Univerzitet u Beogradu

Elektrotehnički fakultet



Rešavanje problema dominirajućeg skupa

na programskom jeziku Java

Diplomski rad

|  |  |
| --- | --- |
| Mentor: | Kandidat: |
| doc. dr Marko Mišić | Nemanja Mehović 2018/0452 |

Beograd, septembar 2022.

Sadržaj

[Sadržaj 2](#_Toc114770361)

[1. Uvod 3](#_Toc114770362)

[2. Dominirajući skup u teoriji grafova 4](#_Toc114770363)

[2.1. Dominirajući skup 4](#_Toc114770364)

[2.1.1. Pokrivanje skupa 6](#_Toc114770365)

[2.1.2. L-redukcija 6](#_Toc114770366)

[2.1.3. Primena dominirajućeg skupa 6](#_Toc114770367)

[2.2. Predmet i cilj istraživanja 8](#_Toc114770368)

[3. Analiza i detalji implementacija algoritma 9](#_Toc114770369)

[3.1. Impementacija grafa 9](#_Toc114770370)

[3.2. Pohlepan algoritam 11](#_Toc114770371)

[3.2.1. Implementacija pohlepnog algoritma 13](#_Toc114770372)

[3.3. Provera svih kombinacija 13](#_Toc114770373)

[3.3.1. Implementacija provere svih kombinacija 13](#_Toc114770374)

[3.4. Rešenja koristeći algoritme za pokrivanje skupa 14](#_Toc114770375)

[3.4.1. Trivijalno rešenje pokrivanje skupa 15](#_Toc114770376)

[3.4.2. Fomin et al. algoritam za pokrivanje skupa 19](#_Toc114770377)

[3.4.3. Rooji et al. algoritam za pokrivanje skupa 23](#_Toc114770378)

[4. Testiranje implementacija 30](#_Toc114770379)

[5. Zaključak 33](#_Toc114770380)

[Literatura 34](#_Toc114770381)

1. Uvod

U teoriji grafova postoje mnogi problemi koji su NP-kompletni. Za probleme ovog tipa je nemoguće naći egzaktna rešenja koja nisu eksponencijalne vremenske složenosti. Primer jednog poznatog NP-kompletnog problem je problem trgovačkog putnika (*traveling salesman problem*). U ovom radu vršićemo analizu jednog drugog NP-kompletnog problema, a to je problem minimalnog dominirajućeg skupa. Takođe analiziraćemo i implementirati pet algoritma za rešavanje samog problema minimalnog dominirajućeg skupa. Problem minimalnog dominirajućeg skupa predstavlja samo jednu varijaciju problema dominirajućeg skupa. Cilj minimalnog dominirajućeg skupa je nalaženje dominirajućeg skupa koji sadrži najmanji mogući broj elementa u sebi, a da se i dalje smatra dominirajućim skupom. U ovom radu pored teoretske analize vršićemo i testiranje implementacija samih algoritma.

Sam rad je podeljen u pet poglavlja. Prvo poglavlje predstavlja uvod. U njemu imamo mali opis sam problema i kog je karaktera.

Drugo poglavlje je dominirajući skup u teoriji grafova. Ovo poglavlje je većim delom teorijska analiza uopšteno problema dominirajućeg skupa. U ovom poglavlju definišemo šta je dominirajući skup, koje varijacije problema postoje i neke primene problema u stvarnom životu. Takođe u ovom poglavlju definisaćemo vezu između dominirajućeg skupa i problema pokrivanje skupa koja će nam biti od koristi pri analizi mogućih algoritma za rešavanje minimalnog dominirajućeg skupa.

Treće poglavlje se bavi analizom i implementacijom algoritma koja smo odlučili da koristimo kao rešenja ovog problema. Samo poglavlje sadrži pet algoritma od koji je jedan aproksimacija tačnog rešenja, drugi trivijalna provera svih kombinacija i ostali su rešenja koristeći algoritam za pokrivanje skupa.

Četvrto poglavlje se odnosi na testiranje samih implementacija iz trećeg poglavlja. U ovom poglavlju ćemo videti kako algoritmi funkcionišu u stvarnosti i porediti ih.

Peto poglavlje je zaključak u njemu ćemo reći da li smo uspeli da uspešno pronađemo algoritam koji bi mogao da se koristi u praksi za rešavanje samog problem.

1. Dominirajući skup u teoriji grafova

Neka *G = (V, E)* predstavlja graf sa *n* čvora gde *V* označava skup svih čvorova u grafu, a *E* označava sve veze u grafu. Postoje različite vrste grafova, u ovom radu ćemo podrazumevati da radimo sa prostim grafovima osim ako se ne naglasi suprotno.

Diagram, shape

Description automatically generated

Slika 2.0.1. Primer prostog grafa

Glavne osobine prostog grafa, kao što možemo videti na slici 2.0.1. je to što ne sadrži petlje niti paralelne grane. Grane se smatraju paralelnim u slučaju ako spajaju dva ista čvora. Dok petljom se smatra svaka grana koja spaja čvor sa samim sobom.

* 1. Dominirajući skup

Skup *D* predstavlja dominirajući skup ako je *D* podskup od *V* takav da svaki čvor koji nije u *D* je susedan barem jednom čvoru iz *D*. Drugim rečima dominirajući skup možemo definisati sledećim formulama:

(2.1.1)

(2.1.2)

Broj dominacije (*domination number*) se predstavlja simbolom γ(*G*) i predstavlja broj čvorova u minimalni dominirajući skup. Sam problem nalaženja dominirajućeg skupa predstavlja NP-kompletan problem. Dokaz ovoga možemo naći u [1]. Posledica ovoga je da ne postoji algoritam za rešavanje problema koji nije eksponencijalne vremenske složenosti osim ako se ne dokaže da je *NP=P*. Problem nalaženja dominirajućeg skupa ima više varijacija, od kojih su sledeće najpoznatije:

* Minimalni dominirajući skup je osnovni oblik problema kod koga je cilj da nađemo skup *D* takav da sadrži što manje čvorova, a da i dalje zadovoljava sve uslove potrebne da bi bio dominirajući skup. Ovaj uslov se nadalje neće spominjati u drugim varijacijama problema, ali i dalje važi.
* Povezani dominirajući skup, koristi se za nalaženje obuhvatnog stabla (*spanning tree*) *T* takvog da sadrži najveći mogući broj listova. Konkretna definicija ove varijante problema je da bilo koji čvor iz *D* može stići do bilo kog drugog čvora iz *D* prateći putanju koja ne napušta *D* [2].
* Ukupan dominirajući skup je skup čvorova takav da svi čvorovi u grafu, uključujući i čvorove koji pripadaju dominirajućem skupu *D*, imaju suseda koji pripada dominirajućem skupu *D* [3]*.*
* K-torka dominirajući skup (*K-tuple dominating set* ) podrazumeva da svaki čvor koji pripada dominirajućem skupu *D* ima barem k suseda koji takođe pripadaju skupu *D* [4].
* K-dominirajući skup je sličan iznad spomenutom k-torka dominirajućem skupu osim što sada svaki čvor koji ne pripada dominirajućem skupu *D* ima barem k suseda koji pripadaju skupu *D* [4].
* Večni dominirajući skup predstavlja dinamičku verziju dominirajućeg skupa u kome se čvor *v* iz dominirajućeg skupa *D* može zameniti susedom *u,* koji ne pripada dominirajućem skupu *D,* tako da novo dobijeni skup *D* ostaje dominirajući skup i ovaj postupak se može ponoviti nad beskonačnoj sekvenci čvorova *v* [5].

Bitna osobina kod nalaženja dominirajućeg skupa, koja će nam biti od koristi prilikom rešavanja problema, je to što se može transformisati u problem pokrivanje skupa (*set cover*). Ovo je moguće postići zato što postoji par linearnih redukcija (L-redukcija) koje to omogućavaju. Većina tačnih algoritma koji se koriste za nalaženje dominirajućeg skupa se bazira na ovom principu transformacije problema [6] i prvi put je korišćeno u [7].

Još jedna važna osobina dominirajućeg skupa jeste veza između njega i problema nezavisnih skupova (*independent sets*). Veza između dominirajućeg skupa i nezavisnih skupa jeste to da je nezavisan skup u isto vreme dominirajući skup ako i samo ako je nezavisan skup maksimalni nezavisni skup (*maximal* *independent set*). Nezavisan skup predstavlja skup čvorova u grafu takav da nijedan element skupa nije sused nekom drugom elementu iz istog skupa. Nezavisan skup je maksimalan nezavisni skup ako nije podskup nijednog drugom nezavisnog skupa.

* + 1. Pokrivanje skupa

Dat nam je skup *U* koji ćemo nazvati univerzum i skup *s* koji sadrži *m* skupova u sebi. Unija svih skupova u *s* daje kao rezultat univerzum, odnosno skup *U*. Problem pokrivanje skupa je pronalaženje minimalnog skupa *C* koji je podskup skupa *s* takav da unija svih elementa skupa *C* daje kao rezultat skup *U*. Drugim rečima ako nam je dat skup *U* = {1, 2, 3, 4, 5} i skup *s* = {{1, 2, 3}, {2, 4}, {3, 4}, {4, 5}} skup *C* će u tom slučaju biti {{1, 2, 3}, {4, 5}}.

