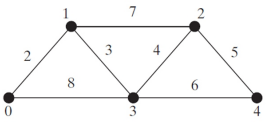
Pokusajte da pokrenete program kako bi ste videli rezultat izvrsavanja. Program kreira graf graph1 za primer sa Slike -2a, u linijama 3–27. Temena grafa graph1 su definisana u linijama 3–5dok su njegove grane definisane u linijama 7–24.

Grane grafa su predstavljene korišćenjem 2D niza. Za svaku vrstu i u nizu, edges[i][0] i edges[i][1] označavaju da postoji grana koja vodi iz temena edges[i][0] u teme edges[i][1] i da težina grane iznosi edges[i][2]. Na primer, {0, 1, 807} (line 8) predstavlja granu koja vodi iz temena 0 (edges[0][0]) u teme 1 (edges[0][1]) i težina joj je 807 (edges[0][2]). {0, 5, 2097} (line 8) predstavlja granu koja ide iz temena 0 (edges[2][0]) u teme 5 (edges[2][1]) težine 2097 (edges[2][2]). U liniji 35 se poziva printWeightedEdges() metod koji za graph1 štampa sve njegove grane.

Program kreira graph2 za primer sa Slike-2a u linijama 37–45. U liniji 47 se poziva printWeightedEdges() metod koji za graph2 štampa sve njegove grane.



a



b

Slika 2.2 Primeri za koje su kreirani grafovi u klasi TestWeightedGraph

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Šta je izlaz sledećeg programa?

```
PriorityQueue<WeightedEdge> q =
    new PriorityQueue<WeightedEdge>();
q.offer(new WeightedEdge(1, 2, 3.5));
q.offer(new WeightedEdge(1, 6, 6.5));
q.offer(new WeightedEdge(1, 7, 1.5));
System.out.println(q.poll().weight);
System.out.println(q.poll().weight);
System.out.println(q.poll().weight);
```

Zadatak 2. Šta je greška u ovom kodu? Popravite je i prikažite izlaz programa.

```
List<PriorityQueue<WeightedEdge>> queues =
    new ArrayList<PriorityQueue<WeightedEdge>>();
queues.get(0).offer(new WeightedEdge(0, 2, 3.5));
queues.get(0).offer(new WeightedEdge(0, 6, 6.5));
queues.get(0).offer(new WeightedEdge(0, 7, 1.5));
queues.get(1).offer(new WeightedEdge(1, 0, 3.5));
queues.get(1).offer(new WeightedEdge(1, 5, 8.5));
queues.get(1).offer(new WeightedEdge(1, 8, 19.5));
System.out.println(queues.get(0).peek()
    .compareTo(queues.get(1).peek()));
```

Zadatak 3. Modifikovati *TestWeightedGraph*, kako bi se napravio fajl za prikaz **graph1**. Fajl napraviti od niza na linijama 7–24. Broj čvorova grafa je 12, koji će biti prikazani u prvoj liniji fajla. Jedna ivica **edge (u, v)** se čuva ako u < v. Fajl treba da bude sledećeg formata:

```
12
0, 1, 807 | 0, 3, 1331 | 0, 5, 2097
1, 2, 381 | 1, 3, 1267
2, 3, 1015 | 2, 4, 1663 | 2, 10, 1435
3, 4, 599 | 3, 5, 1003
4, 5, 533 | 4, 7, 1260 | 4, 8, 864 | 4, 10, 496
5, 6, 983 | 5, 7, 787
6, 7, 214
7, 8, 888
8, 9, 661 | 8, 10, 781 | 8, 11, 810
9, 11, 1187
10, 11, 239
```

Zadatak 4. Izmeniti *GraphView* (prethodna lekcija o grafovima) kako bi se prikazao težinski graf. Napisati program za prikaz grafa sa **Slika-2**.

Poglavlje 3

Minimalno obuhvatno stablo

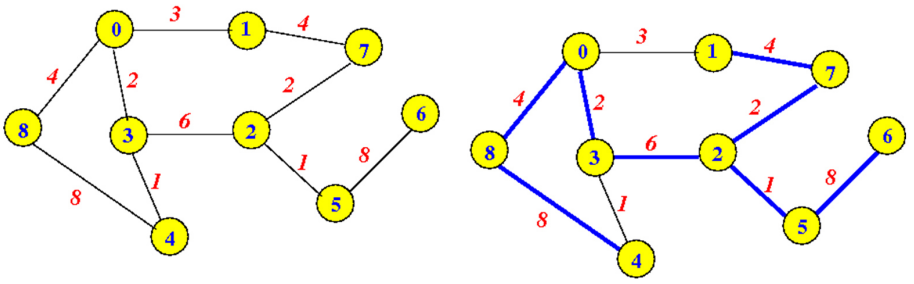
OBUHVATNO STABLO GRAFA

Obuhvatno stablo T jednog neusmerenog grafa G je podgraf koji uključuje sve čvorove grafa G. Da bi se odredilo obuhvatno stablo grafa, graf mora biti povezan

Obuhvatno stablo (engl. spanning tree) *T* grafa *G* je podgraf grafa *G* tako da je *V(T) = V(G)*.

Da bi se odredilo obuhvatno stablo grafa, graf mora biti povezan!

Slika-1 poredi graf *G* i obuhvatno stablo grafa *G*.



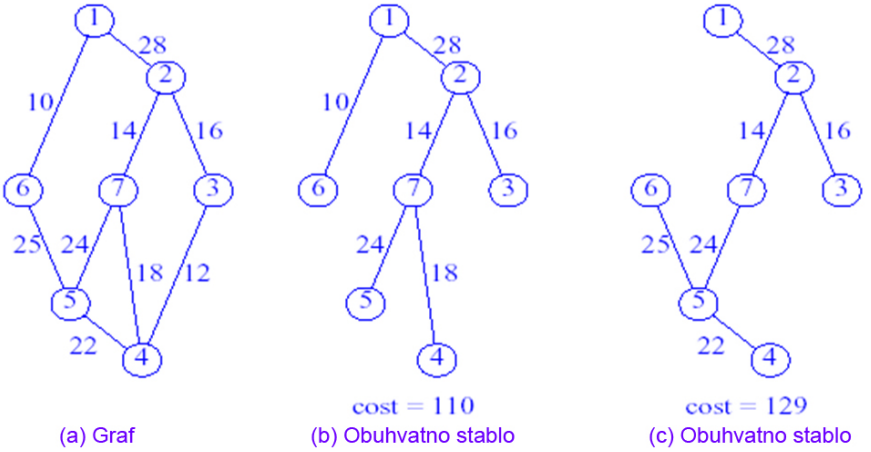
Slika 3.1 Graf G i obuhvatno stablo grafa G

Obuhvatno stablo *T* jednog neusmerenog grafa *G* je podgraf koji uključuje sve čvorove grafa *G*. Taj podgraf je istovremeno i stablo.

Generalno, graf može imati nekoliko obuhvatnih stabala.

Ako su sve grane *G* takođe grane obuhvatnog stabla *T* grafa *G*, tada je *G* stablo i identično je sa *T* (tj, stablo ima jedinstveno obuhvatno stablo i to je ono samo).

Na Slici-2 je prikazan primer grafa i njegovih obuhvatnih stabala.

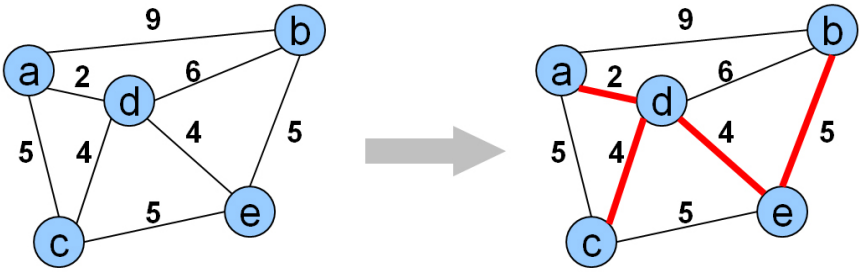


Slika 3.2 Graf i obuhvatno stablo, primer 1

OSNOVI O MINIMALNOM OBUHVATNOM STABLU

Minimalno obuhvatno stablo je podgraf neusmerenog težinskog grafa G sa osobinom: ukupna cena povezanih grana stabla je minimalna među svim mogućim obuhvatnim stablima

Minimalno obuhvatno stablo (engl. Minimum Spanning Tree-MST) je aciklično stablo neusmerenog težinskog grafa koje uključuje sve čvorove *V* i sadrži |*V*|- *1* grana. Ukupna cena povezanih grana je minimalna među svim mogućim obuhvatnim stablima. Pri tome nije nužno da je to jedinstveno stablo. Slika-3 ilustruje kako se kreira MST stablo.

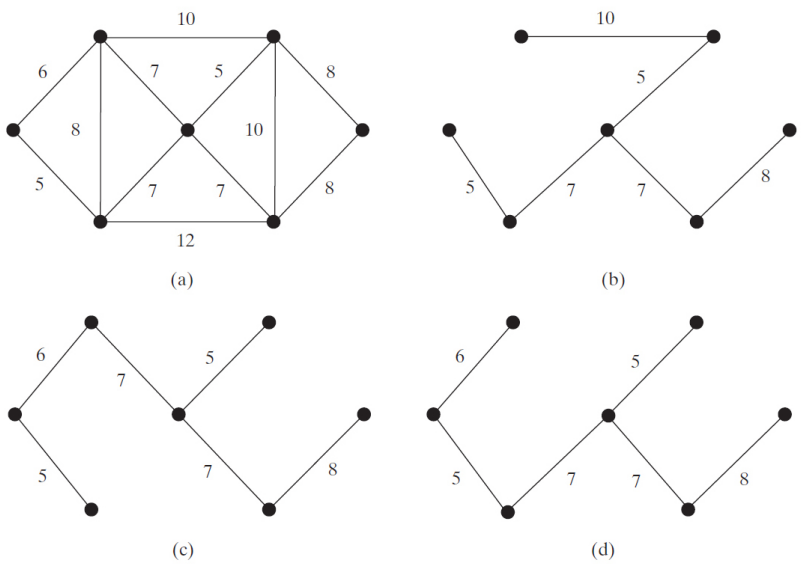


Slika 3.3 Kreiranje minimalnog obuhvatnog stabla - MST.

