Projekt iz predmeta *“Analiza i sinteza algoritama”*

Student: Edin Smajić

Smjer: Teorijska kompjuterska nauka (TKN)

Godina: Treća (III)

Status: Redovan

Broj indeksa: 5810/M

Tema 2

Predmetni asistent: mr. Admir Beširević

Uvod:

U prilogu sa ovom dokumentacijom poslat je i folder pod imenom “Projekt” u kojem se nalaze: “graf.cbp”, “graf.h”, “graf.cpp” i “main.cpp”.

“graf.cbp” - fajl koji pri pokretanju automatski otvara ostale navedene fajlove.

“graf.h” - header fajl, u kojem su navedeni prototipovi funkcija, bez implementacija istih, te su u njemu uključeni neki drugi headeri koji su neophodni za ispravan rad projekta.

“graf.cpp” - source fajl, u kojem su implementirane sve funkcije čiji su prototipovi navedeni u “graf.h”.

“main.cpp” - source fajl, u kojem se vrše testiranja tih funkcija, o kojima će ispod biti više govora.

U fajlu “graf.h” definisana je klasa TezinskiGraf koja ima sljedeće atribute (koji su privatni):

1. vector<list<pair<int,double>>> lista_susjedstva – predstavlja listu susjedstva težinskog grafa.
2. bool usmjeren – vodi računa o tome da li je graf usmjeren ili ne. Ima vrijednost true ukoliko je isti usmjeren, u suprotnom mu je vrijednost false.

Također, klasa TezinskiGraf sadrži jedan konstruktor, te funkciju za dodavanje grane:

1. TezinskiGraf(const int &brojVrhova, const bool &usmjeren = false);
   - Setuje atribut usmjeren na usmjeren
   - Mijenja veličinu atributa lista_susjedstva na brojVrhova

2. void dodajGranu(const int &i, const int &j, const double &tezina);
   - Dodaje granu (i, j) sa težinom tezina u graf
   - Dodaje u i-tu listu atributa lista_susjedstva par: {j, tezina}, ukoliko graf nije usmjeren onda također u j-tu listu atributa lista_susjedstva dodaje par: {i, tezina}

Funkcije:

1) bool BelmanFord(const int &pocetni, vector<double> &udaljenosti);
   - Funkcija rađena na vježbama 6, tada je objašnjena i implementirana.
   - Kroz čitav projekt će se vršiti modifikacija ove funkcije na odgovarajuće načine, u ovisnosti od toga šta se kojim zadatkom traži, naravno svaka od modifikacija će biti nova funkcija.

2) vector<double> belmanFordModifikovan(const int &pocetni);
   - Zadatak 24.1-4
   - Funkcija vraća najmanju udaljenost svih vrhova od vrha pocetni.
   - Ukoliko u grafu postoji ciklus ili više njih negativne dužine, tada za sve vrhove tih ciklusa, te za sve vrhove do kojih postoji put iz bilo kojeg od vrhova koji čine neki negativan ciklus, setujemo udaljenost na -inf tj. minus beskonačno, jer se to zadatkom i traži.
   - Obični BelmanFord smo modifikovali na način da nakon što se izvrši standardna procedura za pronalazak udaljenosti svih vrhova do vrha pocetni, još jednom prođemo kroz sve grane grafa, te ukoliko se treba desiti promjena udaljenosti nekog vrha, taj vrh dodamo u vektor oznaceni, nakon toga prođemo kroz sve vrhove koji se nalaze u vektoru oznaceni i pozivamo funkciju DFS_oznaci nad njima, o njoj više odmah ispod.

3) void DFS_oznaci(const int &vrh, vector<double> &udaljenosti);
   - jedina privatna funkcija u projektu
   - ukoliko udaljenost vrha vrh nije već setovana na -inf, onda setujemo tu udaljenost na -inf, te prođemo kroz sve susjede vrha vrh i pozovemo rekurzivno funkciju DFS_oznaci nad svakim susjedom, a ona će setovati njihovu udaljenost na -inf.