* + 1. L-redukcija

Postoje dve L-redukcije što se tiče problema dominirajućeg skupa i problema pokrivanje skupa. Ove dve L-redukcije se odnose na transformaciju problema dominirajućeg skup u problem pokrivanje skupa i obrnuto. U ovom radu pokrićemo samo L-redukciju od dominirajućeg skupa do pokrivanje skupa pošto nam druga neće trebati.

Ako nam je dat graf *G* = (*V, E)* možemo ga transformisati u (*s, U)* na sledeći način. Univerzum *U* sadrži sve čvorove koje sadrži *V*, a skup *s* konstruišemo tako što za svaki čvor *v* iz *V* napravićemo skup *Sv* koji sadrži čvor *v* i sve njegove susede i zatim skup *Sv* dodamo skupu *s.* Drugim rečima matematički možemo definisati transformaciju na sledeći način:

(2.1.3)

(2.1.4)

Sada ako nam je skup *D* rešenje dominirajućeg skupa onda skup *C* kao rešenje pokrivanje skupa možemo predstaviti na sledeći način:

(2.1.5)

* + 1. Primena dominirajućeg skupa

Dominirajući skup ima razne primene u različitim oblastima. Na primer, u šahu je to problem kraljica (*queens problem*) [8].

Background pattern

Description automatically generated

Slika 2.1.1. Kraljice kako dominiraju šahovsku tablu [8]

Nalazimo najmanji broj kraljica koji možemo staviti na tabli veličine *m* x *m,* a da osiguramo da svako polje na tabli sadrži kraljicu ili je napadnuto od strane kraljice. Ovaj problem se vrlo lako može predstaviti grafom i time rešiti dominirajućim skupom. Možemo videti primer problema na slici 2.1.1.

A picture containing indoor, wire

Description automatically generated

Slika 2.1.2. Primer ad-hoc mobilne mreže sa konstruisanom kičmom [8]

Postoje takođe mnogi problemi koji nisu toliko teoretski, kao problem kraljica, za koje se može koristiti dominirajući skup ili neka njegova varijacija da bi se rešio. Mnogo takvi problemi dolaze iz oblasti računarskih mreže ili telekomunikacija. Kao kod *ad-hoc* mobilnih mreža (*mobile ad-hoc network*) [8]*,* koje predstavljaju samopodešavajuću mrežu mobilnih uređaja. Kod ovog problema se koristi povezani dominirajući skup da bi se uspostavio virtualnu kičmu (*backbone*) mreže kao na slici 2.1.2, kroz koju se radi rutiranje i emitovanje saobraćaja.

* 1. Predmet i cilj istraživanja

U ovom radu vršićemo analizu problema minimalnog dominirajućeg skupa. Probaćemo da nađemo egzaktno rešenje problema koristeći algoritme različitih vremenskih kompleksnosti. Kod samih algoritma prvo ćemo vršiti teorijsku analizu nakon čega ćemo analizirati implementaciju samog algoritma i koje nedostatke sama implementacija ima.

Pored teorijske analize algoritma takođe ćemo meriti performanse samih algoritma u Javi. Koristićemo implementacije koje se spominju u samom radu. Time naš zaključak o samim algoritmima i njihovim performansama neće biti samo teorijski. Imaćemo konkretno vreme izvršavanja svakog algoritma i time ćemo doneti naš zaključak.

1. Analiza i detalji implementacija algoritma

Kao što je rečeno u prethodnom poglavlju sam problem nalaženja dominirajućeg skupa je NP-kompletan. To znači da najbolji tačan algoritam koji možemo naći za rešavanje ovog problema vremenske kompleksnosti oblika *αn* gde je *α* > 1, a *n* predstavlja broj čvorova u grafu. Ovo nas ne sprečava da nađemo algoritam koji nam pruža dovoljno dobru aproksimaciju rešenja u polinomijalnom vremenu.

* 1. Impementacija grafa

Sve implementacije rešenja koja ćemo razmatrati u ovom poglavlju koriste istu reprezentaciju grafa. Implementacija koja se koristi je lista susednosti realizovana uz pomoć *Node* i *Graph* klasa.

public class Graph

**{**

private static Graph instance **=** **null;**

private HashMap**<**String**,** Node**>** allNodes**;**

private List**<**Set**<**Node**>>** allEdges**;**

public static Graph InitializeGraph**(**String InputFileName**);**

public static Graph GetInstance**();**

private Graph**(**String InputFileName**);**

private Node GetCreateNode**(**String id**);**

public int GetNumberOfEdges**();**

public List**<**Node**>** GetAllNodes**();**

public List**<**Set**<**Node**>>** GetAllEdges**();**

**}**

Primer koda 3.1.1.

Kod 3.1.1. nam predstavlja potpis klase *Graph*. Iz njega možemo videti da je graf u samom programu realizovan kao unikat i da se sam graf učitava iz fajla koji se prosleđuje programu kao parametar. Takođe možemo videti da ima polje allEdges koji se može smatrati suvišnim zato što je moguće naći sve veze koristeći samo podatke koje možemo izvući iz allNodes. Čak iako je moguće to uraditi to bi bila vremenski intenzivna operacija, pa smo radi vremenske efikasno žrtvovali memorijsku efikasnost.

public class Node

**{**

private static int globalIdCounter **=** 0**;**

private int id**;**

private String name**;**

private List**<**Node**>** connected**;**

private boolean isDominated**;**

public static void ResetIdCounter**();**

public Node**(**String name**);**

public boolean AddNeighbour**(**Node neighbour**);**

public int NumberOfNeighbours**();**

public String GetName**();**

public List**<**Node**>** GetConnected**();**

public int GetId**();**

public void SetDominated**(**boolean isDominated**);**

public int GetHowManyWouldBeDominated**();**

**}**

Primer koda 3.1.2.

Možemo videti u kodu 3.1.2. kako je zamišljena implementacija čvora u grafu. Tu vidimo da svaki čvor može čuvati svoje trenutno stanje da li je u dominirajućem skupu. To postiže sa poljem isDominated takođe možemo i videti kako je postignuta realizacija liste susednosti koristeći polje connected, koje pretstavlja listu svih suseda trenutnog čvora. Najbitnijom metodom u samoj implementaciji čvora se može smatrati da je GetHowManyWouldBeDominated koja se koristi kao jednostavna heuristika za pohlepan algoritam, sama definicija joj je sledeća:

public int GetHowManyWouldBeDominated**()**

**{**

int count **=** isDominated **?** 0 **:** 1**;**

**for(**Node neighbour **:** connected**)**

**if(!**neighbour**.**isDominated**)**

count**++;**

**return** count**;**

**}**

Primer koda 3.1.3.

* 1. Pohlepan algoritam

Prvi algoritam koji ćemo razmatrati kao rešenje samog problema je jednostavan pohlepan algoritam. On funkcioniše na vrlo prostom principu, koristeći heuristiku određuje koji čvor da ubaci u potencijalni dominirajući skup. Nakon toga proveri da li potencijalni skup zadovoljava sve uslovne potrebne da bi bio dominirajući skup, u slučaju da zadovoljava potencijalni skup postaje dominirajući skup i algoritam se završava. U slučaju da ne zadovoljava uslove potrebne da postane dominirajući skup algoritam nastavlja sa radom sve dok potencijalni skup ne postane dominirajući skup. Heuristika sa kojom će se određivati koji čvor trebamo da ubacimo u potencijalno rešenje, je brojanje koliko čvorova bi postalo dominirano njegovim ubacivanjem u rešenje.

Ovo je jedini algoritam koji ćemo posmatrati, a da nije eksponencijalne vremenske složenosti. Vremenska složenost ovog algoritma se može smatrati da je u teoriji *O*(*n*). To se može dokazati vrlo jednostavno ako uzmemo u obzir da ovaj algoritam u svakoj iteraciji uzima jedan čvor da doda potencijalnom dominirajućem skupu, a znamo da je skup svih čvorova po definiciji dominirajući skup znači možemo maksimalno uzeti *n* čvora, odnosno možemo maksimalno imati *n* iteracija algoritma. Naravno ova aproksimacija vremenske složenosti je vrlo gruba i ne uzima u obzir vreme potrebno za računanje same heuristike čvora, kao ni vreme potrebno da se proveri da li neki potencijalni dominirajući skup zadovoljava sve uslove potrebne da bi postao pravi dominirajući skup. Sama analiza vremenske složenosti se odnosila na najgori slučaj gde bi graf imao 0 veza i *n* izolovanih čvorova.

Ovaj algoritam nam takođe ne garantuje minimalni dominirajući skup, već nam nalazi aproksimaciju. Aproksimacija dominirajućeg skupa se naravno može poboljšati sa boljom heuristikom.