Problem određivanja minimalnog obuhvatnog stabla ima široku primenu. Razmotrimo kompaniju koja ima filijale u velikom broju gradova. Cilj kompanije je da umreži telefonske centrale svake od filijala. Telefonska kompanija naplaćuje uslugu povezivanja različito za različite parove gradova.

Postoji mnogo načina da se gradovi povežu granama međusobno. Najjeftiniji način je da se formira obuhvatno stablo grafa sa najmanjim ukupnim troškovima.

Graf može da ima nekoliko obuhvatnih stabala. Minimalno obuhvatno stablo grafa je ono koje ima najmanju ukupnu težinu. Na primer, stabla na Slikama 4b, 4c, 4d su obuhvatna stabla grafa sa Slike 4a, dok su stabla sa Slike 4c i 4d i minimalna obuhvatna stabla.



Slika 3.4 Stabla na slikama (c) i (d) su minimalna obuhvatna stabla grafa sa slike (a).

Dva osnovna algoritma za određivanje MST stabla su: Primov algoritam i Kruskalov algoritam.

PRIMOV ALGORITAM

Primov algoritam definiše kako izgraditi stablo iterativnim dodavanjem grana dok se ne dobije minimalno obuhvatno stablo. Algoritam se sastoji iz nekoliko koraka

Primov algoritam definiše kako izgraditi stablo iterativnim dodavanjem grana dok se ne dobije minimalno obuhvatno stablo.

Npr, započeti konstruisanje stabla od čvora 0 i razvijati stablo označeno sa *T*. U svakom sledećem koraku, dodati najjeftiniju granu koja ima tačno jednu krajnju tačku u *T*. Procedura je sledeća:

- Graditi *MST* stablo *T* dodavanjem grana, jednu po jednu.
- U svakoj fazi dodati granu sa najmanjom cenom skupu *T* tako da skup odabranih grana još uvek čini stablo.
- Ponoviti postupak dodavanja grana skupu *T* sve dok *T* ne sadrži *N-1* grana.

Primov algoritam može biti napisan na sledeći način:

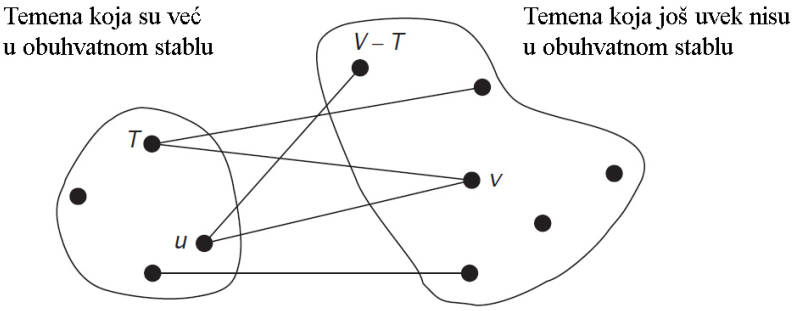
```
minimumSpanningTree() {
 neka je V označava skup čvorova grafa;
 neka je T skup čvorova povezujućeg stabla;
 na početku, dodati početni čvor u T;
 while (size of T < n) {
  naći u u T i v u (V – T) s najmanjom težinom na grani (u, v),
  kao na slici;
  dodati v u T;
```

```
  }
}
```

Analizirajmo primer sa Slike-5.

Algoritam započinje dodavanjem početnog čvora u skup *T*. Zatim se iznova i iznova dodaje čvor (recimo *v*) iz *(V – T)* u skup *T*. *v* je čvor koji je najbliži sused (sa najmanjom težinom) nekom od čvorova iz skupa *T*.

Na primer, postoji pet grana koje povezuju čvorove skupa *T* i *V – T*, kao što je prikazano na Slici-5, i grana *(u, v)* je ona koja ima najmanju težinu.



Slika 3.5 Određivanje čvora u iz skupa T koji povezuje čvor v iz skupa V – T tako da je težina grane najmanja.

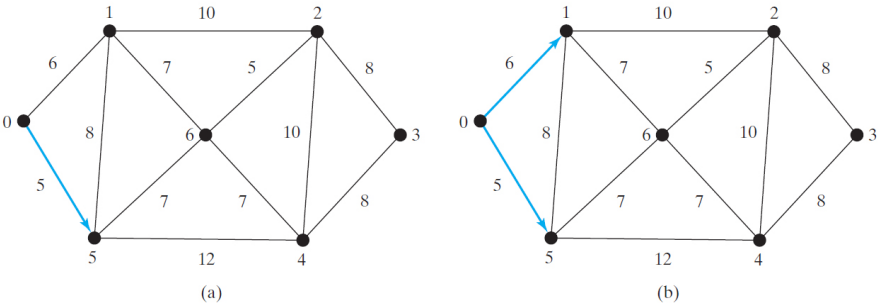
POSTUPAK PRIMOVOG ALGORITMA

U svakoj fazi Primovog algoritma dodati granu sa najmanjom cenom skupu T tako da skup odabranih grana još uvek čini stablo. Postupak ponoviti dok se obuhvate svi čvorovi grafa

Razmotrimo graf sa Slika 6, 7 i 8.

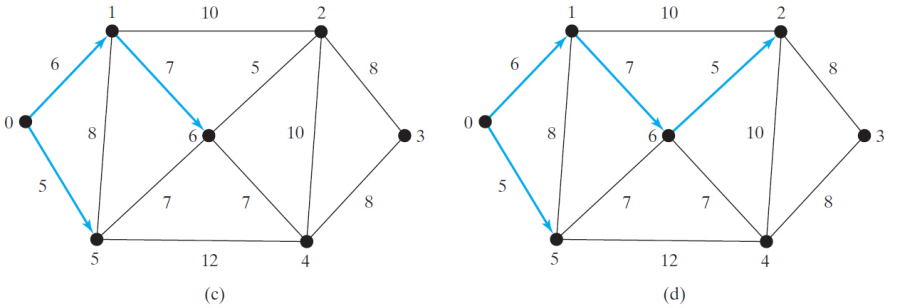
Algoritam dodavanja čvorova u skup *T* se sastoji iz sledećih koraka:

Dodati čvor 0 u *T*. Zatim dodati čvor 5 u *T* pošto grana *Edge(5, 0, 5)* ima najmanju težinu među svim ostalim granama ka čvorovima koji si trenutno u *T* (Slika-6a). Nakon toga dodati čvor 1 u *T*, pošto grana *Edge(1, 0, 6)* ima najmanju težinu među svim ostalim granama ka čvorovima koji su trenutno u *T*, kao na Slici-6b.



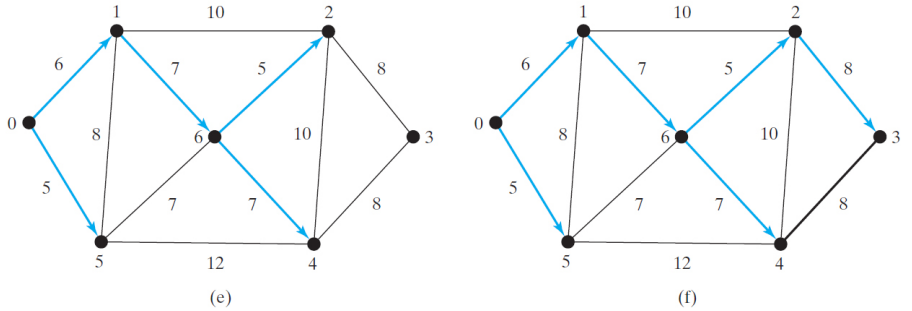
Slika 3.6 Formiranje MST stabla – Koraci a i b.

Zatim dodati čvor 6 u *T*, pošto grana *Edge(6, 1, 7)* ima najmanju težinu, kao na Slici-7c. Nakon toga dodati čvor 2 u *T*, pošto grana *Edge(2, 6, 5)* ima najmanju težinu, kao na Slici-7d.



Slika 3.7 Formiranje MST stabla – Koraci c i d.

Dodati čvor 4 u *T*, pošto grana *Edge(4, 6, 7)* ima najmanju težinu (Slika-8e). Na kraju dodati čvor 3 u *T* pošto grana *Edge(3, 2, 8)* ima najmanju težinu među svim ostalim granama ka čvorovima koji si trenutno u *T*, (Slika-8f).

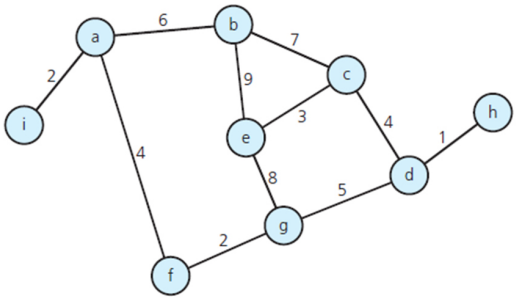


Slika 3.8 Formiranje MST stabla – Koraci e i f.

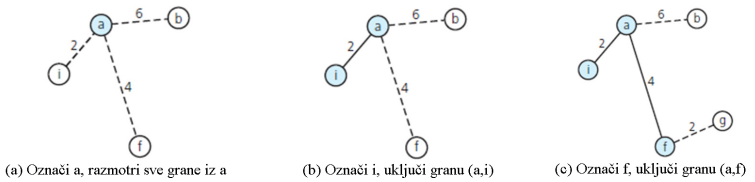
PRIMER KREIRANJA MST STABLA

Svaka sledeća grana koju dodajemo u skup obrađenih grana, skup T, ima najmanju težinu među svim ostalim granama ka čvorovima koji si trenutno u T

Razmotrićemo kreiranje MST stabla na sledećem primeru, Slika-9. Na Slikama 10-12 su koraci kreiranja MST stabla.

