Prebrojavanje razapinjućih stabala grafa

Marković, Bruno

Undergraduate thesis / Završni rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Electrical Engineering and Computing / Sveučilište u Zagrebu, Fakultet elektrotehnike i računarstva**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:168:969031

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-03-20**

Repository / Repozitorij:

FER Repository - University of Zagreb Faculty of Electrical Engineering and Computing repozitory

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1611

PREBROJAVANJE RAZAPINJUĆIH STABALA GRAFA

Bruno Marković

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1611

PREBROJAVANJE RAZAPINJUĆIH STABALA GRAFA

Bruno Marković

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1611

Pristupnik:	**Bruno Marković (0036542159)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	prof. dr. sc. Mario Osvin Pavčević

Zadatak: **Prebrojavanje razapinjućih stabala grafa**

Opis zadatka:

Koliko razapinjućih stabala ima zadani graf je općenito teško pitanje koje se ne može riješiti tehnikama elementarnog prebrojavanja. Za neke klase grafova taj je broj poznat, dok je za neke iznimno težak za prebrojati. Zadatak u ovom radu je razviti vlastiti program koji prebrojava sva različita razapinjuća stabla zadanog grafa. Taj program treba moći razapinjuća stabla konstruirati i prebrojati, a ne samo prebrojati ih. Dobiveni rezultat treba usporediti s poznatim Kirchhoffovim teoremom koji također daje traženi broj. Program treba optimirati kako bi u što kraćem vremenu dao rezultate za grafove s većim brojem vrhova. Analizirati vrijeme izvođenja s obzirom na strukturu grafa.

Rok za predaju rada: 14. lipnja 2024.

Sadržaj

1. Stabla **3**

1.1. Problem broja razapinjućih stabala 3

1.2. Općenito o stablima 3

1.2.1. Razapinjuća stabla 4

1.3. Povijesni pregled 4

1.4. Formule za poznate grafove 5

1.5. Vrste algoritama za pretraživanje 6

1.5.1. Metoda testiraj i odaberi 6

1.5.2. Metoda elementarnih transformacija stabla 7

1.5.3. Metoda sukcesivnog smanjivanja grafa 7

2. Korišteni algoritmi i implementacija **8**

2.1. Kirchhoffova formula 8

2.1.1. Postupak 9

2.1.2. Implementacija 11

2.1.3. Vremenska i prostorna složenost 11

2.2. Algoritam 1 12

2.2.1. Pseudokod 14

2.2.2. Vremenska i prostorna složenost 16

2.3. Mintyev algoritam 17

2.3.1. Ideja i dokaz Mintyevog algoritma 17

2.3.2. Pseudokod za Mintyev algoritam 19

2.3.3. Vremenska i prostorna složenost 22

2.4. Unos podataka i provjere matrica 23

2.4.1. Implementacija unosa podataka 23

2.4.2. Implementacija provjere matrice 24

3. Testiranje i rezultati 26

3.1. Testiranje 26

3.1.1. Model generatora grafova 27

3.2. Rezultati 28

3.2.1. Grafovi 29

4. Zaključak 33

Literatura 34

Sažetak 36

Abstract 37

1. Stabla

1.1. Problem broja razapinjućih stabala

U ovom radu ćemo istražiti problem prebrojavanja razapinjućih stabala grafa. To je jedan od ključnih koncepata teorije grafova koji ima svoju primjenu u raznim područjima znanosti kao što su računarstvo, telekomunikacije, urbana gradnja, elektrotehnika i drugi. Neke od konkretnih primjena su problem optimizacije, dizajn električnih mreža, planiranje transportnih ruta itd. Problem prebrojavanja razapinjućih stabala je dobro teoretski proučen, no sam problem je eksponencijalne vremenske i prostorne složenosti pa samim time postaje teško izračunavanje broja stabala za velike grafove. Cilj ovog istraživanja je opisati i napraviti neke od osnovnih algoritama za prebrojavanje broja razapinjućih stabala iscrpnom metodom pretrage. Osim algoritama koji nalaze sva stabla iscrpnom pretragom postoje i formule koje na mnogo brži način izračunaju ukupni broj razapinjućih stabala. Jedna od najpoznatijih formula među njima je upravo Kirchhoffova formula s kojom ćemo se bolje upoznati u poglavlju 2.1. Problem kod formula je u tome što one izbacuju ukupan broj stabala, no nemamo nikakvu informaciju koja su to stabla. Zbog toga ćemo se u ostatku 2. poglavlja posvetiti algoritmima za iscrpnu pretragu 2.2., 2.3. koji, osim što prebroje sva razapinjuća stabla, mogu i ispisati svako razapinjuće stablo.

1.2. Općenito o stablima

U ovom poglavlju ćemo se dodatno upoznati s pojmom stabla te navesti samu definiciju stabla.

Neka je zadan jednostavan, povezan graf $G = (V,E)$ koji se sastoji od nepraznog konačnog skupa V kojeg nazivamo skup vrhova i konačnog skupa E kojeg nazivamo skup bridova. n predstavlja broj vrhova grafa G, tada znamo da navedeni graf može imati najmanje n -

1 bridova.

Definicija 1. *[4] Povezani graf bez ciklusa naziva se stablo. Ako je graf G stablo onda znamo da:*
1) ne sadrži ciklus
2) ima n - 1 bridova
3) povezan je i svaki brid je ujedno i most
4) dva vrha su povezana točno jednim putem
5) G ne sadrži ciklus, no dodavanjem jednog brida dobit ćemo točno jedan ciklus

1.2.1. Razapinjuća stabla

Razapinjući podgraf grafa $F = (V,E)$ koji ima n vrhova je svaki podgraf $F' = (V, E')$ grafa F koji ima jednak skup vrhova. Kako bismo došli do definicije razapinjućeg stabla pokušajmo prvo napraviti sljedeći postupak. Na prethodnom grafu F pronađemo ciklus te iz navedenog ciklusa otklonimo jedan brid kojeg ćemo nazvati e. Nakon što smo otklonili navedeni brid dobivamo povezani graf F - e. Sada u novo dobivenom grafu opet tražimo ciklus, brišemo jedan brid iz ciklusa i navedeni postupak iterativno ponavljamo dok god novo kreirani povezani graf ima cikluse. Intuitivno zaključujemo da navedeni postupak mora stati čime dobivamo povezani graf bez ciklusa, odnosno stablo 1 koje zovemo razapinjuće stablo zadanog grafa F .

1.3. Povijesni pregled

Problem razapinjućih stabala ima dugu i bogatu povijest. Prebrojavanje razapinjućih stabala izaziva veliki interes znanstvenika koji se bave teorijom grafova. Diljem svijeta znanstvenici se bave ne samo problemom pronalaska svih razapinjućih stabala već i traženjem minimalnog razapinjućeg stabla u slučaju da je graf koji promatramo težinski. U proteklih 50-ak godina osmišljeni su brojni algoritmi koji na zanimljive načine pokušavaju što bolje riješiti probleme vremenske i prostorne složenosti. Povijesno su se razvile tri osnovne tehnike iscrpnog pretraživanja koje ćemo detaljnije opisati u poglavlju 1.5. Osim algoritama za iscrpno pretraživanje postoje i formule za poznate grafove, formulama ćemo se više baviti u poglavlju 1.4.

1.4. Formule za poznate grafove

Već smo ranije spomenuli da se kroz svoju bogatu povijest razvila i velika teoretska podloga za prebrojavanje razapinjućih stabala. Tako da za osnovne grafove imamo zadane formule koje služe za izračunavanje broja razapinjućih stabala. Dokazivanjem samih formula se nećemo baviti u ovom radu te upućujemo čitatelje na literaturu navedenu uz formule.

Potpuni graf

Jednu od najpoznatijih formula za izračun broja razapinjućih stabala T dao je Cayley 1889. godine [6]. Ona navodi da postoji točno n^{n-2} različitih označenih razapinjućih stabala za potpuni graf K_n gdje n predstavlja broj vrhova.

$$T(K_n) = n^{n-2}$$

Ciklus

Kod grafova ciklusa broj vrhova n jednak je broju bridova m, a poznato je kako za graf ciklus vrijedi ako izbacimo jedan brid dobivamo lanac koji je po definiciji stablo. Sada intuitivno možemo zaključiti da micanjem bilo kojeg brida dobivamo stablo, odnosno da je broj razapinjućih stabala T ciklusa C_n jednak broju bridova m, odnosno broju vrhova n

$$T(C_n) = n$$

Kotač

Uzmemo li graf ciklus C_n i dodamo mu jedan vrh koji je povezan sa svim ostalim vrhovima dobivamo graf kotač W_n. Bitno je pripaziti da kod grafa kotača varijabla n ne predstavlja broj vrhova, već je broj vrhova jednak $n + 1$. Za graf kotač formula za izračunavanje broja razapinjućih stabala T je [pogledajte [2]]:

$$W_n = \left(\frac{3+\sqrt{5}}{2}\right)^n + \left(\frac{3-\sqrt{5}}{2}\right)^n - 2$$

Potpuni bipartitni graf

Definicija 2. *[3] Ako skup vrhova grafa G možemo razdvojiti u dva disjunktna skupa A i B tako da svaki brid od G spaja neki vrh skupa A s nekim iz skupa B, onda kažemo da je G* ***bipartitan graf****.*
Potpuni bipartitan graf *je onaj bipartitan graf s particijom skupa vrhova $V(G) = A \cup B$ kod kojeg je svaki vrh iz skupa A spojen sa svakim iz B. Ako je $|A| = r$, te $|B| = s$, onda takav graf označavamo s $K_{r,s}$.*

Formula za broj razapinjućih stabla potpunog bipartitnog grafa je [pogledajte [2]]:

$$T(K_{r,s}) = r^{s-1}s^{r-1}$$

1.5. Vrste algoritama za pretraživanje

Matematičari su kroz povijest razvili brojne algoritme za iscrpno pretraživanje broja razapinjućih stabala. Najveći napredak dogodio se početkom 60-ih godina prošlog stoljeća i traje sve do danas. Algoritmi koji su se razvili koriste različite tehnike za prebrojavanje, razlikuju se u vremenskoj i prostornoj složenosti, no svi algoritmi imaju i nešto zajedničko.
Metode koje su koristili u iscrpnoj pretrazi mogu se raspodijeliti u tri kategorije:
a) metoda Testiraj i odaberi (eng. Test and select method) 1.5.1.
b) Metoda elementarnih transformacija stabla (eng. Elementary tree transformation method) 1.5.2.
c) Metoda sukcesivnog smanjivanja grafa (eng. Successive reduction of graph method) 1.5.3.