A picture containing clock, watch

Description automatically generated

Slika 3.2.1. Primer dobrog rešenja sa prostom heuristikom

Na primer za iznad dat graf naša heuristika bi radila sasvim dobro. Našla bi minimalni dominirajući skup iako nam je heuristika jednostavna i samo broji koliko bi čvora bilo dominirano.

Diagram

Description automatically generated

Slika 3.2.2. Primer lošeg rešenja sa prostom heuristikom

Dok za graf na slici 3.2.2. bi našao dominirajući skup koji sadrži čvor 2 i još tri čvora. Međutim i u ovakvoj situaciji rešenje da nam algoritam vrati i dalje minimalni dominirajući skup je prilično jednostavno. Možemo primetiti da čvorovi 4, 0 i 6 svi imaju samo po jednog suseda, što znači da dominirajući skup mora da sadrži čvorove 4, 0 i 6 ili njihove susede. Dodavanje čvora koji ima samo jedan sused u dominirajući skup nije isplativo pošto dodavanjem njegovog suseda mi ćemo dominirati oba čvora i dodatno bilo koji drugi čvor koji mu je sused. Što znači da nam je popravka za heuristiku vrlo jednostavna, uvek ćemo dodati sused čvora koji ima samo jednog suseda u dominirajući skup. Sa tom promenom dobili bismo dominirajući skup *D* = {1, 3, 5} što je i minimalni dominirajući skup.

Diagram

Description automatically generated

Slika 3.2.3. Primer lošeg rešenja sa unapređenom heuristikom

Čak i sa poboljšanom heuristikom za graf na slici 3.2.3. mi nećemo uspeti da nađemo minimalni dominirajući skup. Ovo naravno ne znači da ne postoji heuristika koja bi nam to omogućila, ali služi kao primer da bilo koja heuristika koja bi zadovoljila svaki mogući graf i i dalje nalazila minimalni dominirajući skup bi bila suviše kompleksna za ovaj rad i vremenska složenost ne bi više bila polinomijalna.

* + 1. Implementacija pohlepnog algoritma

U kodu 3.2.1. možemo videti najbitniji deo implementacije pohlepnog algoritma. Vidimo da je izbor čvora, koji ćemo dodati u potencijalni dominirajući skup, implementirano uz pomoć prioritetnog reda. Za prioritet se uzima heuristika samog čvora. Takođe možemo videti negativnu stranu ove implementacije što je da prioritetan red u Javi ne može sebe da ažurira kada se promene prioriteti samih elementa reda. Zbog čega moramo posle svake iteracije algoritma ažurirati red ručno tako što izbacujemo iz njega elemente čiji su se prioriteti promenili, pa ih zatim vraćamo u red.

priorityQueue**.**addAll**(**Graph**.**GetInstance**().**GetAllNodes**());**

**do**

**{**

Node newNodeToAddToSet **=** priorityQueue**.**peek**();**

newNodeToAddToSet**.**SetDominated**(true);**

dominatingSet**.**add**(**newNodeToAddToSet**);**

Set**<**Node**>** changedGreedyValueSet **=** **new** HashSet**<>();**

changedGreedyValueSet**.**add**(**newNodeToAddToSet**);**

**for(**Node neighbour **:** newNodeToAddToSet**.**GetConnected**())**

**{**

neighbour**.**SetDominated**(true);**

changedGreedyValueSet**.**add**(**neighbour**);**

changedGreedyValueSet**.**addAll**(**neighbour**.**GetConnected**());**

**}**

priorityQueue**.**removeAll**(**changedGreedyValueSet**);**

priorityQueue**.**addAll**(**changedGreedyValueSet**);**

**}while(!**Solver**.**CheckIfDominating**(**dominatingSet**));**

Primer koda 3.2.1.

* 1. Provera svih kombinacija

Najprostiji algoritam koji možemo iskoristiti za nalaženje egzaktnog minimalnog dominirajućeg skupa jeste provera svih kombinacija. Kod ovog algoritma se samo prolazi kroz svaki mogući skup čvorova koji može da se napravi i traži se najmanji skup koji zadovoljava uslove dominirajućeg skupa. Ovaj algoritam je eksponencijalne vremenske kompleksnosti zato što za graf koji ima *n* čvora moguće je sastaviti 2*n*različitih skupova.

* + 1. Implementacija provere svih kombinacija

U kodu 3.3.1. možemo videti da za konstrukciju svih mogućih kombinacija čvorova smo koristili bit mapu. Ograničenje ove implementacije jeste to sto long u Javi ima 64 bita znači teoretski ovu implementaciju ne možemo koristiti za testiranje grafova koji imaju više od 64 čvora. Naravno ako bismo želeli da koristimo ovaj algoritam za veće grafove vreme izvršavanja bi bilo preveliko zbog eksponencijalne vremenske kompleksnosti. Takođe možemo videti da u ovoj implementaciji vremenska kompleksnost nije *O*(*2n*) nego je već *O*(*n2n*) zato sto u svakoj iteraciji takođe moramo proći kroz sve čvorove da bismo videli da li određeni čvor pripada trenutnom skupu.

dominatingSet **=** allNodes**;**

long val **=** **(**long**)** Math**.**pow**(**2**,** allNodes**.**size**());**

**for(**long i **=** 1**;** i **<** val**;** i**++)**

**{**

List**<**Node**>** vals **=** **new** ArrayList**<>();**

**for** **(**int j **=** 0**;** j **<** allNodes**.**size**();** j**++)**

**if** **(((**i **>>** j**)** **&** 1**)** **==** 1**)**

vals**.**add**(**allNodes**.**get**(**j**));**

**if** **(**vals**.**size**()** **>=** dominatingSet**.**size**())**

**continue;**

**if** **(**Solver**.**CheckIfDominating**(**vals**))**

dominatingSet **=** vals**;**

**}**

Primer koda 3.3.1.

* 1. Rešenja koristeći algoritme za pokrivanje skupa

Sve algoritme koje nadalje spominjemo su za rešavanje problema pokrivanje skupa. Znamo da zbog L-redukcija možemo problem dominirajućeg skupa transformisati u problem pokrivanje skupa. Svaki nov algoritam koji navedemo u ovom poglavlju predstavlja tačan algoritam koji je eksponencijalne vremenske složenosti.

Pre nego što pređemo na analizu samih algoritma trebamo definisati par termina kao što su *s*(*e*) i frekvencija od *e*,odnosno *f*(*e*):

(3.4.1)

(3.4.2)

Drugim rečima *s*(*e*) je skup svih skupova koji sadrže element *e*. Dok frekvencija od *e* predstavlja koliko ima skupa koji sadrže element *e.*

Takođe važno je spomenuti da svi algoritmi koji se nadalje rade funkcionišu na principu granaj i smanji (*branch and reduce)* [6], ovakvi tipovi algoritma su prvi put definisani u [9]. Algoritmi koji koriste ovu paradigmu su rekurzivnog tipa i imaju dva skupa pravila. Prvi skup pravila se odnosi na smanjenje. Ova pravila služe da bi trenutnu instancu, koja se izvršava, ujednostavila u manju ekvivalentnu instancu. U slučaju da trenutna instanca ne zadovoljava nijedan uslov za smanjivanje prelazi se na pravila za grananje. Drugi skup pravila se odnosi na grananje. Funkcioniše tako što se naprave dve ili više instance, modifikovanih na neki način u zavisnosti od samog pravila, i algoritam se poziva rekurzivno nad njima.

Analiza algoritma za tačnu vremensku kompleksnost, koji koriste ovu paradigmu, je prilično nezgodo baš zbog toga što su algoritmi rekurzivni i u svakoj iteraciji mogu započeti dve ili više nove instance izvršavanja algoritma. Metoda koja se koristi za analizu ovakvih problema se zove meri i savladaj (*measure and conquer)*. Ovu metodu analize su smislili Foomin et al. u [10]. Zbog same kompleksnosti vremenske analize u ovom radu nećemo se time baviti. Koristićemo vrednosti koje su već izračunate od strane Rooji et al. u [6] i referenciraćemo ih kad bude imalo potrebe.

* + 1. Trivijalno rešenje pokrivanje skupa

Prvi algoritam koji ćemo analizirati predstavlja trivijalno rešenje koristeći granaj i smanji. Ovaj algoritam takođe predstavlja osnovni slučaj ostalih algoritma koje ćemo obraditi u ovom poglavlju i vremenske je kompleksnosti *O*(*2n*) [6]. Tačnije ostali algoritmi predstavljaju unapređenu verziju ovog algoritma.

Text

Description automatically generated

Slika 3.4.1. Pseudokod trivijalnog algoritma [6]

Možemo videti iz pseudokoda na slici 3.4.1. kako sam algoritam funkcioniše. Vidimo da za prekid rekurzije neophodno je da skup skupova *s* bude prazan. Nakon čega u zavisnosti od toga da li je i univerzum *U* prazan odlučuje se da li je ova grana rekurzije našla rešenje ili je ćorsokak. Sam algoritam funkcioniše na vrlo jednostavan način. Uzimamo skup *S* iz *s* tako da *S* predstavlja skup najveće kardinalnosti, drugim rečina skup *S* predstavlja skup sa najviše elementa iz *s*. Posle čega imamo dve situacije da razmotrimo.