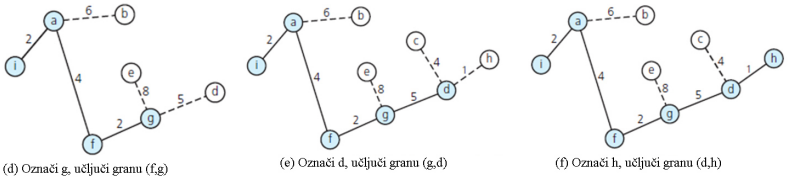


Slika 3.9 Primer Primovog algoritma – Početni graf

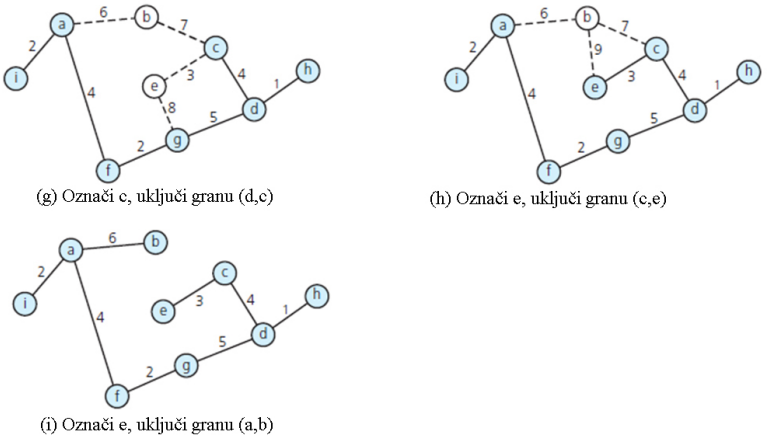


Slika 3.10 10 Primer Primovog algoritma – Koraci a), b) i c)

Krećemo od čvora *a*, Slika-10a. Prvo ubacujemo granu težine 2 (*fg*), Slika- 10b, zatim granu težine 4 (*af*), Slika-10c. Sledeća grana najmanje težine je 2 (*ai*) i nju ubacujemo u skup T, Slika-11d. Među neposećenim granama najmanja je sada težine 5 (*gd*) i nju ubacujemo u T, Slika-7e, a zatim i granu težine 1 (*dh*), Slika11f.



Slika 3.11 Primov algoritam – Koraci d), e) i f)



Slika 3.12 Primov algoritam – Koraci g), h) i i)

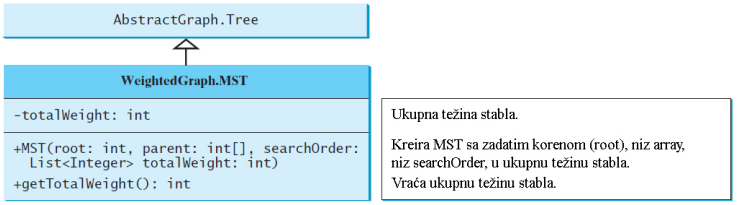
Na Slici-12 je opisan postupak dodavanja ostalih grana – grana težine 4 (*cd* - Slika-12g), grana težine 3 (*ec* - Slika 12h), i na kraju grana težine 6 (*ab*), nakon čega je kreirano MST stablo grafa.

IMPLEMENTACIJA MST ALGORITMA

Metod getMinimumSpanningTree je definisan u okviru klase WeightedGraph. Kao rezultat on vraća instancu klase MST koja je definisana za minimalno obuhvatno stablo

Metod *getMinimumSpanningTree(int v)* je definisan u okviru klase *WeightedGraph*. Kao rezultat on vraća instancu klase *MST*, kao na Slici-1 sekcije *Implementacija težinskog grafa*.

Klasa *MST* je definisana kao unutrašnja klasa u okviru *WeightedGraph*, pri čemu *MST* proširuje klasu *Tree*, kao na Slici-13. Klasu *Tree* smo prikazali u prethodnoj lekciji o bestežinskom grafu i njegovom obuhvatnom stablu.



Slika 3.13 Klasa MST proširuje klasu Tree obuhvatnog stabla.

Metod *getMinimumSpanningTree(int startingVertex)* prvo dodaje početni čvor,*startingVertex* u skup *T*. *T* je lista koja čuva čvorove koji su dodati u obuhvatno stablo. *vertices* je definisan kao zaštićeni podatak u okviru klase *AbstractGraph*, i to je niz u kome se čuvaju čvorovi grafa.

Čvor se dodaje u skup *T* ukoliko je susedan nekom od čvorova koji su već u *T* i ukoliko je rastojanje do njega manje nego do svih ostalih čvorova koji još uvek nisu u *T*. Takav čvor se pronalazi korišćenjem sledeće procedure:

- Za svaki čvor *u* iz *T*, pronaći njegovog suseda sa najmanjom težinom grane do *u*. Svi susedi su smešteni u listu *queues.get(u)*. Metod *queues.get(u).peek()* vraća susednu granu sa najmanjom težinom. Ukoliko je sused već u *T*, obrisati ga. Da bi originalni red sačuvali netaknut, vrši se njegovo kopiranje. Nakon pretrage, metod *queues.get(u).peek()* vraća čvor sa najlakšom granom do *u*.
- Uporediti sve susede i pronaći suseda sa najmanjom težinom. Nakon što se novi čvor doda u *T*, *totalWeight* se ažurira. Nakon što su svi čvorovi dodati u *T*, kreira se instanca klase *MST*. Treba napomenuti da metod neće raditi ako graf nije povezan.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

KLASA MST I METOD GETMINIMUMSPANNINGTREE

Klasa MST je definisana kao unutrašnja klasa u okviru klase WeightedGraph, pri čemu MST proširuje klasu Tree

U nastavku je listing metoda *getMinimumSpanningTree(int startingVertex)*:

```
/** Get a minimum spanning tree rooted at vertex 0 */
  public MST getMinimumSpanningTree() {
    return getMinimumSpanningTree(0);
  }

  /** Get a minimum spanning tree rooted at a specified vertex */
  public MST getMinimumSpanningTree(int startingVertex) {
    List<Integer> T = new ArrayList<Integer>();
    // T initially contains the startingVertex;
```

```
    T.add(startingVertex);

    int numberOfVertices = vertices.size(); // Number of vertices
    int[] parent = new int[numberOfVertices]; // Parent of a vertex
    // Initially set the parent of all vertices to -1
    for (int i = 0; i < parent.length; i++)
    parent[i] = -1;
    double totalWeight = 0; // Total weight of the tree thus far

    // Clone the priority queue, so to keep the original queue intact
    List<PriorityQueue<WeightedEdge>> queues = deepClone(this.queues);

    // All vertices are found?
    while (T.size() < numberOfVertices) {
    // Search for the vertex with the smallest edge adjacent to
    // a vertex in T
    int v = -1;
    double smallestWeight = Double.MAX_VALUE;
    for (int u : T) {
        while (!queues.get(u).isEmpty() &amp;amp;&amp;amp;
            T.contains(queues.get(u).peek().v)) {
        // Remove the edge from queues[u] if the adjacent
        // vertex of u is already in T
        queues.get(u).remove();
        }

        if (queues.get(u).isEmpty()) {
        continue; // Consider the next vertex in T
        }

        // Current smallest weight on an edge adjacent to u
        WeightedEdge edge = queues.get(u).peek();
        if (edge.weight < smallestWeight) {
        v = edge.v;
        smallestWeight = edge.weight;
        // If v is added to the tree, u will be its parent
        parent[v] = u;
        }
    } // End of for

    if (v != -1)
        T.add(v); // Add a new vertex to the tree
    else
        break; // The tree is not connected, a partial MST is found

    totalWeight += smallestWeight;
    } // End of while

    return new MST(startingVertex, parent, T, totalWeight);
  }
```

Klasa *MST* proširuje klasu *Tree*. Da bi kreirali instancu klase *MST*, proslediti *root*, *parent*, *T*, i *totalWeight* konstruktoru. Polja *root*, *parent*, i *searchOrder* su definisana u okviru klase *Tree*, koju smo u okviru prethodnog primera imali kao unutrašnju klasu unutar *AbstractGraph*.

```
/** MST is an inner class in WeightedGraph */
  public class MST extends Tree {
    private double totalWeight; // Total weight of all edges in the tree

    public MST(int root, int[] parent, List<Integer> searchOrder,
        double totalWeight) {
    super(root, parent, searchOrder);
    this.totalWeight = totalWeight;
    }

    public double getTotalWeight() {
    return totalWeight;
    }
  }
```

Analiza vremenske složenosti

Za svaki čvor program kreira prioritetni red za sve grane koje izlaze iz njega. To zahteva $O(log|V|)$ vremena da se ubaci grana u prioritetni red i isto toliko vremena da se grana izbaci iz njega. Stoga, ukupna vremenska složenost programa je $O(|E| \ log \ |V|)$, gde $|E|$ predstavlja broj grana a $|V|$ broj čvorova grafa.