1.5.1. Metoda testiraj i odaberi

Glavna ideja ove metode dolazi iz same definicije stabla 1.2., znamo da svako stablo s n vrhova mora imati $n-1$ bridova m. Dakle, ova metoda se mora sastojati od dvije faze. U prvoj fazi se generiraju sve moguće kombinacije bridova duljine $n-1$, dok se u drugoj fazi ispituje svaka od navedenih kombinacija i provjerava je li zadana kombinacija stablo

ili nije. Sve stabla se ostavljaju, a one kombinacije koje nisu stablo se izbacuju. Poznati algoritmi koji koriste ovu metodu generiranja razapinjućih stabala su: Charov algoritam [9] , Sen Sarma algoritam [10], Naskarov algoritam [11], Oneteov algoritam [12] i mnogi drugi.

1.5.2. Metoda elementarnih transformacija stabla

Metoda elementarnih transformacija stabla kreće od grafa $G(V,E)$ i pomoću određenog algoritma napravi početno stablo. Najčešće koristi pretraživanje u širinu (eng. BFS) ili pretraživanje u dubinu (eng. DFS). Nakon što je generirano početno stablo iz zadanog grafa $G(V,E)$ bridovi grafa su podijeljeni u dvije skupine, unutar jedne skupine nalaze se bridovi koji su unutar početnog stabla dok su u drugoj skupini bridovi koji nisu unutar početnog stabla. Potom generiramo nova stabla tako da u svakom koraku dodajemo i stavljamo brid iz jedne skupine u drugu. Pri tome pazimo da ne kreiramo ciklus i da svi bridovi ostanu povezani. Pri implementaciji ove metode također moramo paziti da ne generiramo duplikate. Ova metoda je generalno dobro prostorno optimirana jer generira samo razapinjuća stabla, odnosno nema dodatnog "smeća" kao u metodi testiraj i odaberi 1.5.1. Neki od poznatih algoritama koji se zasnivaju na metodi elementarnih transformacija su: Hakimijev algoritam [13], Mayedin algoritam [14], Kapoorov algoritam [15], Matsuiev algoritam [16] i mnogi drugi.

1.5.3. Metoda sukcesivnog smanjivanja grafa

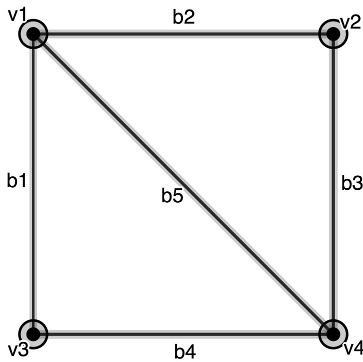
Kao što i samo ime kaže koncept metode sukcesivnog smanjivanja grafa je da se originalni graf smanjuje u manje podgrafove. Takvom metodom originalni problem postaje sve manji jer se bavimo manjim podgrafovima. Najčešće se smanjivanje grafa u podgrafove ponavlja sve dok se ne dobije potpuno jednostavna struktura poput jednog brida ili vrha. Takvim algoritmom pronalazimo broj razapinjućih stabala T originalnog grafa zbrajajući jednostavne podgrafove. Iako ova metoda na prvu ruku izgleda kompliciranije od prethodnih, njezina prednost je u tome što zahtjeva znatno manje provjera od prethodne dvije metode. Neki poznatiji algoritmi koji koriste ovu metodu su: Mintyev algoritam [8], Winterov algoritam [5], Smithov algoritam [7] i mnogi drugi.

2. Korišteni algoritmi i implementacija

Kako bismo izračunali sva razapinjuća stabla T grafa G u ovom radu smo koristili tri različita pristupa koja ćemo objasniti, implementirati te na kraju u 3. poglavlju opisati i usporediti rezultate. Prvi pristup koji smo koristili je bila Kirchhoffova formula. Iako je ona izrazito efikasna po pitanju vremenske, ali i prostorne složenosti, problem koji se javlja je, kao što smo to ranije naveli, da ta metoda ne daje informacije koja razapinjuća stabla postoje već daje samo njihov ukupni broj. Zbog toga nam nije ostala druga opcija nego problemu pristupiti na drugi način odnosno da iscrpnom metodom pretražujemo sva stabla. Iako se možda čini da je prvi pristup u potpunosti nepotreban, to ipak nije tako. Naime, formula je vrlo korisna kada želimo provjeriti ispravnost iscrpnih algoritama jer ona na izrazito brz način izračunava ukupan broj razapinjućih stabala. Primjerice ako je jedan algoritam znatno sporiji od drugoga ne moramo ih međusobno čekati kako bismo provjerili ispravnost bržeg algoritma. Osim za provjeru rješenja formula je korisna i kada nas samo zanima broj razapinjućih stabala što također može biti korisno u mnogim problemima. Prvi algoritam koji smo koristili bazira se na metodi testiraj i odaberi čiju smo ideju objasnili u poglavlju 1.5.1. Drugi algoritam koji smo koristili bazira se na metodi sukcesivnog smanjivanja grafa čiju smo ideju objasnili u poglavlju 1.5.3. Svaki od pristupa koje smo koristili ima svoje prednosti i mane, što ćemo pobliže objasniti u nastavku poglavlja. Kako bismo što bolje objasnili svaki od algoritama koristiti ćemo jedan univerzalni graf u svim primjerima. 2.1.

2.1. Kirchhoffova formula

U ovom poglavlju ćemo predstaviti Kirchhoffov matrični teorem o stablima. Teorem povezuje determinantu matrice i broj razapinjućih stabala grafa. Teorem ćemo implemen-



Slika 2.1. Univerzalni graf

tirati i koristiti ga za izračun broja razapinjućih stabala grafa i provjeru točnosti ostalih algoritama koje napravimo.

2.1.1. Postupak

Prije nego definiramo postupak za izračun broja razapinjućih stabala grafa G, moramo definirati nekoliko pojmova. Prvo trebamo definirati matricu susjedstva A_G.

Definicija 3. *[3] Označimo vrhove grafa $G(V, B)$ s oznakom $V = \{1, 2, 3, .., n\}$, sada definiramo matricu susjedstva $\left[A_{ij}\right]$ kao $n \times n$ čiji je element a_{ij} jednak broju bridova koji spajaju vrh i s vrhom j.*

Za graf iz primjera matrica susjedstva bi glasila:

$$\begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Nakon što smo uspješno definirali matricu susjedstva treba nam dijagonalna matrica D_G, koja se dobije tako da sumu svakog stupca matrice susjedstva stavimo na mjesto u tom stupcu koje presijeca dijagonala matrice, odnosno sumu stupca S_k stavimo u redak

k stupca S_k. Za naš primjer dijagonalna matrica D_G bi bila oblika:

$$\begin{bmatrix} 3 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \end{bmatrix}$$

Kada smo uspješno definirali matricu susjedstva A_G i dijagonalnu matricu D_G grafa G možemo definirati i Laplaceovu matricu L_G.

Definicija 4. *Laplaceova matrica je simetrična matrica reda n gdje n predstavlja broj vrhova grafa G. Laplaceovu matricu L_G dobijemo tako da oduzmemo dijagonalnu matricu od matrice susjedstva:*

$$L_G = D_G - A_G$$

Za pokazni graf 2.1. Laplaceova matrica je oblika:

$$\begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 2 & 0 & -1 \\ -1 & 0 & 2 & -1 \\ -1 & -1 & -1 & 3 \end{bmatrix}$$

Matrica reda $(n-1)$ koja se dobije brisanjem i-tog retka i stupca Laplaceove matrice L_G nazivamo minora Laplaceove matrice L_G^i. Minora prethodne matrice:

$$\begin{bmatrix} 2 & 0 & -1 \\ 0 & 2 & -1 \\ -1 & -1 & 3 \end{bmatrix}$$

Sad kada smo definirali sve potrebno možemo iskazati teorem.

TEOREM 1. *(Matrični teorem o stablima) [1] Neka je zadan graf G s $n \geq 2$ vrhova. Tada je broj razapinjućih stabala $T(G)$ jednak determinanti minore Laplaceove matrice.*

$$T(G) = detL_G^i \qquad \forall i \leq n$$

Konačno kad izračunamo determinantu minore Laplaceove matrice L_G^i dobivamo 8 kao rješenje i ukupan broj razapinjućih stabala. Zbog potpunosti ćemo pokazati sva stabla na slici 2.3.

2.1.2. Implementacija

U prethodnom poglavlju smo opisali postupak računanja razapinjućih stabala pomoću Kirchhoffove formule tako da ćemo u ovom poglavlju dati samo kratki pseudokod:

Algoritam 1: Kirchhoffova formula

```
Input: MatricaSusjedstva
1 Izracunaj dijagonalnu matricu;
2 Izracunaj Laplaceovu matricu;
3 Izvedi minoru Laplaceove matrice;
4 return Determinanta(minora Laplaceove matrice, brojRedakaMinore);
```

Izračun dijagonalne matrice, Laplaceove matrice i minoru Laplaceove matrice smatramo trivijalnim te ih nećemo detaljnije objasniti već ćemo se za kraj fokusirati na funkciju koja služi za izračun determinante matrice.