4) vector<double> najmanjiUlazeciPut(const int &pocetni);
   - zadatak 24.1-5 *
   - Zadatkom se traži da za svaki vrh **v** u grafu pronađemo najkraći put od bilo kojeg drugog vrha **u**, u grafu, do vrha **v**. (s tim da može biti **u**=**v**)

- Modifikujemo običnu BelmanFord funkciju na način da ćemo sada u vektoru udaljenosti čuvati najkraći put, do svakog vrha. Standardno vanjska petlja ima v-1 iteracija, a unutar nje prolazimo kroz sve grane grafa, ažuriramo udaljenosti[it->first] kada ona bude veća od minimuma trenutne grane **it->second** koja se posmatra i zbira udaljenosti[i]+it->second, jer je možda udaljenosti[i] tj. najmanji put do vrha **i** negativan pa sabran sa **it->second** daje manju vrijednost od same grane it->second, na taj način dobijamo najkraći put do svakog vrha (ukoliko nema ciklusa negativne dužine).

5) vector<int> ciklusNegativneDuzine();
   - Zadatak 24.1-6 *
   - Pokrenemo standardnu proceduru kao i za obični BelmanFord-ov algoritam koji će pronaći namanje udaljenosti svih vrhova do vrha 0, pri tome ćemo pamtiti roditelja od svakog od vrhova tj. pamtit ćemo za svaki vrh **it->first**, vrh **i** koji je „uzrokovao“ promjenu najmanje udaljenosti vrha **it->first** do vrha **0**. Ovo nam je potrebno da bi rekonstruisali eventualni cikus negativne dužine.
   - Nakon toga prolazimo kroz sve grane grafa te ukoliko bude ispunjen uslov promjene udaljenosti nekog od vrhova do vrha 0, to znači da imamo ciklus negativne dužine i da je taj vrh dio tog ciklusa ili vrh do kojeg se može doći iz tog ciklusa, u ovom slučaju nećemo nastaviti prolaziti kroz grane grafa, nego ćemo preći na rekonstruisanje ciklusa. To radimo tako što dodamo taj vrh u vektor ciklus, te idemo na njegovog roditelja pa dodamo njega u ciklus, to radimo sve dok ne budemo trebali dodati u vektor ciklus vrh koji je već u tom vektoru. Međutim ovo nije dovoljno, pa da samo možemo vratiti vektor ciklus, jer trebamo izbaciti iz vektora ciklus vrhove do kojih se samo može doći iz ciklusa, koji nisu sastavni dio ciklusa, a to radimo tako što gledamo roditelje vrhova, ukoliko dva vrha, koji su elementi vektora ciklus, imaju istog roditelja, taj roditelj je nužno dio vektora ciklus. Ostalo je da pronađemo samo još koji od ova dva vrha koji imaju istog roditelja, je i sam roditelj nekom elementu ciklusa, kada pronađemo koji je to vrh, tada drugi vrh brišemo iz ciklusa, jer on nije dio ciklusa, nego se do njega samo može doći iz ciklusa. Nakon što izbacimo sve takve elemente koji nisu dio ciklusa, nego se samo do njih može doći iz elemenata koji čine ciklus, trebamo okrenuti vektor ciklus i vratiti

ga. Ako nema promjene udaljenosti nakon procedure običnog BelmanForda tada ćemo samo vratiti prazan vektor kao znak da ne postoji ciklus negativne dužine.