U prvoj situaciji uzimamo skup *S* da je deo rešenja i izbacujemo ga iz skupa skupova *s* kao i sve njegove elemente, što automatski znači da sve njegove elemente isto izbacujemo iz univerzuma *U* pošto se smatraju pokrivenim. Zatim pozivamo algoritam nad novim *s* i *U*. Ako nov rekurzivni poziv vrati vrednost koja nije *null* znamo da skup *S* pripada skupu za pokrivanje trenutnog *s* i *U*.

Kod druge situacije odlučujemo da *S* ne pripada dominirajućem skupu i izbacujemo ga iz *s*. Ponovo pozivamo algoritam rekurzivno nad modifikovanim *s* i ovog puta nad istim univerzumom *U*. Kada se rekurzivni poziv završi i ako vrati vrednost koja nije *null*, znamo da skup koji nam je vratio rekurzivan poziv predstavlja skup koji pokriva trenutno *s* i *U*.

Kada se završe ove situacije poredimo rezultate koje smo dobili i uzimamo najbolji. Pošto pokušavamo da nađemo minimalni skup pokrivanja za *s* i *U* najbolji rezultat će biti onaj skup koji je manji. Ako obe grane vrate *null* to znači da trenutno *s* i *U* nemaju skup pokrivanja i vraćamo *null*. Ovo je samo moguće ako i trenutni poziv algoritma u stvari predstavlja rekurzivni poziv. Možemo biti sigurni ove činjenice, pošto jedini način da ne postoji pokrivajući skup za *s* i *U* je ako su na neki način bili modifikovani. To se može samo dogoditi u slučaju da se radi o nekoj rekurzivnoj grani zato što početno *s* i *U* zavise od *V* grafa kao što možemo videti u formulama 2.1.3. i 2.1.4.

* + - 1. Implementacija trivijalnog algoritma za pokrivanje skupa

Sada kada smo završili sa teorijskom analizom algoritma prelazimo na samu implementaciju. Pre nego sto počnemo analizu cele implementacije algoritma prvo ćemo se fokusirati na par funkcija koje su važne za samu implementaciju i koju ćemo koristiti u kasnijim implementacijama.

protected List**<**List**<**Integer**>>** Take**(**List**<**List**<**Integer**>>** setOfSets**,**

List**<**Integer**>** toBeTaken**,**

HashMap**<**Integer**,** Integer**>** fixIndexesMap**)**

**{**

List**<**List**<**Integer**>>** newSetOfSets **=** **new** ArrayList**<>();**

int j **=** 0**;**

int i **=** 0**;**

**for(**List**<**Integer**>** tmpSet **:** setOfSets**)**

**{**

**if(**tmpSet **!=** toBeTaken**)**

**{**

List**<**Integer**>** modifiedSet **=** tmpSet**.**stream**()**

**.**filter**(**integer **->** **!**toBeTaken**.**contains**(**integer**))**

**.**collect**(**Collectors**.**toList**());**

**if(!**modifiedSet**.**isEmpty**())**

**{**

newSetOfSets**.**add**(**modifiedSet**);**

fixIndexesMap**.**put**(**j**,** i**);**

j**++;**

**}**

**}**

i**++;**

**}**

**return** newSetOfSets**;**

**}**

Primer koda 3.4.1.

U kodu 3.4.1. možemo videti implementaciju funkcije Take, koja se koristi u implementaciji algoritma da simulira prvu granu pseudokoda, odnosno služi da bi uzela određeni skup za rešenj. Vidimo da su dva parametra funkcije setOfSets i toBeTaken, ove promenljive respektivno predstavljaju skup skupova *s* iz pseudokoda i skup maksimalne kardinalnosti *S*. Takođe vidimo da postoji još jedan parametar fixIndexesMap on će se kasnije koristiti u funkciji iz koda 3.4.3. da popravi rešenje koje dobijemo od samog algoritma, dok ovde služi samo da zabeleži promene između novog *s*, koji smo napravili za sledeću iteraciju algoritma, i starog *s*. Sama funkcija radi vrlo jednostavno uzima sve skupove iz setOfSets izbacuje iz njih sve elemente skupa toBeTaken i ako nisu prazni stavlja ih u newSetOfSets koji kasnije vraća kao novo *s*.

protected List**<**List**<**Integer**>>** Remove**(**List**<**List**<**Integer**>>** setOfSets**,**

List**<**Integer**>** toBeTaken**,**

HashMap**<**Integer**,** Integer**>** fixIndexesMap**)**

**{**

List**<**List**<**Integer**>>** newSetOfSets **=** **new** ArrayList**<>();**

int j **=** 0**;**

int i **=** 0**;**

**for(**List**<**Integer**>** tmpSet **:** setOfSets**)**

**{**

**if(**tmpSet **!=** toBeTaken**)**

**{**

newSetOfSets**.**add**(**tmpSet**);**

fixIndexesMap**.**put**(**j**,** i**);**

j**++;**

**}**

i**++;**

**}**

**return** newSetOfSets**;**

**}**

Primer koda 3.4.2.

Možemo videti da funkcija Remove iz koda 3.4.2. isto kao Take iz koda 3.4.1. predstavlja granu iz pseudokoda. U ovoj situaciji ne uzimamo toBeTaken kao deo rešenja nego ga već samo izbacujemo iz setOfSets, što znači da nema potrebe da njegove elemente izbacimo iz svih ostalih skupova. fixIndexesMap kao i u Take označava samo promene između newSetOfSets i setOfSets.

protected void RestoreSolution**(**List**<**Integer**>** solution**,**

HashMap**<**Integer**,** Integer**>** fixIndexes**)**

**{**

**for(**int i **=** 0**;** i **<** solution**.**size**();** i**++)**

solution**.**set**(**i**,** fixIndexes**.**get**(**solution**.**get**(**i**)));**

**}**

Primer koda 3.4.3.

Kod 3.4.3. sadrži definiciju funkcije RestoreSolution koja ima parametre solution i fixIndexes. fixIndexes je isti parametar kao i fixIndexesMap iz funkcija Take i Remove. Dok solution predstavlja rešenje samog algoritma, odnosno skup pokrivanja za trenutni *s* i *U*. Vidimo da ne sadrži druge skupove u sebi nego već cele brojeve. Ovi celi brojevi u stvari predstavljaju indekse samih skupova. Koriste se kao rešenje zbog čega je i potrebno koristiti fixIndexes, da bi se indeksi novog *s* zamenili, pravim indeksima skupova, iz starog *s* odnosno sa indeksima iz skupa za koje smo tražili rešenje. Ove funkcije Take, Remove i RestoreSolution se koriste u sve tri implementacije algoritama koje obrađujemo u ovom potpoglavlju.

protected List**<**Integer**>** MSC**(**List**<**List**<**Integer**>>** setOfSets**,**

List**<**Integer**>** universe**)**

**{**

**if(**setOfSets**.**isEmpty**())**

**{**

**if(**universe**.**isEmpty**())**

**return** **new** ArrayList**<>();**

**return** **null;**

**}**

List**<**Integer**>** maxCardinality **=** GetMaxCardinality**(**setOfSets**);**

int indexMaxCardinality **=** setOfSets**.**indexOf**(**maxCardinality**);**

HashMap**<**Integer**,** Integer**>** fixIndexesMap **=** **new** HashMap**<>();**

List**<**Integer**>** newUniverse **=** universe**.**stream**()**

**.**filter**(**integer **->** **!**maxCardinality**.**contains**(**integer**))**

**.**collect**(**Collectors**.**toList**());**

List**<**List**<**Integer**>>** newSetOfSets **=** Take**(**setOfSets**,** maxCardinality**,**

fixIndexesMap**);**

List**<**Integer**>** C1 **=** MSC**(**newSetOfSets**,** newUniverse**);**

**if(**C1 **!=** **null)**

**{**

RestoreSolution**(**C1**,** fixIndexesMap**);**

C1**.**add**(**indexMaxCardinality**);**

**}**

fixIndexesMap **=** **new** HashMap**<>();**

newUniverse **=** universe**;**

newSetOfSets **=** Remove**(**setOfSets**,** maxCardinality**,** fixIndexesMap**);**

List**<**Integer**>** C2 **=** MSC**(**newSetOfSets**,** newUniverse**);**

**if(**C2 **!=** **null)**

RestoreSolution**(**C2**,** fixIndexesMap**);**

**if(**C1 **==** **null** **&&** C2 **==** **null)**

**return** **null;**

**if(**C1 **==** **null)**

**return** C2**;**

**if(**C2 **==** **null)**

**return** C1**;**

**return** C1**.**size**()** **>** C2**.**size**()** **?** C2 **:** C1**;**

**}**

Primer koda 3.4.4.