Algoritam 2: determinantaMatrice

```
Input: matrica, brojRedaka
1  determinanta = 0;
2  ako je brojRedaka == 1
3  |  return matrica[0][0]
4  ako je brojRedaka == 2
5  |  return matrica[0][0] · matrica[1][1] - matrica[0][1] · matrica[1][0]
6  predznak = 1;
7  i = 0;
8  dok je i < brojRedaka
9  |  temp = minoraMatrice(matrica);
10 |  determinanta += predznak · matrica[0][i] · determinantaMatrice(temp,
   |    brojRedaka - 1);
11 |  predznak = -predznak;
12 |  i++;
13 return detrminanta
```

2.1.3. Vremenska i prostorna složenost

Kao što smo i ranije naglasili Kirchhoffova formula je najbrži i najefikasniji način računanja ukupnog broja razapinjućih stabala. Vremenska i prostorna složenost su mnogo

manje nego u drugim algoritmima. Najviše resursa i vremena u cijelom algoritmu oduzima računanje determinante matrice. Danas postoje mnogi relativno brzi algoritmi za računanje determinante matrice koji imaju vremensku složenost $O(n^3)$. U ovom radu smo htjeli implementirati svoj algoritam, naravno on je dosta sporiji od idealnih algoritama. U našem izračunu determinante svaki redak generira n novih matrica veličine $n-1$ gdje n predstavlja broj redaka odnosno stupaca navedene razine za koji vrijedi $n > 2$. Zaključujemo da je onda vremenska složenost:

$$O(n!)$$

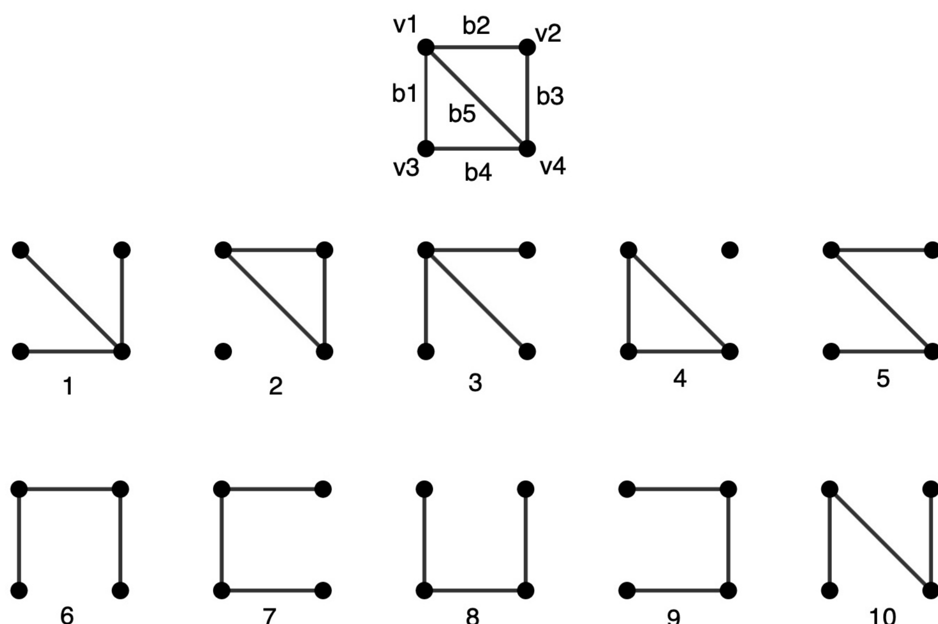
Prostorna složenost je dosta manja zbog alociranja memorije tako da za nju dobivamo vrijednost:

$$O(n^2)$$

Primijetimo da program možemo dodatno ubrzati s pomoću podrezivanja odnosno da u *while* petlji koda 2 dodatno provjeru $ako je matrica[0][i] == 0$ da pređemo na iduću iteraciju. Također bitno je primijetiti da računanje dijagonalne i Laplaceove matrice je puno manje vremenske složenosti te ih zato možemo zanemariti.

2.2. Algoritam 1

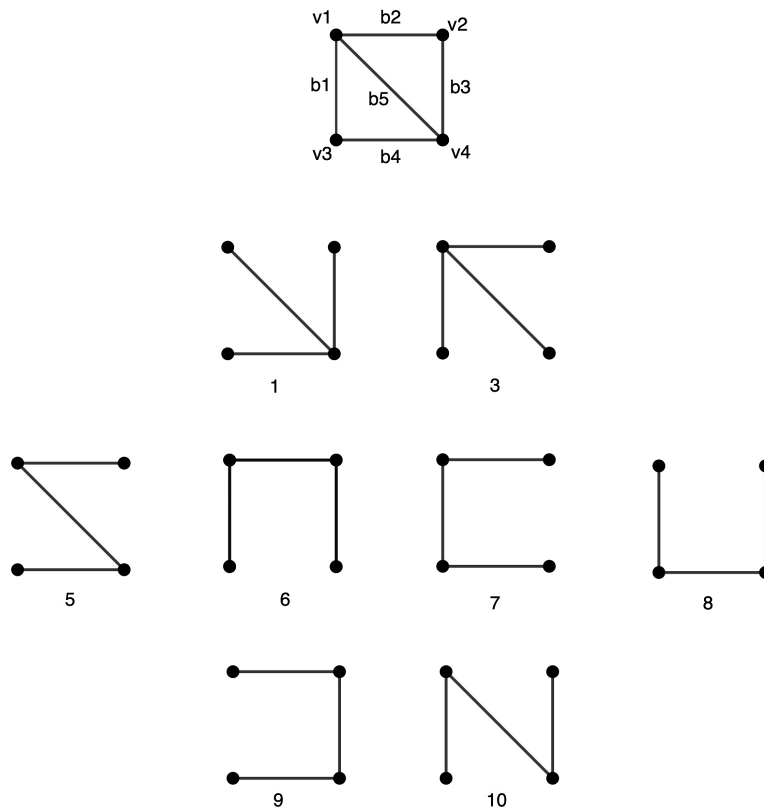
Vjerujemo da se znatiželjni čitatelj tijekom ovog rada već zapitao kako bi se na najlakši mogući način mogao kreirati algoritam za iscrpnu pretragu broja razapinjućih stabala. Ima li bolje mjesto za krenuti od same definicije stabla 1.2. Znamo da razapinjuće stablo od grafa G koji ima n vrhova mora imati točno $n - 1$ bridova. Sada ako uzmemo sve bridove m i stavimo ih u jednu listu možemo kreirati pod liste veličine $n - 1$ koje sadrže različite bridove m grafa G. Ako je sve do sada dobro implementirano imamo sve kombinacije $n - 1$ bridova. Možemo sada to pokazati za naš pokazni graf 2.1. Znamo da graf ima četiri vrha n i pet bridova m ako iskoristimo binomnu formulu zaključujemo da postoji točno $\binom{5}{3}$, odnosno deset kombinacija. Sada ih možemo prikazati na slici.
2.2. Primijetimo da u postupku generiranja nismo pazili da generiramo samo povezane podgrafove koji ne sadrže ciklus. Znači da sada moramo provjeriti svaki podgraf pojedinačno i izbaciti ga ako on nije stablo. U našem primjeru ćemo izbaciti podgrafove dva i četiri. To možemo vidjeti na idućoj slici 2.3. Primijetimo da smo sada dobili sva razapi-



Slika 2.2. Sve kombinacije grafa

njuća stabla grafa 2.1.

Ako analiziramo naš dosadašnji postupak i uspoređujemo ga s metodom testiraj i odbaci 1.5.1. možemo primijetiti da postupak u kojem uzimamo bridove, stavljamo ih u liste i kreiramo podliste upravo odgovara prvoj fazi navedene metode odnosno fazi rasta tj. generiranja podgrafova. Faza u kojoj izbacujemo podgrafove koji nisu stabla upravo odgovara drugoj fazi metode odnosno fazi testiranja i odbacivanja. Primijetimo dokaz je dosta intuitivan i dolazi iz same formule, ako je sve dobro implementirano nemoguće je da postoji razapinjuće stablo grafa, a da ga naš algoritam ne pronađe, također ako je metoda provjere točna sva razapinjuća stabla će ostati, a višak će se maknuti.



Slika 2.3. Sve kombinacije grafa

2.2.1. Pseudokod

Algoritam 3: Pseudokod za Algoritam 1

1 Kreirati listu bridova $listaBridova$;

2 U listu bridova dodati sve bridove u obliku "Vrh1-Vrh2";

3 Napraviti novu listu $sveKombinacijeBridova$ koja će sadržavati liste bridova;

4 $sveKombinacijeBridova = combine(listaBridova, brojVrhova - 1)$;

5 $i = 0$;

6 **dok je** $i <$ *sveKombinacijeBridova.size())*

7 $\quad$ $iteratorLista = sveKombinacijeBridova$.get(i);

8 $\quad$ $privMat$ = pretvoriListuUMatricuSusj($iteratorLista, brojVrhova$);

9 $\quad$ $jeliPovezan$ = DFS($privMat$, $privMat.length$);

10 $\quad$ **ako je** $jeliPovezan$ == *FALSE*

11 $\quad\quad$ izbaci $itaratorLista$ iz $sveKombinacijeBridova$

12 $\quad$ i++;

13 **return** *sveKombinacijeBridova.size()*;

Pseudokod 3 primjenom algoritma 1 iscrpnom metodom pronalazi sva razapinjuća stabla koja zapisuje u listu $sveKombinacijeBridova$. Broj razapinjućih stabala je upravo veličina liste. Veličina liste u pseudokodu je implementirana funkcijom size(), npr. imeListe.size(). Funkcija get($index$) vraća element liste na mjestu $index$. $BrojVrhova$ je varijabla koja predstavlja broj vrhova početnog grafa G čija razapinjuća stabla tražimo. Primijetimo da je po definiciji stabla broj bridova zapravo broj vrhova grafa G minus jedan. Funkcija $pretvoriListuUMatricuSusj$ kao argumente prima listu Stringova koja predstavlja bridove grafa zapisane u obliku "Vrh1-Vrh2" i broj vrhova koji graf koji ona stvara mora imati. Navedena funkcija razdvaja bridove po separatoru "-" čime dobiva vrhove koji joj služe da izgradi matricu susjedstva od liste koju je primila kao argument. Nakon što napravi matricu susjedstva funkcija $pretvoriListuUMatricuSusj$ vraća matricu kao dvodimenzijsko polje Integera. Nakon što smo dobili matricu susjedstva moramo provjeriti je li graf povezan, što vrlo jednostavno činimo funkcijom DFS koja predstavlja pretraživanje u dubinu. Funkcija DFS vraća TRUE ako je graf povezan i FALSE ako graf nije povezan. Za kraj izbacujemo sve grafove koji nisu povezani. Primijetite da smo u pseudokodu to radili čim bismo ispitali je li graf povezan. Međutim, kod većine programskih jezika u stvarnoj implementaciji to ne bi radilo, jer nije ispravno brisati listu dok putujemo kroz nju, već bi problem trebalo riješiti na drugi način primjerice dodavanjem nepovezanih grafova u novu listu te kada sve obiđemo iz liste $sveKombinacijeBridova$ izbaciti elemente nove liste. Za kraj je ostalo objasniti funkciju combine(arg1, arg2) koja kao prvi argument prima listu bridova, a kao drugi argument prima broj koliko elemenata želimo unutar podliste. Cilj nam je, naravno, uzeti $brojVrhova - 1$ jer je to upravo broj bridova koji nam treba. Ako uzmemo navedeni broj kao argument dva dobit ćemo sve $n - 1$ kombinacije bridova m odnosno točno $\binom{m}{n-1}$ kombinacija. U nastavku slijedi pseudokod za funkciju combine().

