SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1304

NEKI ALGORITMI ZA RAZAPINJUĆA STABLA U GRAFU

Dominik Novosel

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1304

NEKI ALGORITMI ZA RAZAPINJUĆA STABLA U GRAFU

Dominik Novosel

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1304

Pristupnik:	**Dominik Novosel (1191249229)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	prof. dr. sc. Mario Krnić

Zadatak: **Neki algoritmi za razapinjuća stabla u grafu**

Opis zadatka:

Osnovni cilj ovog rada je implementacija i analiza nekih algoritama za razapinjuća stabla u povezanom grafu. Nakon uvodnog teorijskog dijela, student treba implementirati Kruskalov i Primov algoritam za nalaženje minimalnog razapinjućeg stabla u grafu. Osim toga, potrebno je implementirati algoritam koji pronalazi razapinjuće stablo s minimalnim produktom težina. Također, student treba osmisliti i analizirati algoritam koji daje ukupan broj razapinjućih stabala u zadanom grafu. Konačno, potrebno je navesti i neke primjene razapinjućih stabala u svakodnevnom životu.

Rok za predaju rada: 14. lipnja 2024.

Zahvaljujem mentoru prof. dr. sc. Mariju Krniću na pomoći i usmjeravanju prilikom pisanja završnog rada. Također, zahvaljujem obitelji i prijateljima, a posebno Luki Čulo, Luki Nestiću, Marku Švalju, Danielu Tumiru i Tinu Vračiću na moralnoj podršci.

Sadržaj

Uvod

Razapinjuća stabla predstavljaju osnovni koncept u teoriji grafova, grani matematike koja se bavi proučavanjem grafova. Razapinjuće stablo nekog grafa je podgraf koji sadrži sve vrhove izvornog grafa i predstavlja stablo, što znači da ne sadrži cikluse. U kontekstu računalnih znanosti i teorije optimizacije, igraju ključnu ulogu u raznim algoritmima i primjenama.

Unutar ovog završnog rada usredotočit ćemo se na nekoliko osnovnih algoritama za pronalaženje minimalnih razapinjućih stabala, razapinjuća stabla čija je ukupna težina bridova minimalna u usporedbi s ostalim razapinjućim stablima istog grafa. Takva stabla također imaju mnoge praktične primjene.

Rad će obuhvatiti implementaciju i analizu dva klasična algoritma za pronalaženje minimalnog razapinjućeg stabla: Kruskalov i Primov algoritam. Algoritme ćemo implementirati unutar programskog jezika Python te detaljnije obraditi i analizirati.

Također ćemo istražiti i malo drugačiji problem od pronalaska minimalnog razapinjućeg stabla, a to je pronalazak razapinjućeg stabla s minimalnim produktom težina. Uočit ćemo kako problem nije u potpunosti trivijalan, ali malenim modifikacijama je lako rješiv.

Zanimat će nas i način računanja ukupnog broja različitih razapinjućih stabala u zadanom grafu. Poznavanje tog podatka može biti korisno u analizi složenosti mreža te razumijevanju strukture grafova.

Na kraju ćemo još navesti neke konkretne primjene razapinjućih stabala u stvarnome životu te tako zaokružiti cjelokupnu temu rada.

1. Teorija grafova

1.1. Osnovni pojmovi

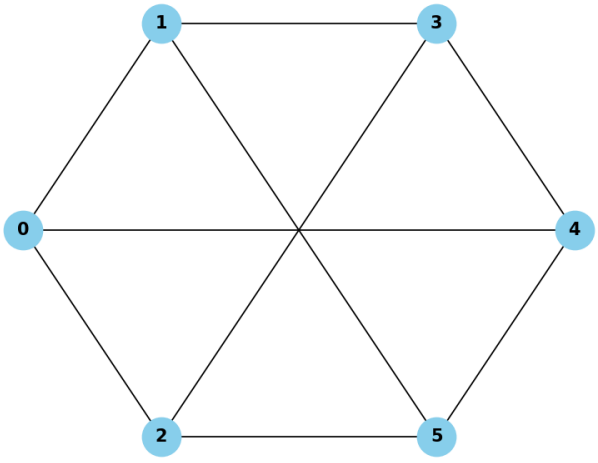
Jednostavan graf G sastoji se od nepraznog konačnog skupa vrhova (čvorova, oznaka V(G)) i konačnog skupa bridova (oznaka E(G)). Formalno ćemo graf pisati kao G = (V, E) ili G = (V(G), E(G)). Neki primjeri jednostavnih grafova su: nul-graf, potpuni graf, ciklički graf, lanac, kotač.

Graf je povezan ako se ne može prikazati kao unija neka dva grafa. U suprotnom kažemo da je graf nepovezan. Svaki se nepovezani graf može prikazati ako unija povezanih grafova. Svaki član unije zovemo komponenta povezanosti.

Podgraf grafa G je graf čiji vrhovi pripadaju skupu V(G), a bridovi skupu E(G).

Šetnja u grafu G je konačan prolazak bridovima oblika $v_0 \rightarrow v_1 \rightarrow ... \rightarrow v_n$ u kojem su svaka dva uzastopna brida ili susjedna ili jednaka. Šetnja u kojoj su svi bridovi različiti zovemo staza, a ako su svi vrhovi, također, različiti onda takvu stazu zovemo put. Za stazu ili put kažemo da su zatvoreni ako je $v_0 = v_n$, odnosno graf sadrži ciklus.

Šuma je graf bez ciklusa, a povezana šuma se naziva stablo. Svojstva stabla S s n vrhova su: S ne sadrži cikluse i ima n – 1 bridova te je povezan graf, a dodavanjem jednog brida dobit ćemo točno jedan ciklus.