6) vector<double> rijesiLinearniProgram();

- U knjizi na str. 666, počevši od naslova „**Constraint graphs**“, str. 667 i str. 668 objašnjeno je kako se rješavaju problem tipa A*x <= b. Ključno je da se za ograničenja koji su oblika xi-xj <= bk pravi grana (j,i) težine bk, te da se dodaje dodatni vrh 0 koji ima granu prema svim ostalim vrhovima grafa, a težine tih grana su 0.
- Teorem ***Theorem 24.9*** kaže sljedeće:

  Theorem 24.9

  Given a system $Ax \leq b$ of difference constraints, let $G = (V, E)$ be the corresponding constraint graph. If G contains no negative-weight cycles, then

  $$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \ldots, \delta(v_0, v_n)) \qquad (24.11)$$

  is a feasible solution for the system. If G contains a negative-weight cycle, then there is no feasible solution for the system.
- Dovoljno je dakle pronaći udaljenosti vrha 0 do svih ostalih vrhova, jer će to po teoremu iznad biti rješenje problema A*x <=b, ali samo ukoliko ne postoji ciklus negativne dužine u datom grafu.
- Ova funkcija je rješenje za većinu zadataka iz dijela 24.4-broj, a konkretno u main.cpp su urađeni zadaci: 24.4-1, 24.4-2 i 24.4-6.

7) vector<double> rijesiLinearniProgramSaSpecOgranicenjem();

- Zadatak 24.4-10
- Podsjetimo se da iz teorije koja se nalazi u knjizi u uvodu u problem linearnog programiranja možemo dodati svakoj varijabli rješenja konstantu, a ono će i dalje ostati rješenje.
  Upravo iz tog razloga dodat ćemo novu varijablu xn+1 u naš graf, te kakvu god vrijednost ona imala nakon rješavanja linearnog problema koji uključuje i nju, tu vrijednost ćemo oduzeti od svih ostalih xi-ova rješenja tog linearnog problema.
- Dakle ako je rješenje našeg linearnog problema x1,x2,...,xn,xn+1, konačno rješenje će biti x1-xn+1, x2-xn+1,...,xn-xn+1.

- Kako dodajemo single ograničenje xk <= bk, odnosno -xk <= bk? Prepostavimo ne umanjujući opštost da se to single ograničenje odnosi na varijablu xn, tj. imamo xn <= bn ili -xn <= bn. Tada dodamo novu varijablu xn+1 u naš problem Ax <= b na način da za ograničenje xn <= bn dodamo granu (xn+1, xn) sa težinom bn, jer xn - xn+1 <= bn, a za ograničenje drugog oblika tj. -xn <= bn <=> xn >= -bn dodamo granu (xn, xn+1) sa težinom bn, jer vrijedi xn - xn+1 >= -bn <=> xn+1 - xn <= bn.

8) vector<double> rijesiLinearniProgram2();

- Zadatak 24.4-11
- Radi potpuno ekvivalentno funkciji rijesiLinearniProgram(), s tim da kako su sada elementi bk realni brojevi, prije same procedure rješavanja linearnog programa moramo sve bk svesti na donji cio broj trenutnog bk, u tom slučaju će rješenje takvog modifikovanog problema biti ujedno rješenje i polaznog problema gdje su bk realni brojevi. Zašto je to tako? To je zato što su varijable xi cijeli brojevi, pa će sve njihove razlike također biti cijeli brojevi, a da bi cijeli broj bio manji ili jednak realnom broju potrebno je, a ujedno i dovoljno da bude manji ili jednak od donje granice tog realnog broja.

9) vector<double> rijesiLinearniProgram3();

- Zadatak 24.4-12 *
- Ovdje ne samo da dozvoljavamo da težine grana budu realne vrijednosti, nego da i dio varijabli xi mogu biti realne vrijednosti. Pokrenemo standardnu proceduru BelmanFord-ovog algoritma kao i u funkciji rijesiLinearniProgram(). Jedina razlika je da sada mijenjamo udaljenost vrha **it->first** do vrha **0** samo u slučaju da je donja granica zbira udaljenosti vrha **i** do vrha **0** sa težinom grane (**i**, **it->first**) manji od trenutne udaljenosti vrha **it->first** do vrha **0**, u tom slučaju će vrijediti sva ograničenja xi – xj <= bk.