Prilikom implementacije algoritma u kodu 3.4.4. pokušali smo što manje da odstupamo od pseudokoda. Vidimo kako se Take i Remove koriste za računanje novog *s*, u kodu newSetOfSets. Kako se rezultati rekurzivnih poziva C1 i C2 popravljaju sa RestoreSolution koristeći fixIndexesMap i kako se na kraju porede da bi se odlučilo koji rezultat da se vrati. Takođe možemo ovde videti kako se računa novo *U*, u kodu newUniverse, kod prve grane filtriranjem svih elementa koji ne pripadaju skupu maksimalne kardinalnosti, dok kod druge grane novo *U* je jednako starom *U*.

* + 1. Fomin et al. algoritam za pokrivanje skupa

Algoritam koji ćemo sada obraditi predstavlja prvu unapređenu verziju algoritma iz poglavlja 3.4.1. Analiza ovog algoritma se može naći u [10] i [6] i vremenske je složenosti *O*(*1.5263n*) [6].

Text, letter

Description automatically generated

Slika 3.4.2. Pseudokod Fomin et al. algoritma [6]

Kao što možemo videti na slici 3.4.2. pseudokod ovog algoritma sadrži ceo pseudokod algoritma na slici 3.4.1. Tako da nećemo se baviti analizom delova koje smo prošli u poglavlju 3.4.1. Već ćemo se baviti samo novim granama koje su uvedene u ovom algoritmu. Nove grane ćemo analizirati istim redosledom kojim se javljaju u pseudokod.

Tako da novu granu koju ćemo obraditi jeste grana koja se javlja na liniji 3. Ova grana je prilično jednostavna i nema potrebe da se na njoj zadržavamo previše. Uslov ulaska u granu jeste ako u univerzum *U* postoji element *e* čija je frekvencija *f*(*e*) jedan. Drugim rečima element *e* se nalazi samo u jednom skupu. Ako takav element postoji onda skup u kome se nalazi se uzima kao deo rešenja. Ovo pravilo je vrlo logično sam problem zahteva da unija svih skupova u pokrivajući skup daje kao rezultat univerzum *U*, a pošto se element *e* nalazi samo u jednom skupu taj skup mora biti deo rešenja. Postupak uzimanja skupa koji sadrži *e* je isti kao i kod prve grane algoritma iz poglavlja 3.4.1.

Druga grana koju ćemo analizirati nalazi se na liniji 5. Kod ove grane gledaju se dva skupa *R* i *Q* iz skupa skupova *s* i uslov ulaska u granu je sledeći. Mora da postoji barem jedan par skupova *R* i *Q* takav da je skup *R* podskup skupa *Q*. Ako je ovaj uslov zadovoljen možemo izbaciti skup *R* iz skupa skupova. Ovo činimo na isti način kao i u drugoj grani kod algoritma u poglavlju 3.4.1 Razlog zbog čega se ovo može raditi je vrlo jednostavan. Ako skup *Q* sadrži sve iste elemente skupa *R* mi nikada nećemo imati potrebu da dodamo skup *R* u naše rešenje. Zato što možemo pokriti sve elemente iz skupa *R* biranjem skupa *Q* dok suprotno nije moguće.

Poslednja nova grana u ovom algoritmu počinje na liniji 7 u pseudokodu datom na slici 3.4.2. Uslov da bi se ušlo u ovu granu jeste da skup *S* sadrži dva ili manje elementa, odnosno da skup najveće kardinalnosti iz skupa skupova *s* ima kardinalnost manju ili jednaku dva. Ovo znači da ne postoji nijedna skup koji sadrži više od dva elementa. Tu osobinu možemo iskoristiti da skup pokrivanja koristeći maksimalno podudaranje kardinalnosti (*maximum matching*). Maksimalno podudaranje je problem u teoriji grafova koji se bavi nalaženjem maksimalnim brojem veza u grafu, takvih da nijedna od izabranih veza ne deli čvor ni sa jednom drugom izabranom vezom. Primer maksimalnog podudaranja možemo videti na slici 3.4.3.

A picture containing watch

Description automatically generated

Slika 3.4.3 primer maksimalnog podudaranja kardinalnosti

Da bi smo ovo iskoristili moramo pretvoriti skupove nazad u graf. Ovo je prilično jednostavno učiniti tako što ćemo sve elemente univerzuma ubaciti u graf kao čvorove dok će skupovi predstavljati veze između čvorova. Rešenje maksimalnog podudaranja možemo iskoristiti na sledeći način. Uzimamo sve skupove koji kada su se pretvorili u veze se nalaze u rešenju maksimalnog podudaranja. Te skupove zatim dodamo u rešenje pokrivajućeg skupa i dodamo takođe ostale skupove koji sadrže po jedan čvor, a da pritom ti čvorovi ne pripadaju nijednoj vezi koja je bila rešenje maksimalnog podudaranja.

* + - 1. Implementacija Fomin et al. algoritama za pokrivanje skupa

Kod analize same implementacije ovog algoritma nećemo vršiti analizu celog koda nego već, kao i kod teorijske analize, samo novih grana u poređenju sa algoritmom iz poglavlja 3.4.1. Prvu stvar koju ćemo analizirati jeste implementaciju grane koja se nalazi na liniji 3 u pseudokodu na slici 3.4.2.

protected List**<**Integer**>** BranchOne**(**List**<**List**<**Integer**>>** setOfSets**,**

List**<**Integer**>** universe**)**

**{**

int indexOfSetWithFrequencyOneElement **=** **-**1**;**

List**<**Integer**>** SetWithFrequencyOneElement **=** **null;**

Set**<**Integer**>** realUniverse **=** **new** HashSet**<>();**

**for(**List**<**Integer**>** tmpSet**:**setOfSets**)**

realUniverse**.**addAll**(**tmpSet**);**

**for** **(**Integer e **:** realUniverse**)**

**{**

int frequency **=** 0**;**

int i **=** 0**;**

**for** **(**List**<**Integer**>** set **:** setOfSets**)**

**{**

**if** **(**set**.**contains**(**e**))**

**{**

frequency**++;**

indexOfSetWithFrequencyOneElement **=** i**;**

SetWithFrequencyOneElement **=** set**;**

**}**

i**++;**

**}**

**if** **(**frequency **==** 1**)**

**break;**

indexOfSetWithFrequencyOneElement **=** **-**1**;**

SetWithFrequencyOneElement **=** **null;**

**}**

**if** **(**indexOfSetWithFrequencyOneElement **!=** **-**1**)**

**{**

final List**<**Integer**>** finalSetWithFOneElement **=**

SetWithFrequencyOneElement**;**

HashMap**<**Integer**,** Integer**>** fixIndexesMap **=** **new** HashMap**<>();**

List**<**Integer**>** newUniverse **=** universe**.**stream**()**

**.**filter**(**integer **->** **!**finalSetWithFOneElement**.**contains**(**integer**))**

**.**collect**(**Collectors**.**toList**());**

List**<**List**<**Integer**>>** newSetOfSets **=** Take**(**setOfSets**,**

SetWithFrequencyOneElement**,**

fixIndexesMap**);**

List**<**Integer**>** setCover **=** MSC**(**newSetOfSets**,** newUniverse**);**

**if** **(**setCover **!=** **null)**

**{**

RestoreSolution**(**setCover**,** fixIndexesMap**);**

setCover**.**add**(**indexOfSetWithFrequencyOneElement**);**

**}**

**return** setCover**;**

**}**

**return** **null;**

**}**

Primer koda 3.4.5.

Implementacija ove grane je prilično jednostavna. Prva stvar koja se obrađuje u kodu jeste traženje da li postoji skup koji sadrži element frekvencije jedan. Vidimo da je ovo realizovano koristeći foreach petlje i promenljive realUniverse koja sadrži elemente svih skupova koji pripadaju skupu skupova, u kodu setOfSets. Nakon toga vidimo u kodu 3.4.5. ako postoji takav element, skup koji sadrži taj element se uzima kao deo rešenja, koristeći metodu Take, koja je definisana u kodu 3.4.1. Ako nema takvog skupa implementacija grane vraća *null*.

protected List**<**Integer**>** BranchTwo**(**List**<**List**<**Integer**>>** setOfSets**,**

List**<**Integer**>** universe**)**

**{**

List**<**Integer**>** Q **=** **null;**

List**<**Integer**>** R **=** **null;**

**for** **(**List**<**Integer**>** setI **:** setOfSets**)**

**{**

**for** **(**List**<**Integer**>** setJ **:** setOfSets**)**

**if** **(**setI **!=** setJ **&&** setI**.**containsAll**(**setJ**))**

**{**

Q **=** setI**;**

R **=** setJ**;**

**break;**

**}**

**if** **(**Q **!=** **null)**

**break;**

**}**

**if** **(**Q **!=** **null)**

**{**

HashMap**<**Integer**,** Integer**>** fixIndexesMap **=** **new** HashMap**<>();**

List**<**List**<**Integer**>>** newSetOfSets **=**

Remove**(**setOfSets**,** R**,** fixIndexesMap**);**

List**<**Integer**>** setCover **=** MSC**(**newSetOfSets**,** universe**);**

**if** **(**setCover **!=** **null)**

RestoreSolution**(**setCover**,** fixIndexesMap**);**

**return** setCover**;**

**}**

**return** **null;**

**}**

Primer koda 3.4.6.