Slika 1.1 Primjer povezanog grafa

1.2. Definicija i svojstva razapinjućih stabala

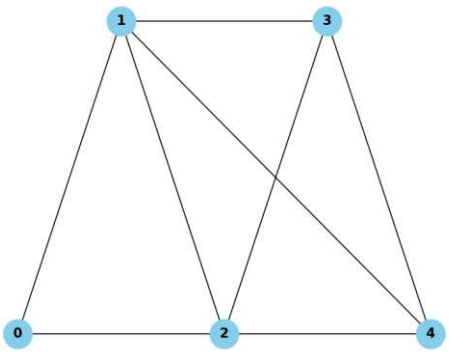
Razapinjuće stablo grafa G je podgraf koji sadrži sve vrhove grafa G i koji je stablo, odnosno povezan i bez ciklusa. Razapinjuća stabla imaju nekoliko važnih svojstava:

- Ako graf ima n vrhova, onda razapinjuće stablo ima n – 1 bridova
- Uklanjanjem jednog brida razapinjućeg stabla, graf će postati nepovezan

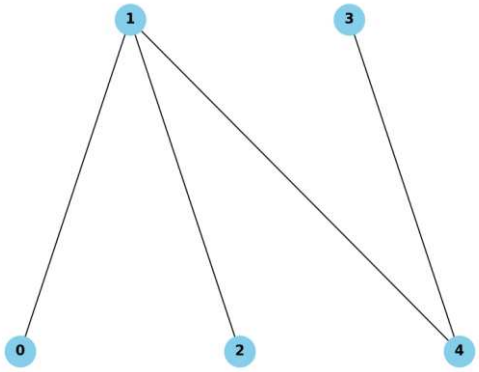
Najjednostavniji postupak dobivanja razapinjućeg stabla jest:

- Zadanom povezanom grafu pronađemo neki ciklus i iz tog ciklusa uklonimo jedan brid te dobijemo G – e
- Ako smo dobili povezan graf bez ciklusa, onda smo gotovi, a inače ponovi postupak iz prvog koraka

Rezultat primjene prethodnih koraka na zadanom grafu (Slika 1.2) može dati nekoliko rješenja, a jedno od njih prikazano je slikom 1.3.



Slika 1.2 Početni graf



Slika 1.3 Razapinjuće stablo početnog grafa

2. Kruskalov algoritam

2.1. Opis algoritma

Kruskalov algoritam je pohlepni algoritam koji daje rješenje problema pronalaska minimalnog razapinjućeg stabla u težinskom grafu. Algoritam funkcionira po principu dodavanja bridova prema rastućem redoslijedu težina pazeći da dodani brid ne stvara ciklus s prethodno odabranim bridovima.

Konstrukcijski postupak Kruskalovog algoritma:

- Sortiraj sve bridove po težini
- Neka je e_1 brid najmanje težine
- Definiramo e_2, e_3, ..., e_{n-1} uzimajući u svakom idućem koraku brid najmanje težine, uzimajući u obzir da taj brid ne stvara ciklus s prije odabranim bridovima e_i

Traženo razapinjuće stablo je podgraf početnog grafa sastavljen od bridova e_1, e_2, ..., e_{n-1}

2.1.1. Dokaz Kruskalovog algoritma

Kako dobiveni graf S ima n-1 bridova te nema ciklusa, S je stablo s n vrhova te zaključujemo da je S razapinjuće stablo. Potrebno je još dokazati da je Kruskalovim algoritmom dobiveno minimalno razapinjuće stablo.

$w(S) = w(e_1) + w(e_2) + ... + w(e_{n-1})$

Pretpostavimo da postoji neko drugo razapinjuće stablo T s težinom bridova manjom od S ($w(T) < w(S)$). Neka je e_k prvi brid niza e_1, e_2, ..., e_{n-1} koji čini stablo S, ali ne i T.

Pogledamo li skup bridova $T \cup \{e_k\}$ uočavamo da takav graf sadrži ciklus C te se u tom ciklusu nalazi i brid e_k. Također zaključujemo kako C sadrži brid e kojeg S ne sadrži, ali T sadrži jer u suprotnom bi nastala kontradikcija s tvrdnjom da je S stablo. Izbacujemo iz C takav brid e te dobivamo novo razapinjuće stablo T'. Iz ovoga slijedi da je $w(e_k) \leq w(e)$ jer smo e_k prethodno odabrali prema načelu minimalnosti, te dobivamo i $w(T') \leq w(T)$. Ponavljanjem postupka izmijenit ćemo stablo T i pretvoriti ga u stablo S, smanjujući mu težinu. Na kraju ćemo dobiti $w(S) \leq w(T)$ što je u kontradikciji s početnom pretpostavkom.

2.2. Implementacija algoritma

U nastavku je prikazan dio koda implementiran za rad Kruskalovog algoritma:

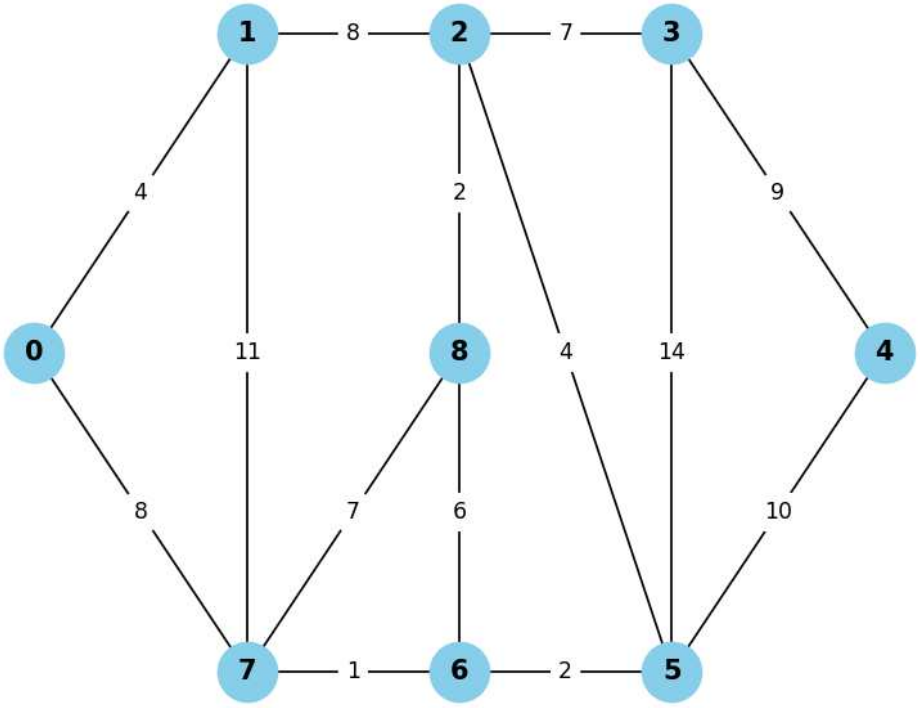
```
def KruskalMST(graph, vertex_edge):
  sorted_vertex_edge = sorted(vertex_edge,key=lambda x: x[2])
  for i in sorted_vertex_edge:
      if provjera_ciklusa(graph, i[0], i[1]):
            graph.append((i[0], i[1], i[2]))
      else:
            continue
  return graph
```

Funkcija prima dva parametra, praznu list *graph* te listu *vertex_edge* sastavljenu od trojki oblika (prvi_vrh, drugi_vrh, tezina_brida) koje predstavljaju bridove početno zadanog grafa. Varijabla *sorted_vertex_edge* predstavlja listu bridova sortiranu prema težini bridova. Koristeći *for* petlju prolazimo po tako sortiranim bridovima te ih dodajemo ako taj brid ne stvara ciklus (provjera tvori li neki brid ciklus izvršava se pomoću funkcije *provjera_ciklusa*). Nakon prolaska kroz cijelu *for* petlju lista *graph* sadržavat će tražene bridove koje vraćamo pomoću *return*.

Detaljnija analiza i objašnjenje cjelokupnog programa nalazi se unutar komentara u kodu, komentar započinje oznakom '#' te završava na kraju retka.

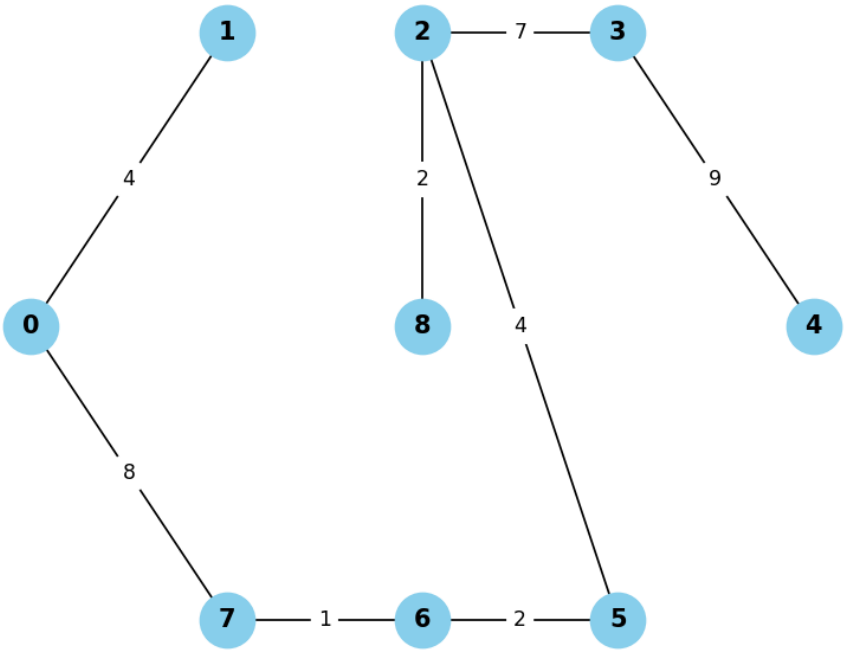
2.3. Primjer rada algoritma

Kako bismo predočili što algoritam radi, prikazat ćemo jedan primjer ulaza i ispisa. Uzmimo za početak graf (Slika 2.1) sa sljedeće definiranim trojkama (*prvi_vrh, drugi_vrh, tezina_brida*), a konkretan primjer pokretanja i ulaza programa prikazan je na slici 2.3 u prva tri retka: (0, 1, 4), (0, 7, 8), (1, 7, 11), (1, 2, 8), (7, 6, 1), (7, 8, 7), (8, 6, 6), (8, 2, 2), (2, 3, 7), (6, 5, 2), (2, 5, 4), (3, 5, 14), (3, 4, 9), (5, 4, 10).