Algoritam 4: Pseudokod za funkciju combine

Input: listaBridova, k

1 Kreirati listu koja sadrži listu bridova podgrafova: $sveKombinacijeBridova$;

2 Kreirati listu Stringova $trenutnaLista$;

3 combineRekurzivno($listaBrid, k, 0, trenutnaLista, sveKombinacijeBridova$);

4 **return** $sveKombinacijeBridova$;

Algoritam 5: Pseudokod za funkciju combineRekurzivno

```
Input: listaBridova, k, pocetak, trenutnaLista, rjesenje
1 ako je k == 0
2     dodaj u rješenje trenutnuListu;
3     return
4 i = pocetak;
5 dok je i < listaBridova.size()
6     u trenutnaLista dodaj listaBridova na mjestu i;
7     combineRekurzivno(listaBridova, k − 1, i + 1, trenutnaLista, rjesenje);
8     iz trenutnaLista izbriši element na zadnjem mjestu;
9     i++;
```

2.2.2. Vremenska i prostorna složenost

Sad kada smo obradili algoritam ostaje nam izračunati vremensku i prostornu složenost.

Prostorna složenost

Već smo ranije rekli da algoritam kreira $\binom{m}{n-1}$ lista veličine $n-1$. Gdje n predstavlja broj vrhova, a m broj bridova početnog grafa G. Znači da je prostorna složenost sigurno barem $\binom{m}{n-1} \cdot (n-1)$. Ne smijemo zaboraviti da algoritam ima i listu za izbacivanje koja opet može biti veličine $\binom{m}{n-1}$ u njoj zapisujemo samo mjesta u početnoj listi koja izbacujemo te zbog toga binomni dio ne množimo s $n-1$. Druga lista zapravo ne utječe na prostornu složenost jer se zbraja s prvom vrijednošću, a manjeg je reda veličine od nje tako da ju možemo zanemariti. Unutar algoritma nema drugih struktura koje utječu na prostornu složenost pa zaključujemo da je prostorna složenost grafa G:

$$O(\binom{m}{n-1} \cdot (n-1))$$

Vremenska složenost

Za vremensku složenost znamo da mora biti veća ili jednaka prostornoj složenosti. Sada možemo pogledati što još dodatno usporava naš algoritam. Znamo da nakon što smo kreirali sve kombinacije bridova moramo proći kroz svaku i provjeriti je li graf povezan.

Za svaku provjeru koristimo algoritam prolaska u dubinu koji ima vremensku složenost $O(V + B)$ gdje je V broj vrhova a B broj bridova. Također znamo da za svaki naš graf vrijedi da je broj bridova jednak $n-1$ a broj vrhova je n. Znači da je vremenska složenost algoritma prolaska kroz dubinu približno jednaka $O(2n)$, pošto je 2 konstanta onda uzimamo da je vremenska složenost jednaka $O(n)$. Kako bismo dobili ukupnu vremensku složenost moramo pomnožiti broj generiranih stabala s vremenskom složenosti pregleda svakog od njih čime dobivamo da je ukupna vremenska složenost Algoritma 1 jednaka:

$$O(\binom{m}{n-1} \cdot (n-1) \cdot n)$$

Što je približno jednako:

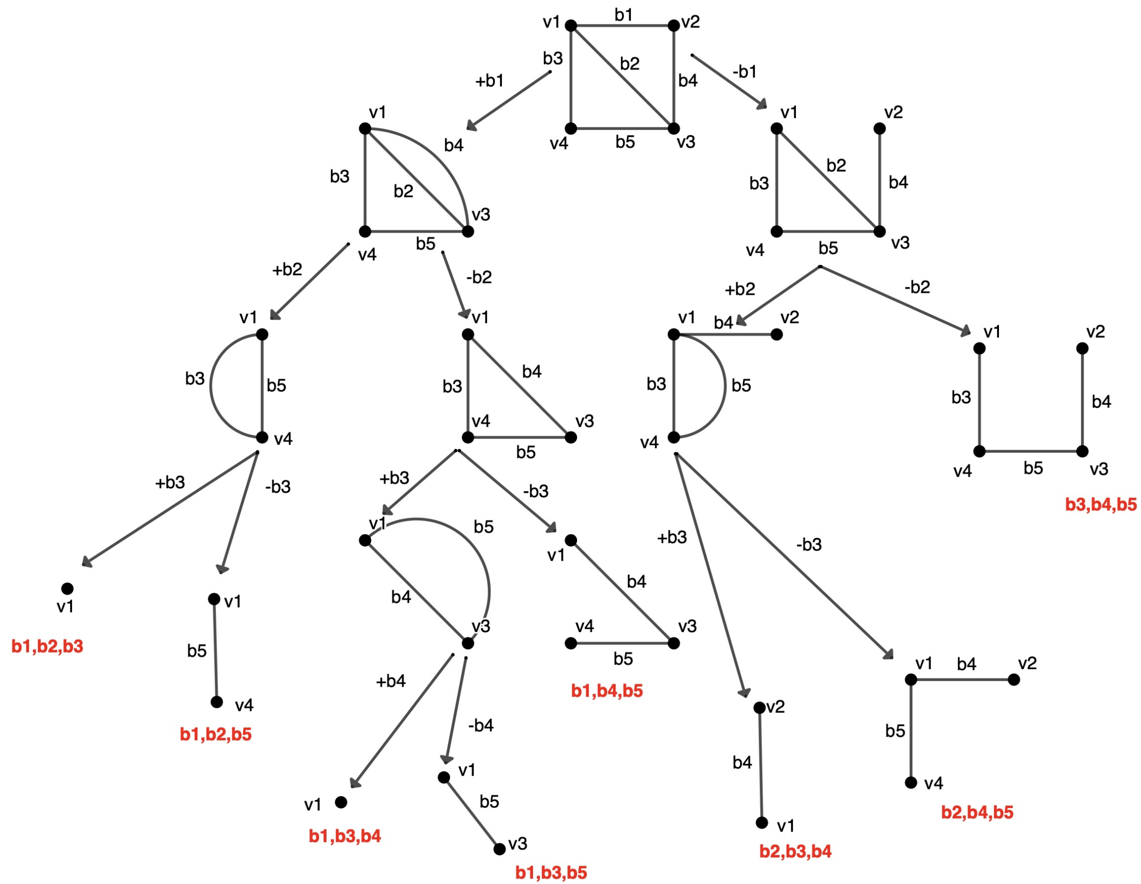
$$\approx O(\binom{m}{n-1} \cdot n^2)$$

2.3. Mintyev algoritam

Drugi algoritam koji smo implementirali za prebrojavanje razapinjućih stabala grafa G bazira se na algoritmu koji je osmislio Minty 1965. godine. Ovaj algoritam koristi metodu sukcesivnog smanjivanja grafa (eng. Successive reduction of graph method) 1.5.3. Ideja algoritma je da koristeći rekurziju krene od početnog grafa $G(V, B)$ te da se izabere jedan brid b_k iz skupa bridova B. Sada generiramo dva grafa G_1 i G_2. Graf G_1 će nastati micanjem brida b_k čime dobivamo graf $G_1 = G - b_k$. Dok će graf G_2 nastati kontrakcijom brida b_k, odnosno sve vrhove koji su incidentni s bridom b_k slijepimo uzimajući pritom u obzir sve bridove s kojima su oba slijepljena vrha incidentna, tako dobivamo graf $G_2 = G \setminus b$. Primijetimo da graf G_2 nije podgraf grafa G . Koristeći ovaj algoritam moramo paziti na uvjet da brid b_k ne smije biti most niti smije biti petlja. Navedeni algoritam se ponavlja dok nismo došli do samo jednog vrha ili da je lista dodanih bridova plus bridovi trenutnog podgrafa jednaka veličine $n-1$. Na slici 2.4. ćemo pokazati kako bi se algoritam ponašao u slučaju pokaznog grafa 2.1.

2.3.1. Ideja i dokaz Mintyevog algoritma

Kao svojevrsni dokaz Mintyevog algoritma ćemo pokazati i dokazati lemu na kojoj se temelji cijeli algoritam.