TESTIRANJE KLASE ZA MST STABLO

Cilj programa je implementacija programa u Javi koji kreira klasu za težinski graf i implementira algoritam za MST stablo (Primov algoritam)

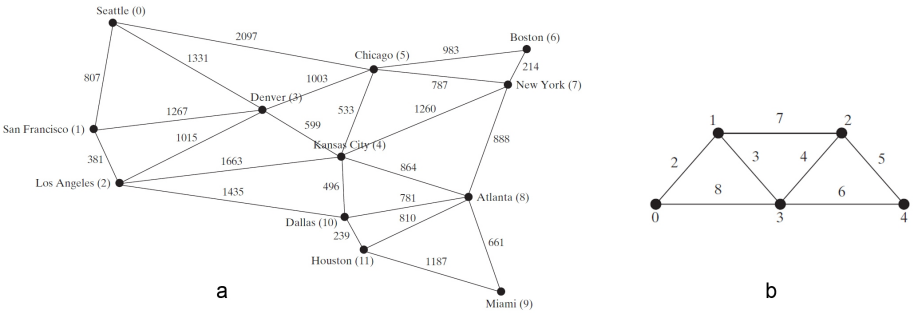
U nastavu je test program koji prikazuje MST za grafove sa Slike-14.

```
public class TestMinimumSpanningTree {
  public static void main(String[] args) {
    String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
      "Denver", "Kansas City", "Chicago", "Boston", "New York",
      "Atlanta", "Miami", "Dallas", "Houston"};

    int[][] edges = {
      {0, 1, 807}, {0, 3, 1331}, {0, 5, 2097},
      {1, 0, 807}, {1, 2, 381}, {1, 3, 1267},
      {2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
      {3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599},
        {3, 5, 1003},
      {4, 2, 1663}, {4, 3, 599}, {4, 5, 533}, {4, 7, 1260},
        {4, 8, 864}, {4, 10, 496},
      {5, 0, 2097}, {5, 3, 1003}, {5, 4, 533},
        {5, 6, 983}, {5, 7, 787},
      {6, 5, 983}, {6, 7, 214},
```

```
      {7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},
      {8, 4, 864}, {8, 7, 888}, {8, 9, 661},
        {8, 10, 781}, {8, 11, 810},
      {9, 8, 661}, {9, 11, 1187},
      {10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},
      {11, 8, 810}, {11, 9, 1187}, {11, 10, 239}
    };

    WeightedGraph<String> graph1 =
      new WeightedGraph<String>(edges, vertices);
    WeightedGraph<String>.MST tree1 = graph1.getMinimumSpanningTree();
    System.out.println("Total weight is " + tree1.getTotalWeight());
    tree1.printTree();

    edges = new int[][]{
      {0, 1, 2}, {0, 3, 8},
      {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
      {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
      {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
      {4, 2, 5}, {4, 3, 6}
    };

    WeightedGraph<Integer> graph2 =
      new WeightedGraph<Integer>(edges, 5);
    WeightedGraph<Integer>.MST tree2 =
      graph2.getMinimumSpanningTree(1);
    System.out.println("Total weight is " + tree2.getTotalWeight());
    tree2.printTree();
  }
}
```

Program kreira težinski graf za primer sa Slike-14a u liniji 27. Zatim poziva metod getMinimumSpanningTree() (line 28) koji vraća objekat tipa MST koji predstavlja minimalno sprežno (obuhvatno) stablo grafa. Pozivom metoda printTree() (line 30) se za prosledjeni MST objekat prikazuju grane koje čine stablo. Primetiti da je MST podklasa od Tree. Metod printTree() je definisan u okviru klase Tree.

Pri kreiranju MST za primer sa Slike-14a, temena se dodaju u sledećem redosledu: Seattle, San Francisco, Los Angeles, Denver, Kansas City, Dallas, Houston, Chicago, New York, Boston, Atlanta, i Miami.

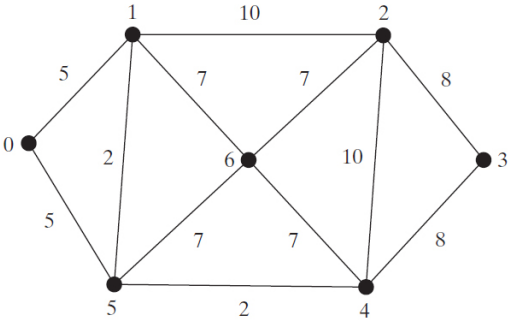


Slika 3.14 Primeri za koje su kreirani grafovi u klasi TestWeightedGraph

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke:

Zadatak 1.Naći minimalno obuhvatno stablo za sledeći graf.



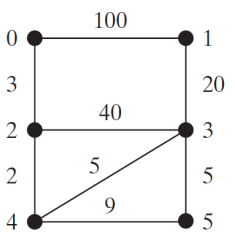
Slika 3.15 Primer za samostalni rad

Zadatak 2. Šta se dešava sa **getMinimumSpanningTree()** metodom u *WeightedGraph* ako graf nije povezan? Proveriti odgovor tako što će se kreirati test program koji pravi nepovezan graf i poziva **getMinimumSpanningTree()** metod. Kako se problem rešava nalaženjem parcijalnog MST?

Zadatak 3. U materijalima je predstavljen Primov algoritam za određivanje minimalnog obuhvatnog stabla. Implementirati u okviru klase *WeightedGraph* metod koji određuje MST korišćenjem Kruskalovog algoritma.

Zadatak 4. U materijalima sa predavanja je predstavljena implementacija Primovog algoritma korišćenjem prioritetnog reda susednih grana. Implementirati Primov algoritam korišćenjem matrice susedstva za težinski graf.

Zadatak 5. Napisati program koji učitava podatke o povezanom grafu iz fajla i prikazuje minimalno obuhvatno stablo. U prvoj liniji fajla se nalazi broj čvorova (*n*), koje su označene indeksima *0, 1, ..., n-1*. Svaka sledeća linija predstavlja grane u obliku *u1*, *v1*, *w1* | *u2*, *v2*, *w2* | .... Svaki triplet predstavlja granu i njenu težinu. Na Slici 1 je primer fajla za odgovarajući graf. Pretpostavka je da je graf neusmeren.U koliko graf ima granu (u,v), takođe ima i granu (v,u), a samo jedna grana je predstabvljena u fajlu.



(a)

```
File
6
0, 1, 100 | 0, 2, 3
1, 3, 20
2, 3, 40 | 2, 4, 2
3, 4, 5 | 3, 5, 5
4, 5, 9
```

(b)

Slika 3.16 Čvorovi i grane težinskog grafa zapisani u fajlu

Zadatak 6. Izmeniti *GraphView* iz prethodne lekcije kako bi se prikazao težinski graf i minimalno obuhvatno stablo za graf sa **Slika-14**. Ivice u MST stablu treba prikazati crvenom bojom.

Poglavlje 4

Problem najkraćih puteva

UVODNA RAZMATRANJA

Problem nakraćeg puta (engl. Shortest Path Problem-SP) je problem pronalaženja puta između dve tačke (ili čvora) u grafu tako da je zbir težina njegovih grana najmanji

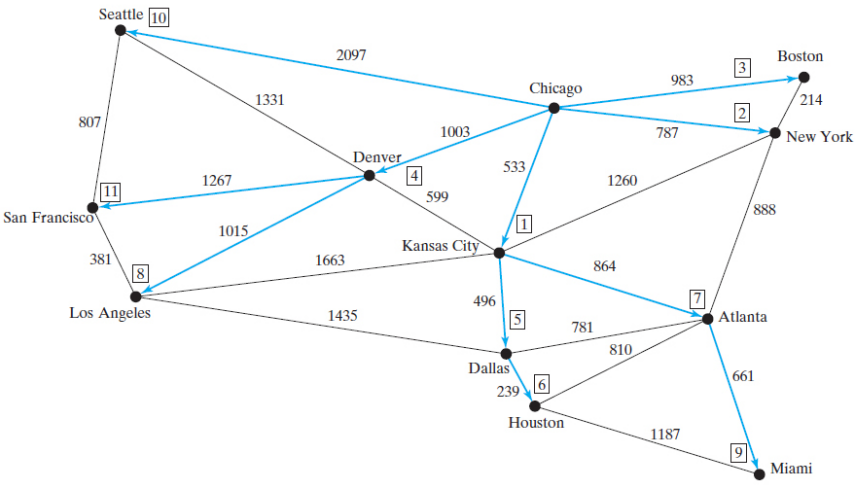
Problem nakraćeg puta (engl. Shortest Path Problem-SP) je problem pronalaženja puta između dve tačke (ili čvora) u grafu tako da je zbir težina grana na putu najmanji.

Problem najkraćeg puta može biti definisan za bilo usmerene, neusmerene ili kombinovane grafove.

U nastavku ćemo pretpostaviti da se *Problem najkraćeg puta* odnosi na usmerene grafove. Ovaj problem se takođe nekad naziva i problem najkraćeg puta jednog para.

Mreža puteva može da se posmatra kao graf sa pozitivnim težinama. Čvorovi predstavljaju raskrsnice a svaka grana grafa predstavlja ulicu između dve raskrsnice.

Težina grane može da predstavlja dužinu ulice, vreme potrebno da se pređe ta ulica ili cena prelaska te ulice. Koristeći usmerene grafove možemo predstaviti i jednosmerne ulice. Ovakvi grafovi su posebni zato što su neke grane bitnije od drugih (npr. autoputevi).



Slika 4.1 Ilustracija algoritma problema najkraćeg puta. Najkraći put od Čikaga do svih ostalih gradova je obojen plavom bojom

Na Slici-1 je ilustracija algoritma problema najkraćeg puta, od grada *Chicago* do svih ostalih gradova.

DIJKSTRA ALGORITAM

Dijkstra algoritam je algoritam koji se zasniva na pamćenju vrednosti d[v] trenutnog najkraćeg puta s za svaki čvor v

Dijkstra algoritam (engl. Dijkstra algorithm) je jedan od algoritama za nalaženje najkraćeg puta u grafu sa nenegativnim težinama grana.

Neka je dat težinski usmereni graf *G* i početni čvor *s* iz *G*. Ako skup svih čvorova grafa obeležimo sa *V*, skup grana sa *E*, tada je svaka grana iz *E*, predstavljena parom čvorova *(u,v)* iz *V*. Takođe, neka svaka grana ima težinu *v*. Dužina puta između dva čvora biće suma težina grana na putu.

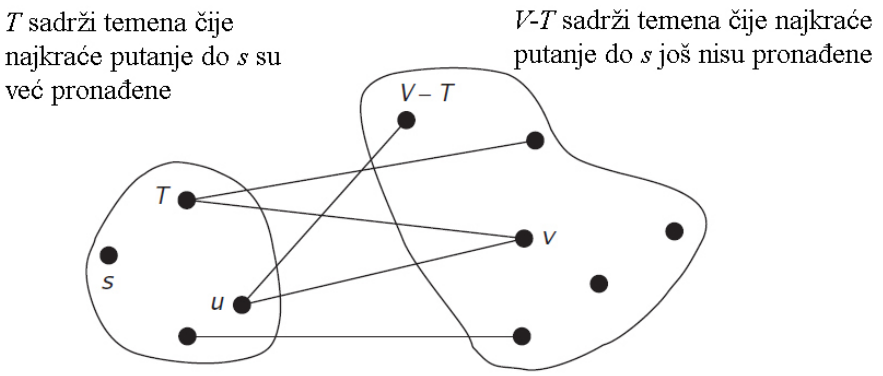
Dijkstra algoritam se zasniva se na pamćenju vrednosti *d[v]* trenutnog najkraćeg puta *s* za svaki čvor *v*. Za početni čvor ta vrednost najpre iznosi *0*, tj. *d[s] = 0*, a za ostale čvorove se uzima vrednost beskonačno. Pri prestanku rada algoritma, *d[v]* dobija vrednost najkraćeg puta *s* ili vrednost beskonačno, ukoliko takav put ne postoji.