U kodu 3.4.6 vidimo implementaciju druge grane koja se nalazi na liniji 5 u pseudokodu datom na slici 3.4.2. Sama implementacija je relativno prosta. Poređuje svaki skup sa svakim dok ne nađe dva skup koja zadovoljavaju uslov da je jedan skup podskup drugog skupa. Ako je pronađen par skupova koji zadovoljavaju ovaj uslov, koristi se funkcija Remove iz koda 3.4.2 da izbaci skup koji predstavlja podskup drugog skupa. U slučaju da nije pronađen par skupova koji zadovoljavaju ovaj uslov implementacija će kao i u prethodnom slučaju vratiti *null*.

Poslednja nova grana ovog algoritma koja je implementirana može se videti u kodu 3.4.7. Ona predstavlja implementaciju grane koja se nalazi na liniji 7 u pseudokodu datom na slici 3.4.2. Sama implementacija koristi već napravljenu implementaciju maksimalnog podudaranja kardinalnosti čiju implementaciju možemo naći [11]. Sama implementacija je prilično prosta i jedina stvar koja bi možda trebala da se naglasi jeste to što se koristi indexMap da bi se pratila konekcija između skupova i veza koje se dobijaju kao rezultat maksimalnog podudaranja.

protected List**<**Integer**>** BranchThree**(**List**<**List**<**Integer**>>** setOfSets**,**

List**<**Integer**>** universe**)**

**{**

List**<**Integer**>** maxCardinality **=**GetMaxCardinality**(**setOfSets**);**

**if** **(**maxCardinality**.**size**()** **<=** 2**)**

**{**

UndirectedGraph**<**Integer**>** graphForEdmondsMatching **=**

**new** UndirectedGraph**<>();**

**for** **(**Integer tmp **:** universe**)**

graphForEdmondsMatching**.**addNode**(**tmp**);**

**for** **(**List**<**Integer**>** tmp **:** setOfSets**)**

**if** **(**tmp**.**size**()** **==** 2**)**

graphForEdmondsMatching**.**addEdge**(**tmp**.**get**(**0**),** tmp**.**get**(**1**));**

graphForEdmondsMatching **=** EdmondsMatching

**.**maximumMatching**(**graphForEdmondsMatching**);**

List**<**Integer**>** setCover **=** **new** ArrayList**<>();**

Set**<**Set**<**Integer**>>** sets **=** **new** HashSet**<>();**

HashMap**<**Set**<**Integer**>,** Integer**>** indexMap **=** **new** HashMap**<>();**

int i **=** 0**;**

**for** **(**List**<**Integer**>** tmpList **:** setOfSets**)**

**{**

Set**<**Integer**>** tmpSet **=** **new** HashSet**<>(**tmpList**);**

**if** **(!**sets**.**contains**(**tmpSet**))**

**{**

indexMap**.**put**(**tmpSet**,** i**);**

sets**.**add**(**tmpSet**);**

**}**

i**++;**

**}**

**for** **(**Integer vertex **:** graphForEdmondsMatching**)**

**{**

Set**<**Integer**>** edges **=**

**new** HashSet**<>(**graphForEdmondsMatching**.**edgesFrom**(**vertex**));**

edges**.**add**(**vertex**);**

**for** **(**Set**<**Integer**>** check **:** sets**)**

**{**

**if** **(**check**.**equals**(**edges**)** **&&**

**!**setCover**.**contains**(**indexMap**.**get**(**check**)))**

setCover**.**add**(**indexMap**.**get**(**check**));**

**}**

**}**

**return** setCover**;**

**}**

**return** **null;**

**}**

Primer koda 3.4.7.

* + 1. Rooji et al. algoritam za pokrivanje skupa

Ovo je poslednji algoritam koji ćemo analizirati u ovom radu. On je takođe najbolji po pitanju vremenske složenosti od svih egzaktnih algoritma koje smo do sada radili. Predstavlja direktno poboljšanje algoritma iz poglavlja 3.4.2. Algoritam je prvi put osmišljen u radu [6] gde je vršena i analiza njegove vremenske složenosti i zaključilo se da je *O*(*1.4969n*).

Text, letter

Description automatically generated

Slika 3.4.3. Pseudokod Rooji et al. algoritma [6]

Kao što smo u pseudokodu algoritma na slici 3.4.2. videli da sadrži sve elemente pseudokoda algoritma sa slike 3.4.1. u sebi, isto tako možemo primeti da pseudokod dat na slici 3.4.3. sadrži u sebi sve elemente iz pseudokoda na slici 3.4.2. Samim tim isto kao u poglavlju 3.4.2. nećemo vršiti analizu celog algoritma već samo novih grana.

Prvu novu granu koju primećujemo kod samog algoritma jeste grana koja počinje na liniji 7. Možemo videti da je ulov za ovu granu da postoje elementi *e1* i *e2* takvi da zadovoljavaju uslov *s*(*e1*) podskup *s*(*e2*). Ovo znači da skup skupova koji sadrže u sebi element *e2* takođe sadrži sve skupove koji u sebi sadrže element *e1*. Ako postoje dva elementa koji zadovoljavaju ovaj uslov onda možemo izbaciti element *e2* iz univerzuma i svih skupova. Znamo da ovo možemo učiniti zato što cilj algoritma je da pokriva sve skupove, odnosno da unija svih skupova koji su deo rešenja bude unija svih elementa. Izbacivanjem elementa *e2* iz svih skupova možda na prvi pogled deluje kao da se suprotstavlja tom cilju pošto algoritam neće znati da treba da pokrije i element *e2*. Ovo u stvari nije problem zato što ako je zadovoljen uslov da je *s*(*e1*) podskup *s*(*e2*) time znamo da svaki skup koji sadrži element *e1* će u isto vreme sadržati *e2*. Samim tim kada u rešenje ubacimo skup koji pokriva *e1* mi ćemo indirektno pokrivati i *e2*.

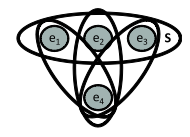
Shape, circle

Description automatically generated

Slika 3.4.4. Primer skupova koji zadovoljavaju uslov *s*(*e1*) podskup *s*(*e2*)

Na slici 3.4.4. možemo i videti kako ovo pravilo radi. Izbacivanjem *e2* nam neće promeniti izbor skupova. I dalje ćemo imati tri skupa od kojih moramo izabrati dva kao rešenje. Samo što ovaj put element *e2* uopšte ne moramo da uzimamo u obzir.

Sledeća grana koju ćemo analizirati počinje na liniji 9 u pseudokodu na slici 3.4.3. U njoj možemo videti uslov za ulazak u granu koji deluje malo kompleksnije u odnosu na prethodne uslove koje smo analizirali, ali on je i dalje sasvim logičan. Sam uslov se može prevesti na sledeći način. Mora postojati skup *R* koji sadrži elemente frekvencije dva, takav da unija *Q*, izuzev samih elementa iz skupa *R*, sadrži manje elementa u odnosu na koliko skup *R* ima elementa sa frekvencijom dva. U ovom slučaju *Q* je unija svih skupova koji u sebi sadrže bilo koji element iz skupa *R* čija je frekvencija dva. Ako je ovaj uslov tačan možemo uzeti skup *R* kao deo rešenja. Ovaj uslov kao i prethodni je najlakše potvrditi preko primer.



Slika3.4.5. Primer skupova koji zadovoljavaju uslov grane [6]

Na slici 3.4.5. možemo videti skupove koji će zadovoljiti prethodni uslov. Ako uzmemo da je skup *R* u ovom slučaju skup *S* možemo videti da nema potrebe razmatrati svaki skup pojedinačno. Ako uzmemo prethodno uslov i napravimo skup *Q* on će biti unija skupova {*e1*, *e4*}, {*e2*, *e4*} i {*e3*, *e4*} to bi obično značilo da je skup *Q* = {*e1*, *e2*, *e3*, *e4*}, ali po pravilu uslova izbacujemo sve elemente koji pripadaju skupu *R* što znači da je *Q* = {*e4*}. Pošto skup *Q* sadrži samo jedan element, a skup *R* sadrži tri elementa, čija je frekvencija dva, to znači da uzimamo skup *R* kao deo rešenja što bi bila i tačna oduka za dobijanje minimalnog pokrivanja skupa.