Slika 2.4. Primjer Minty

Lema 1. *[2] Neka je e brid grafa G. Za T(G) broj razapinjućih stabala grafa G vrijedi*

$$T(G) = T(G - e) + T(G\backslash e) \tag{2.1}$$

Dokaz. Ako uzmemo sva razapinjuća stabla od grafa G, njih možemo podijeliti u dvije skupine. Prva skupina će biti sva razapinjuća stabla koja sadržavaju brid e, a druga skupina su sva stabla koja ne sadržavaju brid e. Pošto smo samo podijelili ukupni broj stabala $T(G)$ u dvije skupine čiji je presjek prazan skup i dalje vrijedi uvjet da je ukupni broj stabala grafa jednak zbroju razapinjućih stabala od dvije grupe. Ako broj razapinjućih stabala prve skupine bez brida e označimo s x, a s y označimo broj stabala druge grupe koja sadrži brid e zaključujemo da je broj razapinjućih stabala grafa G jednak $T(G) = x + y$. Uočimo, da za svako stablo grafa G koje ne sadrži brid e vrijedi da je ono ujedno i stablo podgrafa $T(G-e)$ pa zaključujemo da je $x = T(G-e)$. Svako stablo iz $G\backslash e$ dobivamo iz stabla grafa G koje sadrži brid e postupkom konkatenacije brida e navedenog stabla. Ako obrnemo postupak, svako razapinjuće stablo od $G\backslash e$ daje jedno

razapinjuće stablo od G koji sadrži brid e pa iz toga zaključujemo da je $y = T(G \backslash e)$. □

Nakon što smo dokazali prethodnu lemu intuitivno možemo zaključiti da ostatak algoritma vrijedi. Naime ako nastavimo iskorištavati Lemu 1 2.1, ali pri svakoj iteraciji uzimamo novi brid sve dok ne dođemo do jednostavne strukture dobivamo upravo Mintyev algoritam.

2.3.2. Pseudokod za Mintyev algoritam

Algoritam 6: Main Minty

1. Učitati matricu susjedstva za proizvoljni graf: $matSusj$;
2. Pretvoriti matricu susjedstva u matricu incidencije: $matInc$;
3. Dodati ekstra redak u $matInc$ koji će biti index brida;
4. Kreirati listu za pohranu dodanih bridova: $list$;
5. MintyRekFja($matInc, list, brojRedaka + 1, brojBridova, KlasaMatrica, razina = 0$);

Klasa matrica predstavlja klasu u Javi koja pohranjuje podatke o originalnoj matrici incidencije, broju vrhova, broju stupaca te brojaču razapinjućih stabala. Kada nađemo neko novo razapinjuće stablo povećamo brojač. Sada ćemo dati pseudokod za svaku od funkcija.

Algoritam 7: MintyRekFja

Ulazni argumenti: NovaMatInc, rj, brRed, brojStupaca, KlasaMatrica, razina

1. lijevoRekurzivno($NovaMatInc, rj, brRed, brSt, KlasaMatrica, raz$);
2. očisti listu rješenja rj;
3. desnoRekurzivno($NovaMatInc, rj, brRed, brSt, KlasaMatrica, raz$);

NovaMatInc predstavlja matricu incidencije koju ću dobiti brisanjem brida b_k ($G_1 = G - b_k$) i kontrakcijom brida b_k ($G_2 = G \setminus b$). Navedena matrica će se predati nižoj razini kako bi niža razina napravila isti postupak na grafu G_1 ili G_2. rj predstavlja listu koja sadrži bridove rješenja koji su dodani kontrakcijom. $brRed$ predstavlja broj redaka matrice incidencije, a $brojStupaca$ predstavlja broj stupaca. $KlasaMatrica$ je već objašnjena. $razina$ predstavlja razinu unutar rekurzije na kojoj se nalazimo.

Algoritam 8: lijevoRekurzivno

```
   Ulazni argumenti: NovaMatInc, rj, brRed, brojStupaca, KlasaMatrica, razina
1  ako je brRed == 2
2  |  povećaj broj razapinjućih stabala u KlasaMatrica;
3  |  return
4  ako je (rješenja.size() + brojStupaca) == brRed - 2
5  |  povećaj broj razapinjućih stabala u KlasaMatrica;
6  |  return
7  nadi brid s najmanjim indexom;
8  kontrahiraj najmanji brid;
9  dodaj najmanji brid u listu rješenja;
10 ako je brojRedaka kontrahirane matrice == 2
11 |  povećaj broj razapinjućih stabala u KlasaMatrica;
12 |  return
13 ako je brojRjesenja + broj stupaca kontrahirane matrice == brRed - 2
14 |  povećaj broj razapinjućih stabala u KlasaMatrica;
15 |  return
16 izbaci prazne stupce;
17 ako je novonastala matrica zadovoljava uvjete za izlazak iz petlje provjeri
     povezanost i povećaj broj razapinjućih stabala;
18 lijevoRekurzivno(kontrahMat, rj, novBrRed, novBrSt, KlasaMatrica, ++
     raz);
19 izbriši zadnje rješenje;
20 - -razina;
21 desnoRekurzivno(kontrahMat, rj, novBrRed, novBrSt, KlasaMatrica, ++
     raz);
22 - -razina;
```

Uočite da provjeravamo je li broj redaka jednak dva umjesto da provjeravamo je li broj redaka jednak jedan. Razlog tome je što prvi redak predstavlja index brida. Funkcija .size() vraća veličinu liste. Kontrahiranje retka činimo tako da u prvom stupcu matrice nađemo prve dvije jedinice odnosno dva vrha koja su spojena. Kada smo ih

našli brišemo prvi stupac te redak u kojem je jedinica s većim indexom. Prije brisanja retka jedinice s većim indexom moramo napraviti operaciju xor između dva retka koja sadrže jedinice i rezultat operacije xor upisujemo u redak jedinice s nižim indexom. Možemo navedeni postupak pokazati na primjeru kontrakcije brida b_1 početnog grafa sa slike 2.4. Matrica incidencije grafa izgleda ovako:

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

Sada pratimo postupak. U prvom stupcu nađemo index jedinica to je u našem slučaju index nula i jedan. Sada radimo operaciju xor nad redcima nula i jedan i rezultat operacije zapisujemo u prvi redak.

$$\begin{bmatrix} x & 1 & 1 & 1 & 0 \\ x & x & x & x & x \\ x & 1 & 0 & 1 & 1 \\ x & 0 & 1 & 0 & 1 \end{bmatrix}$$

Za kraj brišemo redak jedan i prvi stupac, čime dobivamo navedenu matricu:

$$\begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

Možemo primijetiti da ona točno odgovara grafu $+b1$ sa slike 2.4.

Ostaje nam pokazati funkciju desnoRekurzivno:

Algoritam 9: desnoRekurzivno

```
Ulazni argumenti: NovaMatInc, rj, brRed, brojStupaca, KlasaMatrica, razina
1  ako je (rj.size() + brojStupaca) == (brRed - 2)
2  |   povećaj broj razapinjućih stabala u KlasaMatrica;
3  |   return
4  izbrisši prvi redak;
5  provjeri povezanost nove matrice: matIncDesna;
6  ako je nije povezana
7  |   return
8  ako je (rj.size() + broj redaka matIncDesna) == (KlasaMatrica.brRed - 2)
9  |   povećaj broj razapinjućih stabala u KlasaMatrica;
10 |   return
11 lijevoRekurzivno(matIncDesna, rj, brRed, brojStupaca −
     1, KlasaMatrica, + + razina);
12 - -razina;
13 izbriši zadnje rješenje iz liste rj;
14 desnoRekurzivno(matIncDesna, rj, brRed, brojStupaca −
     1, KlasaMatrica, + + razina);
15 - -razina;
```

2.3.3. Vremenska i prostorna složenost

Iako je Minty napravio algoritam, nije izračunao vremensku i prostornu složenost algoritma, Tek je kasnije Smith [7] izračunao vremensku i prostornu složenost algoritma. Prema njegovim podacima vremenska složenost je:

$$O(n + m + m \cdot \tau(G))$$

Gdje n predstavlja broj vrhova grafa, m predstavlja broj bridova, a $\tau(G)$ predstavlja broj razapinjućih stabala grafa G.

Prostornu složenost je izračunao kao:

$$O(n + m)$$

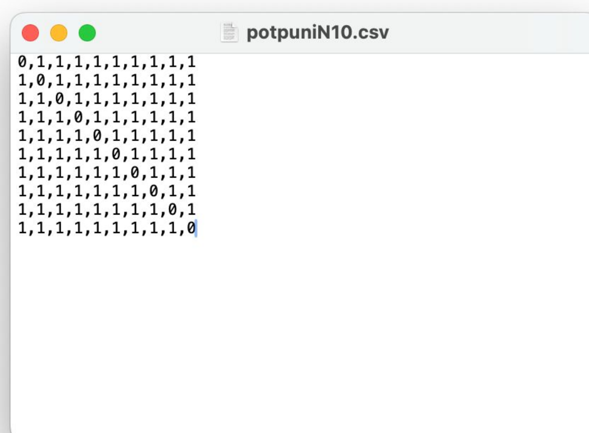
Uočimo da je prostorna složenost izrazito mala. Bitno je naglasiti da ako želimo postići željenu prostornu složenost ne smijemo spremati stabla već ih odmah ispisati po pronalasku. Ako spremamo stabla prostorna složenost raste. Problem kod ispisivanja rješenja je u tome što je operacija ispisivanja iznimno spora što znači da moramo žrtvovati vremensku složenost ako želimo maksimizirati prostornu složenost. Navedeni problem je naveo i Minty u svom članku [8].

2.4. Unos podataka i provjere matrica

Nakon što smo objasnili svaki algoritam ostaje nam objasniti još jednu stvar u procesu izrade koja je bitna, ali je ostala u drugom planu. Kako algoritmi dobiju podatke o grafu čija razapinjuća stabla tražimo?

2.4.1. Implementacija unosa podataka

Kao što smo naveli ranije svaki graf se može predstaviti matricom susjedstva i matricom incidencije. Naš program može na dva načina doći do matrice susjedstva, ovisno o korisničkom izboru. Korisnik može izabrati da sam unese željenu matricu. Kako pisanje matrice može biti iscrpno i repetitivno odlučili smo se za pristup da korisnik spremi matricu u .txt file te da pri upisu poštuje csv zapis (eng. comma-separated values). Korisnik predaje putanju do csv datoteke programu koji to čita i pretvara u matricu susjedstva. Tako korisnik može više puta pokrenuti program za istu matricu bez da je mora svaki put ponovno pisati. U nastavku dajemo primjer za potpuni graf s deset vrhova. 2.5.