Osnovna operacija Dijkstra algoritma je oslobađanje grana: ukoliko postoji grana iz *u* ka *v*, tada trenutno najkraći put *s* (*d[u]*) može dobiti vrednost sume *d[v]* i težine grane *(u,v)*. Dakle, njegova dužina će iznositi *d[u] + w (u,v)*, ukoliko je ova vrednost manja od *d[v]*. Proces oslobađanja grana se nastavlja dok vrednost *d[v]* ne određuje najkraći put *s* za svaki čvor *v*.

Analizirajmo primer sa Slike-2.

Tokom izvršavanja algoritma, izdvajaju se dva skupa čvorova *T* i *V-T*. U skupu T su oni čvorovi za koje je poznata vrednost *d[v]*, a u skupu *V-T* svi ostali. Na početku je skup *T* prazan, a u svakoj iteraciji jedan čvor se premešta iz *V-T* u *T*. To je onaj čvor koji ima najmanju vrednost *d[u]*.

Na kraju se oslobađaju sve grane *(u,v)* prema prethodno opisanom postupku.



Slika 4.2 Dijkstra algoritam: Pronaći teme u skupa koje je povezano sa v skupa V – T tako da je vrednost d[u] + w(u, v) najmanja moguća

PSEUDO-KOD ALGORITMA

Pri prestanku rada algoritma, d[v] dobija vrednost najkraćeg puta s ili vrednost beskonačno, ukoliko takav put ne postoji. Rezultat algoritma je stablo sa korenom u izvornom čvoru

Uprošćeni oblik pseudo-koda je dat u nastavku:

```
shortestPath(s) {
 Neka je sa V oznacen skup čvorova u grafu;
 Neka je T skup koji sadrži čvorove čije su putanje do čvora s poznate;
 Inicijalno T sadrži izvorni čvor s sa vrednošću troškova puta cost[s] = 0;

 while (veličina skupa T < n) {
pronadji v u skupu V – T sa najmanjim troškom puta cost[u] + w(u, v) medju svim u u
T;
  dodati v u skup T I postaviti vrednost cost[v] = cost[u] + w(u, v);
 }
}
```

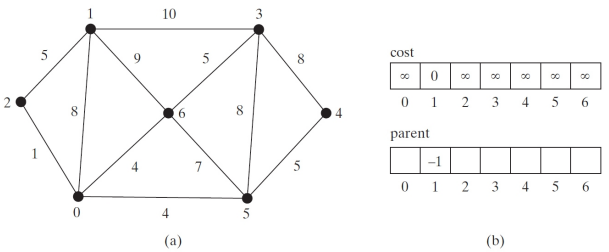
Algoritam koristi niz *cost[v]* za čuvanje troškova najkraćih putanja od čvora *v* do izvornog čvora *s*. *cost[s]* je *0*. Inicijalno dodeliti beskonačno veliku vrednost članu *cost[v]* da vi se označilo da ne postoji poznata putanja od *v* do *s*. Neka je sa *V* označen skup svih čvorova, i neka je sa *T* označen skup čvorova čiji su troškovi putanje od *s* do njih poznati. Inicijalno, izvorni čvor *s* je deo skupa *T*. Algoritam iznova i iznova pronalazi čvor *u* iz *T* i čvor *v* iz *V – T* takve da je vrednost *cost[u] + w(u, v)* najmanja, i zatim pomera *v* u skup *T*. Ovde je sa *w(u, v)* označena težina grane *(u, v)*.

Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.

POSTUPAK DIJKSTRA ALGORITMA

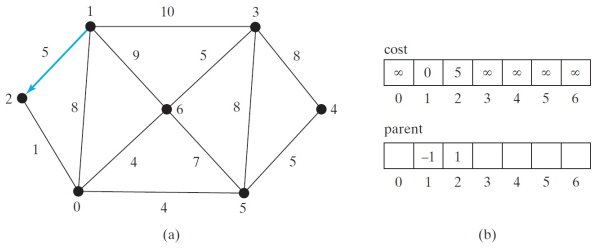
Tokom izvršavanja algoritma, izdvajaju se dva skupa čvorova T i V-T.

Analizirajmo primer sa Slike-3a. Pretpostavimo da je početni čvor 1. Stoga, *cost[1]* je 0 a troškovi za sva ostala temena su inicijalo postavljeni na beskonačno (Slika-3b). Koristimo niz *parent[i]* da bi naznačili roditelja temena i na putanji. Logično, roditelj korena ima vrednost -1.



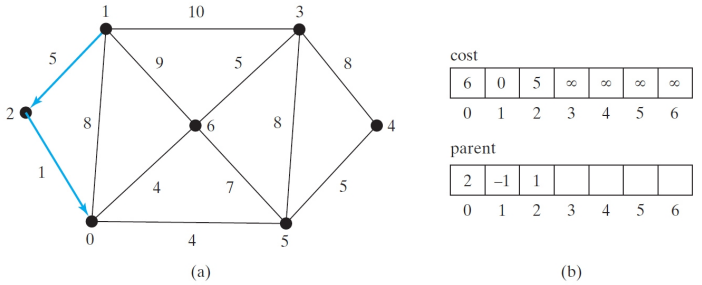
Slika 4.3 Algoritam će pronaći sve nakraće putanje od polaznog čvora 1

Početno postavimo da skup T sadrži samo polazno teme. Temena 2, 0, 6, i 3 su sesedna sa temenima u *T*, i teme 2 je najbliže (ima najmanje troškove) do početnog temena 1, pa dodajmo 2 u skup *T*. *cost[2]* sada postaje 5, (Slika-4).



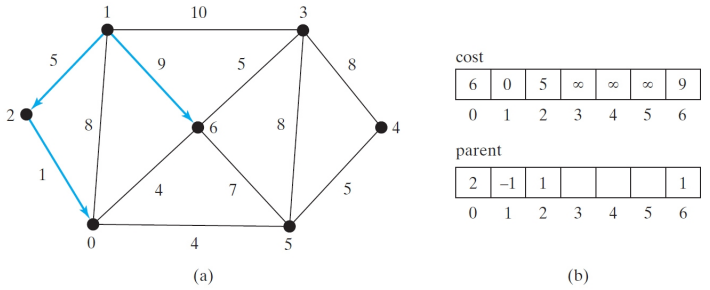
Slika 4.4 Sada su temena 1 i 2 u skupu T

T sada sadrži {1, 2}. Temena 0, 6, i 3 su susedna sa temenima u *T*, I teme 0 ima najkraću putanju do polaznog čvora 1, pa dodajemo 0 u *T*. *cost[0]* sada postaje 6, (Slika-5).



Slika 4.5 Temena {1, 2, 0} su u skupu T

Sada *T* sadrži {1, 2, 0}. Temena 3, 6, i 5 su susedi temenima u *T*, a teme 6 ima njkraću putanju do polaznog temena 1, pa dodajemo 6 u T. *cost[6]* sada postaje 9, (Slika-6).



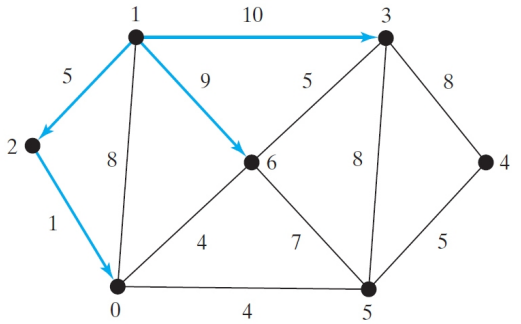
Slika 4.6 Temena {1, 2, 0, 6} su u skupu T

POSTUPAK IZVRŠAVANJA DIJKSTRA ALGORITMA

U skupu T su oni čvorovi za koje je poznata vrednost d[v], a u skupu V-T svi ostali

Nastavljamo sa analizom primera sa prethodnog slajda.

Skup *T* sada sadrži {1, 2, 0, 6}. Temena 3 i 5 su susedna sa temenima koja su već u *T*, i oba imaju putanju iste dužine do polaznog čvora 1. Stoga proizvoljno biramo jedno od ta dva temena, ili 3 ili 5. Dodajmo 3 u *T*. *cost[3]* sada postaje 10 (Slika-7).



(a)

cost

0	1	2	3	4	5	6
6	0	5	10	∞	∞	9

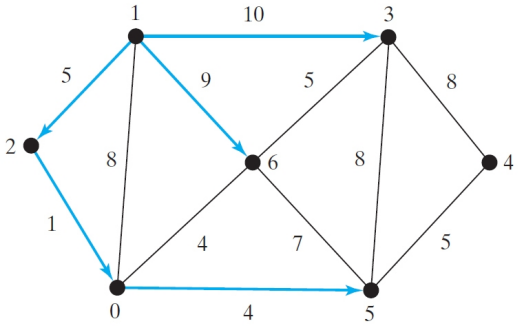
parent

0	1	2	3	4	5	6
2	–1	1	1			1

(b)

Slika 4.7 Temena {1, 2, 0, 6,3} su u skupu T

Skup *T* sadrži elemente {1, 2, 0, 6, 3}. Temena 4 i 5 su susedna sa temenima u *T*, a teme 5 ima najkraću putanju do polaznog čvora 1, pa dodajemo 5 u *T*. *cost[5]* postaje 10 (Slika-8).