Poslednja grana koju ćemo analizirati počinje na liniji 11 u pseudokodu datom na slici 3.4.3. Uslov ove grane je prilično jednostavan, zahteva samo da postoji skup *R* koji sadrži tačno dva elementa koji su frekvencije dva. Ako postoji skup *R* koji zadovoljava ovaj uslov radi se sledeće. Nalaze se skupovi *R1* i *R2* takvi da predstavljaju skupove u kojima se elementi skupa *R* nalaze. Znamo da postoje samo dva takva skupa pošto je uslov grane da *R* ima dva elementa i da oba elementa imaju frekvenciju dva, odnosno da se pojavljuju u dva skupa. Sada napravimo novi skup *Q* koji predstavlja uniju skupa *R1* i *R2* i onda iz njega izbacimo elemente skupa *R*. Posle toga napravimo novi *s* i *U* takve da iz *s* izbacimo skupove *R*,*R1* i *R2*, a ubacimo *Q*, dok novo *U* pravimo tako što izbacimo elemente *R* iz njega. Kada pokrenemo algoritam nad novim *s* i *U* ako rešenje koje dobijemo nazad sadrži skup *Q* to znači da trebamo vrati ti isto to rešenje, ali da zamenimo skup *Q* skupovima *R1* i *R2*. Ako rešenje koje smo dobili nazad ne sadrži skup *Q* to znači da možemo samo dodati skup *R* rešenju da bi smo pokrili njegove elemente.

A picture containing text, transport, handcart, clipart

Description automatically generated

Slika 3.4.6. Primer koji sadrži skupove sa tačno dva elementa frekvencije dva

Na slici 3.4.6. možemo videti tačno kako ova grana funkcioniše. Ako uzmemo da je *R =* {*e1*, *e4*} tada možemo naći skupove *R1* i *R2* iste kao na slici 3.4.6. Kada sastavimo *Q* i prosledimo ga sledećoj iteraciji algoritma dobićemo nazad rešenje koje sadrži skup *Q* u sebi. Sada sa znamo da *Q* u rešenju trebamo zameniti sa *R1* i *R2* i njega vratiti kao pravo rešenje. Ovo kao što vidimo sa slike 3.4.6. trebamo uraditi da bi smo dobili minimalni pokrivajući skup.

* + - 1. Implementacija Rooji et al. algoritma za pokrivanje skupa

Kao i kod implementacije Fomin et al. algoritma u ovoj sekciji nećemo vršiti analizu celog koda već samo najbitnijih delova.

HashMap**<**Integer**,** Set**<**Set**<**Integer**>>>** subSetOfElements **=** **new** HashMap**<>();**

**for** **(**Integer e **:** universe**)**

**{**

Set**<**Set**<**Integer**>>** setOfE **=** **new** HashSet**<>();**

**for** **(**List**<**Integer**>** tmpSet **:** setOfSets**)**

**if** **(**tmpSet**.**contains**(**e**))**

**{**

Set**<**Integer**>** containsE **=** **new** HashSet**<>(**tmpSet**);**

setOfE**.**add**(**containsE**);**

**}**

subSetOfElements**.**put**(**e**,** setOfE**);**

**}**

Integer elementToRemove **=** **null;**

**for** **(**Integer e1 **:** universe**)**

**{**

Set**<**Set**<**Integer**>>** subSetOfE1 **=** subSetOfElements**.**get**(**e1**);**

**for** **(**Integer e2 **:** universe**)**

**{**

Set**<**Set**<**Integer**>>** subSetOfE2 **=** subSetOfElements**.**get**(**e2**);**

**if** **(**e1 **!=** e2 **&&** subSetOfE2**.**containsAll**(**subSetOfE1**))**

**{**

elementToRemove **=** e2**;**

**break;**

**}**

**}**

**if** **(**elementToRemove **!=** **null)**

**break;**

**}**

Primer koda 3.4.7.

U kodu 3.4.7. možemo videti deo implementacije prve grane koje smo analizirali u poglavlju 3.4.3. Ovaj deo koda se konkretno odnosi na proveru da li postoje elementi *e1* i *e2* takvi da zadovoljavaju uslov da je *s*(*e1*) podskup *s*(*e2*). Možemo videti da je sama implementacija vrlo jednostavna ali vremenski intenzivna. Prva stvar koja se radi u samoj implementaciji jeste pravljenje svih skupova *s*(*e*) gde *e* predstavlja bilo koji element iz *U*. Nakon čega se radi poređenje samih skupova *s*(*e*). Koje predstavlja poređenje svaki sa svakim.

HashMap**<**Integer**,** Integer**>** frequencyMap **=** FrequencyMapCalc**(**setOfSets**,** universe**);**

List**<**Integer**>** setToTake **=** **null;**

int indexSetToTake **=** 0**;**

int i **=** 0**;**

**for** **(**List**<**Integer**>** r **:** setOfSets**)**

**{**

Set**<**Integer**>** setR **=** **new** HashSet**<>();**

Set**<**Integer**>** unionOfQ **=** **new** HashSet**<>();**

**for** **(**Integer e **:** r**)**

**if** **(**frequencyMap**.**get**(**e**).**equals**(**2**))**

**{**

**for** **(**List**<**Integer**>** tmpSet **:** setOfSets**)**

**if** **(**tmpSet**.**contains**(**e**))**

unionOfQ**.**addAll**(**tmpSet**);**

setR**.**add**(**e**);**

**}**

unionOfQ **=** unionOfQ**.**stream**().**filter**(**integer **->** **!**r**.**contains**(**integer**))**

**.**collect**(**Collectors**.**toSet**());**

**if** **(**unionOfQ**.**size**()** **<** setR**.**size**())**

**{**

setToTake **=** r**;**

indexSetToTake **=** i**;**

**break;**

**}**

i**++;**

**}**

Primer koda 3.4.8.

Sledeći deo koda koji ćemo analizirati, dat u kodu 3.4.8. predstavlja implementaciju nalaženja skupa *R* takvog da zadovoljava uslov dat u drugoj grani koju smo analizirali u poglavlju 3.4.2. Slično kao i kod prethodne grane sama implementacija je prilično vremenski intenzivna zato što moramo iterirati kroz skupove dok ne nađemo jedan koji nam zadovoljava uslov. Takođe konstrukcija samog skupa *Q*, u kodu označenog unionOfQ se radi u još jednoj petlji koja ponovo iterira kroz sve skupove da bi odredila koji skup da doda svojoj uniji. Takođe možemo videti da se u implementaciji koristi promenljiva frequencyMap. Ona u sebi čuva frekvenciju svih elementa koji su deo univerzuma. Ovo se uradilo da ne bi moralo u svakoj iteraciji petlje da se računa frekvencija elementa.

List**<**Integer**>** Q **=** **new** ArrayList**<>();**

List**<**List**<**Integer**>>** R1R2 **=** **new** ArrayList**<>();**

List**<**Integer**>** indexesR1R2 **=** **new** ArrayList**<>();**

i **=** 0**;**

**for** **(**List**<**Integer**>** tmpSet **:** setOfSets**)**

**{**

**if** **((**tmpSet**.**contains**(**r**.**get**(**0**))** **||**

tmpSet**.**contains**(**r**.**get**(**1**)))** **&&** tmpSet **!=** r**)**

**{**

Q**.**addAll**(**tmpSet**);**

R1R2**.**add**(**tmpSet**);**

indexesR1R2**.**add**(**i**);**

**}**

i**++;**

**}**

Q**.**remove**(**r**.**get**(**0**));**

Q**.**remove**(**r**.**get**(**1**));**

final List**<**Integer**>** finalR **=** r**;**

HashMap**<**Integer**,** Integer**>** fixIndexesMap **=** **new** HashMap**<>();**

List**<**List**<**Integer**>>** newSetOfSets **=** **new** ArrayList**<>();**

List**<**Integer**>** newUniverse **=** universe**.**stream**()**

**.**filter**(**integer **->** **!**finalR**.**contains**(**integer**))**

**.**collect**(**Collectors**.**toList**());**

i **=** 0**;**

int j **=** 0**;**

**for** **(**List**<**Integer**>** tmpSet **:** setOfSets**)**

**{**

**if** **(**tmpSet **!=** r **&&** tmpSet **!=** R1R2**.**get**(**0**)** **&&** tmpSet **!=** R1R2**.**get**(**1**))**

**{**

newSetOfSets**.**add**(**tmpSet**);**

fixIndexesMap**.**put**(**j**,** i**);**

j**++;**

**}**

i**++;**

**}**

int QIndex **=** newSetOfSets**.**size**();**

newSetOfSets**.**add**(**Q**);**

Primer koda 3.4.9.

Poslednji deo koda koji ćemo analizirati se odnosi na nalaženja skupova *R1* i *R2* kao i konstrukcije skupa *Q* za uslov na liniji 11 dat u pseudokodu na slici 3.4.3. Samo implementaciju možemo videti u kodu 3.4.9. Primećujemo da je nalaženje skupova *R1* i *R2* kao i konstrukcija samog skupa *Q* trivijalna operacija. Takođe vidimo u samom kodu da se sada fixIndexesMap ne namešta koristeći funkcija Remove niti Take. Ne možemo koristiti te funkcije u ovom slučaju zato što iz skupa skupova trebamo izbaciti dva skupa *R1* i *R2*,dok su te funkcije zamišljene da omoguće izbacivanje samo jednog skupa. U kodu isto vidimo da postoji promenljiva indexesR1R2 ova promenljiva nam samo pamti indekse skupova *R1* i *R2* u slučaju da trebamo da ih ubacimo u rezultat.

1. Testiranje implementacija

U ovom poglavlju bavićemo se testiranjem implementacija algoritama iz poglavlja 3. Samo testiranje je zamišljeno da funkcioniše na vrlo prost način. Kao ulazni parametar samog programa prosleđuje se fajl koji u sebi sadrži graf, sve veze u grafu. Program je fiksno namešten da pokrene jedanaest iteracija svakog algoritma i da zabeleži koliko je trajalo izvršavanje svake iteracije.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Imena | Broj čvorova | Broj grana | Prosečan stepen čvora | Maksimalni stepen čvora |
| books | 105 | 441 | 8.4 | 25 |
| dolphins | 62 | 159 | 5.13 | 12 |
| jazz | 198 | 2742 | 27.7 | 100 |
| karate | 34 | 78 | 4.59 | 17 |
| facebook\_combined.txt | 4039 | 88234 | 43.7 | 1045 |
| graph 6 nodes.txt | 6 | 7 | 2.33 | 3 |
| graph 7 nodes.txt | 7 | 10 | 2.86 | 4 |
| graph 8 nodes.txt | 8 | 14 | 3.5 | 4 |
| graph 11 nodes.txt | 11 | 18 | 3.27 | 5 |

Tabela 4.0.1 Karakteristike grafova

Treba se naglasiti da prva iteracija samog testa se ne gleda u rezultatima. Ovo je zbog toga kako sama Java funkcioniše i njenog JIT (*Just in time*) kompajlera, koji može usporiti rad algoritma prvi put kada se pokreće. Važno je takođe reći da su algoritmi provera svih kombinacija i trivijalni algoritam za pokrivanje skupa ograničeni da se pokreću samo kada graf ima manje od četrdeset čvorova.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Imena | Pohlepan algoritam | Provera svih kombinacija | Trivijalno pokrivanje skupa | Fomin et al. algoritam | Rooji et al. algoritam |
| books | 1.027 | / | / | 120.273 | 68.290 |
| dolphins | 0.782 | / | / | 4.558 | 5.643 |
| jazz | 3.242 | / | / | 168.213 | 217.700 |
| karate | 0.280 | 3874650.288 | 13682.927 | 1.453 | 0.949 |
| facebook\_combined.txt | 25.357 | / | / | / | / |
| graph 6 nodes.txt | 0.074 | 0.119 | 0.319 | 0.126 | 0.101 |
| graph 7 nodes.txt | 0.111 | 0.246 | 0.426 | 0.164 | 0.112 |
| graph 8 nodes.txt | 0.099 | 0.303 | 0.329 | 0.531 | 1.035 |
| graph 11 nodes.txt | 0.120 | 0.601 | 0.586 | 0.291 | 0.268 |

Tabela 4.0.2. minimalno vreme izvršavanja algoritma [u ms] nad datim grafovima

Prazna polja u tabeli 4.0.2. označavaju da algoritam ne može u razumnom vremenu da se završi za dati graf. Možemo videti u tabeli 4.0.2. da se odnos brzina izvršavanja algoritma i odnos njihovih vremenskih kompleksnosti slažu, odnosno algoritmi koji su imali manju vremensku kompleksnost se uglavnom brže izvršavaju. Ali takođe je važno naglasiti da i dalje ima odstupanja naročito kod Fomin et al. i Rooji et al. algoritma. Čak iako je vremenska složenost Rooji et al. algoritma teoretski manja možemo iznad videti da se u nekim grafovima sporije izvršava u odnosu na Fomin et al.

Implementacija toga kako se vršilo testiranje se može videti u kodu 4.0.1.

final int NUMBER\_OF\_ITERATIONS **=** 11**;**

Solver**[]** solvers **=** **{new** GreedySolver**(),new** FominSolver**(),**

**new** RooijSolver**(),** **new** TrivialSolver**(),**

**new** AllCombinationsSolver**()};**

long**[][]** timeValues **=** **new** long**[**solvers**.**length**][**NUMBER\_OF\_ITERATIONS**];**

**for** **(**int i **=** 0**;** i **<** solvers**.**length**;** i**++)**

**{**

print**(**"Solving: " **+** solvers**[**i**].**toString**());**

**for** **(**int j **=** 0**;** j **<** NUMBER\_OF\_ITERATIONS**;** j**++)**

**{**

**if** **(**Graph**.**GetInstance**().**GetAllNodes**().**size**()** **<=** 40 **||**

i **<** **(**solvers**.**length **-** 2**))**

**{**

print**(**"Iteration: " **+** j**);**

print**(**"Time: " **+** System**.**currentTimeMillis**());**

**if** **(**j **!=** **(**NUMBER\_OF\_ITERATIONS **-** 1**))**

solvers**[**i**].**Solve**();**

**else**

PrintSolverResult**(**solvers**[**i**]);**

timeValues**[**i**][**j**]** **=** solvers**[**i**].**GetTimeSpent**();**

solvers**[**i**].**Reset**();**

**}**

**else**

timeValues**[**i**][**j**]** **=** Long**.**MAX\_VALUE**;**

**}**

**}**

long**[][]** timeValuesAccountForJit **=** **new** long**[**solvers**.**length**][**NUMBER\_OF\_ITERATIONS **-** 1**];**

**for** **(**int i **=** 0**;** i **<** solvers**.**length**;** i**++)**

System**.**arraycopy**(**timeValues**[**i**],** 1**,**

timeValuesAccountForJit**[**i**],** 0**,** NUMBER\_OF\_ITERATIONS **-** 1**);**

Primer koda 4.0.1.

1. Zaključak

Primarna stvar koju je ovaj rad hteo da postigne jeste analiza i poređenje algoritma za rešavanje problema minimalnog dominirajućeg skupa. Pre početka analize samih algoritama teorijski smo uzeli da objasnimo sve glavne koncepte potrebne za rešavanje problema. Ovo je urađeno u drugom poglavlju.

Nakon toga vršili smo analizu samih algoritama i njihovih implementacija u poglavlju tri. Ovo poglavlje je većim delom uzimalo definicije iz poglavlja dva i koristilo za neki praktičan rad. Takođe smo videli kako se algoritmi za pokrivanje skupa nadograđuju jedan drugog i kako su povezani u istom konceptu za rešavanje problema.

Na kraju u poglavlju četiri vršili smo testiranje samih algoritma koristeći različite grafove u cilju da vidimo kako se njihove teorijske osobine preslikavaju u realnosti. Iz ovih testova možemo doneti sledeće zaključke.

* Ako je aproksimacija rešenja prihvatljiva kao rezultat najbolje ili ako su grafovi sa kojima se radi veći od 100 čvorova, a broj veza mali treba se koristiti pohlepan algoritam. Ima najmanju vremensku složenost i jedini je uspevao da reši grafove većih dimenzija.
* Ako nije prihvatljiva aproksimacija rešenja ili su veličine grafova razumne, onda može se koristiti Fomin et al. algoritam ili Rooji et al. algoritam. Sama razlika u testovima je bila vrlo mala ali u slučaju da je potrebno egzaktno rešenje za veći graf ipak je bolje koristiti Rooji et al. algoritam.
* Provera svih kombinacija i trivijalno rešenje pokrivanje skupa skoro nikad nema potrebe da se koriste. Jedina situacija u kojoj bi mogli da se efikasno iskoriste jeste kod vrlo malih grafova sa manje od 30 čvorova, a čak i tad nema razloga da se ne koristi neki od iznad spomenutih algoritama.

Literatura

1. M.R. Garey, D.S. Johnson, *Computers and Intractability, A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, 1979.
2. Ding-Zhu Du, Peng-Jun Wan, *Connected Dominating Set: Theory and Applications*, Springer, 2013.
3. E.J. Cockayne, R.M. Dawes, S.T. Hedetniemi, ‘Total dominations in graphs’, *Networks*, 10: 211–219, 1980.
4. K. Ralf, L. Christian, ‘Hardness results and approximation algorithms of k-tuple domination in graphs’, *Information Processing Letters*, 89 (2): 75–83, 2004.
5. S. Finbow, S. Gaspers, ME. Messinger, P. Ottaway, ‘A note on the eternal dominating set problem’, *International Journal of Game Theory*, 47: 543–555, 2018.
6. J.M.M. van Rooij, H.L. Bodlaender, ‘Exact algorithms for dominating set’, *Discrete Applied Mathematics*, 159(17): 2147-2164, 2011.
7. F. Grandoni, ‘A note on the complexity of minimum dominating set’, *Journal of Discrete Algorithms,* 4(2): 209–214, 2006.
8. Y. Talebi, H. Rashmanlou, ‘Application Of Dominating Sets In Vague Graphs’, *Applied Mathematics, E-Notes*, 17: 251–267, 2017.
9. M. Davis, H. Putnam, ‘A computing procedure for quantification theory’, *Journal of the ACM*, 7:201–215, 1960.
10. F.V. Fomin, F. Grandoni, D. Kratsch, ‘A measure & conquer approach for the analysis of exact algorithms’, *Journal of the ACM*, 56(5), 2009.
11. Keith Schwarz, Edmonds Matching implementacija, <https://www.keithschwarz.com/interesting/code/edmonds-matching/> , 21/09/2022.