Slika 2.5. Primjer unosa

Osim csv datoteke program nudi opciju i da sam generira matricu. Ako se korisnik odluči za navedenu opciju mora izabrati broj redaka matrice koji želi, šansu da se stvori brid između dva vrha (decimalni broj odvojen točkom) i broj grafova koji želi da se generira. Nakon što je korisnik uspješno odredio sve parametre program generira željeni broj matrica susjedstva te za svaku pojedinačno prebroji razapinjuća stabla. Način generiranja matrice ćemo detaljno objasniti u poglavlju Testiranje i rezultati 3.1.1.

2.4.2. Implementacija provjere matrice

Prije pokretanja algoritama za prebrojavanje razapinjućih stabala grafa, moramo osigurati da je matrica susjedstva ispravna i da je graf jednostavan, povezan te da su težine bridova jednake jedan. Ako program sam generira matricu onda ne trebamo nužno provjeriti sve uvjete jer se sam proces generiranja treba pobrinuti za određene uvjete kao što su da je težina svih bridova jedan i da je graf jednostavan. Također moramo se pobrinuti i za mnoge rubne slučajeve ako korisnik sam unosi graf. Matrica mora biti kvadratna, odnosno program mora prepoznati ako neki redak ili stupac imaju previše ili premalo elemenata. Trebamo se i pobrinuti da vrijedi ako je matrica kvadratna dimenzija $n \times n$, tada za svaki element matrice vrijedi $\text{matrica}[i][j] = \text{matrica}[j][i]$. Neovisno o načinu unosa grafa program se mora osigurati da je graf povezan, to može napraviti jednostavnom pretragom u širinu ili dubinu. Primjer pseudokoda funkcije za provjeru ispravnosti matrice:

Algoritam 10: Pseudokod za provjeru ispravnosti matrice

Ulazni argumenti: *matrica*

```
1  ako je matrica nije kvadratna
2      printf("Kriva matrica susjedstva");
3      return FALSE
4  ako je elementNaDijagonali != 0
5      printf("Nije jednostavan graf");
6      return FALSE
7  ako je elementMatrice != 0 i elementMatrice != 1
8      printf("Graf je tezinski ili sadrzi elemente koji nisu brojevi");
9      return FALSE
10 ako je matrica[i][j] != matrica[j][i]
11     printf("Bridovi su krivi");
12     return FALSE
13 ako je !DFS(matrica)
14     printf("Graf nije povezan");
15     return FALSE
16 return TRUE
```

3. Testiranje i rezultati

Nakon što smo sve temeljno teoretski obradili i implementirali algoritme i formulu ostaje nam još samo provjeriti točnost rješenja i prodiskutirati rezultate. U ovom poglavlju ćemo detaljno objasniti postupak testiranja, opisati model i vrstu provjere rješenja. Povući ćemo paralelu s tehnikama ispitivanja i predmetom Programsko inženjerstvo. Nakon što se uvjerimo da su rješenja točna u poglavlju 3.2. ćemo dodatno proučiti rješenja pogledati brzinu izvođenja i usporediti s teoretskom podlogom obrađenom u poglavlju 2.

3.1. Testiranje

Ranije smo naglasili da postoje formule za poznate grafove, tako znamo formulu za broj razapinjućih stabala potpunog grafa, grafa kotača, potpunog bipartitnog grafa i drugih 1.4. Ova informacija će nam biti od koristi u daljnjem postupku testiranja rješenja.

Prva faza

Prvo smo implementirali Kirchhoffovu formulu jer je najbrža i izrazito je korisna za provjeru rješenja ostalih algoritama. U toj početnoj fazi jedini način provjere jesmo li dobro implementirali Kirchhoffovu formulu bio je da ručno unosimo primjere poznatih grafova za koje već postoje formule ili grafova za koja smo sami prebrojali sva stabla. Nakon što smo se uvjerili da je formula dobro implementirana krenuli smo s implementacijom ostala dva algoritma.

Druga faza

Nakon što smo implementirali sva tri načina prebrojavanja mogli smo uspoređivati međusobno rješenja kako bi se uvjerili da je rješenje točno čak i za grafove za koje ne postoji

formula. U drugoj fazi smo ručno nalazili i unosili grafove te provjeravali daju li algoritmi ista rješenja. Vjerujemo da je zainteresirani čitatelj i sam već zaključio da navedena metoda nije pouzdana te da za istinsku provjeru treba provjeriti mnogo više grafova nego što jedna osoba ručno može ispitati u razumnom vremenu. U tu svrhu smo pristupili funkcijskom ispitivanju programa.

Treća faza

Funkcijsko ispitivanje se zasniva na tome da se programski kod promatra kao crna kutija koju treba ispitati. Ispitivač šalje ulaz i uspoređuje izlaz koji crna kutija daje. Kako bismo proces što više automatizirali napravili smo generator matrica susjedstva. Svrha generatora je da mi odaberemo broj vrhova grafa, šansu stvaranja brida između dva vrha i broj grafova koje želimo generirati, a da generator prema zadanim parametrima generira matrice, pokrene program i pošalje informaciju ako algoritmi nisu dali isto rješenje. Pretpostavka koju koristimo tijekom ovog ispitivanja je da barem jedan postupak daje točno rješenje za svaki graf. Nakon što sve implementiramo ostaje nam samo pokrenuti generator na velikom broju grafova kako bi provjerili na što većem uzorku točnost programskog rješenja.

Funkcijsko ispitivanje ima mnogo vrsta kao što su: kombinacijsko ispitivanje, podjela na ekvivalentne particije, analiza graničnih vrijednosti. Vrsta funkcijskog ispitivanja koje smo koristili zove se fuzz ili slučajno ispitivanje. U nastavku ćemo dodatno opisati model, slučajnu varijablu i generator koji smo koristili kod fuzz ispitivanja.

3.1.1. Model generatora grafova

Generator grafova stvara grafove prema zadanoj vjerojatnosti povezivanja između vrhova. Korisnik mu zadaje željenu vjerojatnost za stvaranje brida između dva proizvoljna vrha. Navedeni postupak je implementiran koristeći pseudoslučajne brojeve.

Vjerojatnost

U ovom modelu vjerojatnost je definirana od strane korisnika i ostaje konstantna tijekom cijelog postupka generiranja grafa. Ona predstavlja vjerojatnost da dva vrha budu povezana s bridom.

Pseudoslučajni brojevi

Vjerojatnost implementiramo s pomoću klase Random u Javi koja generira pseudoslučajne brojeve. Nakon što dobijemo pseudoslučajni broj uspoređujemo ga s vjerojatnosti, ako je broj manji od vjerojatnosti dva vrha povezujemo bridom.

Pseudokod

Algoritam 11: Generator grafova

```
   Ulazni argumenti: vjerojatnost, matrica
1  Random rand = new Random();
2  za i ← 0 do brojRedakaMatrice
3      za j ← i do brojRedakaMatrice
4          ako je i == j :
5              matrica[i][j] = 0;
6              CONTINUE;
7          ako je rand.nextDouble() < vjerojatnost :
8              matrica[i][j] = 1;
9              matrica[j][i] = 1;
10             CONTINUE;
11         matrica[i][j] = 0;
12         matrica[j][i] = 0;
13 return matrica
```

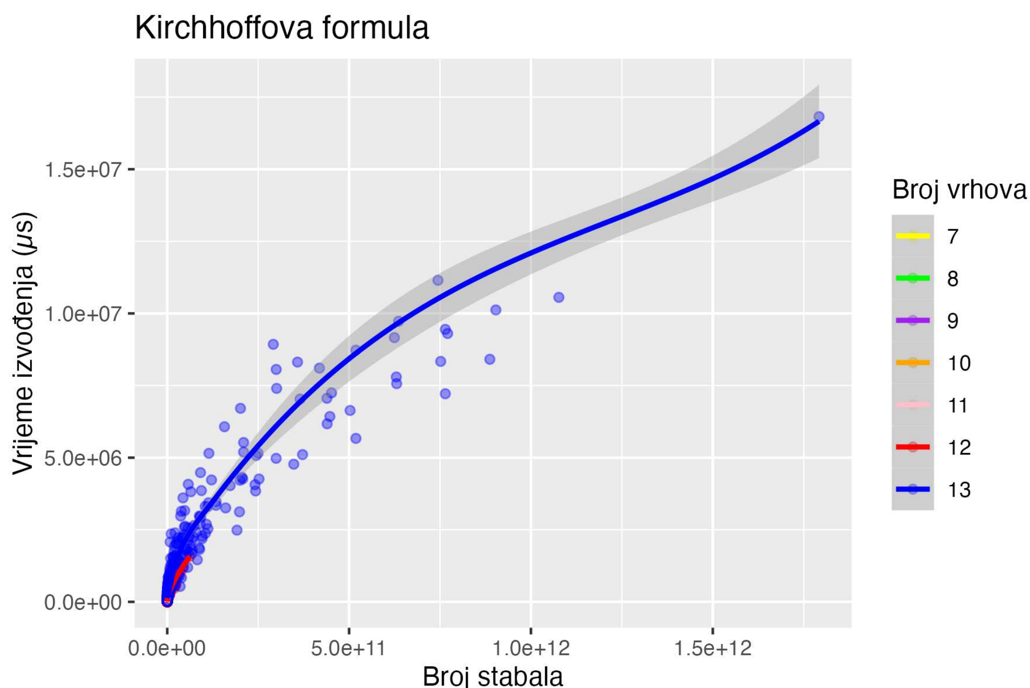
3.2. Rezultati

Vrijeme izvođenja formule i algoritama smo teoretski pokrili u poglavlju 2. Ostaje pogledati koliko dugo se izvode algoritmi u konkretnim primjerima. Iako je za učinkovit i brz kod bitno da je dobro optimiziran, na vrijeme izvođenja utječe i brzina računala na kojem se kod pokreće kao i programski jezik. Zbog toga je potrebno naglasiti specifikacije računala koje smo koristili za izvođenje koda i prikupljanje podataka koje ćemo vizualizirati u poglavlju Grafovi 3.2.1. Kod koji se izvrši i za koji pružamo informacije o vremenu izvođenja napisan je u programskom jeziku Java, koristeći Java Development Kit (JDK) verziju 19.0.2. JDK 19.0.2 pruža potrebne alate i biblioteke za razvoj i izvršenje Java aplikacija te je odabran zbog svoje stabilnosti i kompatibilnosti s ciljanim platformama. Kod se izvršavao na računalu koje ime 32 GB RAM-a i procesor Apple M1 MAX.

