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

Grafovi

Lekcija 05

PRIRUČNIK ZA STUDENTE

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

Lekcija 05

GRAFOVI

- Grafovi
- Poglavlje 1: Uvod u grafove
- Poglavlje 2: Predstavljanje grafova
- Poglavlje 3: Implementacija grafova u Javi
- Poglavlje 4: Vizuelizacija grafa
- Poglavlje 5: Obilazak grafa i obuhvatno stablo
- Poglavlje 6: Obilazak grafa u dubinu
- Poglavlje 7: Obilazak grafa u širinu
- Poglavlje 8: Studija slučaja: Problem devet novčića
- 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:

- Definicijom grafova i osnovnom terminologijom u vezi sa grafovima
- Osnovnim tipovima grafova (usmereni i neusmereni grafovi, bestežinski i težinski grafovi)
- Predstavljanjem grafova korišćenjem programskih jezika.
- Predstavljanjem grana grafa korišćenjem matrice susedstva i liste susedstva
- Implementacijom grafova u programskom jeziku Java
- Obilaskom grafova u širinu i dubinu i kreiranjem odgovarajućih obuhvatnih stabala kod grafova.

Graf je veoma korisna struktura za rešavanje mnogih realnih problema. Na primer, problem određivanja najmanjeg broja letova između dva grada može biti modeliran korišćenjem grafa gde čvorovi predstavljaju gradove a grane predstavljaju letove između povezanih gradova.

Takođe i ulice mogu biti predstavljene preko grafova - u većini slučajeva ulice su dvosmerne, dok ima i onih ulica koje su jednosmerne.

Jedna od osnovnih operacija pri radu sa grafovima je obilazak grafova. Obilazak grafa ima široku primenu i mi ćemo se u nastavku, između ostalog, baviti i ovom temom.

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

Poglavlje 1

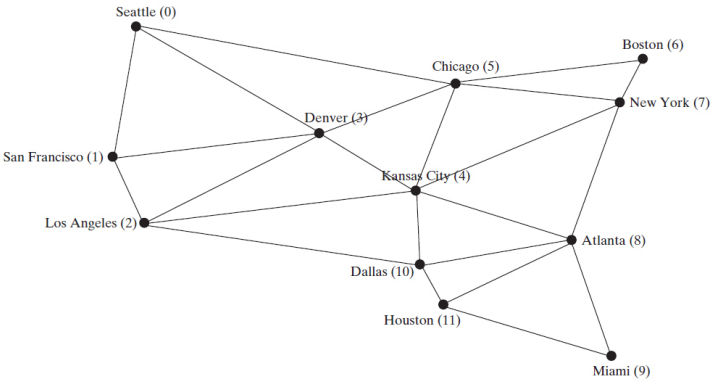
Uvod u grafove

DEFINICIJA GRAFA

Problem određivanja najmanjeg broja letova između dva grada može biti modeliran korišćenjem grafa gde čvorovi predstavljaju gradove a grane predstavljaju letove između povezanih gradova

Graf je veoma korisna struktura za rešavanje mnogih realnih problema. Na primer, problem određivanja najmanjeg broja letova između dva grada može biti modeliran korišćenjem grafa gde čvorovi predstavljaju gradove a grane predstavljaju letove između povezanih gradova, Slika-1.

Problem određivanja najmanjeg broja povezujućih letova između dva grada se svodi na određivanje najkraće putanje između dva čvora u grafu.



Slika 1.1 Graf može biti korišćen da se predstavi mapa letova između skupa gradova.

Graf sa Slike-1 predstavlja letove između gradova.

Kao što je poznato, graf se sastoji iz nepraznog skupa čvorova (temena ili tačaka), i skupa grana (ili ivica - eng. edge) koje povezuju ove čvorove (temena).

Radi pogodnosti, definisaćemo graf kao $G = (V, E)$, gde V predstavlja skup temena (eng. vertices) a E predstavlja skup grana. Na primer, skupovi E i V za primer sa Slike-3, mogu biti predstavljeni na sledeći način:

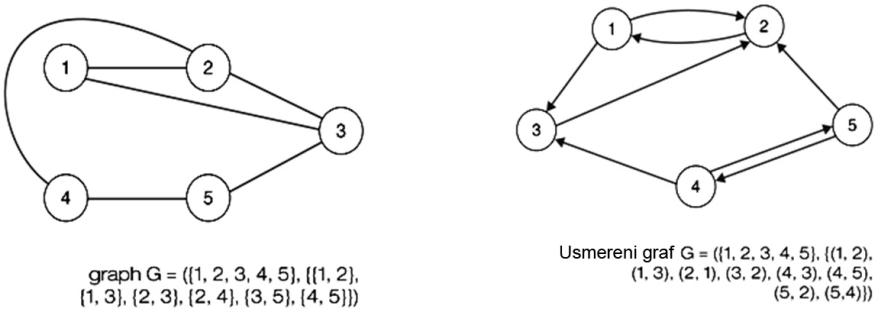
```
V = {"Seattle", "San Francisco", "Los Angeles",
  "Denver", "Kansas City", "Chicago", "Boston", "New York",
  "Atlanta", "Miami", "Dallas", "Houston"};

E = {{"Seattle", "San Francisco"},{"Seattle", "Chicago"},
```

```
  {"Seattle", "Denver"}, {"San Francisco", "Denver"},
  ...
};
```

OSNOVNI TIPOVI GRAFOVA

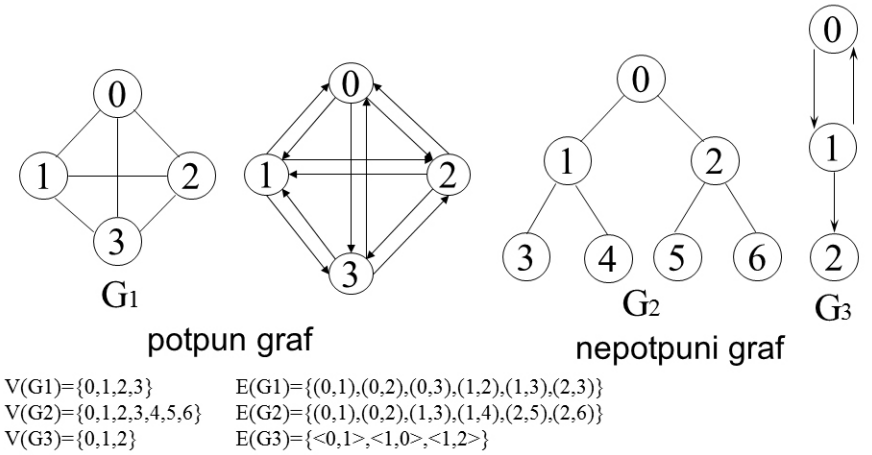
Graf može biti usmeren ili neusmeren. U usmerenom (eng. directed) grafu, svaka grana ima smer koji određuje da li je moguće pomeriti se kroz tu granu od jednog do drugog čvora

Graf može biti **usmeren** ili **neusmeren**. Grane usmerenog grafa dozvoljavaju obilazak u jednom smeru. Grane neusmerenog grafa dozvoljavaju obilazak u oba smera. Slika-2 ilustruje neusmereni i usmereni graf.



Slika 1.2 Graf može biti neusmeren (a) ili usmeren (b).

Potpuni graf je onaj koji ima granu između svakog para čvorova, Slika-3. Potpuni neusmeri graf ima *n(n-1)/2* grana, a potpuni usmeri graf ima *n(n-1)* grana.

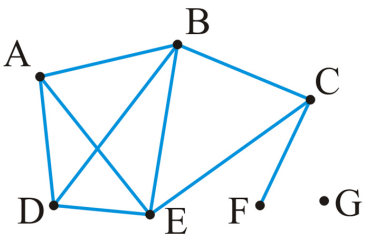


Slika 1.3 Potpuni i nepotpuni graf

NEUSMERENI GRAFOVI

Neusmereni graf je graf u kojem grane nemaju orijentaciju. ADT neusmerenog grafa se definiše kao skup čvorova V = {v1, v2, ..., vn}, kome je pridružen skup parova {vi, vj} , tj. grana

Neusmereni graf (engl. undirected graph) je onaj u kojoj ivice nemaju orijentaciju. Grana *(a,b)* je identična sa granom *(b,a)*, odnosno nisu uređeni parovi, ali su skupovi *{U, V}* čvorova. Maksimalan broj grana u neusmerenom grafu bez samostalne petlje je *(n - 1) / 2*. Slika-4 prikazuje neusmereni graf.



Slika 1.4 Neusmereni graf

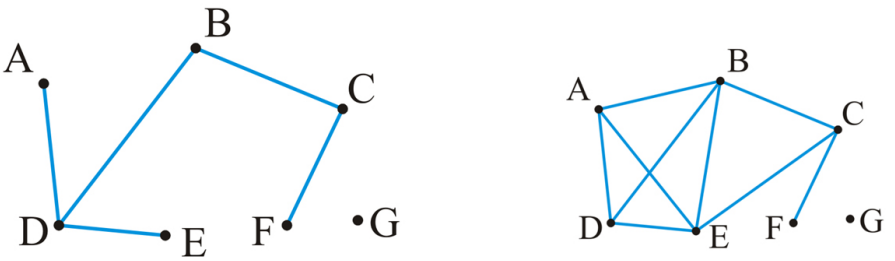
Stepen neusmerenog grafa

Stepen čvora je definisan kao broj susednih čvorova, Slika-6.

- stepen(A) = stepen(D) = stepen(C) = 3
- stepen(B) = stepen(E) = 4
- stepen(F) = 1
- stepen(G) = 0

Podgraf neusmerenog grafa

Podgraf je graf podskupa čvorova i podskupa grana koji povezuju podskup čvorova u originalnom grafu, Slika-5.



Slika 1.5 Podgrafovi

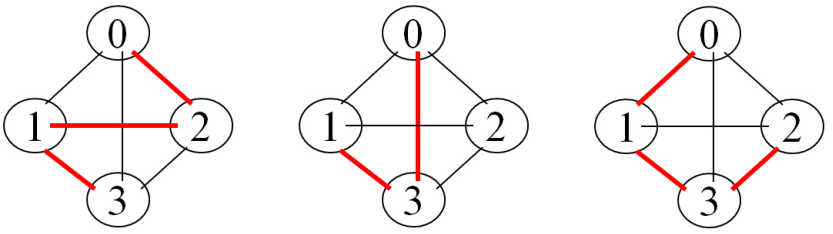
PUTANJE I POVEZANOST NEUSMERENOG GRAFA

Put od čvora vp do čvora vq u grafu G je sekvenca čvorova vp, vi1, vi2, ..., vin, vq, tako da su (vp, vi1), (vi1, vi2), ..., (vin, vq) grane neusmerenog grafa

Putanje u grafu

Put od čvora v_p do čvora v_q u grafu G je sekvenca čvorova v_p, v_{i1}, v_{i2}, ..., v_{in}, v_q, tako da su (v_p, v_{i1}), (v_{i1}, v_{i2}), ..., (v_{in}, v_q) grane neusmerenog grafa. Primeri putanja su (0, 2), (2, 1), (1, 3), i ove putanje se često pišu kao 0, 2, 1, 3.

Dužina putanje je broj grana na tom putu.



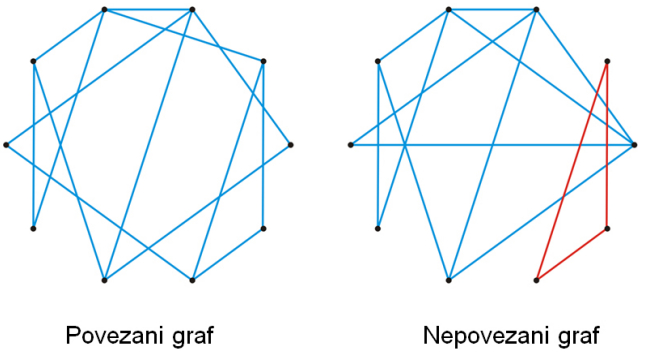
Slika 1.6 Putanje neusmerenih grafova

Povezanost grafova

Kaže se da su dva temena v_i, v_j povezana ako postoji put od v_i, do v_j.

Graf je povezan ukoliko postoji put između bilo koja dva čvora u grafu.

Slika-7 prikazuje povezani i nepovezani graf.



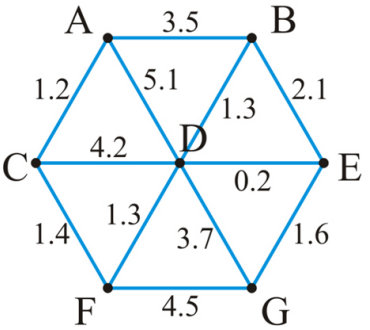
Slika 1.7 Povezani i nepovezani grafovi.

TEŽINSKI NEUSMERENI GRAF

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 graf (engl. weighted graph) je onaj u kome svaka grana ima težinu (engl. cost) putovanja između čvorova.

Težinski graf je triplet (V, E, W) gde je (V, E) graf (usmeren ili neusmeren), W funcija iz E u R, realna (integer or rationals). Za granu e, W(e) se zove težina za e. Težina može biti povezana sa svakom granom grafu, može predstavljati rastojanje, potrošnju energije, troškove, itd. Takav graf se naziva težinski graf.



Slika 1.8 Primer težinskog grafa

Dužina putanje unutar težinskog grafa je zbir svih grana koje čine putanju.

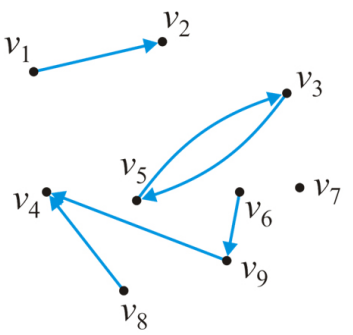
USMERENI GRAFOVI

U usmerenom grafu (directed graphs), grane grafa imaju smer. Grane su uređeni parovi (vj, vk) koje označavaju povezanost od vj do vk, pri čemu se grana (vj, vk) razlikuje od grane (vk,vj)

U usmerenom grafu (engl. directed graphs), grane grafa imaju smer. Grane su uređeni parovi (v_j,v_k) koje označavaju povezanost od v_j do v_k. Grana (v_j,v_k) se razlikuje od grane (v_k,v_j).

Ulice su usmereni grafovi: u većini slučajeva, ulice su dvosmerne, samo neke su jednosmerne.

Dat je graf sa 9 temena V = {v1, v2, ...v9}. Ovih 6 parova (v_j,v_k) su usmerene grane E = {(v1, v2), (v3, v5), (v5, v3), (v6, v9), (v8, v4), (v9, v4)}, Slika-9.



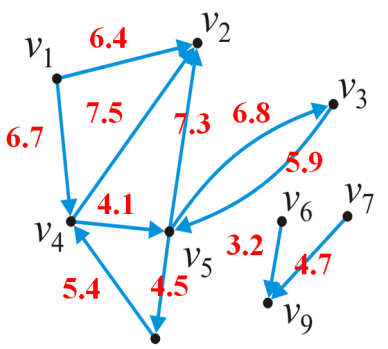
Slika 1.9 Usmereni graf

Maksimalni broj usmerenih grana u usmerenom grafu je dat kao:

$$|E| \leq 2\binom{|V|}{2} = 2\frac{|V|(|V|-1)}{2} = |V|(|V|-1) = O(|V|^2)$$

Kod težinskih usmerenih grafova (engl. weighted directed graphs), svakoj grani je pridružena vrednost.