Slika 2.1 Prikaz početnog grafa

Prolaskom kroz cijeli algoritam dobit ćemo novu listu *graph* oblika: [(7, 6, 1), (8, 2, 2), (6, 5, 2), (0, 1, 4), (2, 5, 4), (2, 3, 7), (0, 7, 8), (3, 4, 9)]. (Slika 2.2 i Slika 2.3)



Slika 2.2 Prikaz dobivenog rješenja

```
PS C:\Users\domin\Desktop\FER\sem6\ZAVRAD> python kruskal.py (0, 1, 4) (0, 7, 8) (1, 7, 11) (1
, 2, 8) (7, 6, 1) (7, 8, 7) (8, 6, 6) (8, 2, 2) (2, 3, 7) (6, 5, 2) (2, 5, 4) (3, 5, 14) (3, 4
, 9) (5, 4, 10)
[(7, 6, 1), (8, 2, 2), (6, 5, 2), (0, 1, 4), (2, 5, 4), (2, 3, 7), (0, 7, 8), (3, 4, 9)]
37
```

Slika 2.3 Zadnja dva retka prikazuju dobiveni graf i sumu njegovih težina

2.4. Analiza složenosti algoritma

Glavni koraci te procesi algoritma koje treba gledati kako bismo napravili vremensku analizu algoritma jesu:

- Sortiranje bridova
    - Sortiranje *vertex_edge* ima složenost O(E log E), gdje E predstavlja broj bridova u grafu
- Prolazak kroz bridove te dodavanje novih bridova u stablo
    - Prolazak *for* petlje kroz bridove nakon sortiranja je O(E)
- Provjera ciklusa
    - Svaka provjera stvaranja ciklusa koristi algoritam složenosti O(E * V), gdje V predstavlja broj vrhova, a E broj bridova

Kombiniranjem svih glavnih koraka dolazimo do zaključka kako implementirani algoritam ima vremensku složenost $O(E^2 * V)$.

Prostorna složenost u priloženom algoritmu lako je vidljiva. Kada funkcija *nacrtaj_graf* ne bi bila implementirana, složenost bi bila O(V), međutim za bolji prikaz i razumijevanje ćemo ju ipak koristiti te će složenost biti O(V + E).

3. Primov algoritam

3.1. Opis algoritma

Primov algoritam je drugi, također pohlepni, algoritam za pronalaženje minimalnog razapinjućeg stabla, koji koristi drugačiji pristup od Kruskalovog algoritma. Primov algoritam započinje s jednim vrhom kojeg svojevoljno odabiremo i postupno dodajemo bridove i susjedne vrhove, ne stvarajući cikluse. Uvijek gledamo samo one bridove čiji vrh još nije unutar trenutnog stabla, vrh je susjed od nekog već dodanog vrha u stablu te uvijek dodajemo bridove najmanje težine.

Koraci Primovog algoritma:

- Inicijaliziraj stablo s jednim vrhom
- Pronađi bridove koji povezuju bilo koji vrh stabla sa susjednim vrhovima, gledaj samo one bridove čiji se jedan od vrhova ne nalazi u stablu dok se drugi vrh nalazi
- Odaberi brid najmanje težine unutar tih bridova
- Dodaj brid ako ne stvara ciklus u trenutnom stablu
- Ponovi drugi korak dok stablo ne sadrži sve vrhove

3.2. Implementacija algoritma

Funkcija Primovog algoritma (Kôd 3.1) pomaže za bolje razumijevanje algoritma.

```
def PrimMST(graph, vertex_edge, pocetak):
  closed = set()
  closed.add(pocetak)
  open = []
  while True:
      for i in vertex_edge:
            if (i[0] == pocetak) and (i[1] not in closed):
                  open.append(i)
            if (i[1] == pocetak) and(i[0] not in closed):
```

```
                    open.append(i)
            sorted_open = sorted(open, key=lambda x: x[2])
            index = []
            for ind, i in enumerate(sorted_open):
                  if provjera_ciklusa(graph, i[0], i[1]):
                        graph.append((i[0], i[1], i[2]))
                        index.append(ind)
                        if i[0] not in closed:
                              pocetak = i[0]
                              closed.add(i[0])
                        elif i[1] not in closed:
                              closed.add(i[1])
                              pocetak = i[1]
                        break
                  else:
                        index.append(ind)
                        continue
            index.reverse()
            for i in index:
                  sorted_open.pop(i)
            open = sorted_open
            if len(open) == 0:
                  break
      return graph
```

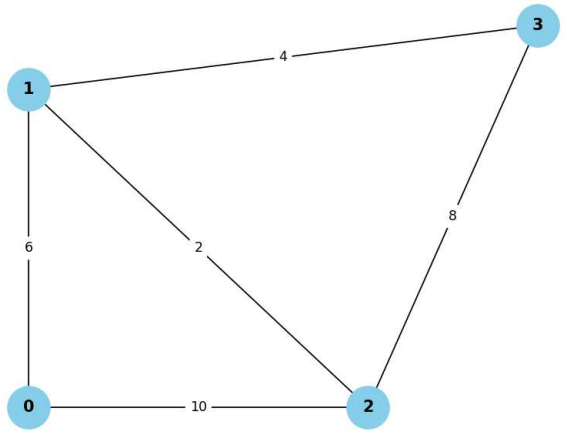
Kôd 3.2 – Funkcija Primovog algoritma

Funkcija prima tri parametra, *graph* koji je prazna lista gdje pohranjujemo rješenje, *vertex_edge* jest lista ekvivalentna kao u Kruskalovom algoritmu (bridovi i težine početnog grafa) te *pocetak* koji nam daje broj početnog vrha. Skup *closed* nam služi kako bismo pamtili koje smo vrhove već stavili unutar našeg stabla, a lista *open* za pamćenje potencijalnih bridova koji se mogu dodati u idućem koraku. Algoritam se ponavlja sve dok ne dodamo sve vrhove u stablo. Za početak pronađemo potencijalne bridove koje ćemo dodati u stablo te ih sortiramo rastućim poretkom prema težini. Zatim među tim bridovima se pronađe onaj koji ne stvara ciklus te je najmanje težine. Ako smo došli do kraja, lista *open* će biti prazna te će algoritam vratiti pronađeno stablo.

Detaljna analiza i objašnjenje programa nalazi se unutar komentara u kodu, komentar započinje oznakom '#' te završava na kraju retka.