3.2.1. Grafovi

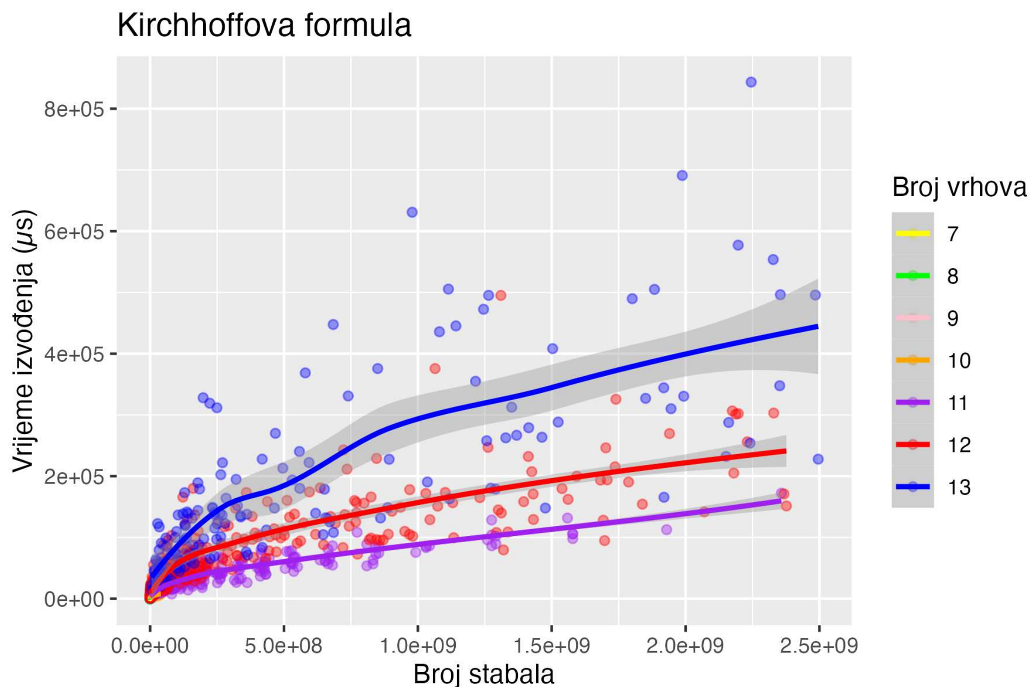
Kirchhoffova formula

Za Kirchhoffovu formulu smo naveli da vremenska složenost ovisi o broju vrhova grafa te da ona iznosi $O(n!)$ gdje n predstavlja broj vrhova. Sada možemo pogledati graf ovisnosti vremena izvođenja o broju vrhova i stabala. Možemo primijetiti da na grafu dominira



Slika 3.1. Kirchhoffova formula - vrijeme

vrijeme grafova koji imaju trinaest vrhova, osim što ima najviše vrhova grafovi s trinaest vrhova imaju najviše stabala što također usporava algoritam. No, kako bismo mogli bolje opisati i dokazati teoretsku pretpostavku da vrhovi utječu na vrijeme izvođenja formule moramo graf zumirati odnosno gledati vrijeme izvođenja kad je broj stabala jednak. Za to će nam poslužiti sljedeća slika. Na slici 3.2. možemo primijetiti da su linije očekivanja paralelne. Odnosno, čak i kada imamo isti broj stabala vrijeme izvođenja će biti veće za graf koji ima veći broj vrhova. Primijetimo i da se krivulja ponaša slično kao i logaritamska krivulja, odnosno u početku brzo raste nakon čega se brzina rasta usporava. Navedena opservacija je bitna jer pokazuje da broj stabala nije ključan kada računamo vrijeme izvođenja, odnosno da broj vrhova igra glavnu ulogu što je ujedno bio i rezultat teoretskog izračuna. Broj stabala i dalje ima ulogu, kao što je vidljivo na grafu, no ona je rezultat toga što broj grafova određuje popunjenost matrice odnosno određuje koliko će

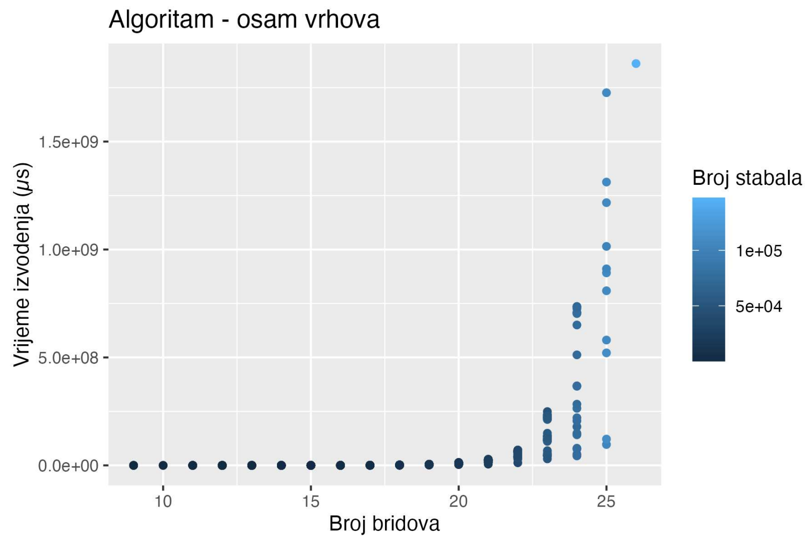


Slika 3.2. Kirchhoffova formula - vrijeme za jednaki broj stabala

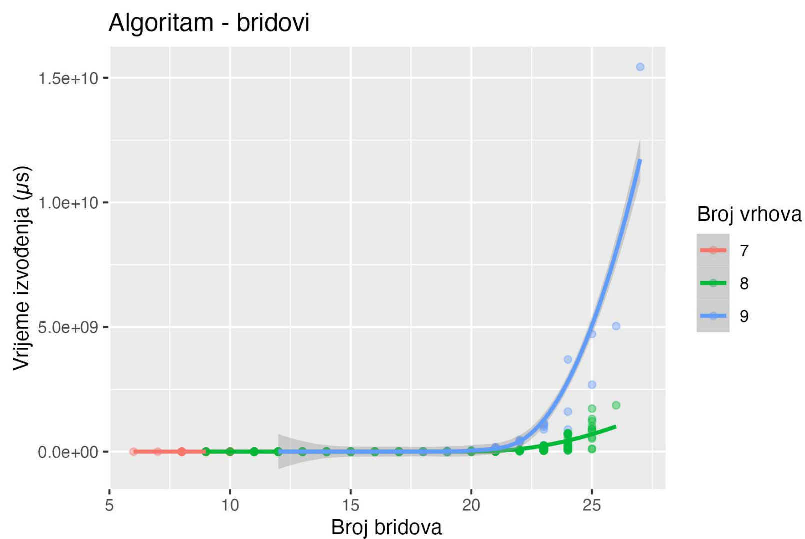
biti podrezivanja u procesu računanja determinante. Što je matrica praznija to će biti više podrezivanja, samim time će ušteda biti veća, no to neće smanjiti red veličine očekivanja.

Algoritam 1

Algoritam 1 je najsporiji od tri načina izračuna broja stabala koji smo implementirali. Za njega smo naveli da vremenska složenost ovisi o broju vrhova n te broju bridova m. Vremenska složenost algoritma je $\approx O(\binom{m}{n-1} \cdot n^2)$. Podatke koje ćemo prikazivati za algoritam 1 su većinski za graf s osam vrhova. Razlog tome je što za veći broj vrhova vrijeme izvođenja toliko naraste da bi prikupljanje dovoljnog broja podataka za prikazati kvalitetan graf trajalo predugo. Generirali smo i grafove s devet vrhova, no za pojedine grafove je trebalo i više od četiri sata za izračun broja vrhova. Ako uzmemo u obzir za dobar prikaz treba više od tisuću grafova dolazimo do zaključka da je optimalno izabrati grafove s osam vrhova. Na slici 3.3. vidimo da vrijeme izvođenja ovisi o broju bridova i da ono raste izrazito brzo. Ostaje nam dokazati da je vrijeme izvođenja veće za graf koji ima veći broj vrhova ako dva grafa imaju jednaki broj bridova. To ćemo pokazati na sljedećoj slici. Na slici 3.4. vidimo da je plava linija veća ili jednaka zelenoj liniji za svaki broj bridova. Time smo pokazali da vrijeme izvođenja ovisi ne samo o broju bridova već i o broju vrhova grafa G. Linije predstavljaju očekivano vrijeme izvođenja u ovisnosti



Slika 3.3. Algoritam 1 - bridovi/vrijeme



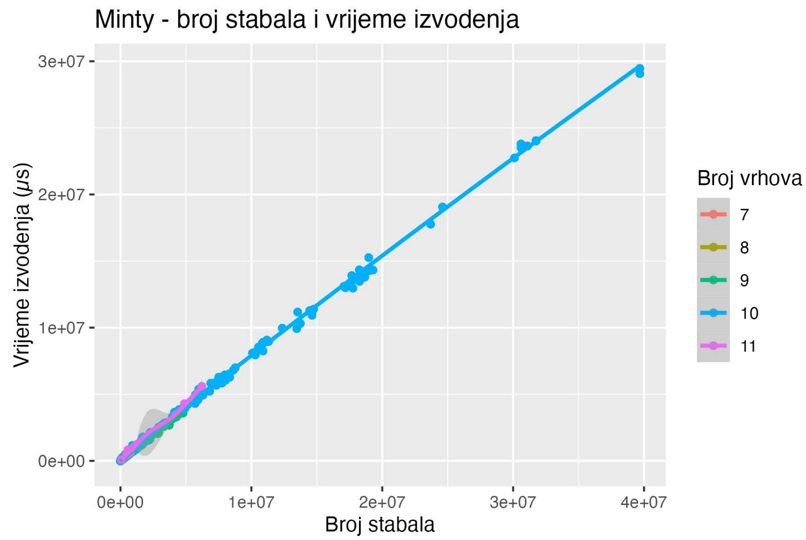
Slika 3.4. Algoritam 1 - bridovi/vrijeme/Vrh

o broju bridova. S pomoću prethodna dva grafa smo pokazali da se teoretski izračun poklapa s implementacijom.