(a)

cost

0	1	2	3	4	5	6
6	0	5	10	∞	10	9

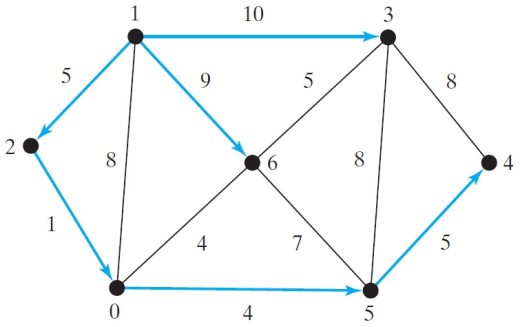
parent

0	1	2	3	4	5	6
2	–1	1	1		0	1

(b)

Slika 4.8 Temena {1, 2, 0, 6, 3, 5} su u skupu T

Sada *T* sadrži {1, 2, 0, 6, 3, 5}. Najmanji troškovi putanje koja povezuje 4 i 1 je 15 (Slika-9).



(a)

cost

0	1	2	3	4	5	6
6	0	5	10	15	10	9

parent

0	1	2	3	4	5	6
2	–1	1	1	5	0	1

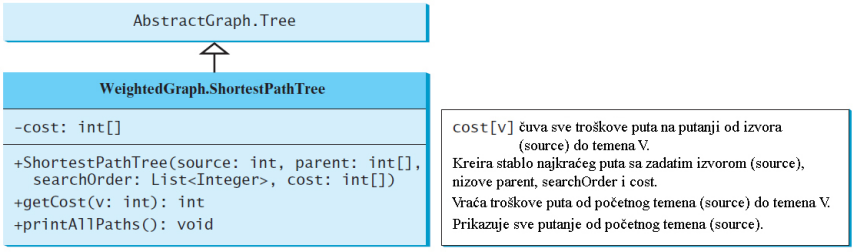
(b)

Slika 4.9 Temena {1, 2, 0, 6, 3, 5,4} su u skupu T

IMPLEMENTACIJA DIJKSTRA ALGORITMA

Čvor se dodaje u stablo T ukoliko je susedan jednom od čvorova koji su već u T i ukoliko je trošak do njega preko nekog od čvorova koji je već u T najmanji

Algoritam pronalazi najkraću putanju od izvornog čvora, što ustvari formira stablo čiji je koren u izvornom čvoru. Ovaj način određivanja najkraćeg puta se zove *stablo najkraćeg puta iz jednog izvora* (single-source all-shortestpath tree, ili jednostavno stablo najkraće putanje). Da bi modelirali ovo stablo, neophodno je definisati klasu pod nazivom *ShortestPathTree* koja proširuje klasu *Tree* kao što je prikazano na Slici-10. Klasa *ShortestPathTree* je definisana kao unutrašnja klasa unutar *WeightedGraph*.



Metod koji pronalazi najkraću putanju je *getShortestPath(int sourceVertex)*. Metod prvo dodaje izvorni čvor *sourceVertex* u skup *T*. *T* je lista koja čuva čvorove čije su putanje do izvornog čvora poznate. *vertices* je niz koji čuva sve čvorove grafa. Svakom čvoru se dodeljuju određeni troškovi.

Troškovi izvornog čvora su naravno nula. Troškovi svih ostalih čvorova se postavljaju na beskonačno veliki broj.

```
/** Find single source shortest paths */
  public ShortestPathTree getShortestPath(int sourceVertex) {
    // T stores the vertices whose path found so far
    List<Integer> T = new ArrayList<Integer>();
    // T initially contains the sourceVertex;
    T.add(sourceVertex);

    // vertices is defined in AbstractGraph
    int numberOfVertices = vertices.size();

    // parent[v] stores the previous vertex of v in the path
    int[] parent = new int[numberOfVertices];
    parent[sourceVertex] = -1; // The parent of source is set to -1

    // cost[v] stores the cost of the path from v to the source
    double[] cost = new double[numberOfVertices];
    for (int i = 0; i < cost.length; i++) {
    cost[i] = Double.MAX_VALUE; // Initial cost set to infinity
    }
    cost[sourceVertex] = 0; // Cost of source is 0
```

```
    // Get a copy of queues
    List<PriorityQueue<WeightedEdge>> queues = deepClone(this.queues);

    // Expand T
    while (T.size() < numberOfVertices) {
    int v = -1; // Vertex to be determined
    double smallestCost = Double.MAX_VALUE; // Set to infinity
    for (int u : T) {
        while (!queues.get(u).isEmpty() &amp;amp;&amp;amp;
            T.contains(queues.get(u).peek().v)) {
        queues.get(u).remove(); // Remove the vertex in queue for u
        }

        if (queues.get(u).isEmpty()) {
        // All vertices adjacent to u are in T
        continue;
        }

        WeightedEdge e = queues.get(u).peek();
        if (cost[u] + e.weight < smallestCost) {
        v = e.v;
        smallestCost = cost[u] + e.weight;
        // If v is added to the tree, u will be its parent
        parent[v] = u;
        }
    } // End of for

    T.add(v); // Add a new vertex to T
    cost[v] = smallestCost;
    } // End of while

    // Create a ShortestPathTree
    return new ShortestPathTree(sourceVertex, parent, T, cost);
  }
```

METOD GETSHORTESTPATH I KLASA SHORTESTPATHTREE

Da bi modelirali stablo najkraćeg puta, moguće je definisati klasu pod nazivom ShortestPathTree koja proširuje klasu Tree. Metod koji pronalazi najkraću putanju je getShortest

Metod *getShortestPath* mora da obriše elemente iz reda u cilju pronalaska jednog koji ima najmanje ukupne troškove. Da bi zadržali originalni red netaknut, red je kloniran. Čvor se dodaje u skup *T* ukoliko je susedan jednom od čvorova koji su već u *T* i ukoliko je trošak do njega preko nekog od čvorova koji je već u *T* najmanji. Takav čvor *s* pronalazi na osnovu sledeće procedure:

- Za čvor *u* iz *T*, pronaći njegovu susednu granu *e* koja ima najmanju težinu do *u*. Sve grane susedne čvoru *u* se čuvaju u nizu*queues.get(u)*. Metod*queues.get(u).peek()* vraća onu susednu granu najmanje težine. Ukoliko je *e.v* već u skupu *T*, obrisati *e* iz *queues.get(u)*. Zatim *queues.get(u).peek()* vraća granu *e*, takvu da *e* ima najmanju težinu do čvora *u* i važi da *e.v* nije još uvek u skupu *T* .
- Uporediti sve ove grane i pronaći onu sa najmanjom vrednošću za *cost[u] + e.getWeight()*.

Nakon što je novi čvor dodat u *T* , troškovi čvora se ažuriraju. Jednom kad su svi čvorovi dodati u *T*, kreira se instanca klase *ShortestPathTree*. Može se primetiti da metod neće raditi ako graf nije povezan. Mada, to se može naknadno korigovati.

Klasa *ShortestPathTree* proširuje klasu *Tree*. Da bi kreirali instance klase *ShortestPathTree*, prosleđuju se konstruktoru parametric *sourceVertex*, *parent*, *T*, i *cost*. Čvor *sourceVertex* postaje koren stabla. Polja *root*, *parent*, i *searchOrder* su definisana u klasi *Tree*, koja je definisana kao unutrašnja klasa od *AbstractGraph*.

```
/** ShortestPathTree is an inner class in WeightedGraph */
  public class ShortestPathTree extends Tree {
    private double[] cost; // cost[v] is the cost from v to source

    /** Construct a path */
    public ShortestPathTree(int source, int[] parent,
        List<Integer> searchOrder, double[] cost) {
    super(source, parent, searchOrder);
    this.cost = cost;
    }

    /** Return the cost for a path from the root to vertex v */
    public double getCost(int v) {
    return cost[v];
    }

    /** Print paths from all vertices to the source */
    public void printAllPaths() {
    System.out.println("All shortest paths from " +
        vertices.get(getRoot()) + " are:");
    for (int i = 0; i < cost.length; i++) {
        printPath(i); // Print a path from i to the source
        System.out.println("(cost: " + cost[i] + ")"); // Path cost
    }
    }
  }
```

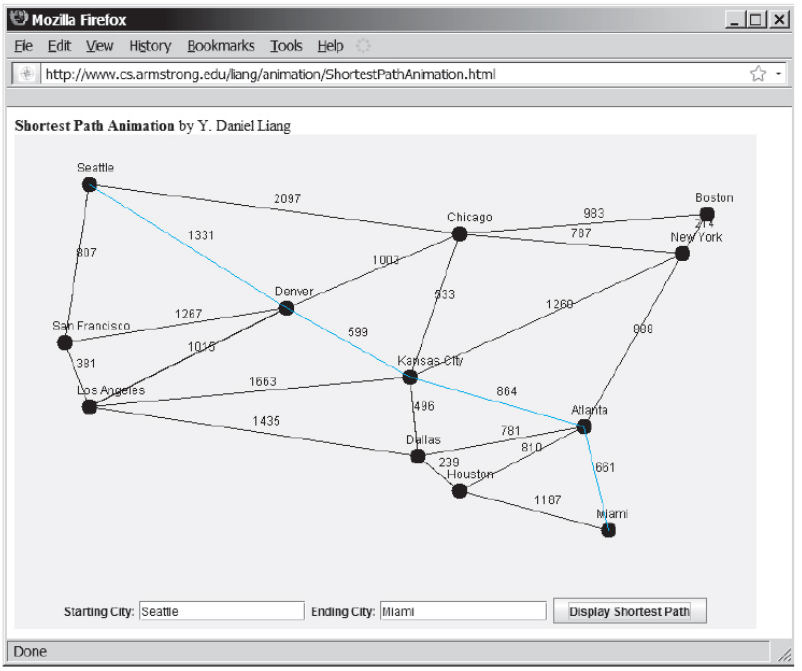
ANALIZA DIJKSTRA ALGORITMA

Dijkstra algoritam je implementiran u osnovi na isti način kao i Primov algoritam. Vremenska složenost je O(|E| log |V|),

Idite na link www.cs.armstrong.edu/liang/animation/ShortestPathAnimation.html da bi se opoznali sa interaktivnim GUI program koji služi za određivanje najkraćih putanja izmedju dva gradam, Slika-11.

Dijkstra algoritam je implementiran u osnovi na isti način kao i Primov algoritam. Stoga, vremenska složenost Dijkstra algortima je $O(|E| \log |V|)$, gde je sa $|E|$ označen broj grana a sa $|V|$ broj čvorova u grafu.