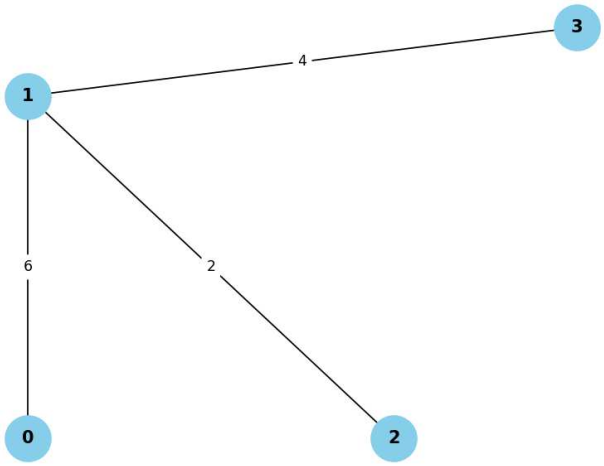
3.3. Primjer rada algoritma

Funkcionalnost algoritma može se provjeriti i konkretnim primjerima. Uzmimo graf (Slika 3.1) sa sljedećim definiranim trojkama (*prvi_vrh, drugi_vrh, tezina_brida*): (0, 1, 6), (0, 2, 10), (1, 2, 2), (1, 3, 4), (2, 3, 8) te uzmimo u obzir da je početni vrh iz kojeg krećemo vrh 0. Prikaz pokretanja programa iz command prompt-a je prikazan na slici 3.2.



Slika 3.1 Početno zadani graf

Izvršavanjem algoritma dobit ćemo napunjenu listu *graph* koja predstavlja minimalno razapinjuće stablo oblika: [(1, 2, 2), (1, 3, 4), (0, 1, 6)], (Slika 3.2).



Slika 3.2 Rješenje Primovim algoritmom

```
PS C:\Users\domin\Desktop\FER\sem6\ZAVRAD> python prim.py (0, 1, 6) (0, 2, 10) (1, 2, 2) (1, 3
, 4) (2, 3, 8)
[(0, 1, 6), (1, 2, 2), (1, 3, 4)]
12
```

Slika 3.3 Primjer ulaza i dobivenog izlaza programa

3.4. Analiza složenosti algoritma

Glavne funkcionalnosti te dijelovi programa koji najviše pridonose vremenskoj složenosti algoritma jesu:

- Provjera ciklusa
    - Ukupna vremenska složenost funkcije *provjera_ciklusa* jest O(E * V), a to je slučaj kada ćemo proći kroz sve vrhove i bridove stabla
- Vanjska petlja Primovog algoritma doprinosi s O(V)
- Sortiranje liste O(E * log E)
- Petlja koja iterira preko svih bridova O(E)

Kombiniranjem svih funkcionalnosti dolazimo do zaključka kako implementirani algoritam ima vremensku složenost O(V * V * E + V * E * logE). Pretpostavimo li da je E proporcionalan V dobivamo da je ukupna vremenska složenost $O(V^3)$.

Ekvivalentno Kruskalovom algoritmu, prostorna složenost jest O(V + E) zbog funkcije za prikaza grafa.

4. Algoritam za minimalni produkt težina

4.1 Opis algoritma

Rješenje problema pronalaska minimalnog produkta težina čini se sličnim kao problem pronalaska minimalnog razapinjućeg stabla, no ispostavlja se da nije. Međutim, malim modifikacijama problem možemo svesti na upravo taj prethodni problem. Naime, primijenimo li logaritamsku transformaciju na težine grafa dobivamo upravo problem pronalaska minimalnog razapinjućeg stabla, zbog svojstava logaritama, te je Primovim algoritmom zadatak lako rješiv.

4.2 Implementacija algoritma

Sljedeći kod je prikaz implementacije u kojoj je prikazano rješenje Primovim algoritmom, ali samo dio pretvorbe u logaritamske vrijednosti te pozivi funkcija:

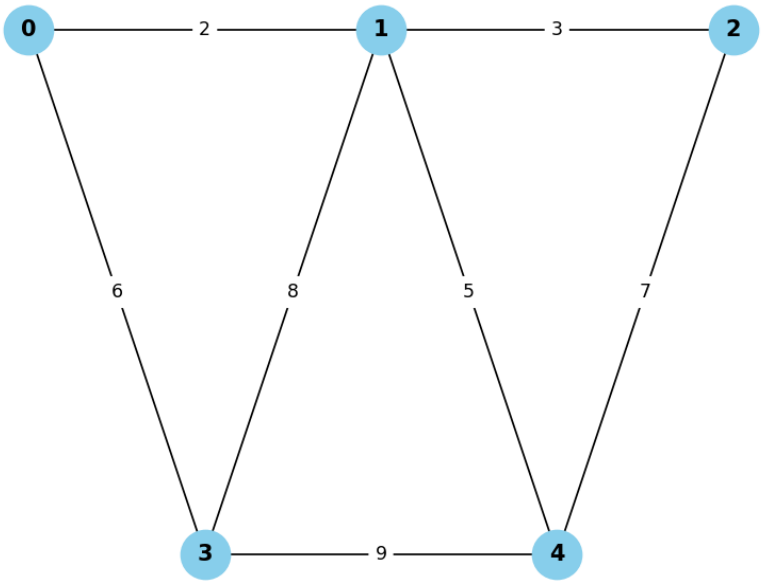
```
def vertex_edge_logarit(vertex_edge, logaritamske_vrijednosti):
      for brid in vertex_edge:
            pomoc = round(math.log(brid[2]), 3)
            logaritamske_vrijednosti.append((brid[0], brid[1], pomoc))

vertex_edge_logarit(vertex_edge, vert_ed_log)
PrimMST(graph, vert_ed_log, 0)
```

Funkcija prima dva parametra, a to su *vertex_edge* lista koja sadrži početno zadani graf, te *logaritamske_vrijednosti* koja je početno prazna lista. Prikazana funkcija će napuniti listu *logaritamske_vrijednosti* s novim (logaritamskim) vrijednostima koje ćemo moći koristiti.

Cjelokupna analiza i objašnjenje programa nalazi se unutar komentara u kodu, iako se program ne razlikuje puno od Primovog algoritma.

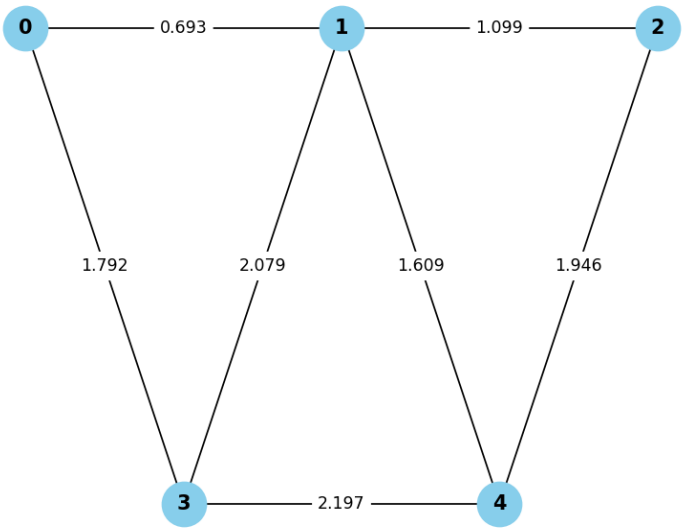
4.3 Primjer rada algoritma

Za kvalitetniju analizu i bolje razumijevanje razmotrimo idući primjer (Slika 4.1) s klasičnim ulazom (Slika 4.4): (0, 1, 2), (1, 2, 3), (0, 3, 6), (1, 3, 8), (4, 3, 9), (1, 4, 5), (4, 2, 7).



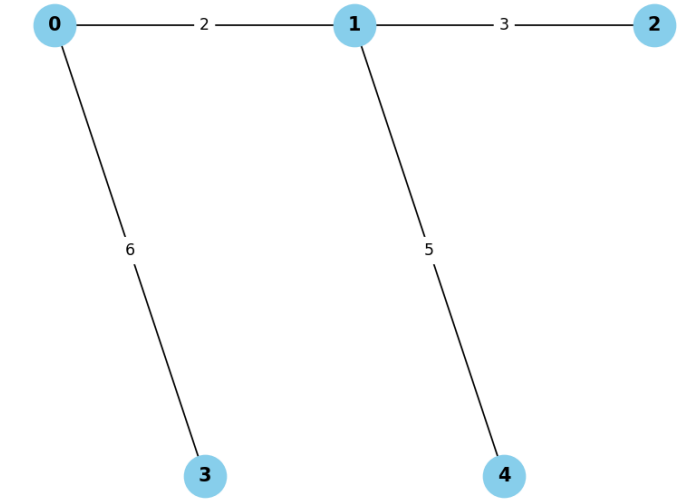
Slika 4.1 Početni graf

Prolaskom kroz funkciju *vertex_edge_logarit* početnom grafu zamjenjuju se težine s logaritamskim vrijednostima (Slika 4.2).



Slika 4.2 Težine zamijenjene logaritamskim vrijednostima

Kako sada na dobivenom grafu možemo koristiti poznate algoritme za pronalazak minimalno razapinjućeg stabla, koristeći Primov algoritam dobivamo rješenje (Slika 3.3).



Slika 4.3 Rješenje Primovim algoritmom

Iz dobivenog grafa sada lako možemo iščitati rješenje našeg problema, pomnožimo sve težine vidljive na posljednjem grafu (6 * 2 * 5 * 3 = 180). (Slika 4.4)

```
PS C:\Users\domin\Desktop\FER\sem6\ZAVRAD> python min_product.py (0, 1, 2) (1, 2, 3) (0, 3, 6)
 (1, 3, 8) (4, 3, 9) (1, 4, 5) (4, 2, 7)
[(0, 1, 2), (1, 2, 3), (1, 4, 5), (0, 3, 6)]
180
```

Slika 4.4 Ulaz i izlaz algoritma pokrenut iz command prompt-a

4.4 Analiza složenosti

Kako su korištene iste funkcije kao i u Primovom algoritmu, osim funkcije *vertex_edge_logarit* čija je složenost O(E), što je u našem slučaju zanemarivo za cjelokupnu vremensku složenost, zaključujemo da je $O(V^3)$ vremenska složenost prikazanoga algoritma.

Prostorna složenost algoritma jest O(V + E) zbog prolaska listom početnih bridova te izmjenom njihovih težina.

5. Algoritam za broj razapinjućih stabala

5.1. Opis algoritma

Cayleyev teorem govori ako postoji točno n^{n-2} različitih označenih stabala s n vrhova. Ovaj teorem je koristan i govori nam podatak o sveukupnom broju razapinjućih stabala ako je zadani graf s n vrhova potpun.

Kirchhoffov teorem nam daje način za izračun broja razapinjućih stabala, na svim grafovima, kao determinanta posebne matrice.

Konstrukcijski postupak dobivanja broja razapinjućih stabala može se prikazati u nekoliko koraka:

- Kreiranje matrice susjedstva zadanog grafa
- Sve dijagonalne elemente potrebno je zamijeniti sa stupnjem čvora, ako vrh 1 ima dva susjedna vrha onda ćemo na mjesto (1, 1) u matrici zapisati broj 2
- Sve ne dijagonalne elemente koji su različiti od 0 potrebno je zamijeniti brojem -1
- Izračunaj kofaktor za bilo koji element dobivene matrice
- Dobiveni kofaktor jest ukupan broj razapinjućih stabala za zadani graf

Determinanta matrice dobiva se uklanjanjem bilo kojeg retka i stupca iz Laplaceove matrice. Determinanta matrice je uvijek ista (do na predznak), neovisno koji smo redak i stupac uklonili.

5.2. Implementacija algoritma

Zbog veličine pojedinih funkcija implementiranih u programu, prikazan je samo poziv tih funkcija i opis što se dobije njihovim izvršavanjem.

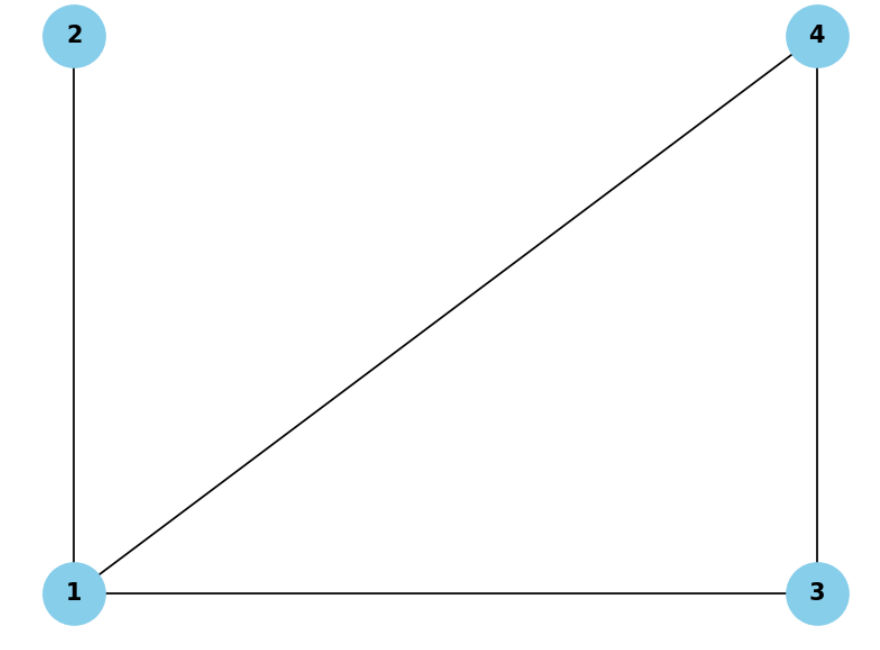
```
stvori_matricu(matrica, len(pamti_vrhove), vertex_edge)
preoblikuj_matricu(matrica)
```

```
izracunaj_broj_razapinjucih_stabala(matrica, 0, 0)
```

Pozivom prve funkcije listu *matrica* punimo s listom redova matrice, odnosno dobivamo matricu susjedstva iz početnog grafa. Druga funkcija radi istu zadaću kao drugi i treći korak algoritma, popunjava dijagonalu sa stupnjem čvora, a ostale elemente zamjenjuje s -1. Zadnjom funkcijom se izvodi četvrti korak algoritma gdje se prvo ukloni neki redak i stupac (prikazanim pozivom se uklanja prvi redak i prvi stupac matrice) te se određenom manipulacijom nad redovima matrica dovede u gornje trokutasti oblik za lagano izračunavanje determinante, odnosno rješenje traženog problema.

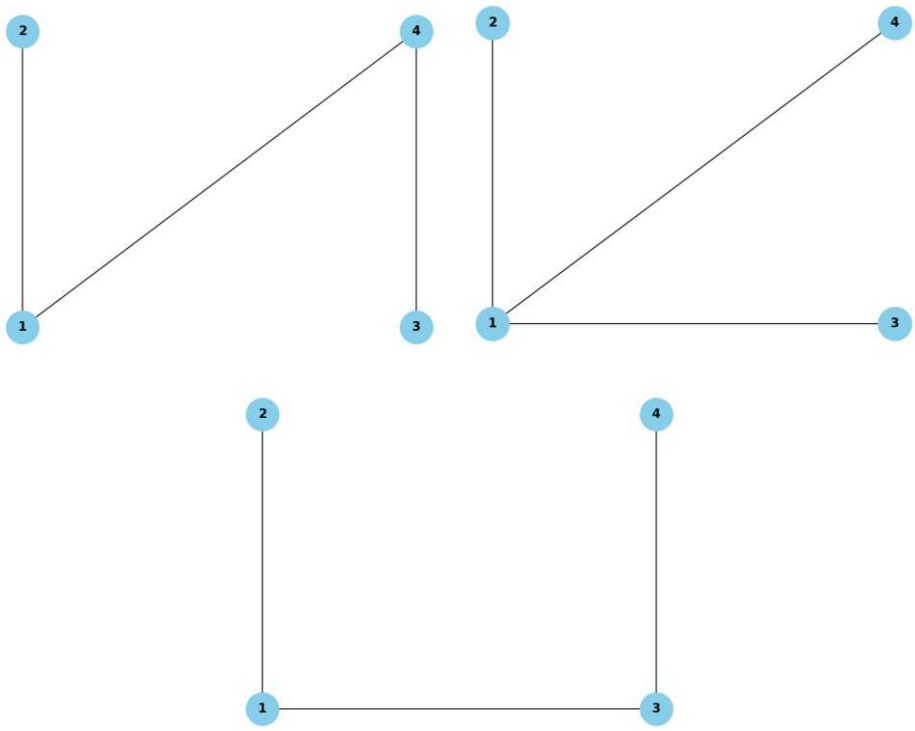
5.3. Primjer rada algoritma

Za razliku od prethodnih algoritama, težina bridova nije bitna te će zato ulazni podaci biti oblika (*prvi_vrh, drugi_vrh*). Uzmimo za primjer graf koji nije potpun (Slika 5.1): (1, 2), (1, 3), (1, 4), (3, 4).



Slika 5.1 Grafički prikaz ulaza

Analizirajući graf lako je vidljivo da je broj mogućih razapinjućih stabala tri, a to su:



Implementirani algoritam će zaista pronaći točan broj razapinjućih stabala. Zadatak je riješen na jednostavnom primjeru kako bi bilo očito rješenje svima, ali algoritam i za daleko složenije grafove daje točna rješenja. Ako bismo isprobali rješenje za primjer grafa ekvivalentnog onomu iz Kruskalovog primjera dobili bismo broj 662 (Slika 5.2).

```
PS C:\Users\domin\Desktop\FER\sem6\ZAVRAD> python kirchhoff_tm.py (0, 1) (0, 7) (1, 7) (1, 2)
(7, 6) (7, 8) (8, 6) (8, 2) (2, 3) (6, 5) (2, 5) (3, 5) (3, 4) (5, 4)
662
```

Slika 5.2 Pokretanje programa za graf kao na slici 2.1

5.4. Analiza složenosti algoritma

Program koristi matrično množenje za pronalazak rješenja problema koji traje $O(V^3)$, izvodi se N puta (potencija matrice susjedstva) te je ukupna vremenska složenost algoritma $O(V^3 * \log N)$.

Prostorna složenost određena je veličinom matrice susjedstva i matrice rezultata. Znamo da su matrice dimenzije V x V pa je prostorna složenost programa $O(V^2)$.

6. Primjene razapinjućih stabala

U svakodnevnom životu imamo nekoliko primjena razapinjućih stabala koje se koriste:

- Dizajn mreže
    - Minimizacija troškova veza odabirom najjeftinijih bridova
- Obrada slike
    - Identifikacija područja sličnog intenziteta ili boje, korisno za zadatke segmentacije i klasifikacije
- Analiza društvenih mreža
    - Identifikacija važnih veza i odnosa među pojedincima ili grupama
- Navigacijski sustavi
    - Pronalazak optimalnih ruta između više lokacija za minimizaciju vremena putovanja ili udaljenosti
- Projektiranje sustava za opskrbu vodom
    - Izgradnja mreže cijevi za distribuciju vode kako bi se minimizirali troškovi instalacije i održavanja
- Logističke mreže
    - Optimizacija ruta za dostavu robe kako bi se smanjili troškovi transporta i vrijeme isporuke

Zaključak

Ovim radom pružen je detaljan pregled algoritama za pronalazak minimalnog razapinjućeg stabla grafova, s naglaskom na Kruskalov i Primov algoritam. Kroz analizu složenosti uočavaju se prednosti i mane svakog algoritma.

Kirchhoffovim teoremom i Laplaceovim matricama pokazalo se efikasno i pouzdano pronalaženje broja razapinjućih stabala, što otvara dodatne mogućnosti za daljnja istraživanja u teoriji grafova.

Dodatnim modifikacijama nekih teških problema, zadatak se može svesti na trivijalnu primjenu Kruskalovog, odnosno Primovog algoritma kao što se to može vidjeti na problemu pronalaska minimalnog produkta težina razapinjućeg stabla.

Iako su obrađeni algoritmi učinkoviti za manje grafove, uviđamo potrebu za efikasnijim tehnikama za veće grafove.

Razapinjuća stabla pokazala su svoju važnost u raznim primjenama, čineći ih neizostavnim alatom u svakodnevnom životu.

Literatura

[1] D. Kovačević, M. Krnić, A. Nakić, M. O. Pavčević, *Diskretna matematika 1*

[2] Poveznica: https://www.geeksforgeeks.org/spanning-tree/

[3] Poveznica: https://www.geeksforgeeks.org/what-is-minimum-spanning-tree-mst/

[4] Poveznica: https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/

[5] Poveznica: https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/

[6] Poveznica: https://www.geeksforgeeks.org/minimum-product-spanning-tree/

[7] Poveznica: https://www.geeksforgeeks.org/videos/kirchhoffs-theorem-for-calculating-number-of-spanning-trees-of-a-graph/

Sažetak

Neki algoritmi za razapinjuća stabla u grafu

Rad istražuje različite algoritme pronalaska razapinjućih stabala u grafovima poput Kruskalovog, Primovog te algoritma za minimalni produkt. Analizira se vremenska i prostorna složenost svakog algoritma te se uspoređuju prednosti i nedostaci.

Također su prikazani primjeri stvarne implementacije u svakodnevnom životu. Kroz teorijsku podlogu i praktične primjere rad daje sveobuhvatan pregled funkcioniranja i korištenja ovih algoritama za rješavanje konkretnih problema.

Za svaki algoritam i obrađeno poglavlje prikazan je primjer pokretanja konkretnog programa te dobiveni ispisi i grafovi.

Ključne riječi korištene u završnome radu:

- Razapinjuće stablo
- Graf
- Kruskalov algoritam
- Primov algoritam
- Minimalni produkt
- Svakodnevne primjene

Summary

Algorithms for Spanning Trees in Graphs

This paper explores various algorithms for finding spanning trees in graphs, such as Kruskal's algorithm, Prim's algorithm, and the minimum product algorithm. It analyzes the time and space complexity of each algorithm, comparing their advantages and disadvantages.

Additionally, examples of real-world implementation in everyday life are presented. Through theoretical background and practical examples, the paper provides a comprehensive overview of the functioning and usage of these algorithms for solving specific problems.

For each algorithm and discussed chapter, an example of running a specific program is provided, along with the resulting outputs and graphs.

Keywords used in the thesis**:**

- Spanning tree
- Graph
- Kruskal's algorithm
- Prim's algorithm
- Minimum product
- Everyday applications