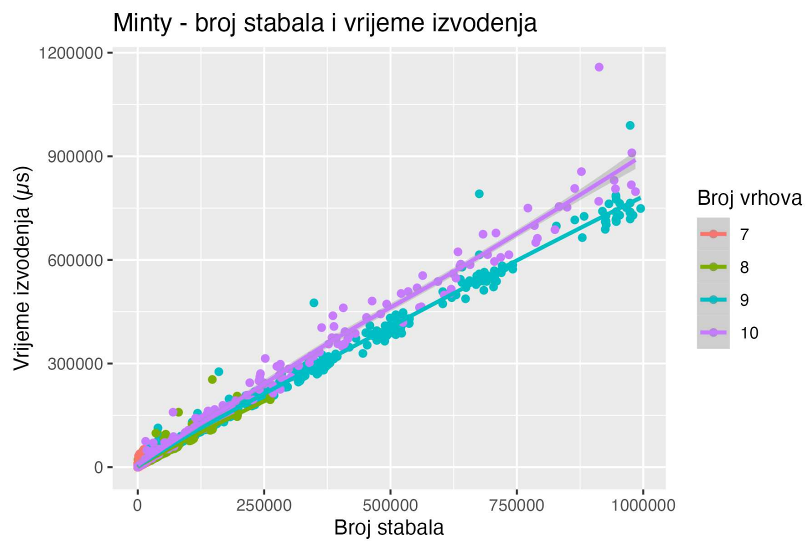
Mintyev algoritam

Za razliku od prethodna dva algoritma vremenska složenost Mintyevog algoritma ovisi ponajviše o broju razapinjućih stabala. Razlog tome je što Mintyev algoritam generira samo razapinjuće stabla.

Na slici 3.5. možemo primijetiti linearnu ovisnost vremena izvođenja i broja razapinjućih stabala što upravo odgovara formuli $O(n + m + m \cdot \tau(G))$. Možemo zumirati graf i pogledati ovisi li vrijeme izvođenja o broju vrhova grafa G. Možemo primijetiti da su linije



Slika 3.5. Minty - stabla/vrijeme



Slika 3.6. Minty - zumirano stabla/vrijeme

paralelne i da se razlikuju samo za neku malu konstantu. Zaključujemo da broj vrhova n nema ključnu ulogu u određivanju vremena izvođenja već služi kao svojevrsna konstanta, odnosno da najveći utjecaj na vremensku složenost ima upravo broj razapinjućih stabala $\tau(G)$. Prisjetimo se da to upravo odgovara formuli koju smo naveli u poglavlju 2.3.3.

4. Zaključak

Efikasno pronalaženje svih razapinjućih stabala je složen i iscrpljujući proces. Problem prebrojavanja stabala već je dobro teoretski opisan, postoje poznate formule, metode i algoritmi za prebrojavanje. Ipak teoretska podloga ne čini problem jednostavnim. Kompleksnost zadatka leži u veličini prostora rješenja i pronalaženju što efikasnijeg algoritma. U radu je implementirana i objašnjena Kirchhoffova formula kao i dva algoritma, prvi koristi metodu testiraj i odaberi, a drugi, Mintyev algoritam, koristi metodu sukcesivnog smanjivanja grafa. Mintyev algoritam je brži od prethodnog, no treba imati na umu da je ovo složen problem s velikim prostorom rješenja. Zbog eksponencijalnog rasta prostora rješenja vrijeme izvođenja Mintyevog algoritma može postati predugo za velike brojeve razapinjućih stabala. Ako želimo efikasnije riješiti problem treba posegnuti za naprednijim tehnikama prebrojavanja kao što je Winterov algoritam [5].

U daljnjem razvoju može se dodatno optimizirati kod. Jedan od primjera je u Kirchhoffovoj formuli koristiti napredne tehnike za računanje determinante. Trenutna vremenska složenost $O(n!)$ naprednim tehnikama računanja determinante može se smanjiti na $O(n^3)$.

Usprkos dugogodišnjem interesu matematičara za ovaj problem, nadamo se da će ovaj rad motivirati buduće znanstvenike da istraže i razviju još bolje metode za prebrojavanje svih razapinjućih stabala.

Literatura

[1] G. Kirchhoff, *Über die Auflösung der Gleichungen, auf welche man bei der Untersuchung der linearen Verteilung galvanischer Ströme geführt wird*, Ann. Phys. Chem., vol. 72, pp. 497-508, 1847.

[2] D. Kablar, *Prebrojavanje razapinjućih stabala grafa*, Acta mathematica Spalatensia Series didactica, vol. 5, pp. 48, 2022.

[3] D. Kovačević, M. Krnić, A. Nakić, M.O. Pavčević, *Diskretna matematika 1*, FER, vol. 3, pp. 63-83, 2020.

[4] D. Kovačević, M. Krnić, A. Nakić, M.O. Pavčević, *Diskretna matematika 1*, FER, vol. 6, pp. 117-129, 2020.

[5] P. Winter, *An algorithm for the enumeration of spanning trees*, BIT Numerical Mathematics, vol. 26, pp. 44-62, 1986.

[6] A. Cayley, *A theorem on trees*, Quart. J. Pure Appl. Math., vol. 23, pp. 376-378, 1889.

[7] M.J. Smith, *Generating Spanning Trees*, Department of Computer Science, University of Victoria, USA, 1997.

[8] G.J. Minty, *A simple algorithm for listing all the trees of a graph*, IEEE Transactions on Circuit Theory, vol. 12, pp. 120, 1965.

[9] T. Chartrand, E.L. John, *Generation of trees, two-trees and storage of master forests*, IEEE Transactions on Circuit Theory, vol. 15, no. 2, pp. 128-138, 1968.

[10] S. Sen Sarma, A. Rakshit, R. Sen, A. Choudhury, *An efficient tree generation algorithm*, Journal of the Institution of Electronics and Telecommunication Engineers,

vol. 27, no. 3, pp. 105-109, 1981.

[11] S. Naskar, K. Basuli, S. Sen Sarma, *Generation of all spanning trees of a simple, symmetric, connected graph*, National Seminar on Optimization Technique, Department of Applied Mathematics, University of Calcutta, 2007.

[12] C.E. Onete, M.C.C. Onete, *Enumerating all the spanning trees in an un-oriented graph—a novel approach*, XIth International Workshop on Symbolic and Numerical Methods, Modeling and Applications to Circuit Design (SM2ACD), 2010.

[13] S.L. Hakimi, *On trees of a graph and their generation*, Journal of the Franklin Institute, vol. 272, no. 5, pp. 347-359, 1961.

[14] W. Mayeda, S. Seshu, *Generation of trees without duplications*, IEEE Transactions on Circuit Theory, vol. 12, pp. 181-185, 1965.

[15] S. Kapoor, H. Ramesh, *Algorithms for enumerating all spanning trees of undirected and weighted graphs*, SIAM Journal on Computing, vol. 24, no. 2, pp. 247-265, 1995.

[16] T. Matsui, *An algorithm for finding all the spanning trees in undirected graphs*, Department of Mathematical Engineering and Information Physics, University of Tokyo, vol. 16, pp. 237-252, 1993.

Sažetak

Broj razapinjućih stabala grafa općenito je zahtjevan zadatak koji se ne može riješiti jednostavnim tehnikama elementarnog prebrojavanja. Iako je u općenitom slučaju teško prebrojati sva razapinjuća stabla zbog veličine prostora rješenja, postoje grafovi za koje je poznat broj razapinjućih stabala. Cilj ovog rada je opisati i implementirati formulu i algoritme za neke od osnovnih metoda prebrojavanja razapinjućih stabala. Prvo je implementirana Kirchhoffova formula koja je najefikasniji način za dobiti broj razapinjućih stabala, no ona ne pronalazi stabla već samo daje njihov ukupni broj. Zbog toga smo posegnuli za drugim algoritmima i posvetili se nekim od već razrađenih metoda prebrojavanja stabala. Napravili smo algoritam koji koristi metodu testiraj i odaberi. Taj algoritam ima visoku vremensku složenost te samim time nije efikasan. Zbog toga smo implementirali i Mintyev algoritam čija složenost raste linearno s brojem razapinjućih stabala. Iako je Mintyev algoritam mnogo brži od prethodnog, treba imati na umu da skup rješenja brzo raste s brojem bridova i vrhova te zbog toga izvođenje može biti dugotrajno za velike grafove.

Ključne riječi: Graf; Stabla; Razapinjuće stablo; Kirchhoffova formula; Mintyev algoritam; Metoda sukcesivnog smanjivanja grafa; Metoda testiraj i odaberi

Abstract

The task of counting the number of spanning trees in a graph is generally challenging and cannot be solved using simple elementary counting techniques. Although it is difficult to count all spanning trees in the general case due to the size of the solution space, there are graphs for which the number of spanning trees is well known. This work aims to describe and implement the formula and algorithms for some of the basic methods of counting spanning trees. First, Kirchhoff's formula was implemented, which is the most efficient way to obtain the number of spanning trees, but it only gives their number without finding the actual trees. Therefore, we resorted to other algorithms and focused on some of the well-established methods for counting trees. We developed an algorithm that uses the test and select method, which has a high-time complexity and is therefore inefficient. Consequently, we implemented Minty's algorithm, whose complexity grows linearly with the number of spanning trees. Although Minty's algorithm is much faster than the previous one, it should be noted that the solution space grows rapidly with the number of edges and vertices, and therefore, the execution time can be long for large graphs.

Keywords: Graph; Trees; Spanning Tree;Kirchhoff's Formula; Minty's Algorithm; Successive reduction of graph method; Test and select method