Dijkstra algoritam je kombinacija pohlepnog algortima i dinamičkog programiranja. On je pohlepni algoritam u smislu da uvek dodaje novi čvor koji ima najmanje troškove ka izvornom čvoru. On čuva najkraća rastojanja svakog poznatog čvora u odnosu na izvor i koristi te vrednosti da ne bi vršio ponovna izračunavanja, pa se stoga može zaključiti da Dijkstra algoritam koristi dinamičko progrmiranje.



Slika 4.10 Animirani alat koji prikazuje najkraća rastojanja izmedju gradova.

TESTIRANJE DIJKSTRA ALGORITMA

Cilj programa je implementacija programa u Javi koji kreira klasu za težinski graf i implementira algoritam za MST stablo (Primov algoritam) i algoritam za određivanje najkraćeg

Test program koji prikazuje najkraće puteve od grada Chicago do svih ostalih gradova sa Slike-1 je naveden u nastavku.

```
public class TestShortestPath {
  public static void main(String[] args) {
    String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
      "Denver", "Kansas City", "Chicago", "Boston", "New York",
      "Atlanta", "Miami", "Dallas", "Houston"};
```

```
    int[][] edges = {
      {0, 1, 807}, {0, 3, 1331}, {0, 5, 2097},
      {1, 0, 807}, {1, 2, 381}, {1, 3, 1267},
      {2, 1, 381}, {2, 3, 1015}, {2, 4, 1663}, {2, 10, 1435},
      {3, 0, 1331}, {3, 1, 1267}, {3, 2, 1015}, {3, 4, 599},
        {3, 5, 1003},
      {4, 2, 1663}, {4, 3, 599}, {4, 5, 533}, {4, 7, 1260},
        {4, 8, 864}, {4, 10, 496},
      {5, 0, 2097}, {5, 3, 1003}, {5, 4, 533},
        {5, 6, 983}, {5, 7, 787},
      {6, 5, 983}, {6, 7, 214},
      {7, 4, 1260}, {7, 5, 787}, {7, 6, 214}, {7, 8, 888},
      {8, 4, 864}, {8, 7, 888}, {8, 9, 661},
        {8, 10, 781}, {8, 11, 810},
      {9, 8, 661}, {9, 11, 1187},
      {10, 2, 1435}, {10, 4, 496}, {10, 8, 781}, {10, 11, 239},
      {11, 8, 810}, {11, 9, 1187}, {11, 10, 239}
    };

    WeightedGraph<String> graph1 =
      new WeightedGraph<String>(edges, vertices);
    WeightedGraph<String>.ShortestPathTree tree1 =
      graph1.getShortestPath(graph1.getIndex("Chicago"));
    tree1.printAllPaths();

    // Display shortest paths from Houston to Chicago
    System.out.print("Shortest path from Houston to Chicago: ");
    java.util.List<String> path = tree1.getPath(11);
    for (String s: path) {
      System.out.print(s + " ");
    }

    edges = new int[][]{
      {0, 1, 2}, {0, 3, 8},
      {1, 0, 2}, {1, 2, 7}, {1, 3, 3},
      {2, 1, 7}, {2, 3, 4}, {2, 4, 5},
      {3, 0, 8}, {3, 1, 3}, {3, 2, 4}, {3, 4, 6},
      {4, 2, 5}, {4, 3, 6}
    };
    WeightedGraph<Integer> graph2 =
      new WeightedGraph<Integer>(edges, 5);
    WeightedGraph<Integer>.ShortestPathTree tree2 =
      graph2.getShortestPath(3);
    tree2.printAllPaths();
  }
}
```

Program kreira težinski graf za primer sa Slike-1 u liniji 27. Zatim poziva metod getShortestPath(graph1.getIndex("Chicago")) koji kao rezultat vraća objekat tipa Path u kome se nalaze sve najkraće putanje koje od grada Chicago vode do svih ostalih gradova. Pozivom metoda printAllPaths() vrši se prikazivanje svih putanja (line 30).

Grafička ilustracija svih najkraćih putanja koje vode iz grada Chicago je prikazana na Slici-1. Najkraće putanje od grada Chicago do ostalih gradova su pronadjene u sledećem redosledu: Kansas City, New York, Boston, Denver, Dallas, Houston, Atlanta, Los Angeles, Miami, Seattle, and San Francisco.

ZADACI ZA SAMOSTALNO VEŽBANJE

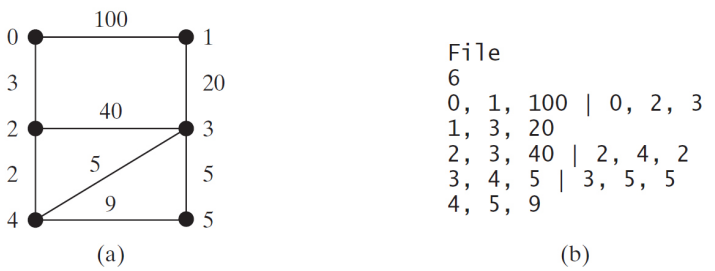
Na osnovu materijala sa predavanja i vežbi uraditi samostalno sledeće zadatke:

Zadatak 1. Prikazati korake izvršavanja Dijkstra algoritma za put od Bostona do svih ostalih gradova na **Slika-1**.

Zadatak 2. U materijalima je implementiran Dijkstra algortiam korišćenjem prioritetnog reda susednih grana. Implementirati Dijkstra algoritam u okviru klase *WeightedGraph* korišćenjem matrice susedstva za tešinski graf.

Zadatak 3. Napisati program koji učitava podatke o povezanom grafu iz fajla . U prvoj liniji fajla se nalazi broj čvorova (*n*), koje su označene indeksima *0, 1, ..., n-1*. Svaka sledeća linija predstavlja grane u obliku *u1*, *v1*, *w1* | *u2*, *v2*, *w2* | .... Svaki triplet predstavlja granu i njenu težinu.

Na Slici 1 je primer fajla za odgovarajući graf. Pretpostavka je da je graf neusmeren.U koliko graf ima granu (u,v), takođe ima i granu (v,u), a samo jedna grana je predstabvljena u fajlu.



Slika 4.11 Čvorovi i grane težinskog grafa zapisani u fajlu

Treba napisati program koji od korisnika zahteva da unese ime fajla, zatim dva čvora, a da kao rešenje prikaže najkraće rastojanje između ova dva čvora. Na primer, za graf sa slike 1, najkraći put između čvorova 0 i 1 treba biti prikazan kao 0 2 4 3 1.

U nastavku je primer prostog izlaza:

```
Enter a file name: WeightedGraphSample2.txt (Enter)
Enter two vertices (integer indexes): 0 1 (Enter)
The number of vertices is 6
Vertex 0: (0, 2, 3) (0, 1, 100)
Vertex 1: (1, 3, 20) (1, 0, 100)
Vertex 2: (2, 4, 2) (2, 3, 40) (2, 0, 3)
Vertex 3: (3, 4, 5) (3, 5, 5) (3, 1, 20) (3, 2, 40)
Vertex 4: (4, 2, 2) (4, 3, 5) (4, 5, 9)
```

```
Vertex 5: (5, 3, 5) (5, 4, 9)
A path from 0 to 1: 0 2 4 3 1
```

Zadatak 4. Izmeniti *GraphView* iz prethodne lekcije (Grafovi), tako da prikaže težinski graf i najkraći put između dva zadata grada kao što je prikazano na *Slika-11.* Treba dodati novi atribut klase *GraphView* **path**. Ako putanja nije **null**, ivice u putanji se prikazuju crvenom . Ako grad ne postoji na mapi treba ispisati poruku korisniku.

Poglavlje 5

Problem devet novčića sa težinama

UVODNA RAZMATRANJA

Problem devet novčića sa težinama može biti sveden na problem najkraće putanje u težinskom grafu

U prethodnoj lekciji Grafovi I u okviru poslednje sekcije predstavljen je Problem devet novčića koji je rešen korišćenjem BFS algoritma obilaska grafa u širinu. U nastavku će biti predstavljen jedan od oblika istog problema koji se rešava primenom algoritma za određivanje najkraćeg puta. Problem devet novčića se sastoji u određivanju najmanjeg broja obrtanja novčića koji vodi do slučaja kada su svi novčići okrenuti glavom na dole. Pri svakom potezu ovrće se glavni novčić i njegovi susedi. Problem devet novčića sa težinama dodeljuje broj obrtanja kao težinu pri svakom potezu. Na primer, moguće je izvršiti promenu iz oblika sa slike 1a u oblik sa slike 1b, obrtanjem prvog novčića u prvoj vrsti i svih njegovih suseda. Stoga, težina ove promene je 3, jer su tri novčića obrnuta.

H	H	H
T	T	T
H	H	H

(a)

T	T	H
H	T	T
H	H	H

(b)

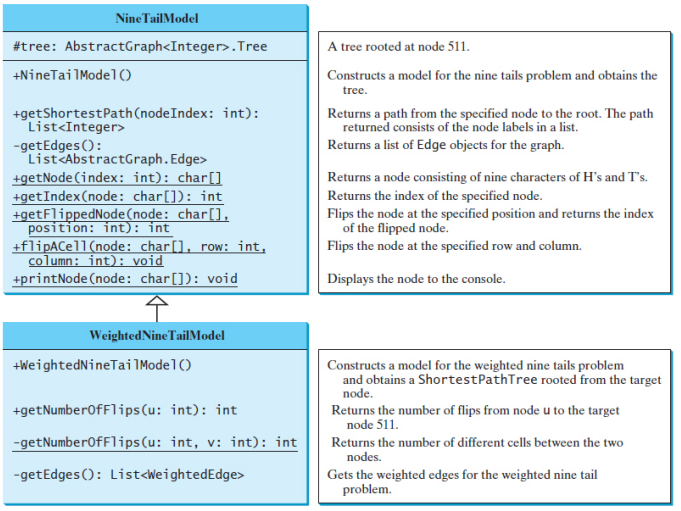
Slika 5.1 Težina svakog poteza je broj obrtanja novčića po jednom potezu.