Za razliku od težinski neusmerenih grafova, ako su (v_j, v_k) i (v_j, v_k) grane, ne zahteva se da imaju istu težinu (engl. weight), Slika-10.



ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

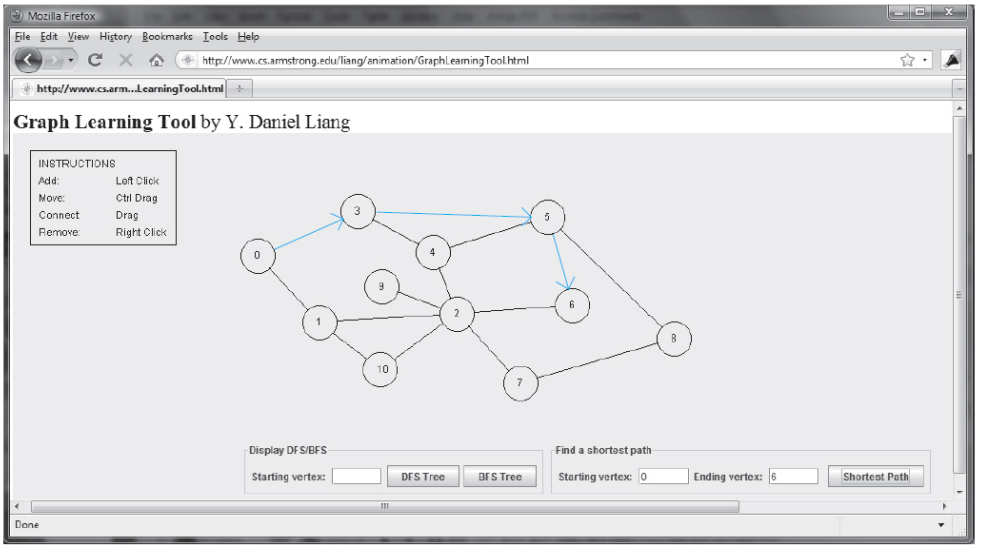
Zadatak 1. Šta je graf? Objasniti sledeće termine: neusmereni graf, usmereni graf, težinski graf, stepen čvora, paralelna ivica, jednostavan graf, kompletan graf, povezan graf, ciklus, podgraf, stablo i obuhvatno stablo.

Zadatak 2. Koliko ima ivica kompletan graf sa 5 čvorova? Koliko ivica ima stablo sa 5 čvorova?

Zadatak 3. Koliko ivica ima kompletan graf sa n čvorova? Koliko ivica ima stablo sa n čvorova?

Sa grafovima se možete upoznati pomoću interaktivne aplikacije na sledećem linku: www.cs.armstrong.edu/liang/animation/GraphLearningTool.html

Izgled aplikacije je prikazan na slici **Slika-11**. Alat omogućava dodavanje, brisanje i pomeranje ivica kao i dodavanje čvorova klikom miša. Takođe, mogu se pronađi stabla pretrage u širinu i stabla pretrage u dubinu, kao i najkraći put između dva zadata čvora.



Slika 1.10 Pomoću miša možete kreirati graf i prikazati DFS/BFS stabla, kao i najlraće putanje.

Poglavlje 2

Predstavljanje grafova

PREDSTAVLJANJE ČVOROVA GRAFA

Čvorovi grafa mogu biti predstavljeni korišćenjem niza ili povezane liste. Takođe, čvorovi mogu biti objekti bilo kog tipa podatka

Čvorovi grafa mogu biti predstavljeni korišćenjem niza ili povezane liste. Na primer, sve nazive gradova sa Slike-3 sekcije *Uvod u grafove*, možemo smestiti u sledeći niz:

```
String[] vertices = {"Seattle", "San Francisco", "Los Angeles",
  "Denver", "Kansas City", "Chicago", "Boston", "New York",
  "Atlanta", "Miami", "Dallas", "Houston"};
```

Čvorovi mogu biti objekti bilo kog tipa podatka. Pretpostavimo da su gradovi objekti koji sadrže informacije kao što su ime, broj stanovnika, i ima gradonačelnika. Stoga je čvor grafa moguće definisati na sledeći način:

```
City city0 = new City("Seattle", 608660, "Mike McGinn");
...
City city11 = new City("Houston", 2099451, "Annise Parker");
City[] vertices = {city0, city1, ... , city11};
```

gde je klasa *City* definisana kao:

```
public class City {
 private String cityName;
 private int population;
 private String mayor;
 public City(String cityName, int population, String mayor) {
  this.cityName = cityName;
  this.population = population;
  this.mayor = mayor;
 }
 …
}
```

Čvorovi grafa mogu najjednostavnije biti označeni indeksima 0, 1, 2, . . . , *n* - 1, za graf koji ima N čvorova. Stoga će *vertices[0]* predstavljati grad *"Seattle"*, *vertices[1]* će predstavljati *"San Francisco"*, i tako dalje, kao što je prikazano na Slici-1.

vertices[0]	Seattle
vertices[1]	San Francisco
vertices[2]	Los Angeles
vertices[3]	Denver
vertices[4]	Kansas City
vertices[5]	Chicago
vertices[6]	Boston
vertices[7]	New York
vertices[8]	Atlanta
vertices[9]	Miami
vertices[10]	Dallas
vertices[11]	Houston

Slika 2.1.1 Niz čuva nazive čvorova.

PREDSTAVLJANJE GRANA GRAFA

Grane grafa mogu biti predstavljene korišćenjem: niza grana, objekata specijalno kreiranih za grane, dok su najuobičajeni načini predstavljanja: matrica susedstva i lista susedstva

Predstavljanje grana: Niz grana

Grane grafa koji je baziran na nizu (engl. edge array) se mogu predstaviti korišćenjem dvodimenzionalnog niza.

Na primer, sve grane grafa sa Slike-3 sekcije *Uvod u grafove* možemo predstaviti korišćenjem sledećeg 2D niza:

```
int[][] edges = {
    {0, 1}, {0, 3}, {0, 5},
    {1, 0}, {1, 2}, {1, 3},
    {2, 1}, {2, 3}, {2, 4}, {2, 10},
    {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
    {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
    {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
    {6, 5}, {6, 7},
    {7, 4}, {7, 5}, {7, 6}, {7, 8},
    {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
    {9, 8}, {9, 11},
    {10, 2}, {10, 4}, {10, 8}, {10, 11},
    {11, 8}, {11, 9}, {11, 10}
    };
```

Ovo predstavljanje grana grafa se naziva *niz grana*.

Predstavljanje grana korišćenjem objekta klase: Edge

Još jedan od načina da se predstave grane grafa je da se definišu grane kao objekti i da se niz grana smesti u niz tipa *java.util.ArrayList*.

Klasa *Edge* za granu grafa može biti definisana kao:

```
public static class Edge {
    public int u; // Starting vertex of the edge
    public int v; // Ending vertex of the edge
    public Edge(int u, int v) {
    this.u = u;
    this.v = v;
    }
  }
```

Na primer, moguće je smestiti grane grafa sa Slike-3 sekcije *Uvod u grafove* korišćenjem liste na sledeći način:

```
java.util.ArrayList<Edge> list = new java.util.ArrayList<Edge>();
list.add(new Edge(0, 1));
list.add(new Edge(0, 3));
list.add(new Edge(0, 5));
...
```

Čuvanje objekata *Edge* u listi tipa *ArrayList* je korisno ako se ne zna unapred broj grana u grafu. Iako je predstavljanje grana grafa korišćenjem niza grana ili niza objekata lako i intuitivno, ono nije efikasno za interno procesiranje.

U nastavku će biti opisano predstavljanje grafa korišćenjem *matrice susedstva* ili *povezane liste susedstva*. Ove dve strukture podataka su veoma efikasne za rad sa grafovima.

2.1 Matrica susedstva

UVOD U MATRICU SUSEDSTVA

Graf može biti predstavljen i kvadratnom matricom čiji je red jednak broju čvorova grafa. Element aij jednak je broju grana koje polaze iz čvora vi, a završavaju se u čvoru vj

Graf može biti predstavljen i kvadratnom matricom čiji je red jednak broju čvorova grafa. Element a_{ij} jednak je broju grana koje polaze iz čvora v_i, a završavaju se u čvoru v_j.

Ova matrica se naziva *matrica susedstva grafa* i obeležava se sa A.

Osnovne karakteristike matrice susedstva:

- Kvadratna matrica koja se sastoji iz logičkih ili binarnih vrednosti (***true*** ili ***false***, *0* ili *1*).
- Ukoliko se graf sastoji iz *N* cvorova, onda će matrica imati N vrsta i N kolona.

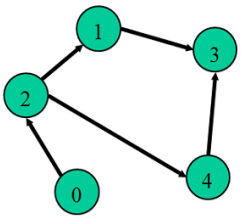
- Za dva čvora oznacena sa *I* i *J*, element u matrici susedva koji se nalazi u *I*-toj vrsti i *J*-toj koloni će biti ***true*** (1) ukoliko postoji veza između *I* i *J*, a u suprotnom će biti ***false*** (0).

Za neusmereni graf, matrica će biti simetrična duž dijagonale. Za težinske grafove, matrica susedstva ima težinu grana grafa duž dijagonale i beskonačnost (∞) na svako drugoj poziciji.

Matrica susedstva može biti implementirana korišćenjem 2D niza:

```
boolean[][] adjMatrix;
adjMatrix = new boolean[5][5];
```

Na Slici-1 je prikazan primer grafa i njegove matrice susedstva.



	0	1	2	3	4
0	false	false	true	false	false
1	false	false	false	true	false
2	false	true	false	false	true
3	false	false	false	false	false
4	false	false	false	true	false

Slika 2.2.1 Primer grafa predstavljenog pomoću matrice susedstva

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

PRIMERI IMPLEMENTACIJE MATRICE SUSEDSTVA

Za dva čvora označena sa I i J, element u matrici susedstva koji se nalazi u I-toj vrsti i J-toj koloni će biti true (1) ukoliko postoji veza između I i J, a u suprotnom će biti false (0)

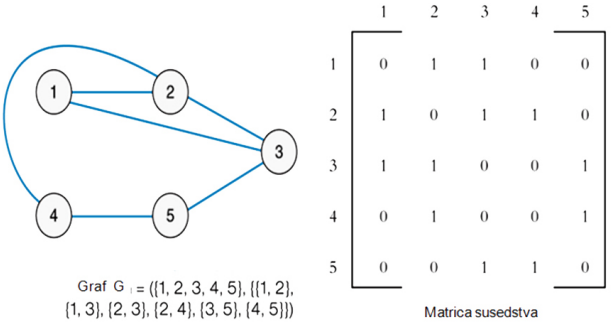
Pretpostavimo da graf ima *N* čvorova. Moguće je koristiti 2D matricu dimenzija NxN, pod nazivom *adjacencyMatrix*, da se predstave grane grafa. Svaki element u 2D matrici ima vrednost 0 ili 1. Vrednost člana *adjacencyMatrix[i][j]* je *1*ako postoji grana između čvorova *(i,j)*. U suprotnom, vrednosti člana *adjacencyMatrix[i][j]* je *0*. Ukoliko graf nije usmeren, matrica je simetrična, pošto je *adjacencyMatrix[i][j]* isto kao i *adjacencyMatrix[j][i]*.

Na primer, grane grafa sa Slike-3 sekcije *Uvod u grafove* mogu korišćenjem matrice susedstva biti predstavljene na sledeći način:

```
int[][] adjacencyMatrix = {
 {0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0}, // Seattle
 {1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0}, // San Francisco
 {0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0}, // Los Angeles
 {1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0}, // Denver
 {0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0}, // Kansas City
 {1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0}, // Chicago
 {0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0}, // Boston
 {0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0}, // New York
```

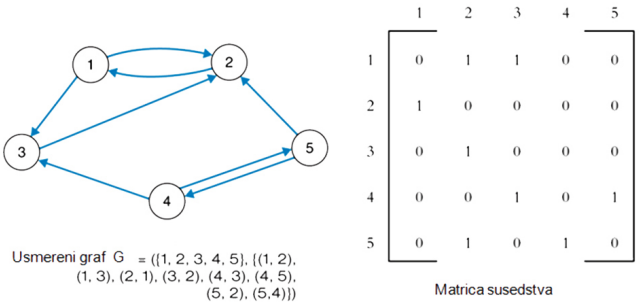
```
 {0, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 1}, // Atlanta
 {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1}, // Miami
 {0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1}, // Dallas
 {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0}  // Houston
};
```

Matrica susedstva se razlikuje za usmerene i neusmerene grafove. Na Slici 2 je mastrica susedstva za neusmereni graf.



Slika 2.2.2 Primer matrice susedstva – neusmereni graf

Na Slici-3 je primer matrice susedstva za usmereni graf.



Slika 2.2.3 Primer matrice susedstva – usmereni graf

2.2 Lista susedstva

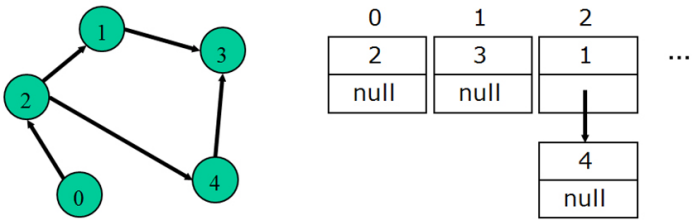
KARAKTERISTIKE LISTE SUSEDSTVA

Za svaki čvor grafa G = (V , E), lista susedstva sadrži sve čvorove koji su susedni s njim u G. Za svaki čvor grafa se čuva pokazivač na povezanu listu u kojoj se nalaze njegovi susedi

Za svaki čvor grafa *G = (V , E)*, lista susedstva sadrži sve čvorove koji su susedni s njim u *G*.

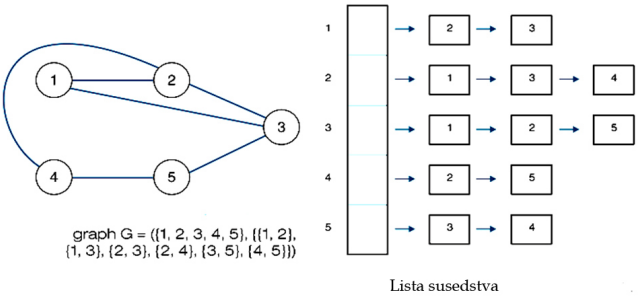
Lista pokazivača je data za svaki čvor grafa. Ovi pokazivači su počeci povezane liste (engl. linked list) čvorova do kojih mogu doći grane grafa. Za težinski dijagram, ova lista takođe uključuje težinu za svaku granu.

Na Slici-1 je primer implementacije grafa korišćenjem liste susedstva.



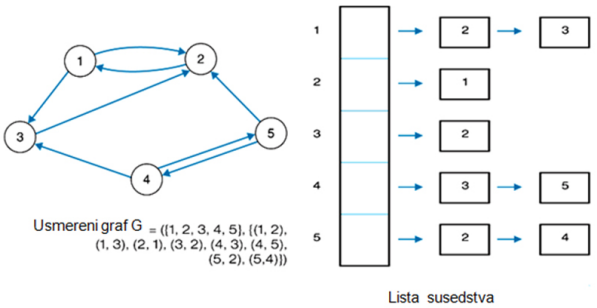
Slika 2.3.1 . Predstavljanje grafa korišćenjem liste povezanosti za svaki čvor

Lista susedstva se, kao i matrica susedstva, razlikuje za usmerene i neusmerene grafove. Na Slici-2 je prikazana matrica susedstva za neusmereni graf.



Slika 2.3.2 Primer liste susedstva za neusmereni graf

Na Slici-3 je primer liste susedstva za usmereni graf.



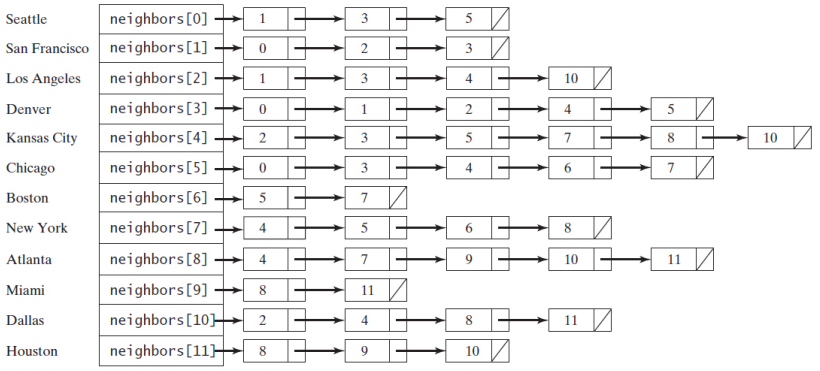
Slika 2.3.3 . Primer liste susedstva za usmereni graf

IMPLEMENTACIJA LISTE SUSEDSTVA

Da bi predstavili grane korišćenjem liste susedstva neophodno je kreirati niz povezanih listi. Povezana lista za čvor i niza sadrži sve čvorove j tako da postoji grana između čvorova (i,j)

Da bi predstavili grane korišćenjem liste susedstva neophodno je kreirati niz povezanih listi. Niz će imati *n* ulaza, a svaki element niza je povezana lista. Povezana lista za čvor *i* sadrži sve čvorove *j* tako da postoji grana između čvorova *(i,j)*.

Na primer, da bi predstavili grane u grafu sa Slike-3 sekcije *Uvod u grafove*, moguće je kreirati niz povezanih listi na sledeći način, Slika-4:



Slika 2.3.4 Grane u grafu su predstavljene korišćenjem povezanih listi.

Obe, matrica susedstva i lista susedstva, mogu biti korišćene u programu u cilju postizanja efikasnijeg algoritma.

Na primer, nephodno je *O(1)* konstantno vreme da se proveri da li su dva čvora povezana korišćenjem martrice susedstva, a neophodno je linearno vreme *O(m)* da se oštampaju sve grane u grafu korišćenjem liste susedstva gde je *m* broj grana.

Zbog fleksibilnosti i uprošćenja, u nastavku ćemo koristiti listu baziranu na nizu (array lists) da predstavimo nizove. Takođe, koristićemo liste bazirane na nizovima umesto povezanih listi, jer naši algoritmi jedino zahtevaju pretraživanje susednih čvorova u listi. Korišćenje nizova je mnogo efikasnije za algoritme koji će biti predstavljeni u nastavku lekcije.

Korišćenjem liste bazirane na nizu, lista povezanosti grafa sa Slike-3 sekcije *Uvod u grafove* može biti kreirana na sledeći način:

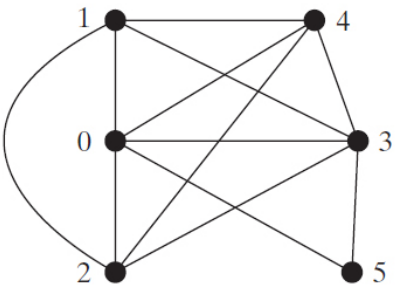
```
List<ArrayList<Integer>> neighbors = new ArrayList<List<Integer>>();
neighbors.add(new ArrayList<Integer>());

neighbors.get(0).add(1); neighbors.get(0).add(3);
neighbors.get(0).add(5);
neighbors.add(new ArrayList<Integer>());
neighbors.get(1).add(0); neighbors.get(1).add(2);
neighbors.get(1).add(3);
...
...
```

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Kako se predstavljaju čvorovi grafa? Kako se predstavljaju ivice grafa pomoću niza ivica? Kako se predstavlja ivica koristeći objekat ivice? Kako se predstavljaju ivice grafa pomoću matrice susedstva? Kako se predstavljaju ivice pomoću liste susedstva?

Zadatak 2. Predstaviti sledeći graf pomoću niza ivica, liste objekata ivica, matrice susedstva i liste susedstva.



Slika 2.3.5 Zadatak za samostalni rad

Poglavlje 3

Implementacija grafova u Javi

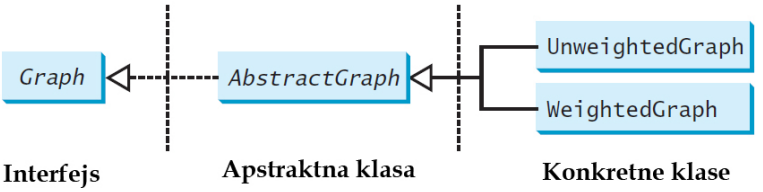
HIJERARHIJA KLASA I INTERFEJSA

Po ugledu na Okvir kolekcija u Javi (JCF) moguće je kreirati hijerarhiju klasa i interfejsa za rad sa grafovima

Okvir kolekcija u Javi (Java Collections Framework) je najbolji primer kreiranja složenih struktura podataka.

Osnovni elementi i mogućnosti svake strukture podatka su definisane u interfejsima (*Collection*, *Set*, *List*, *Queue*). Imamo apstraktne klase koje delimično implementiraju te interfejse (tj. *AbstractCollection*, *AbstractSet*, *AbstractList*) i konkretne klase koje obezbeđuju konkretnu implementaciju (tj, *HashSet*, *LinkedHashSet*, *TreeSet*, *ArrayList*, *LinkedList*, *PriorityQueue*).

Na sličan način je moguće kreirati hijerarhiju za rad sa grafovima. Definisaćemo interfejs pod nazivom *Graph* koji se sastoji iz osnovnih operacija za rad sa grafovima i apstraktnu klasu *AbstractGraph* koja delimično implementira interfejs *Graph*. Zatim ćemo kreirati konkretne grafove *UnweightedGraph* i *WeightedGraph* za portrebe težinskih i bestežinskih grafova (Slika-1).



Slika 3.1 Modeliranje grafa korišćenjem interfejsa, abstraktnih klasa i konkretnih klasa

Koje su osnovne operacija za rad sa grafovima?

U osnovni, neophodno je da sa nađe broj čvorova u grafu, da se pristupi svakom čvoru grafa, da se pristupi čvoru sa određenim indeksom, itd. Takođe je neophodno kreirati metode za obilazak grafa po dubini i širini.

METODE ZA RAD SA GRAFOVIMA

Klasa AbstractGraph delimično implementira interfejs Graph. Klase UnweightedGraph i WeightedGraph implementiraju interfejs Graph za portrebe težinskih i bestežinskih grafova

Na Slici-2 je UML dijagram metoda za rad sa grafovima.

Klasa *AbstractGraph* ne uvodi nijedan novi metod u odnosu na interfejs *Graph*. Lista čvorova i lista susedstva se definišu u okviru klase *AbstractGraph*. Korišćenjem ovih polja moguće je implementirati sve metode interfejsa *Graph*.

Klasa *UnweightedGraph* jednostavno proširuje klasu *AbstractGraph* sa pet konstruktora za kreiranje konkretnog grafa, tj instance tipa *Graph*. Klasa *UnweightedGraph* nasleđuje sve metoda apstraktne klase *AbstractGraph*, ali ne uvodi nove metode niti predefiniše bilo koji postojeći metod.

Slika 3.2 Izgled Graph interfejsa i osnovnih operacija za rad sa grafovima

TESTIRANJE KREIRANOG GRAFA

Cilj primera je testiranje implementiranih klasa i interfejsa u Javi za grafove: Graph, AbstractGraph i UnweightedGraph

Za testiranje opcija kreiranog grafa možemo da iskoristimo sledeću probnu klasu:

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

```
    // Edge array for graph in Figure 27.1
    int[][] edges = {
    {0, 1}, {0, 3}, {0, 5},
    {1, 0}, {1, 2}, {1, 3},
    {2, 1}, {2, 3}, {2, 4}, {2, 10},
    {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
    {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
    {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
    {6, 5}, {6, 7},
    {7, 4}, {7, 5}, {7, 6}, {7, 8},
    {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
    {9, 8}, {9, 11},
    {10, 2}, {10, 4}, {10, 8}, {10, 11},
    {11, 8}, {11, 9}, {11, 10}
    };

    Graph<String> graph1 =
    new UnweightedGraph<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.printEdges();

    // List of Edge objects for graph in Figure 27.3(a)
    String[] names = {"Peter", "Jane", "Mark", "Cindy", "Wendy"};
    java.util.ArrayList<AbstractGraph.Edge> edgeList
    = new java.util.ArrayList<AbstractGraph.Edge>();
    edgeList.add(new AbstractGraph.Edge(0, 2));
    edgeList.add(new AbstractGraph.Edge(1, 2));
    edgeList.add(new AbstractGraph.Edge(2, 4));
    edgeList.add(new AbstractGraph.Edge(3, 4));
    // Create a graph with 5 vertices
    Graph<String> graph2 = new UnweightedGraph<String>
    (edgeList, java.util.Arrays.asList(names));
    System.out.println("\nThe number of vertices in graph2: "
    + graph2.getSize());
    System.out.println("The edges for graph2:");
    graph2.printEdges();
  }
}
```

Rezultat programa biće:

```
The number of vertices in graph1: 12
The vertex with index 1 is San Francisco
The index for Miami is 9
The edges for graph1:
Seattle (0): (0, 1) (0, 3) (0, 5)
San Francisco (1): (1, 0) (1, 2) (1, 3)
```

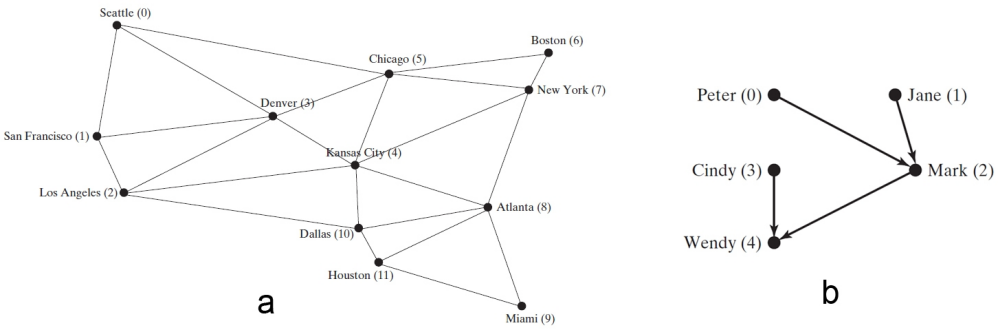
```
Los Angeles (2): (2, 1) (2, 3) (2, 4) (2, 10)
Denver (3): (3, 0) (3, 1) (3, 2) (3, 4) (3, 5)
Kansas City (4): (4, 2) (4, 3) (4, 5) (4, 7) (4, 8) (4, 10)
Chicago (5): (5, 0) (5, 3) (5, 4) (5, 6) (5, 7)
Boston (6): (6, 5) (6, 7)
New York (7): (7, 4) (7, 5) (7, 6) (7, 8)
Atlanta (8): (8, 4) (8, 7) (8, 9) (8, 10) (8, 11)
Miami (9): (9, 8) (9, 11)
Dallas (10): (10, 2) (10, 4) (10, 8) (10, 11)
Houston (11): (11, 8) (11, 9) (11, 10)
The number of vertices in graph2: 5
The edges for graph2:
Peter (0): (0, 2)
Jane (1): (1, 2)
Mark (2): (2, 4)
Cindy (3): (3, 4)
Wendy (4):
```

OPIS PROGRAMA ZA TESTIRANJE GRAFOVA

Prethodni program kreira grafove graph1 i graph2 za primere sa Slike-3

Program kreira graph1 za graf sa slike 3a, u linijama 3–24. Temena grafa graph1 definisana u linijama 3–5, dok su grane definisane u linijama 8–21. Grane su predstavljene korišćenjem dvodimenzionalnog niza. Za svaku vrstu i niza, edges[i][0] i edges[i][1] označavaju da postoji grana koja ide od temena edges[i][0] do temena edges[i][1]. Na primer, prva vrsta , {0, 1}, predstavlja gradu od temena vertex 0 (edges[0][0]) do temena 1 (edges[0][1]). Vrsta {0, 5} predstavlja granu od temena 0 (edges[2][0]) do temena 5 (edges[2][1]). Graf je kreiran u liniji. U liniji 32 se poziva metod printEdges() koji za graf graph1 prikazuje sve njegove grane.

Program kreira graph2 za graf sa Slike 3b u linijama 35–44. Grane grafa graph2 su definisane u linijama 38–41. graph2 je kreiran korišćenjem liste objekata tipa Edge u liniji 44. U liniji 48 se poziva metod printEdges() koji za graf graph2 prikazuje sve njegove grane..



Slika 3.3 Primeri kreiranih grafova

Napomenimo da oba grafa graph1 i graph2 sadrže temena tipa string. Temena su asocirana sa indeksima 0, 1, . . . , n-1. Indeks predstavlja poziciju temena u nizu vertices. Na primer, indeks temena Miami je 9.

INTERFEJS GRAPH

Interfejs Graph definiše skup osnovnih operacija za rad sa grafovima. Klasa AbstractGraph koristi liste bazirane na nizovima umesto povezanih listi za kreiranje liste susedstva

Sada ćemo da se fokusiramo na kreiranja odgovoarajućih interfejsa i klasa. Predstavićemo redom interfejs Graph, klasu AbstractGraph, i klasu UnweightedGraph.

Interfejs *Graph* može biti definisan na sledeći način:

```
public interface Graph<V> {
  /** Return the number of vertices in the graph */
  public int getSize();

  /** Return the vertices in the graph */
  public java.util.List<V> getVertices();

  /** Return the object for the specified vertex index */
  public V getVertex(int index);

  /** Return the index for the specified vertex object */
  public int getIndex(V v);

  /** Return the neighbors of vertex with the specified index */
  public java.util.List<Integer> getNeighbors(int index);

  /** Return the degree for a specified vertex */
  public int getDegree(int v);

  /** Print the edges */
  public void printEdges();

  /** Clear the graph */
  public void clear();

  /** Add a vertex to the graph */
  public boolean addVertex(V vertex);

  /** Add an edge (u, v) to the graph */
  public boolean addEdge(int u, int v);

  /** Add an edge to the graph */
  public boolean addEdge(Edge e);

  /** Remove a vertex v from the graph */
```

```
  public default boolean remove(V v) {
    return true; // Implementation left as an exercise
  }

  /** Remove an edge (u, v) from the graph */
  public default boolean remove(int u, int v) {
    return true; // Implementation left as an exercise
  }

  /** Obtain a depth-first search tree */
  public UnweightedGraph<V>.SearchTree dfs(int v);

  /** Obtain a breadth-first search tree */
  public UnweightedGraph<V>.SearchTree bfs(int v);
}
```

IMPLEMENTACIJA KLASE ABSTRACTGRAPH

Klasa AbstractGraph implementira interfejs Graph i koristi listu susedstva za označavanje grana u grafu

Implementacija apstraktne klase *AbstractGraph* u Javi se može videti u nastavku. Klasa *AbstractGraph* implementira interfejs *Graph* i koristi listu susedstva za označavanje grana u grafu.

Klasa AbstractGraph definiše niz vertices (line 4) za čuvanje temena i niz neighbors (line 5) za čuvanje grana grafa u listi susedstva. neighbors.get(i) vraća sve grane susedne temenu i. Četiri predefinisana konstruktora su definisana u linijama 9–42 u cilju kreiranja grafa, ili grafa definisanog niza ili liste grana i temena. Metod createAdjacencyLists(int[][] edges, int numberOfVertices) kreira listu susedstva na osnovu niza grana (lines 45–50). Metod createAdjacencyLists(List<Edge> edges, int numberOfVertices) kreira listu susedstva na osnovu liste grana tipa Edge (lines 53–58).

Metod getNeighbors(u) (lines 81–87) vraća listu temena koji su susedni temenu u. Metod clear() (lines 106–110) briše sve grane i temena iz grafa. Metod addVertex(u) (lines 112–122) dodaje novo teme u niz vertices i vraća vrednost true. U slučaju da teme već postoji u grafu, metod vraća vrednost false (line 120).

Metod addEdge(e) (lines 124–139) dodaje novu granu u listu susednih grana. Metod vraća false ako grana već postoji.

Metod printEdges() (lines 103–112) prikazuje listu susednih temena i grana za svako teme grafa.

```
import java.util.*;

public abstract class AbstractGraph<V> implements Graph<V> {
  protected List<V> vertices = new ArrayList<>(); // Store vertices
  protected List<List<Edge>> neighbors
```

```
  = new ArrayList<>(); // Adjacency lists

/** Construct an empty graph */
protected AbstractGraph() {
}

/** Construct a graph from vertices and edges stored in arrays */
protected AbstractGraph(V[] vertices, int[][] edges) {
  for (int i = 0; i < vertices.length; i++)
    addVertex(vertices[i]);

  createAdjacencyLists(edges, vertices.length);
}

/** Construct a graph from vertices and edges stored in List */
protected AbstractGraph(List<V> vertices, List<Edge> edges) {
  for (int i = 0; i < vertices.size(); i++)
    addVertex(vertices.get(i));

  createAdjacencyLists(edges, vertices.size());
}

/** Construct a graph for integer vertices 0, 1, 2 and edge list */
protected AbstractGraph(List<Edge> edges, int numberOfVertices) {
  for (int i = 0; i < numberOfVertices; i++)
    addVertex((V)(new Integer(i))); // vertices is {0, 1, ...}

  createAdjacencyLists(edges, numberOfVertices);
}

/** Construct a graph from integer vertices 0, 1, and edge array */
protected AbstractGraph(int[][] edges, int numberOfVertices) {
  for (int i = 0; i < numberOfVertices; i++)
    addVertex((V)(new Integer(i))); // vertices is {0, 1, ...}

  createAdjacencyLists(edges, numberOfVertices);
}

/** Create adjacency lists for each vertex */
private void createAdjacencyLists(
    int[][] edges, int numberOfVertices) {
  for (int i = 0; i < edges.length; i++) {
    addEdge(edges[i][0], edges[i][1]);
  }
}

/** Create adjacency lists for each vertex */
private void createAdjacencyLists(
    List<Edge> edges, int numberOfVertices) {
  for (Edge edge: edges) {
    addEdge(edge.u, edge.v);
  }
}
```

```
@Override /** Return the number of vertices in the graph */
public int getSize() {
  return vertices.size();
}

@Override /** Return the vertices in the graph */
public List<V> getVertices() {
  return vertices;
}

@Override /** Return the object for the specified vertex */
public V getVertex(int index) {
  return vertices.get(index);
}

@Override /** Return the index for the specified vertex object */
public int getIndex(V v) {
  return vertices.indexOf(v);
}

@Override /** Return the neighbors of the specified vertex */
public List<Integer> getNeighbors(int index) {
  List<Integer> result = new ArrayList<>();
  for (Edge e: neighbors.get(index))
    result.add(e.v);

  return result;
}

@Override /** Return the degree for a specified vertex */
public int getDegree(int v) {
  return neighbors.get(v).size();
}

@Override /** Print the edges */
public void printEdges() {
  for (int u = 0; u < neighbors.size(); u++) {
    System.out.print(getVertex(u) + " (" + u + "): ");
    for (Edge e: neighbors.get(u)) {
      System.out.print("(" + getVertex(e.u) + ", " +
        getVertex(e.v) + ") ");
    }
    System.out.println();
  }
}

@Override /** Clear the graph */
public void clear() {
  vertices.clear();
  neighbors.clear();
}
```

```
@Override /** Add a vertex to the graph */
public boolean addVertex(V vertex) {
  if (!vertices.contains(vertex)) {
    vertices.add(vertex);
    neighbors.add(new ArrayList<Edge>());
    return true;
  }
  else {
    return false;
  }
}

/** Add an edge to the graph */
protected boolean addEdge(Edge e) {
  if (e.u < 0 || e.u > getSize() - 1)
    throw new IllegalArgumentException("No such index: " + e.u);

  if (e.v < 0 || e.v > getSize() - 1)
    throw new IllegalArgumentException("No such index: " + e.v);

  if (!neighbors.get(e.u).contains(e)) {
    neighbors.get(e.u).add(e);
    return true;
  }
  else {
    return false;
  }
}

@Override /** Add an edge to the graph */
public boolean addEdge(int u, int v) {
  return addEdge(new Edge(u, v));
}

/** Edge inner class inside the AbstractGraph class */
public static class Edge {
  public int u; // Starting vertex of the edge
  public int v; // Ending vertex of the edge

  /** Construct an edge for (u, v) */
  public Edge(int u, int v) {
    this.u = u;
    this.v = v;
  }

  public boolean equals(Object o) {
    return u == ((Edge)o).u &amp;amp;&amp;amp; v == ((Edge)o).v;
  }
}

@Override /** Obtain a DFS tree starting from vertex v */
/** To be discussed in Section 28.6 */
public Tree dfs(int v) {
```

```
  List<Integer> searchOrder = new ArrayList<>();
  int[] parent = new int[vertices.size()];
  for (int i = 0; i < parent.length; i++)
    parent[i] = -1; // Initialize parent[i] to -1

  // Mark visited vertices
  boolean[] isVisited = new boolean[vertices.size()];

  // Recursively search
  dfs(v, parent, searchOrder, isVisited);

  // Return a search tree
  return new Tree(v, parent, searchOrder);
}

/** Recursive method for DFS search */
private void dfs(int u, int[] parent, List<Integer> searchOrder,
    boolean[] isVisited) {
  // Store the visited vertex
  searchOrder.add(u);
  isVisited[u] = true; // Vertex v visited

  for (Edge e : neighbors.get(u)) {
    if (!isVisited[e.v]) {
      parent[e.v] = u; // The parent of vertex e.v is u
      dfs(e.v, parent, searchOrder, isVisited); // Recursive search
    }
  }
}

@Override /** Starting bfs search from vertex v */
/** To be discussed in Section 28.7 */
public Tree bfs(int v) {
  List<Integer> searchOrder = new ArrayList<>();
  int[] parent = new int[vertices.size()];
  for (int i = 0; i < parent.length; i++)
    parent[i] = -1; // Initialize parent[i] to -1

  java.util.LinkedList<Integer> queue =
    new java.util.LinkedList<>(); // list used as a queue
  boolean[] isVisited = new boolean[vertices.size()];
  queue.offer(v); // Enqueue v
  isVisited[v] = true; // Mark it visited

  while (!queue.isEmpty()) {
    int u = queue.poll(); // Dequeue to u
    searchOrder.add(u); // u searched
    for (Edge e: neighbors.get(u)) {
      if (!isVisited[e.v]) {
        queue.offer(e.v); // Enqueue w
        parent[e.v] = u; // The parent of w is u
        isVisited[e.v] = true; // Mark it visited
      }
```

```
      }
    }

    return new Tree(v, parent, searchOrder);
  }

  /** Tree inner class inside the AbstractGraph class */
  /** To be discussed in Section 28.5 */
  public class Tree {
    private int root; // The root of the tree
    private int[] parent; // Store the parent of each vertex
    private List<Integer> searchOrder; // Store the search order

    /** Construct a tree with root, parent, and searchOrder */
    public Tree(int root, int[] parent, List<Integer> searchOrder) {
      this.root = root;
      this.parent = parent;
      this.searchOrder = searchOrder;
    }

    /** Return the root of the tree */
    public int getRoot() {
      return root;
    }

    /** Return the parent of vertex v */
    public int getParent(int v) {
      return parent[v];
    }

    /** Return an array representing search order */
    public List<Integer> getSearchOrder() {
      return searchOrder;
    }

    /** Return number of vertices found */
    public int getNumberOfVerticesFound() {
      return searchOrder.size();
    }

    /** Return the path of vertices from a vertex to the root */
    public List<V> getPath(int index) {
      ArrayList<V> path = new ArrayList<>();

      do {
        path.add(vertices.get(index));
        index = parent[index];
      }
      while (index != -1);

      return path;
    }
```

```
    /** Print a path from the root to vertex v */
    public void printPath(int index) {
      List<V> path = getPath(index);
      System.out.print("A path from " + vertices.get(root) + " to " +
        vertices.get(index) + ": ");
      for (int i = path.size() - 1; i >= 0; i--)
        System.out.print(path.get(i) + " ");
    }

    /** Print the whole tree */
    public void printTree() {
      System.out.println("Root is: " + vertices.get(root));
      System.out.print("Edges: ");
      for (int i = 0; i < parent.length; i++) {
        if (parent[i] != -1) {
          // Display an edge
          System.out.print("(" + vertices.get(parent[i]) + ", " +
            vertices.get(i) + ") ");
        }
      }
      System.out.println();
    }
  }
}
```

IMPLEMENTACIJA KLASA UNWEIGHTEDGRAPH

Klasa UnweightedGraph jednostavno proširuje klasu AbstractGraph

Klasa *UnweightedGraph* jednostavno proširuje klasu *AbstractGraph* sa pet konstruktora za kreiranje konkretnog grafa, tj. instance tipa *Graph*. Klasa *UnweightedGraph* nasleđuje sve metoda apstraktne klase *AbstractGraph*, ali ne uvodi nove metode niti predefiniše bilo koji postojeći metod. Implementacija klase *UnweightedGraph* je data u nastavku:

```
import java.util.*;

public class UnweightedGraph<V> extends AbstractGraph<V> {
  /** Construct an empty graph */
  public UnweightedGraph() {
  }

  /** Construct a graph from vertices and edges stored in arrays */
  public UnweightedGraph(V[] vertices, int[][] edges) {
    super(vertices, edges);
  }

  /** Construct a graph from vertices and edges stored in List */
  public UnweightedGraph(List<V> vertices, List<Edge> edges) {
    super(vertices, edges);
  }
```

```
  /** Construct a graph for integer vertices 0, 1, 2 and edge list */
  public UnweightedGraph(List<Edge> edges, int numberOfVertices) {
    super(edges, numberOfVertices);
  }

  /** Construct a graph from integer vertices 0, 1, and edge array */
  public UnweightedGraph(int[][] edges, int numberOfVertices) {
    super(edges, numberOfVertices);
  }
}
```

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Koje su razlike, a koje sličnosti između klasa *Graph*, *AbstractGraph* i *UnweightedGraph*.

Zadatak 2. Za kod iz klase *TestGraph*, šta je rezultat sledećih naredbi:

graph1.getIndex("Seattle");

graph1.getDegree(5);

graph1.getVertex(4);

Poglavlje 4

Vizuelizacija grafa

KLASA GRAPHVIEW

U cilju da se prikaže graf, neophodno je da se zna pozicija na ekranu gde će svaki od čvorova biti prikazan. Graf može biti prikazan korišćenjem panela pod nazivom GraphView

U ovoj sekciji ćemo opisati načine na koje je moguće vizualizovati grafove. U cilju da se prikaže graf, neophodno je da se zna pozicija na ekranu gde će svaki od čvorova biti prikazan. U tu svhu će biti definisan interfejs *Displayable* koji ima metode za dobija informacija o *X* i *Z* koordinati, kao i imenu čvora (u ovom slučaju grada), dok čvorovi treba da budu instance interfejsa.

```
public interface Displayable {
  public int getX(); // Get x-coordinate of the vertex
  public int getY(); // Get x-coordinate of the vertex
  public String getName(); // Get display name of the vertex
}
```

Graf koji sada ima čvorove koji proširuju interfejs *Displayable* može biti prikazan korišćenjem panela pod nazivom *GraphView*.

Da bi prikazali graf u panelu, jednostavno treba kreirati instancu panel *GraphView* tako što prosledimo graf kao argument. Klasa za čvorove grafa mora da implementira interfejs *Displayable* u cilju da se prikažu čvorovi. Za svaki čvor sa indeksom *i*, pozivanjem funkcije *graph.getNeighbors(i)* dobijamo njegovu listu suseda. Na osnovu ove liste, moguće je pronaći sve njegove susedne čvorove, pa se može nacrtati linija koja povezuje čvor *i* sa njemu odgovarajućim susedom.

```
import javafx.scene.Group;
import javafx.scene.layout.BorderPane;
import javafx.scene.shape.Circle;
import javafx.scene.shape.Line;
import javafx.scene.text.Text;

public class GraphView extends BorderPane {
  private Graph<? extends Displayable> graph;
  private Group group = new Group();

  public GraphView(Graph<? extends Displayable> graph) {
    this.graph = graph;
    this.setCenter(group); // Center the group
```

```
    repaintGraph();
  }

  private void repaintGraph() {
    group.getChildren().clear(); // Clear group for a new display

    // Draw vertices and text for vertices
    java.util.List<? extends Displayable> vertices
      = graph.getVertices();
    for (int i = 0; i < graph.getSize(); i++) {
      double x = vertices.get(i).getX();
      double y = vertices.get(i).getY();
      String name = vertices.get(i).getName();

      group.getChildren().add(new Circle(x, y, 16));
      group.getChildren().add(new Text(x - 8, y - 18, name));
    }

    // Draw edges for pairs of vertices
    for (int i = 0; i < graph.getSize(); i++) {
      java.util.List<Integer> neighbors = graph.getNeighbors(i);
      double x1 = graph.getVertex(i).getX();
      double y1 = graph.getVertex(i).getY();
      for (int v: neighbors) {
        double x2 = graph.getVertex(v).getX();
        double y2 = graph.getVertex(v).getY();

        // Draw an edge for (i, v)
        group.getChildren().add(new Line(x1, y1, x2, y2));
      }
    }
  }
}
```

PRIMER ISCRTAVANJA GRAFA

Za potrebe vizualizacije grafa kreiran je aplet DisplayUSMap. Pri startovanju apleta otvara se panel tipa GraphView, u kome se kreira graf sa čvorovima koji su tipa City

Naredni listing predstavlja primer iscrtavanja grafa sa slike 1.

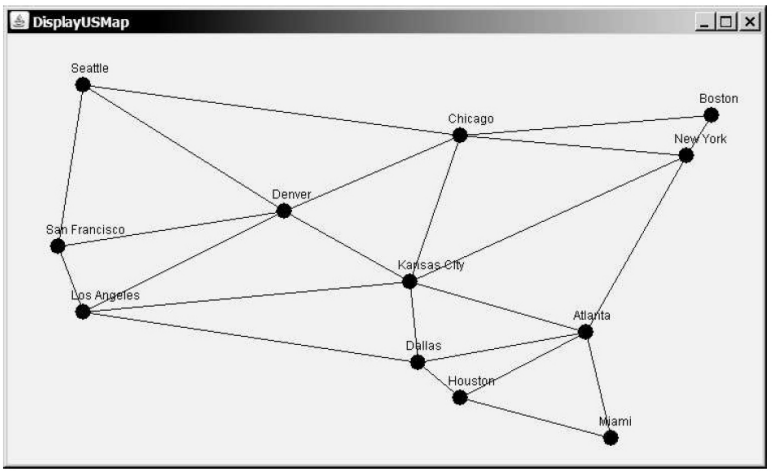
```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class DisplayUSMap extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    City[] vertices = {new City("Seattle", 75, 50),
```

```
      new City("San Francisco", 50, 210),
      new City("Los Angeles", 75, 275), new City("Denver", 275, 175),
      new City("Kansas City", 400, 245),
      new City("Chicago", 450, 100), new City("Boston", 700, 80),
      new City("New York", 675, 120), new City("Atlanta", 575, 295),
      new City("Miami", 600, 400), new City("Dallas", 408, 325),
      new City("Houston", 450, 360) };

    // Edge array for graph in Figure 28.1
    int[][] edges = {
      {0, 1}, {0, 3}, {0, 5}, {1, 0}, {1, 2}, {1, 3},
      {2, 1}, {2, 3}, {2, 4}, {2, 10},
      {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
      {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
      {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
      {6, 5}, {6, 7}, {7, 4}, {7, 5}, {7, 6}, {7, 8},
      {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
      {9, 8}, {9, 11}, {10, 2}, {10, 4}, {10, 8}, {10, 11},
      {11, 8}, {11, 9}, {11, 10}
    };

    Graph<City> graph = new UnweightedGraph<>(vertices, edges);

    // Create a scene and place it in the stage
    Scene scene = new Scene(new GraphView(graph), 750, 450);
    primaryStage.setTitle("DisplayUSMap"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  static class City implements Displayable {
    private double x, y;
    private String name;

    City(String name, double x, double y) {
      this.name = name;
      this.x = x;
      this.y = y;
    }

    @Override
    public double getX() {
      return x;
    }

    @Override
    public double getY() {
      return y;
    }

    @Override
    public String getName() {
      return name;
```

```
    }
  }

  /**
   * The main method is only needed for the IDE with limited
   * JavaFX support. Not needed for running from the command line.
   */
  public static void main(String[] args) {
    launch(args);
  }
}
```

Klasa *City* je definisana sa ciljem da se modeliraju čvorovi tako da sadrže i naziv grada i koordinate. Program kreira graf sa čvorovima koji su tipa *City*. S obzirom da klasa *City* implementira interfejs *Displayable*, objekat klase *GraphView* će prikazati graf u panelu.

Kao primer za vežbu, u cilju boljeg razumevanja grafova, možete dodati novi grad (npr. Detroit) sa odgovarajućim granama ka ostalim čvorovima grafa.



Slika 4.1 Vizualizacija grafa korišćenjem java.swing - Daniel Liang, 2013.

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Da li će program *DisplayUSMap* i dalje da radi, ako se linije 30–34 u klasi *GraphView* zamene sledećim kodom? Zašto?

```
if (i < v) {
    int x2 = graph.getVertex(v).getX();
    int y2 = graph.getVertex(v).getY();
    // Draw an edge for (i, v)
    getChildren().add(new Line(x1, y1, x2, y2));
}
```

Zadatak 2. Za objekat **graph1** kreiran u *TestGraph*, da li se kreiranje *GraphView* objekta može izvršiti na sledeći način?

```
GraphView view = new GraphView(graph1);
```

Poglavlje 5

Obilazak grafa i obuhvatno stablo

OBILAZAK GRAFA

Obilazak grafa u dubinu i širinu su dva najuobičajenija načina obilaska čvorova u grafu. Ideja je pronaći sve moguće čvorove putujući nekom od putanja od startnog čvora

Obilazak grafa se obično sastoji iz sledećih koraka:

- Polazimo od unapred definisanog čvora
- Pronaći sve čvorove koje možemo dosegnuti ako pođemo iz startnog čvora putujući nekom od mogućih putanja

To može biti slučaj kada se traže informacije koje se nalaze u čvorovima ili ako se želi distribuirati informacija svakom čvoru.

Ono o čemu treba voditti računa je to da jedan čvor u toku obilaska grafa možemo posetiti samo jednom, što znači da ne sme biti cikličnih ponavljanja. Stoga moramo uvesti neki parametar koji će nam uvek kazati da li je neki čvor tokom obilaska putanje posećen ili ne.

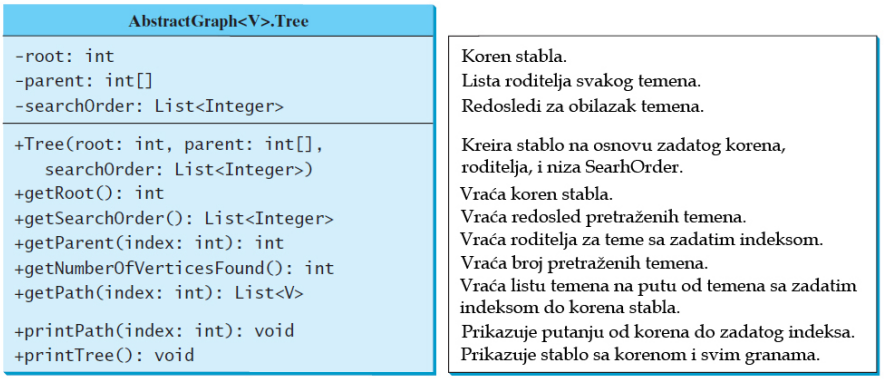
U nastavku je prikazan primer korišćenja niza *marked* koji može biti korišćen u tu svrhu:

```
boolean[] marked;
marked = new boolean[g.size()];
```

Postoje dva najuobičajenija načina obilaska grafa:

- *obilazak u dubinu* (eng. depth-first search) i
- *obilazak u širinu* (eng. breadth-first search).

Oba obilaska rezultiraju u *obuhvatnom stablu* (spanning tree) koja mogu biti modelirana korišćenjem klase koja je prikazana na Slici-1. Primetimo da je *Tree* unutrašnja klasa klase *AbstractGraph*. Klasa *AbstractGraph<V>.Tree* se razlikuje od interfejsa *Tree* koga smo definisali u lekciji *L09 -Stabla I*. *AbstractGraph.Tree* je specijalna klasa dizajnirana za opis relacija roditelj-deca kod čvorova grafa.



Slika 5.1 Klasa Tree opisuje relacije čvorova kada su u pitanju odnosi između roditelja i direktnih naslednika.

IMPLEMENTACIJA KLASE ZA OBUHVATNO STABLO

Klasa Tree je definisana kao unutrašnja klasa klase AbstractGraph. Ona služi za čuvanje obuhvatnog stabla koje je dobijeno odgovarajućim obilaskom grafa

Klasa *Tree* je definisana kao unutrašnja klasa klase *AbstractGraph*. Kod ove klase je dat u nastavku:

```
/** Tree inner class inside the AbstractGraph class */
public class Tree {
 private int root; // The root of the tree
 private int[] parent; // Store the parent of each vertex
 private List<Integer> searchOrder; // Store the search order

 /** Construct a tree with root, parent, and searchOrder */
 public Tree(int root, int[] parent, List<Integer> searchOrder) {
  this.root = root;
  this.parent = parent;
  this.searchOrder = searchOrder;
 }

 /** Return the root of the tree */
 public int getRoot() {
  return root;
 }

 /** Return the parent of vertex v */
 public int getParent(int v) {
  return parent[v];
 }

 /** Return an array representing search order */
 public List<Integer> getSearchOrder() {
  return searchOrder;
```

```
 }

 /** Return number of vertices found */
 public int getNumberOfVerticesFound() {
  return searchOrder.size();
 }

 /** Return the path of vertices from a vertex to the root */
 public List<V> getPath(int index) {
  ArrayList<V> path = new ArrayList<V>();

  do {
   path.add(vertices.get(index));
   index = parent[index];
  }
  while (index != -1);

  return path;
 }

 /** Print a path from the root to vertex v */
 public void printPath(int index) {
  List<V> path = getPath(index);
  System.out.print("A path from " + vertices.get(root) + " to " +
    vertices.get(index) + ": ");
  for (int i = path.size() - 1; i >= 0; i--)
   System.out.print(path.get(i) + " ");
 }

 /** Print the whole tree */
 public void printTree() {
  System.out.println("Root is: " + vertices.get(root));
  System.out.print("Edges: ");
  for (int i = 0; i < parent.length; i++) {
   if (parent[i] != -1) {
    // Display an edge
    System.out.print("(" + vertices.get(parent[i]) + ", " +
      vertices.get(i) + ") ");
   }
  }
  System.out.println();
 }
}
```

Klasa *Tree* definiše sedam metoda.

Metod *getRoot()* vraća koren obuhvatnog stabla. Pozivom metoda *getSearchOrder()* dobija se sekvenca posećenih čvorova u redosledu kako su oni i posećeni.

Metod *getParent(v)* pronalazi roditelja čvora *v*, tj. čvora preko kojeg se u pretrazi došlo do čvora *c*.

Metod *getNumberOfVerticesFound()* vraća broj posećenih čvorova. Ukoliko je broj posećenih čvorova manji od broja čvorova u grafu može se zaključiti da taj graf nije povezan.

Metod *getPath(index)* vraća listu čvorova na putanji od odgovarajućeg čvora, definisanog indeksom, do korena (*root*) obuhvatnog stabla.

Metod *printPath(v)* prikazuje putanju od korena (*root*) do čvora *v*.

Korišćenjem metoda *printTree()* moguće je prikazati sve grane obuhvatnog stabla.

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Da li AbstractGraph<V>.Tree implementira *Tree* interfejs implementiran u *Tree.java*?

Zadatak 2. Koji metod se koristi da bi se našao roditelj nekog čvora u stablu?

Poglavlje 6

Obilazak grafa u dubinu

UVODNA RAZMATRANJA

DFS algoritam obilazi dubinu bilo koje posebne putanje pre istraživanja svoje širine. Koristiti stek (ili rekurziju) da bi ste čuvali čvorove koje treba posetiti

Kod obilaska grafa u dubinu koriste se sledeća pravila:

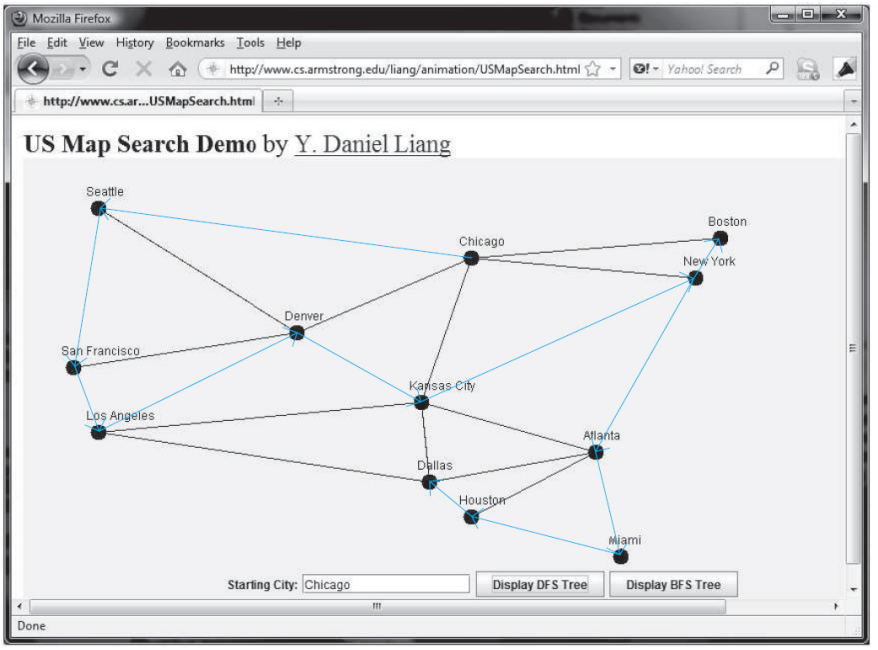
- Koristiti stek (ili rekurziju) da bi ste čuvali čvorove koje treba posetiti
- Iz zadatog čvora grafa *J*, koji ima susede označene sa *I, I+1,...N*, posetiti prvo sve moguće grane čvora *I* pre nego što se posete susedi označeni sa *I+1 ... N*.

Pretraga po dubini - *DFS* je algoritam za obilazak konačnog grafa. *DFS* posećuje decu čvorove pre posete braći, to jest, *DFS* obilazi dubinu bilo koje posebne putanje pre istraživanja svoje širine.

Algoritam za obilazak grafa u dubinu se može napisati na sledeći način:

```
Tree dfs(vertex v) {
 visit v;
 for each neighbor w of v
 if (w has not been visited) {
  dfs(w);
 }
}
```

You can use an array named isVisited to denote whether a vertex has been visited. Initially, isVisited[i] is false for each vertex i. Once a vertex, say v, is visited, isVisited[v] is set to true. The graphical illustration of the DFS starting from Chicago is shown in
Slika-1. For an interactive GUI demo of DFS, go to www.cs.armstrong.edu/liang/animation/USMapSearch.html.



Slika 6.1.1 A DFS search starts from Chicago.

POSTUPAK DFS OBILASKA GRAFA

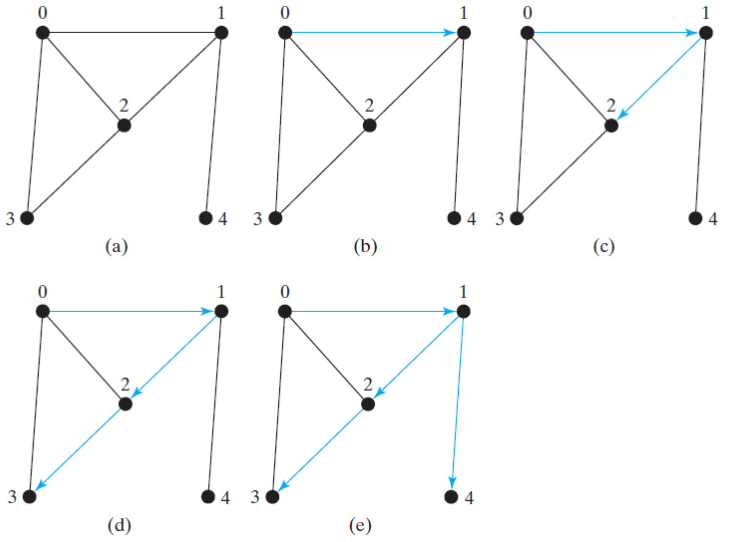
Iz zadatog čvora grafa J, koji ima susede označene sa I, I+1,...N, neophodno je prvo posetiti sve moguće grane čvora I pre nego što se posete susedi označeni sa I+1 ... N

Uzmimo u razmatranje graf sa Slike-2a. Pretpostavimo da algoritam obilaska po dubini započinjemo iz čvora *0*. Prvo posetimo *0* a zatim nekog od njegovih suseda, recimo *1*. Sada je *1* posećen, kao na Slici-2b.

Čvor *1* ima tri suseda - *0*, *2*, i *4*. Pošto je *0* već posećen, ostali su nam čvorovi *2* ili *4*. Izaberimo čvor *2*. Sada je *2* posećen, kao što je označeno na Slici-1c. Čvor *2* ima tri suseda: *0*, *1*, i *3*. Pošto su *0* i *1* već posećeni, izaberimo *3*. Sada je *3* posećen, kao što je prikazano na Slici-2d. Do ovog trenutka obišli smo čvorove: *0*, *1*, *2*, *3*.

Pošto su posećeni svi susedi čvora *3* vraćamo se (backtrack) na čvor *2*. Pošto su i svi susedi čvora *2* posećeni vraćamo se na čvor *1*. Iz čvora *1* je ostalo da posetimo suseda *4*. Stoga, posećujemo čvor *4*, kao na Slici-2e. Pošto su svi susedi čvora *4* posećeni vraćamo se na čvor *1*. Pošto su i svi susedi čvora *1* posećeni vraćamo se na čvor *0*. Na kraju, pošto su i svi susedi čvora *0* posećeni, pretraga je završena.

Pošto se sve grane i svi čvorovi posećuju najviše jednom, vremenska složenost *dfs* algoritma je u najgorem slučaju $O(|E| + |V|)$, gde $|E|$ označava broj grana a $|V|$ broj čvorova.



Slika 6.1.2 Slika 2. Postupak obilaska po dubini u primeru grafa sa 5 čvorova- Daniel Liang, 2013.

IMPLEMENTACIJA DFS ALGORITMA

DFS algoritam je veoma jednostavno implementirati korišćenjem rekurzije. Za svaki neposećeni sused čvora v, metod se rekurzivno poziva da bi se nastavila pretraga po dubini

Prethodno opisani algoritam za *DFS* obilazak koristi rekurziju pa je prirodno koristiti i rekurziju u toku implementacije. Kao alternativa moguće je koristiti stek. Metod *dfs(int v)* je implementiran u okviru klase *AbstractGraph*. Izvorni kod ove metode je dat u nastavku:

```
public Tree dfs(int v) {
    List<Integer> searchOrder = new ArrayList<Integer>();
    int[] parent = new int[vertices.size()];
    for (int i = 0; i < parent.length; i++)
    parent[i] = -1; // Initialize parent[i] to -1

    // Mark visited vertices
    boolean[] isVisited = new boolean[vertices.size()];

    // Recursively search
    dfs(v, parent, searchOrder, isVisited);

    // Return a search tree
    return new Tree(v, parent, searchOrder);
  }

  /** Recursive method for DFS search */
  private void dfs(int v, int[] parent, List<Integer> searchOrder,
    boolean[] isVisited) {
    // Store the visited vertex
    searchOrder.add(v);
```

```
    isVisited[v] = true; // Vertex v visited

    for (int i : neighbors.get(v)) {
    if (!isVisited[i]) {
        parent[i] = v; // The parent of vertex i is v
        dfs(i, parent, searchOrder, isVisited); // Recursive search
    }
    }
  }
```

Metod *dfs(int v)* vraća instancu klase *Tree* sa čvorom *v* kao korenom obuhvatnog stabla. Metod pakuje pretražene čvorove u listu *searchOrder*, roditelj svakog čvora se čuva u nizu *parent*, a koristi niz *isVisited* da se označi da li je čvor već posećen ili ne.

Metod *dfs(int v)* poziva pomoćni metod *dfs(v, parent, searchOrder, isVisited)* da bi se obavila pretraga po dubini.

U rekurzivnom pomoćnom metodu, pretraga počinje od čvora *v*. Čvor *v* se dodaje u niz *searchOrder* i markira se kao posećen. Za svaki neposećeni sused čvora *v*, metod se rekurzivno poziva da bi se nastavila pretraga po dubini. Kada se poseti čvor *i* roditelj čvora *i* se čuva u nizu *parent[i]*.

Metod *dfs* se završava tek kada su obiđeni svi čvorovi povezanog grafa.

TESTIRANJE OBILASKA GRAFA U DUBINU

Cilj primera je testiranje algoritma za obilazak grafa u dubinu implementiranog u okviru klase AbstractGraph

U nastavku je primer test klase:

```
public class TestDFS {
  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}, {0, 3}, {0, 5},
    {1, 0}, {1, 2}, {1, 3},
    {2, 1}, {2, 3}, {2, 4}, {2, 10},
    {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
    {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
    {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
    {6, 5}, {6, 7},
    {7, 4}, {7, 5}, {7, 6}, {7, 8},
    {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
    {9, 8}, {9, 11},
    {10, 2}, {10, 4}, {10, 8}, {10, 11},
    {11, 8}, {11, 9}, {11, 10}
```

```
    };

    Graph<String> graph =
    new UnweightedGraph<String>(edges, vertices);
    AbstractGraph<String>.Tree dfs =
    graph.dfs(graph.getIndex("Chicago"));

    java.util.List<Integer> searchOrders = dfs.getSearchOrder();
    System.out.println(dfs.getNumberOfVerticesFound() +
    " vertices are searched in this DFS order:");
    for (int i = 0; i < searchOrders.size(); i++)
    System.out.print(graph.getVertex(searchOrders.get(i)) + " ");
    System.out.println();

    for (int i = 0; i < searchOrders.size(); i++)
    if (dfs.getParent(i) != -1)
    System.out.println("parent of " + graph.getVertex(i) +
        " is " + graph.getVertex(dfs.getParent(i)));
  }
}
```

Rezultat programa biće

```
12 vertices are searched in this DFS order:
Chicago Seattle San Francisco Los Angeles Denver
Kansas City New York Boston Atlanta Miami Houston Dallas
parent of Seattle is Chicago
parent of San Francisco is Seattle
parent of Los Angeles is San Francisco
parent of Denver is Los Angeles
parent of Kansas City is Denver
parent of Boston is New York
parent of New York is Kansas City
parent of Atlanta is New York
parent of Miami is Atlanta
parent of Dallas is Houston
parent of Houston is Miami
```

PRIMENA DFS OBILASKA

Obilazak po dubini se može koristiti kao rešenje za ogroman broj problema. Jedan od najpoznatijih primera je rešavanje Problema konjićevog skoka korišćenjem Hamiltonovih putanja

Obilazak po dubini se može koristiti kao rešenje za ogroman broj problema, kao što su:

- Ispitivanje povezanosti grafa. Pretražiti graf počev od bilo kog čvora. Ukoliko je broj posećenih čvorova jednak ukupnom broju čvorova u grafu onda je graf povezan. U suprotnom, graf nije povezan.
- Ispitivanje da li postoji putanja između dva čvora grafa.

- Određivanje putanje između dva čvora.
- Određivanje svih povezanih komponenti. Povezana komponenta je maksimalni povezani podgraf u kome je svaki par čvorova povezan putanjom.
- Ispitivanje da li postoji ciklus (ciklična putanja) u grafu.
- Određivanje ciklusa u grafu.
- Pronalaženje **Hamiltonove putanje/ciklusa**. *Hamiltonova putanja* u grafu je putanja koja posećuje svaki čvor grafa samo jednom. *Hamiltonov ciklus* posećuje svaki čvor grafa samo jednom i vraća sa u početni čvor putanje.

Prvih šest problema mogu biti veoma jednostavno rešeni korišćenjem metoda *dfs*. Da bi pronašli Hamiltonovu putanju/ciklus, neophodno je da istražimo sva *DFS* obuhvatna stabla da bi pronašli najdužu moguću putanju. Hamiltonova putanja/ciklus ima veliku primenu, uključujući rešavanje *Problema konjićevog skoka* (eng. Knight's tour problem).

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

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Nacrtati DFS stablo za graf sa **Slika-3a**, krenuti od čvora sa nazivom Atlanta.

Zadatak 2. Nacrtati DFS stablo za graf sa **Slika-3b,** krenuti od čvora A.

Zadatak 3. Koja je povratna vrednost pozivanja **dfs(v)**?

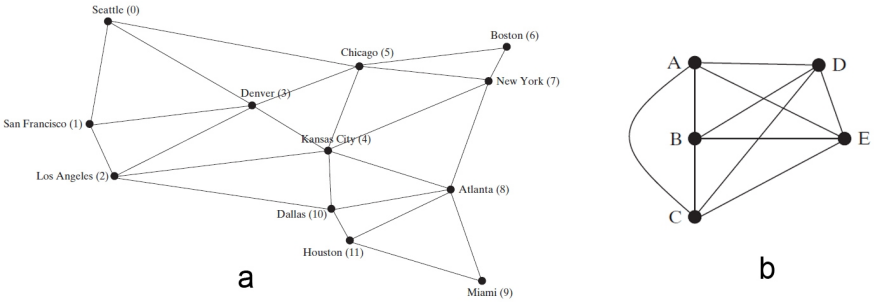
Zadatak 4. Prethodno implementiran DFS koristi rekurziju. Drugi način je upotreba steka za simuliranje rekurzije, gde je kod prikazan ispod. Pronađite grešku i ispravite je.

```
// Wrong version
Tree dfs(vertex v) {
    push v into the stack;
    mark v visited;

    while (the stack is not empty) {
        pop a vertex, say u, from the stack
        visit u;
        for each neighbor w of u
            if (w has not been visited)
                push w into the stack;
    }
}
```

Zadatak 5. Kreirati novu klasu pod nazivom *MyGraph* kao podklasu klase *UnweightedGraph* koja sadrži metod koji pronalazi sve povezane komponente u grafu. Metod ima sledeće zaglavlje:

```
public List<List<Integer>> getConnectedComponents();
```

Metod kao rezultat vraća listu tipa *List<List<Integer>>*. Svaki element u listi je druga lista koja sadrži sve čvorove u povezanoj komponenti grafa.



Slika 6.1.3 Primeri za samostalni rad

Zadatak 6. Graf je bipartitivan ako njegovi čvorovi mogu biti razdvojeni u dva podskupa tako da ne postoje grane koje spajaju čvorove istog podskupa. Dodati novi metod u klasu AbstractGraph koji ispituje da li je graf bipartitivan.

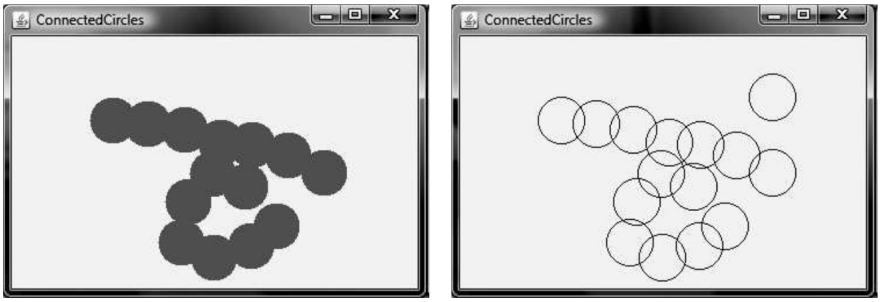
Zadatak 7. Dodati novi metod u klasu AbstractGraph koji kao rezultat vraća dva biparititivna skupa čvorova ukoliko je graf bipartitivan. Metod vraća listi tipa List koja sadrži dve podliste, pri čemu svaka od lista sadrži skup čvorova. Ukoliko graf nije bipartitivan, metod vraća null.

6.1 Studija slučaja: Problem spojenih krugova

POSTAVKA PROBLEMA

Problem povezanih krugova se sastoji u određivanju da li su svi krugovi u dvodimenzionalnoj ravni povezani. Problem može biti veoma lako rešen korišćenjem obilaska po dubini - DFS

DFS algoritam ima široku primenu. U ovoj sekciji se vrši primena DFS algoritma u cilju rešavanja problema povezanih krugova. U okviru problema spojenih krugova, neophodno je odrediti da li su sve kružnice u 2D ravni povezane. Ukoliko su sve kružnice povezane, onda se one oboje kao ispunjene kružnice, kao što je prikazano na Slici-1a. U suprotnom, one neće biti ispunjene bojom, kao na Slici-1b.



a b

Slika 6.2.1 Može se primeniti DFS algoritam u cilju uspitivanja da li su kružnice povezane ili ne.

Program će biti napisan tako da će korisnik klikom na belu pozadinu apleta, koja trenutno nije ispunjena kružnicom, moći da kreira novu kružnicu. Kako se kružnice dodaju u prostor, tako se vrši osvežavanje prikaza i kružnice se boje u zavisnosti od toga da li su povezane ii ne.

Kreiraćemo graf da modeliramo problem. Svaka kružnica je čvor grafa. Dve kružnice su povezane ako se preklapaju. Primenićemo *DFS* agloritam, tako da ako je broj čvorova u obuhvatnom *DFS* stablu jednak ukupnom broju čvorova grafa onda je graf povezan i krugovi će biti obojeni.

IMPLEMENTACIJA ALGORITMA

U slučaju da je broj čvorova u obuhvatnom DFS stablu jednak ukupnom broju čvorova grafa onda je graf povezan i krugovi će biti obojeni

Program je dat u listingu koji sledi:

```
Show Code With Line Numbers
import javafx.application.Application;
import javafx.geometry.Point2D;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.Circle;
import javafx.stage.Stage;

public class ConnectedCircles extends Application {
  @Override // Override the start method in the Application class
  public void start(Stage primaryStage) {
    // Create a scene and place it in the stage
    Scene scene = new Scene(new CirclePane(), 450, 350);
    primaryStage.setTitle("ConnectedCircles"); // Set the stage title
    primaryStage.setScene(scene); // Place the scene in the stage
    primaryStage.show(); // Display the stage
  }

  /** Panel for displaying circles */
  class CirclePane extends Pane {
```

```
  public CirclePane() {
    this.setOnMouseClicked(e -> {
      if (!isInsideACircle(new Point2D(e.getX(), e.getY()))) {
        // Add a new circle
        getChildren().add(new Circle(e.getX(), e.getY(), 20));
        colorIfConnected();
      }
    });
  }

  /** Returns true if the point is inside an existing circle */
  private boolean isInsideACircle(Point2D p) {
    for (Node circle: this.getChildren())
      if (circle.contains(p))
        return true;

    return false;
  }

  /** Color all circles if they are connected */
  private void colorIfConnected() {
    if (getChildren().size() == 0)
      return; // No circles in the pane

    // Build the edges
    java.util.List<Edge> edges =
        new java.util.ArrayList<>();
    for (int i = 0; i < getChildren().size(); i++)
      for (int j = i + 1; j < getChildren().size(); j++)
        if (overlaps((Circle)(getChildren().get(i)),
            (Circle)(getChildren().get(j)))) {
          edges.add(new Edge(i, j));
          edges.add(new Edge(j, i));
        }

    // Create a graph with circles as vertices
    Graph<Node> graph = new UnweightedGraph<>
      ((java.util.List<Node>)getChildren(), edges);
    UnweightedGraph<Node>.SearchTree tree = graph.dfs(0);
    boolean isAllCirclesConnected = getChildren().size() == tree
        .getNumberOfVerticesFound();

    for (Node circle: getChildren()) {
      if (isAllCirclesConnected) { // All circles are connected
        ((Circle)circle).setFill(Color.RED);
      }
      else {
        ((Circle)circle).setStroke(Color.BLACK);
        ((Circle)circle).setFill(Color.WHITE);
      }
    }
  }
}
```

```
  public static boolean overlaps(Circle circle1, Circle circle2) {
    return new Point2D(circle1.getCenterX(), circle1.getCenterY()).
      distance(circle2.getCenterX(), circle2.getCenterY())
      <= circle1.getRadius() + circle2.getRadius();
  }

  /**
   * The main method is only needed for the IDE with limited
   * JavaFX support. Not needed for running from the command line.
   */
  public static void main(String[] args) {
    launch(args);
  }
}
```

JavaFX klasa *Circle* ima polja **x, y** i **radius**, koji određuju lokaciju i poluprečnik. Takođe definiše se **contains** metod za proveru da li se tačka nalazi u krugu. Metod **overlaps** proverava da li se dva kruga preklapaju.

Kada korisnik klikne na neko polje izvan postojećih krugova, novi krug se kreira sa centrom gde je kliknut taster miša, i podaci o novom krugu se dodaju u listu *circles*.

Da bi ispitali da li su krugovi povezani, program generiše graf. Krugovi su čvorovi (temena) grafa. Zatim se vrši kreiranje grana u grafu između dva čvora ukoliko se kružnice koje odgovaraju tim čvorovima preklapaju. Primenom DFS algoritma kreira se *DFS obuhvatno stablo*.

Metoda klase *Tree* pod nazivom *getNumberOfVerticesFound()* vraća broj posećenih čvorova. Ukoliko je ovaj broj jednak ukupnom broju krugova, svi krugovi su povezani i biće ispunjeni odgovarajućom bojom.

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke

Zadatak 1. Kako je graf za problem spojenih krugova napravljen?

Zadatak 2. Kada se klikne mišem unutar kruga, da li program kreira novi krug?

Zadatak 3. Kako program zna da li su svi krugovi povezani?

Zadatak 4. Modifikovati *ConnectedCircles*, tako da se omogući uzimanje i pomeranje kruga mišem (drag and move).
Zadatak 5. Modifikovati *ConnectedCircles*, tako da se omogući brisanje kruga na klik unutar kruga.

Poglavlje 7

Obilazak grafa u širinu

UVODNA RAZMATRANJA

Kod BFS obilaska se koristiti red čekanja za smestanje čvorova koji će tek biti posećeni. Ideja je da se za čvor J prvo posete svi susedi čvora J, pa tek onda susedi suseda čvora J

Kod obilaska grafa u širinu koriste se sledeća pravila:

- Koristiti red čekanja da se u njemu smeste čvorovi koji će tek biti posećeni
- Ako krenete iz čvora *J*, koji ima susede označene *sa I, I+1,...N*, onda prvo poseti sve njegove susede *(I,...N)*, pa tek onda susede suseda čvora *J*.

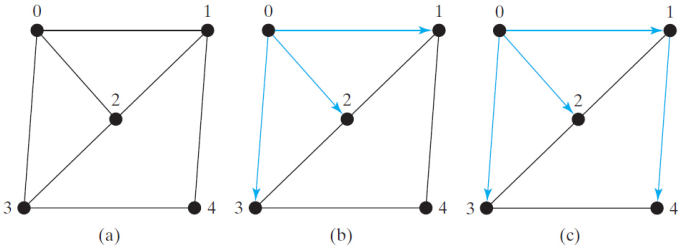
Algoritam za pretragu po širini može biti predstavljen na sledeći način:

```
bfs(vertex v) {
 kreirati novi prazni red za cuvanje čvorova;
 dodaj v u red;
 oznaci v kao posecen;

 while (red nije prazan) {
  iz reda izbaci čvor, recimo u, koriscenjem dequeue;
  cvor u dodaj u listu posecenih cvorova;
  for svaki sused w cvora u
   if w nije poseceno {
    dodaj w u red;
    oznaci w kao posecen;
   }
 }
}
```

Razmotrimo graf sa Slike-1a. pretpostavimo da obilazak po dubini započinjemo iz čvora *0*. Posetimo prvo čvor *0* a zatim posetimo sve njegove susede: *1*, *2*, i *3*, kao što je prikazano na Slici-1b.

Čvor *1* ima tri suseda: *0*, *2*, i *4*. Pošto su *0* i *2* već posećeni, ostalo je da posetimo čvor *4*, kao što je prikazano na slici 1c. Čvor *3* ima tri suseda *0*, *2*, i *4*, koji su već posećeni. Čvor *4* ima dva suseda *1* i *3* koji su takođe posećeni. Stoga, pretraga je završena.

Pošto se svaka grana i svaki čvor grafa posećuju samo jednom, vremenska složenost *BFS* metoda je $O(|E| + |V|)$, gde je $|E|$ broj grana a $|V|$ broj čvorova u grafu.



Slika 7.1 Postupak obilaska grafa u širinu za polazni čvor 0 - Daniel Liang, 2013.

IMPLEMENTACIJA BFS ALGORITMA

Metod bfs(int v) može biti definisan u interfejsu Graph a implementiran u okviru klase AbstractGraph. Metod kao rezultat vraća instancu klase Tree sa čvorom v kao korenom stabla

Metod *bfs(int v)* je definisan u interfejsu *Graph* a implementiran u okviru klase*AbstractGraph*. U nastavku je implementacija:

```
@Override /** Starting bfs search from vertex v */
  public Tree bfs(int v) {
    List<Integer> searchOrder = new ArrayList<Integer>();
    int[] parent = new int[vertices.size()];
    for (int i = 0; i < parent.length; i++)
    parent[i] = -1; // Initialize parent[i] to -1

    java.util.LinkedList<Integer> queue =
    new java.util.LinkedList<Integer>(); // list used as a queue
    boolean[] isVisited = new boolean[vertices.size()];
    queue.offer(v); // Enqueue v
    isVisited[v] = true; // Mark it visited

    while (!queue.isEmpty()) {
    int u = queue.poll(); // Dequeue to u
    searchOrder.add(u); // u searched
    for (int w : neighbors.get(u)) {
        if (!isVisited[w]) {
        queue.offer(w); // Enqueue w
        parent[w] = u; // The parent of w is u
        isVisited[w] = true; // Mark it visited
        }
    }
    }

    return new Tree(v, parent, searchOrder);
  }
```

Metod kao rezultat vraća instancu klase *Tree* sa čvorom *v* kao korenom stabla. Metod čuva posećene čvorove u listi *searchOrder*, roditelje svakog čvora u nizu *parent*, koristi povezanu listu za red, i koristi niz *isVisited* da se označi da li je čvor posećen ili ne.

Obilazak započinjemo iz proizvoljnog čvora *v*. Čvor *v* se dodaje u red, i označava kao posećen. Metod zatim ispituje svaki čvor *u* iz reda čekanja i dodaje ga u niz *searchOrder* prilikom posete. Metod dodaje u red svakog neposećenog suseda *w* čvora *u*, postavlja njegovog roditelja na *u*, i označava ga kao posećenog.

TESTIRANJE OBILASKA GRAFA U ŠIRINU

Cilj primera je testiranje algoritma za obilazak grafa u širinu implementiranog u okviru klase AbstractGraph

Testiranje obilaska u širinu korišćenjem pretodno definisane metode BFS. Demonstracija GUI za testiranje obilaska grafova po širini se može naći na adresi http://www.cs.armstrong.edu/liang/animation/USMapSearch.html.

U nastavku je data test klasa:

```
public class TestBFS {
  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}, {0, 3}, {0, 5},
    {1, 0}, {1, 2}, {1, 3},
    {2, 1}, {2, 3}, {2, 4}, {2, 10},
    {3, 0}, {3, 1}, {3, 2}, {3, 4}, {3, 5},
    {4, 2}, {4, 3}, {4, 5}, {4, 7}, {4, 8}, {4, 10},
    {5, 0}, {5, 3}, {5, 4}, {5, 6}, {5, 7},
    {6, 5}, {6, 7},
    {7, 4}, {7, 5}, {7, 6}, {7, 8},
    {8, 4}, {8, 7}, {8, 9}, {8, 10}, {8, 11},
    {9, 8}, {9, 11},
    {10, 2}, {10, 4}, {10, 8}, {10, 11},
    {11, 8}, {11, 9}, {11, 10}
    };

    Graph<String> graph =
    new UnweightedGraph<String>(edges, vertices);
    AbstractGraph<String>.Tree bfs =
    graph.bfs(graph.getIndex("Chicago"));

    java.util.List<Integer> searchOrders = bfs.getSearchOrders();
    System.out.println(bfs.getNumberOfVerticesFound() +
    " vertices are searched in this order:");
    for (int i = 0; i < searchOrders.size(); i++)
    System.out.println(graph.getVertex(searchOrders.get(i)));

    for (int i = 0; i < searchOrders.size(); i++)
    if (bfs.getParent(i) != -1)
```

```
        System.out.println("parent of " + graph.getVertex(i) +
        " is " + graph.getVertex(bfs.getParent(i)));
  }
}
```

Rezultat programa biće:

```
12 vertices are searched in this order:
Chicago Seattle Denver Kansas City Boston New York
San Francisco Los Angeles Atlanta Dallas Miami Houston
parent of Seattle is Chicago
parent of San Francisco is Seattle
parent of Los Angeles is Denver
parent of Denver is Chicago
parent of Kansas City is Chicago
parent of Boston is Chicago
parent of New York is Chicago
parent of Atlanta is Kansas City
parent of Miami is Atlanta
parent of Dallas is Kansas City
parent of Houston is Atlanta
```

PRIMENA BFS ALGORITMA

Mnogi od problema koji se mogu rešiti korišćenjem DFS algoritma se mogu rešiti i primenom BFS algoritma. Jedan od prepoznatljivih problema je ispitivanje bipartitivnosti grafa

Mnogi od problema koji se mogu rešiti korišćenjem *DFS* algoritma se mogu rešiti i primenom *BFS* algoritma. *BFS* algoritam se može korisiti za rešavanje sledećih problema:

- Ispitivanje povezanosti grafa. Graf je povezan ako postoji putanja između svaka dva čvora u grafu.
- Ispitivanje da li postoji putanja između dva čvora u grafu.
- Određivanje najkraće putanje između dva čvora. Može se dokazati da je putanja između korena i bilo kog čvora grafa u *BFS obuhvatnom stablu* ustvari najkraća putanja od korena stabla do tog čvora
- Određivanje svih povezanih komponenti. Povezana komponenta je najveći povezani podgragf u kome je svaki par čvorova povezan putanjom.
- Ispitivanje da li postoji ciklična putanja u grafu
- Određivanje ciklične putanje u grafu.
- Ispitati da li je graf bipartitivan (Graf je bipartitivan ako čvorovi grafa mogu biti podeljeni u dva odvojena podskupa takva da ne postoje grane između čvorova istog podskupa).

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

ZADACI ZA SAMOSTALNO VEŽBANJE

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

Zad 1. Nacrati BFS stablo za graf sa slike **Slika-2a**, početi iz čvora Atlanta.

Zad 2. Nacrati BFS stablo za graf sa slike **Slika-2b,** početi iz čvora A.

Zad 3. Pokazati da putanja između bilo kog čvora i korena u BFS stablu je najkraća putanja između korena i tog čvora.

Zad 4. Dodati novi metod u klasu *AbstractGraph* koji pronalazi putanju između dva čvora grafa. Metod ima sledeće zaglavlje:

```
public List<List<Integer>> getConnectedComponents();
```

```
public List<Integer> getPath(int u, int v);
```

Metod vraća listu tipa *List<Integer>* koja sadrži sve čvorove na putanji od *u* do *v* u ovom redosledu. Korišćenjem BFS pristupa, moguće je dobiti kao rezultat najkraću putanju od *u* do *v*. Ukoliko ne postoji najkraća putanja od *u* do *v*, metod treba da vrati *null*.

Zad 5. Dodati novi metod u klasi *AbstractGraph* da bi odredili da li postoji ciklična putanja u grafu.

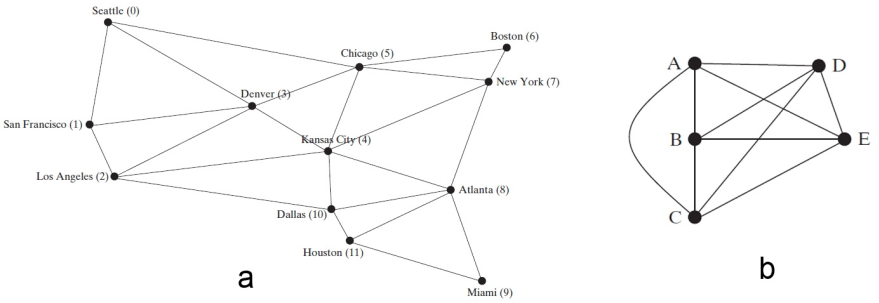
Zad 6. Dodati novi metod u klasu *AbstractGraph* koji pronalazi cikličnu putanju. Metod ima sledeće zaglavlje:

```
public List<Integer> getACycle(int u);
```

Metod vraća listu tipa *List* koja sadrži sve čvorove u cikličnoj putanji, počevši od čvora *u*. Ukoliko graf ne sadrži cikličnu putanju metod vraća *null*.

Zad 8. Napisati program koji čita podatka o povezanom grafu iz fajla. Graf je smešten u fajlu (čvorovi, lista povezanosti). Program treba da zahteva od korisnika da unese naziv fajla, zatim dva čvora, a kao rezultat treba da prikaže najkraću putanju između ova dva čvora

Zad 9. Modifikovati *GraphView* klasu to tako da se doda novo polje **tree** sa set metodom. Ivice stabla treba prikazati crvenom bojom. Program treba da zahteva grad od kojeg se pravi BFS stablo.



Slika 7.2 Primeri za samostalni rad

Poglavlje 8

Studija slučaja: Problem devet novčića

POSTAVKA PROBLEMA

Zadatak je pronaći minimalni broj pomeranja koji vode do slučaja kada će svi novčiči biti okrenuti glavnom na dole. Problem 9 novčića može biti sveden na problem najkraće putanje

Problem devet novčića je u sledećem. Devet novčića su postavljena u 2D matricu dimenzija 3x3, pri čemu se neke postavljene pismom a neke glavnom na dole (glava i pismo – eng. head and tail). Dozvoljen potez je da se uzme bilo koji novčić koji je okrenut glavom na gore i okrenuti ga na suprotnu stranu, zajedno sa svim novčićima koji su mu susedi (ovo ne uključuje novčiće koji su dijagonalno susedni). Zadatak je pronaći minimalni broj pomeranja koji vode do slučaja kada će svi novčiči biti okrenuti glavnom na dole. Na primer, započeti sa 9 novčića postavljenih kao na slici 1a. Usled obrtanja drugog novčića u poslednjoj vrsti, novčići su postavljeni kao na slici 1b. Nakon što se obrne drugi novčić u prvoj vrsti, svih devet novčića će biti postavljeno glavnom na dole, kao na slici 1c.

H	H	H
T	T	T
H	H	H

(a)

H	H	H
T	H	T
T	T	T

(b)

T	T	T
T	T	T
T	T	T

(c)

Slika 8.1 Problem je rešen kada su svi novčići okrenuti glavom na dole.

Napisaćemo program koji od korisnika zahteva da unese početnu konfiguraciju novćića i koji će da prikaže rešenje, kao što je prikazano u sledećem primeru jednog od izvršavanja programa:

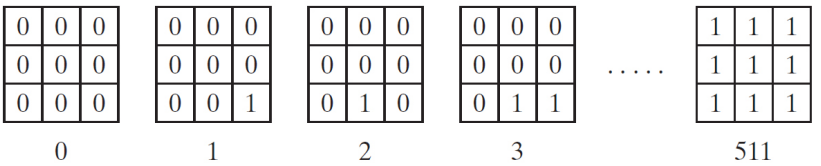
```
Enter the initial nine coins Hs and Ts:  HHHTTTHHH   (ENTER)
The steps to flip the coins are

HHH
TTT
HHH

HHH
THT
TTT
```

```
TTT
TTT
TTT
```

FORMIRANJE GRAFA

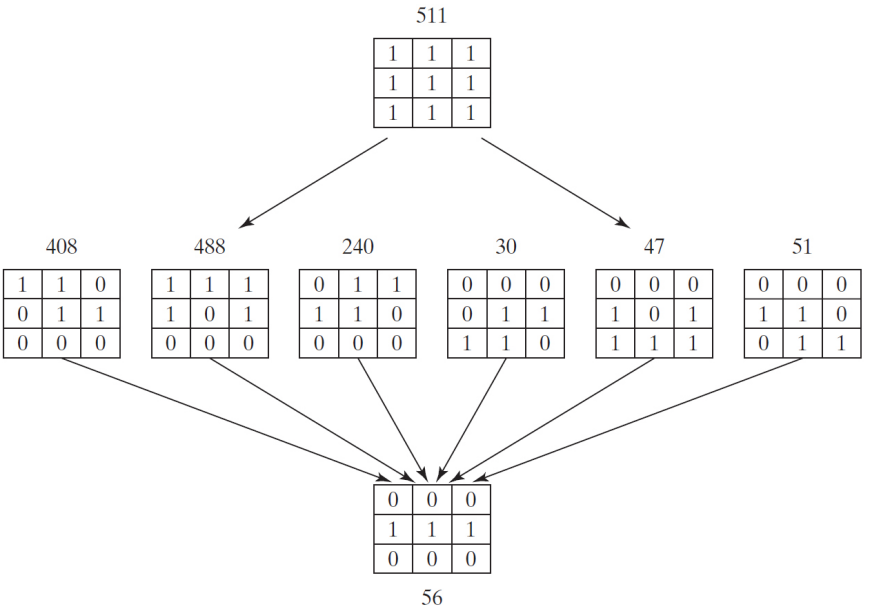
Svako stanje jednog od devet novćića predstavlja čvor u grafu. Pritom dodeljujemo granu između čvorova v i u ukoliko postoji dozovljen potez transformacije iz stanja u u stanje v

Svako stanje jednog od devet novćića predstavlja čvor u grafu. Na primer, tri stanja sa slike 1 odgovaraju trima čvorovima grafa. Radi pogodnosti, koristićemo matricu da bismo predstavili sve čvorove, i koristićemo vrednost 0 za glavu (head), a 1 za pismo (tail). Pošto imamo 9 polja i svako polje matrice može biti **0** ili **1**, postoji ukupno 512 čvorova u grafu označenih sa **0**, **1**, . . . , i **511**, kao što je prikazano na slici 2.



Slika 8.2 Postoji ukupno 512 čvorova u grafu označenih indeksima: 0, 1, 2, . . . , 511.

Dodeljujemo granu između čvorova*v* i *u* ukoliko postoji dozovljen potez transformacije iz *u* u *v*. Slika 3 prikazuje deo grafa. Primetimo da postoji grana između čvorova *511* i *47*, pošto je moguće obrnuti ćeliju iz čvora *47* tako da dobijemo čvor *511*. Poslednji čvor a slike 2 predstavlja konačno stanje devet novčića postavljenih glavom na dole. Ovakvo stanje ćemo označiti kao ciljni čvor (target node). Stoga, ciljni čvor je označen sa *511*. Pretpostavimo da početno stanje novčića u Problemu 9 novčića odgovara čvoru *s*.



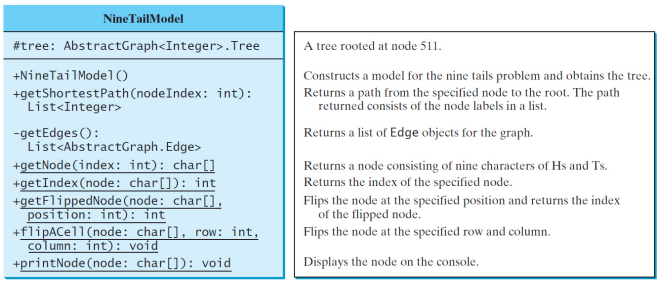
Slika 8.3 Ako čvor u postane čvor v usled obrtanja ćelija, dodeliti usmerenu granu između čvorova v i u.

Problem se svodi na određivanje najkraće putanje od čvora *s* do ciljnog čvora (*511*), što je identično određivanju puta od čvora s do ciljnog čvora pomoću BFS obuhvatnog stabla čiji je koren u ciljnom čvoru.

KLASA NINETAILMODES

Jednom kad je graf kreiran formira se obuhvatno BFS stablo sa korenom u ciljnom čvoru. Klasa NineTailModel sadrži metode za dobijanje najkraće putanje od ciljnog do bilo kog čvora

Sada se zadatak svodi na kreiranje grafa koji sadrži 512 čvorova označenih labelama 0, 1, 2, . . . , 511, i grane koje povezuju čvorove. Jednom kad je graf kreiran, formirati obuhvatno BFS stablo sa korenom u čvoru 511. Na osnovu BFS stabla, možemo pronaći najkraću putanju od korena do bilo kog čvora. Kreiraćemo klasu *NineTailModel*, koja sadrži metode za dobijanje najkraće putanje od ciljnog čvora do bilo kog čvora u grafu. UML dijagram klase je prikazan na slici 4



Slika 8.4 UML dijagram klase NineTailModel koja modelira problem devet novčića korišćenjem grafova.

Vizuelno, čvor je prestavljen pomoću 3 * 3 matrice sa slovima *H* (head) i T (tail). U programu, korišćen je jednodimenzionalni niz od devet karaktera da se predstavi čvor. Na primer, čvor 1 sa slike 2 je predstavljen kao {'H', 'H', 'H', 'H', 'H', 'H', 'H', 'H', 'T'}.

Metod *getNode(index)* vraća čvor na specifičnom indeksu. Na primer, metod *getNode(0)* vraća čvor koji ima sva polja *H*. Metod *getNode(511)* vraća čvor koji ima sva polja *T*.

Metod *getFlippedNode(char[] node, int position)* obrće vrednosti polja na određenoj poziciji, i sve njegove susede, tako da kao rezultat vraća indeks čvora koji se dobija nakon obrtanja odgovarajućih ćelija. Na primer, za čvor *56* sa slike 3, obrtanjem novčića na poziciji *0*, dobiće se čvor *51*. Ukoliko se za čvor *56* obrne novčić na poziciji *1*, dobiće se čvor *47*.

Metod *flipACell(char[] node, int row, int column)* obrće novćić koji se nalazi na odgovarajućem polju određenom indeksom vrste i kolone. Na primer ako se za čvor *56* obrne novčić u nultoj vrsti i nultoj koloni dobiće se čvor *408*. Ukoliko se za čvor *56* obrne novčić u drugoj vrsti i nultoj koloni dobiće se čvor *30*.

IMPLEMENTACIJA KLASE NINETAILMODEL

Klasa NineTailModel sadrži promenljivu tree koja čuva obuvatno stablo grafa dobijeno BFS obilaskom grafa. Pritom je ciljni čvor sa indeksom 511 proglašen za koren obuhvatnog stabla

U nastavku je dat primer implementacije klase *NineTailModel*

```
import java.util.*;

public class NineTailModel {
  public final static int NUMBER_OF_NODES = 512;
  protected AbstractGraph<Integer>.Tree tree; // Define a tree

  /** Construct a model */
  public NineTailModel() {
    // Create edges
    List<AbstractGraph.Edge> edges = getEdges();

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

    // Obtain a BSF tree rooted at the target node
    tree = graph.bfs(511);
  }

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

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

    return edges;
  }

  public static int getFlippedNode(char[] node, int position) {
    int row = position / 3;
    int column = position % 3;
```

```
    flipACell(node, row, column);
    flipACell(node, row - 1, column);
    flipACell(node, row + 1, column);
    flipACell(node, row, column - 1);
    flipACell(node, row, column + 1);

    return getIndex(node);
  }

  public static void flipACell(char[] node, int row, int column) {
    if (row >= 0 &amp;amp;&amp;amp; row <= 2 &amp;amp;&amp;amp; column >= 0
&amp;amp;&amp;amp; column <= 2) {
    // Within the boundary
    if (node[row * 3 + column] == 'H')
        node[row * 3 + column] = 'T'; // Flip from H to T
    else
        node[row * 3 + column] = 'H'; // Flip from T to H
    }
  }

  public static int getIndex(char[] node) {
    int result = 0;

    for (int i = 0; i < 9; i++)
    if (node[i] == 'T')
        result = result * 2 + 1;
    else
        result = result * 2 + 0;

    return result;
  }

  public static char[] getNode(int index) {
    char[] result = new char[9];

    for (int i = 0; i < 9; i++) {
    int digit = index % 2;
    if (digit == 0)
        result[8 - i] = 'H';
    else
        result[8 - i] = 'T';
    index = index / 2;
    }

    return result;
  }

  public List<Integer> getShortestPath(int nodeIndex) {
    return tree.getPath(nodeIndex);
  }

  public static void printNode(char[] node) {
    for (int i = 0; i < 9; i++)
```

```
    if (i % 3 != 2)
        System.out.print(node[i]);
    else
        System.out.println(node[i]);

    System.out.println();
  }
}
```

Konstruktor klase kreira graf sa *512* čvorova, i svaku granu na osnovu mogućih transformacija iz jednog u drugi čvor. Na osnovu grafa se kreira obuhvatno BFS stablo sa korenom u čvoru *511*.

Da bi kreirali grane, metod *getEdges* proverava svaki čvor *u* sa ciljem da vidi da li on obrtanjem može da se pretvori u čvor *v*. Ukoliko je moguće, dodati par *(v, u)* u listu grana, odnosno u niz *Edge*. Metod *getFlippedNode(node, position)* pronalazi indeks čvora koji se dobija obrtanjem neke *H* ćelije i njenih suseda u okviru čvora. Metod *flipACell(node, row, column)* ustvari obrće *H* ćeliju i njene susede u okviru čvora.

Metod *getIndex(node)* je implementiran na sličan način kao i pretvaranje iz binarnog u decimalni broj, pri čemu je node niz karaktera. Metod *getNode(index)* vraća čvor (niz karaktera) koji se sastoji iz slova *H* i *T*.

Metod *getShortestpath(nodeIndex)* poziva *getPath(nodeIndex)* sa ciljem da dobije najkraću putanju od odgovarajuće čvora do ciljnog čvora u grafu.

TESTIRANJE PROGRAMA

Korisnik unosi početnu konfiguraciju novčića u 3x3 matrici, a kao rezultat dobija najkraću putanju odnosno korake do ciljne konfiguracije gde su svi novčići okrenuti glavom na dole

Naredni listing od korisnika traži da unese početnu konfiguraciju devet novčića i kao rezultat štampa putanju od početnog do ciljnog čvora.

```
import java.util.Scanner;

public class NineTail {
  public static void main(String[] args) {
    // Prompt the user to enter nine coins H and T's
    System.out.print("Enter the initial nine coins Hs and Ts: ");
    Scanner input = new Scanner(System.in);
    String s = input.nextLine();
    char[] initialNode = s.toCharArray();

    NineTailModel model = new NineTailModel();
    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()));
  }
}
```

Program od korisnika traži da unese početni čvor koji se sastoji od 9 slova dobijenih kombinacijom karaktera *H* i *T*, u obliku stringa, zatim pretvra string u niz karaktera, kreira graf i obuhvatno BFS stablo, određuje najraću putanju od početnog do ciljnog čvora, i prikazuje spisak čvorova na putanji.

ZADACI ZA SAMOSTALNI RAD

Uraditi samostalno sledeće zadatke:

Zadatak 1. Šta se dešava posle izvršenja naredbe **getIndex("HTHTTTHHH".toCharArray())**? Šta vraća poziv **getNode(46)** in *NineTailModel*?

Zadatak 2. Ako se lnije 26 i 27 zamene u **NineTailModel**, da li će program i dalje da radi? Zašto?

Zadatak 3. U problemu 9 novčića, kada se baci novčić, horizontalna i vertikalno susedna polja se takođe menjaju. Izmeniti program, tako da se sva susedna polja promene, što znači uključiti i dijagonalno susedna.

Zadatak 4. *NineTail* predstavlja rešenje za problem 9 novčića. Izmeniti problem tako da se rešava za **4 x** 4, tj. 16 novčića. Treba paziti da možda neće postojati rešenje za početne konfiguracije. Ako je to slučaj, prijaviti da rešenje ne postoji.

Zadatak 5. Problem četiri novčića koristi **3 * 3** matricu. Neka postoji 16 novčića u 4 x 4 matrici. Napisati program koji će pronaći broj početnih konfiguracija novčića koje nemaju rešenje.

Zaključak

REZIME

Na osnovu svega obrađenog možemo izvesti sledeći zaključak:

Graf je apstraktni tip podataka koji se sastoji iz skupa čvorova (temena) i skupa grana koje povezuju čvorove. Osnovni termini kod grafova su: čvorovi, grane, susedi, usmerenost grane, težina grane, samostalna petlja, paralelne grane, povezani graf, podgraf, ciklus, obuhvatno stablo.

Graf može biti usmeren ili neusmeren. U usmerenom (directed) grafu, svaka grana ima smer koji određuje da li je moguće pomeriti se kroz tu granu od jednog do drugog čvora. Graf takođe može biti bestežinski i težinski. Težinski graf (weighted graph) je onaj u kome svaka grana ima težinu putovanja između čvorova. Težina može predstavljati rastojanje, potrošnju energije, troškove, itd.

Čvorovi grafa mogu biti predstavljeni korišćenjem niza ili povezane liste. Takođe, čvorovi mogu biti objekti bilo kog tipa podatka. Grane grafa mogu biti predstavljene korišćenjem: niza grana, objekata specijalno kreiranih za grane, dok su najuobičajeni načini predstavljanja: matrica susedstva i lista susedstva.

Graf može biti predstavljen kvadratnom matricom čiji je red jednak broju čvorova grafa. Element a_{ij} jednak je broju grana koje polaze iz čvora v_i, a završavaju se u čvoru v_j. Ova matrica se naziva matrica susedstva. Da bi predstavili grane korišćenjem liste susedstva neophodno je kreirati niz povezanih listi. Niz će imati n ulaza, a svaki element niza je povezana listi.

Obilazak grafa u dubinu i obilazak u širinu su dva najuobičajenija načina obilaska čvorova u grafu. Ideja je pronaći sve moguće čvorove putujući nekom od putanja od startnog čvora.

DFS algoritam obilazi dubinu bilo koje posebne putanje pre istraživanja svoje širine. Koristiti stek (ili rekurziju) da bi ste čuvali čvorove koje treba posetiti. *DFS algoritam* je veoma jednostavno implementirati korišćenjem rekurzije. Za svaki neposećeni sused čvora v, metod se rekurzivno poziva da bi se nastavila pretraga po dubini

Kod *BFS* obilaska se koristiti red čekanja da se u njemu smeste čvorovi koji će tek biti posećeni. Ideja je da se za čvor J prvo posete svi susedi čvora J, pa tek onda susedi suseda čvora J. BFS algoritam je veoma jednostavno implementirati korišćenjem redova.

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] https://en.wikipedia.org/wiki/Depth-first_search

[4] https://en.wikipedia.org/wiki/Breadth-first_search

[5] https://www.tutorialspoint.com/data_structures_algorithms/depth_first_traversal.htm

[6] https://www.tutorialspoint.com/data_structures_algorithms/breadth_first_traversal.htm

[7] https://en.wikipedia.org/wiki/Topological_sorting