Težinski problem 9 novčića može biti sveden na određivanje najkraćeg puta od startnog čvora do ciljnog čvora u težinskom grafu. I u ovom slučaju graf ima 512 čvorova. Kreirati ponovo granu od čvora *v* do *u* ukoliko je moguća promena iz čvora *u* u čvor *v*. Dodeliti broj obrtanja kao težinu svakoj grani.

KLASA WEIGHTEDNINETAILMODEL

Klasu WeightedNineTailModel proširuje klasu NineTailModel, koja kreira težinski graf tipa WeightedGraph i formira stablo sa najkraćom putanjom

Podsećanja radi, u prethodnoj lekciji smo definisali klasu *NineTailModel* za modeliranje problema 9 novčića. Moguće je sada definisati klasu pod nazivom *WeightedNineTailModel* koja proširuje klasu *NineTailModel*, kao što je prikazano na slici 2.



Slika 5.2 Klasa WeightedNineTailModel proširuje klasu NineTailModel.

Klasa *NineTailModel* kreira graph tipa *Graph* i formira obuhvatno stablo tipa *Tree* sa korenom u ciljnom čvoru 511. Klasa *WeightedNineTailModel* je skoro ista kao i klasa *NineTailModel* sem što kreira težinski graf *WeightedGraph* i formira stablo sa najkraćom putanjom korišćenjem metoda *ShortestPathTree* sa korenom u ciljnom čvoru 511. Klasa *WeightedNineTailModel* je kreirana tako da proširuje *NineTailModel*. Metod *getNumberOfFlips(int u, int v)* vraća broj obrtanja nočvića pri promeni iz čvora *u* u čvor *v*. Metod *getNumberOfFlips(int u)* vraća ukupan broj obrtanja od čvora *u* do ciljnog čvora.

IMPLEMENTACIJA KLASE WEIGHTEDNINETAILMODEL

Klasa WeightedNineTailModel proširuje NineTailModel i kreira težinski graf tipa WeightedGraph u cilju modeliranja Problema 9 novčića sa težinama

U narednom listingu je implementacija klase *WeightedNineTailModel*.

```
import java.util.*;

public class WeightedNineTailModel extends NineTailModel {
  /** Construct a model */
  public WeightedNineTailModel() {
    // Create edges
    List<WeightedEdge> edges = getEdges();

    // Create a graph
    WeightedGraph<Integer> graph = new WeightedGraph<Integer>(
    edges, NUMBER_OF_NODES);

    // Obtain a shortest path tree rooted at the target node
    tree = graph.getShortestPath(511);
  }
```

```
  /** Create all edges for the graph */
  private List<WeightedEdge> getEdges() {
    // Store edges
    List<WeightedEdge> edges = new ArrayList<WeightedEdge>();

    for (int u = 0; u < NUMBER_OF_NODES; u++) {
    for (int k = 0; k < 9; k++) {
        char[] node = getNode(u); // Get the node for vertex u
        if (node[k] == 'H') {
        int v = getFlippedNode(node, k);
        int numberOfFlips = getNumberOfFlips(u, v);

        // Add edge (v, u) for a legal move from node u to node v
        edges.add(new WeightedEdge(v, u, numberOfFlips));
        }
    }
    }

    return edges;
  }

  private static int getNumberOfFlips(int u, int v) {
    char[] node1 = getNode(u);
    char[] node2 = getNode(v);

    int count = 0; // Count the number of different cells
    for (int i = 0; i < node1.length; i++)
    if (node1[i] != node2[i]) count++;

    return count;
  }

  public int getNumberOfFlips(int u) {
    return (int)((WeightedGraph<Integer>.ShortestPathTree)tree)
    .getCost(u);
  }
}
```

Klasa *WeightedNineTailModel* proširuje *NineTailModel* i kreira težinski graf tipa *WeightedGraph* u cilju modeliranja Problema 9 novčića sa težinama. Za svaki čvor *u*, metod *getEdges()* pronalazi sve čvorove *v* koji se dobijaju obrtanjem čvora *u* i dodeljuje broj obrtanja kao težinu grani *(v, u)*.

Metod *getNumberOfFlips(int u)* vraća broj obrtanja pri promeni iz čvora *u* u ciljni čvor, što predstavlja troškove putanje od čvora *u* do ciljnog čvora.

WeightedNineTailModel klasa pronalazi *ShortestPathTree* sa korenom u ciljanom čvoru 511 (line 14). Stablo je definisano kao protected atribut u *NineTailModel* i *ShortestPathTree* je potklasa klase *Tree*. Metode definisane u *NineTailModel* koriste atribut **tree**.

Metod **getNumberOfFlips(int u)** (lines 49–52) vraća broj obrtanja od čvora **u** do ciljnog čvora, što predstavlja cenu puta od čvora **u** do ciljnog čvora. Ova cena se može dobiti pozivanjem **getCost(u)** metoda definisanog u klasi *ShortestPathTree* (line 51).

TESTIRANJE PROGRAMA

Program od korisnika traži da unese početni čvor a kao rezultat prikazuje minimalan broj obrtanja novčića do ciljnog čvora

U narednom listingu je program koji od korisnika traži da unese početni čvor, a kao rezultat prikazuje minimalan broj obrtanja novčića do ciljnog čvora.

```
import java.util.Scanner;

public class WeightedNineTail {
  public static void main(String[] args) {
    // Prompt the user to enter nine coins' Hs and Ts
    System.out.print("Enter an initial nine coins' Hs and Ts: ");
    Scanner input = new Scanner(System.in);
    String s = input.nextLine();
    char[] initialNode = s.toCharArray();

    WeightedNineTailModel model = new WeightedNineTailModel();
    java.util.List<Integer> path =
    model.getShortestPath(NineTailModel.getIndex(initialNode));

    System.out.println("The steps to flip the coins are ");
    for (int i = 0; i < path.size(); i++)
    NineTailModel.printNode(
        NineTailModel.getNode(path.get(i).intValue()));

    System.out.println("The number of flips is " +
    model.getNumberOfFlips(NineTailModel.getIndex(initialNode)));
  }
}
```

Mogući primer izvršavanja programa bi bio:

```
Enter an initial nine coins Hs and Ts: HHHTTTHHH (ENTER)
The steps to flip the coins are
HHH
TTT
HHH
HHH
THT
TTT
TTT
TTT
TTT
The number of flips is 8
```

Kao i kod običnog Problema 9 novčića, program zahteva od korisnika da unese podatke o početnom čvoru u obliku stringa koji se sastoji iz 9 proizvoljnih karaktera *H* ili *T*, zatim pretvara string u niz karaktera, kreira model i određuje najkraću putanju od početnog čvora do ciljnog čvora, prikazuje spisak čvorova na putanju, i prikazuje ukupan broj obrtanja novčića na putu od početnog do ciljnog čvora.

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. U lekcijama, broj obrtanja (flips) se računa kao težina poteza. Izmeniti program, ako se računa da je težina poteza tri puta veća od broja obrtanja.

Zadatak 2. Ispada da oba modela, *NineTailModel* i *WeightedNineTailModel* daju isti najkraći put. Da li je to tačno?. (Uputstvo: Neka su **tree1** i **tree2** stabla koja počinju od čvora **511** dobijenog redom iz **NineTailModel** i **WeightedNineTailModel**. Ako je dubina čvora **u** ista i u **tree1** i **tree2**, dužina puta od u do cilja je ista u oba modela)

Zadatak 3. Težinska varijanta problema 9 novčića koristi 3 x 3 matricu. Neka sada imate 16 novčića u 4 x 4 matrici. Kreirajte novi model *WeightedTailModel16*. Napravit instancu te klase i sačuvati objekat fajl.

Zadatak 4. *WeightedNineTail* predstavlja pogled za problem 9 novčića. Izmeniti program da bude za 4 x 4 16 tails problem. Program treba da učita podatke iz fajla kreiranog u prethodnom zadatku.

Zaključak

REZIME

Na osnovu svega obrađenog možemo zaključiti sledeće:

Težinski graf (engl. weighted graph) je onaj u kome svaka grana ima težinu putovanja između čvorova. Težina može predstavljati rastojanje, potrošnju energije, troškove, itd... Težinski grafovi mogu biti predstavljeni na sličan način kao bestežinski grafovi, osim što treba imati i podatak o težini grane. Jedan od načina da se predstave težinske grane je da se definišu grane kao objekti. Uvek je korisno čuvati grane koje izlaze iz čvora u nekom redu sa prioritetom da bi mogli da pristupate granama u rastućem poretku u odnosu na vrednost težine.

Minimalno obuhvatno stablo je podgraf neusmerenog težinskog grafa *G* sa osobinom: ukupna cena povezanih grana stabla je minimalna među svim mogućim obuhvatnim stablima. Osnovni algoritmi za određivanje minimalnog obuhvatnog stabla su Primov algoritam i Kruskalov algoritam.

Problem nakraćeg puta (engl. Shortest Path Problem-SP) je problem pronalaženja puta između dve tačke (ili čvora) u grafu tako da je zbir težina njegovih grana najmanji. Najpoznatiji algoritmi za određivanje najkraćeg puta su Belman-Ford, Flojd-Varšal i Dijkstra algoritam.

Dijkstra algoritam pronalazi najkraću putanju od izvornog čvora, što ustvari formira stablo čiji je koren u izvornom čvoru. Ovaj način određivanja najkraćeg puta se zove stablo najkraćeg puta iz jednog izvora - single-source all-shortestpath tree (ili jednostavno stablo najkraće putanje).

REFERENCE

Korišćena literatura

[1] D. Liang, Introduction to Java Programming, Comprehensive Version, deseto izdanje, Prentice Hall, 2014

[2] M.A. Weiss, Data Structures and Problem Solving Using Java, treće izdanje, Addison Wesley, 2005.

[3] 5. R. Sedgewick, K.Wayne, Algorithms, 4. Izdanje, Pearson Education, 2011.

[4] https://en.wikipedia.org/wiki/Kruskal%27s_algorithm

[5] http://www.geeksforgeeks.org/greedy-algorithms-set-2-kruskals-minimum-spanning-tree-mst/

[6] https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm