Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Gregor Ažbe

Reševanje problemov na grafih z izločevalnimi algoritmi

diplomska naloga

visokošolski strokovni študijski program
prve stopnje
računalništvo in informatika

Mentor: doc. dr. Jurij Mihelič

Ljubljana, 2019

Copyright. Rezultati diplomske naloge so intelektualna lastnina avtorja in Fakultete za računalništvo in informatiko Univerze v Ljubljani. Za objavo in koriščenje rezultatov diplomske naloge je potrebno pisno privoljenje avtorja, Fakultete za računalništvo in informatiko ter mentorja.

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo: Reševanje problemov na grafih z izločevalnimi algoritmi

Tematika naloge:

Ena izmed najbolj znanih, enostavnih in tudi učinkovitih metod snovanja algoritmov je požrešna metoda. Temelji na postopni gradnji končne rešitve, kjer na vsakem koraku izberemo komponento, ki se zdi najbolj obetavna. V okviru diplomske naloge preučite požrešni metodi soroden pristop, kjer začetna rešitev sestoji iz vseh možnih kandidatov, tekom samega postopka pa najmanj obetavne kandidate skušamo izločiti. Izberite nekaj problemov za katere menite, da bi se izločevalna metoda lahko uporabila. Implementirajte tako požrešni kot izločevalni algoritem, nato pa ju oba eksperimentalno primerjajte na izbranih problemih.

Zahvaljujem se mentorju doc. dr. Juriju Miheliču za strokovno svetovanje in potrpežljivost pri nastajanju diplomske naloge. Rad bi se zahvalil tudi lektorici Snežni Poljanšek za lektoriranje. Nenazadnje pa bi se rad zahvalil še svoji družini in zaročenki Niki za podporo pri študiju in pisanju diplome.

Kazalo

Seznam uporabljenih kratic

kratica	**angleško**	**slovensko**
NP	non-deterministic polynomial-time	nedeterministični polinomski čas
P	polynomial-time	polinomski čas

Povzetek

Naslov: Reševanje problemov na grafih z izločevalnimi algoritmi

Avtor: Gregor Ažbe

Diplomska naloga obravnava reševanje problemov na grafih s pomočjo izločevalnih algoritmov. Vsebuje teoretično izhodišče in eksperimentalno delo. V njem primerjamo reševanje izbranih problemov s požrešnimi in izločevalnimi algoritmi. Ti dve vrsti algoritmov smo preizkusili na problemih dominantne množice, barvanja grafov in vozliščnega pokritja. V nalogi predstavimo probleme, implementirane algoritme in rezultate primerjav med njimi.

Ključne besede: požrešni algoritem, izločevalni algoritem, graf.

Abstract

Title: Solving problems in graphs using elimination algorithms

Author: Gregor Ažbe

This diploma thesis deals with solving problems in graphs using elimination algorithms. It contains theoretical basis and experimental work. We compare solving of selected problems using greedy and elimination algorithms. They were tested on dominating set, graph coloring and vertex cover problems. In this thesis we present the problems, the implementations and the results of the comparison between them.

Keywords: greedy algorithm, elimination algorithm, graph.

Poglavje 1

Uvod

Če velja da $P \neq NP$, potem pri NP-težkih problemih optimalne rešitve ne moremo najti v polinomskem času. Take probleme zato pogosto rešujemo s približnimi algoritmi. Eden od načinov približnega reševanja je uporaba požrešnih algoritmov. Njihova prednost je v tem, da hitro pridejo do dopustne rešitve. S pomočjo naše diplomske naloge želimo požrešne algoritme izboljšati. Naš namen je bil z izboljšavo doseči boljše rezultate, kot s klasičnimi požrešnimi algoritmi. To smo želeli doseči tako, da smo problem reševali v obratnem vrstnem redu.

Pri požrešnem algoritmu reševanje problema začnemo s prazno množico rešitev. Nato vsakem koraku izberemo trenutno najboljšo možnost in jo dodamo v množico rešitev. V tej diplomski nalogi pa se ukvarjamo s tako imenovanim izločevalnim pristopom, kjer je glavna ideja, da v začetku množica rešitev vsebuje vse elemente, iz katere nato odstranjujemo najslabše elemente. Odstranimo jih lahko le, če je po odstranitvi rešitev še pravilna. V tej diplomski nalogi smo preučili požrešne in izločevalne algoritme. Nato smo jih preizkusili na različnih problemih na grafih.

Na začetku izdelave diplomske naloge smo izbrali nekaj problemov. Izbrali smo probleme dominantne množice, barvanja grafov in vozliščnega pokritja. Vsi problemi so NP-težki. Nato smo za njih implementirali požrešni in izločevalni algoritem ter primerjali rezultate.

V nadaljevanju si bomo natančneje ogledali oba algoritma, ki ju bomo primerjali. Opisali bomo njune lastnosti in kako v splošnem delujeta.

1.1 Požrešni algoritmi

Požrešni algoritmi so algoritmi, pri katerih pri vsakem koraku s pomočjo hevristike izberemo lokalno najboljšo možnost. Pri tem upamo, da bo najboljša lokalna možnost vodila v globalni optimum. Na ta način v vsakem koraku izločimo velik del možnih rešitev, ki jih ne obravnavamo.

Za določene probleme požrešni algoritem vedno vrne najboljšo možno rešitev. To je v primeru, če za določen problem izbira lokalnih maksimumov v zaporedju odločitev, vodi v globalni maksimum. Če lahko dokažemo, da to velja, lahko trdimo, da je za ta problem požrešni algoritem natančen (npr. Kruskalov algoritem za problem najmanjšega vpetega drevesa [5]).

Za večino problemov pa ni nujno, da z uporabo požrešnega algoritma pridemo do optimalnega rezultata. Kljub temu pa se požrešni algoritmi uporabljajo tudi za nekatere take probleme, saj vrnejo dovolj dober rezultat. Ker v zaporedju odločitev vedno izberejo najboljšo lokalno možnost in se nanjo več ne vračajo, so preprosti in učinkoviti. Algoritem pa lahko izboljšamo z uporabo različnih dodatnih hevristik, ki vodijo do uspešnejšega rezultata. Ena od njegovih dobrih lastnosti je tudi to, da ga lahko uporabimo za reševanje veliko različnih problemov.

V splošnem algoritem poteka v naslednjih korakih. Najprej inicializiramo podatkovne strukture, ki jih potrebujemo pri algoritmu. Nato izberemo najboljšo lokalno možnost. Preverimo, če bo v primeru odločitve za to možnost rešitev še pravilna. Če je rešitev še pravilna, možnost dodamo v rešitev. Nato ta postopek ponavljamo, dokler nismo zadovoljni z rešitvijo.

Algoritem 1: Požrešni algoritem

Vhod: I

Izhod: S

$inicializacija()$

while *nismo zadovoljni z rešitvijo* **do**

  $x \leftarrow NajboljsaLokalnaMoznost(I)$

  if $JeVeljavno(S, x)$ **then**

    $S \leftarrow S \cup x$

1.2 Izločevalni algoritmi

V tej točki bomo predstavili izločevalne algoritme. Njihov namen je izboljšati reševanje problemov, ki jih rešujemo s požrešnimi algoritmi.

Pri izločevalnih algoritmih reševanje problema začnemo z množico vseh možnosti. Nato v vsakem koraku izločimo trenutno najslabšo možnost. Možnost lahko izločimo le, če je po izločitvi rešitev še veljavna. Če rešitev v primeru izločanja ni pravilna, možnost dodamo v množico rešitev.

Algoritem 2: Izločevalni algoritem

Vhod: input data I

Izhod: solution S

init()

$S \leftarrow I$

while *notSatisfied(S)* **do**

  $x \leftarrow WorstLocalOption(I)$

  if $canEliminate(I, x)$ **then**

    $I \leftarrow I - x$

  else

    $S \leftarrow S \cup x$

1.3 Osnovni pojmi

Graf $G(V, E)$ je podatkovna struktura, ki je definirana z množico vozlišč V in povezav $E \subseteq V \times V$.

Soseščina $\mathcal{N}(v)$ vozlišča $v \in V$ v grafu G(V, E) je množica, ki vsebuje vsa sosednja vozlišča vozlišča v.

Definicija 1.3.1 *Soseščina vozlišča*

Naj bo $G(V, E)$ neusmerjen graf. Soseščina $\mathcal{N}(v) = \{u \in V \mid (u, v) \in E\}$ predstavlja množico vseh sosednjih vozlišč vozlišča v.

Stopnja vozlišča $\deg(v)$ je število vozlišč, ki so sosednja vozlišču v.

Definicija 1.3.2 *Stopnja vozlišča*

Stopnja $deg(v)$ vozlišča v je moč soseščine $\mathcal{N}(v)$ vozlišča v, t.j. $deg(v) = |\mathcal{N}(v)|$.

Največja stopnja grafa je enaka stopnji vozlišča z največjo stopnjo.

Definicija 1.3.3 *Največja stopnja grafa*

Največja stopnja $\Delta(G)$ grafa $G(V, E)$ je enaka $\max(deg(v) : v \in V)$.

1.4 Pregled vsebine

V nadaljevanju bomo opisali izbrane probleme in implementirane algoritme za te probleme.

Najprej si bomo ogledali problem dominantne množice. Opisali bomo, kako delujeta požrešni in izločevalni algoritem za ta problem. Primerjali bomo rezultate merjenja uspešnosti in učinkovitosti.

Nato si bomo ogledali problem barvanja grafov. Problem bomo opisali in si ogledali požrešni ter izločevalni algoritem. Pri izločevalnem algoritmu si bomo ogledali še podatkovno strukturo disjunktnih množic, ki smo jo uporabili za izboljšanje učinkovitosti. Nato bomo še primerjali uspešnost in učinkovitost obeh algoritmov.

Kot zadnji problem si bomo ogledali še problem vozliščnega pokritja. Pri tem problemu bomo prav tako pogledali, kako delujeta požrešni in izločevalni algoritem. Nato si bomo na primeru ogledali, zakaj z izločevalnim algoritmom pričakujemo boljše rezultate. Na koncu si bomo ogledali še primerjavo obeh algoritmov.

Na koncu diplomske naloge bomo v zaključku povzeli opravljeno delo in rezultate opravljenih meritev. Nato bomo predstavili še možnosti nadaljevanja raziskovalnega dela na tem področju.

Poglavje 2

Najmanjša dominantna množica

V tem poglavju predstavimo problem najmanjše dominantne množice.

Problem najmanjše dominantne množice je optimizacijski problem. Pri tem problemu moramo za podani graf najti najmanjšo dominantno množico. Vozlišče v dominantni množici pokrije samo sebe in svoja sosednja vozlišča. Tako morajo vozlišča, ki so v tej množici, pokriti vsa vozlišča v grafu.

Problem najmanjše dominantne množice ima veliko uporabnih aplikacij. Ena od aplikacij tega problema je uporabna pri analizi socialnih omrežij. Če želimo ljudi hitro obvestiti o kakšnem dogodku, npr. o naravni nesreči, lahko to storimo hitreje, če poznamo dominantno množico grafa socialnega omrežja za obveščanje. V tem grafu vsako vozlišče predstavlja eno osebo, povezava med dvema vozliščema pa pomeni, da lahko komunicirata direktno. Novico tako lahko sporočimo le osebam v dominantni množici, te pa sporočilo posredujejo vsem, s katerimi lahko komunicirajo direktno. Na tak način novica doseže vse ljudi v omrežju [2].

Dominantno število grafa $\gamma(G)$ je velikost najmanjše dominantne množice za graf G.

Problem dominantne množice je NP-težek. Ker ne vemo, ali velja $P = NP$, zanj še ne poznamo algoritma, ki bi ga rešil v polinomskem času. V

nadaljevanju bomo obravnavali dva približna algoritma, ki rešujeta dani problem.

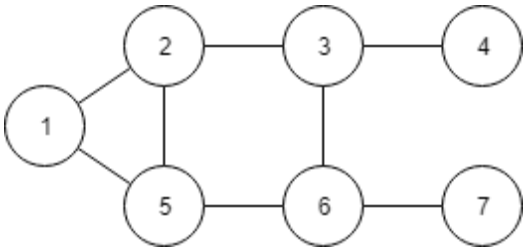
2.1 Definicija problema

Dominantna množica je taka podmnožica vozlišč, da ima vsako vozlišče v grafu vsaj eno sosednje vozlišče, ki pripada tej množici, ali pa je vozlišče samo del te množice [10].

Definicija 2.1.1 *Dominantna množica*

Naj bo $G(V, E)$ graf. Množica $D \subseteq V(G)$ je dominantna množica, če za vsako vozlišče $v \in V(G) - D$ velja $\exists u \in \mathcal{N}(v) : u \in D$.

Dominantnih množic je v poljubnem grafu lahko veliko. Na spodnji sliki 2.1 je primer grafa, pri katerem je približno 60 različnih dominantnih množic. Ena od dominantnih množic je npr. {1, 2, 3, 7} ali pa {2, 4, 6, 7}. Množica z vsemi vozlišči v grafu je vedno dominantna množica. Tudi najmanjših dominantnih množic je lahko več. V grafu na sliki 2.1 jih je 12. Dve od teh sta množici {2, 3, 6} in {1,4,7}.



Slika 2.1: Dominantna množica

2.2 Požrešni algoritem

Požrešni algoritem obdeluje vozlišča od vozlišča z največjim številom sosedov do tistega z najmanjšim. V vsakem koraku se trenutno vozlišče doda v

dominantno množico. Nato se izbriše iz grafa in poleg tega še vsi njegovi sosedi. Ta vozliča izbrišemo zato, ker so pokrita in jih v nadaljevanju ni potrebno več obravnavati.

Vozlišča obdelujemo po stopnji nenaraščajoče, ker tako pokrijemo največ vozlišč v trenutnem koraku algoritma. S tovrstno hevristiko dosežemo, da je potrebno čim manj vozlišč za to, da pokrijemo celoten graf.

Algoritem 3: Požrešni algoritem iskanja dominantne množice

Vhod: $G(V, E)$
Izhod: D
$D \leftarrow \emptyset$
while $V \neq \emptyset$ **do**
  $v \leftarrow$ vozlišče z največjim številom sosedov
  $D \leftarrow D \cup \{v\}$
  Izbriši v in $\mathcal{N}(v)$ iz G

Glavna zanka naredi $O(n)$ korakov, brisanje vozlišč v vsakem koraku glavne zanke pa prav tako $O(n)$. Torej je časovna zahtevnost algoritma $O(n^2)$.

2.3 Izločevalni algoritem

Izločevalni algoritem za dominantno množico je opisan v [7, 8]. V tem podpoglavju bomo preučili ta algoritem in ga nato primerjali s klasičnim požrešnim algoritmom.

Izločevalni algoritem za problem dominantne množice uporablja dve pomožni podatkovni strukturi. Prva je polje *covCnt*, druga pa polje *score*.

V polju *covCnt* imamo zapisano, koliko vozlišč trenutno pokriva posamezno vozlišče. V začetku za vsako vozlišče v polje *covCnt* zapišemo število sosednjih vozlišč. Temu številu prištejemo ena, saj vsako vozlišče pokriva tudi samo sebe.

V polju *score* pa imamo zapisano oceno posameznega vozlišča. Z njo

algoritem izbire najbolj primerno vozlišče za procesiranje. Na začetku je to polje enako *covCnt*.

Algoritem se ponovi n-krat, če je n število vseh vozlišč v grafu. V vsakem koraku algoritem izbere vozlišče v z najmanjšo oceno v polju *score*. Nato preveri, če ima katero od sosednjih vozlišč v polju *covCnt* vrednost 1. Če tako vozlišče obstaja, pomeni, da je vozlišče v edino, ki še lahko pokrije tako vozlišče. Torej se mora trenutno vozlišče dodati v dominantno množico. Nato se *covCnt* za vsa sosednja vozlišča nastavi na 0. S tem označimo, da je vozlišče že pokrito.

Če tako vozlišče ne obstaja, vozlišča v ne potrebujemo v dominantni množici. Ker je zaradi vrstnega reda procesiranja vozlišče v najmanj primerno za rešitev, ga iz rešitve odstranimo. To storimo tako, da za trenutno in vsa sosednja vozlišča vrednost v polju *covCnt* zmanjšamo za 1. Tako dosežemo, da vozlišče v ne pokriva več sebe in sosednjih vozlišč.

Ocena vozlišča v in njegovih sosednjih vozlišč pa se poviša za ena in tako se ta vozlišča pomaknejo proti koncu vrste za procesiranje.

Na koncu glavne zanke oceno vozlišča v nastavimo na neskončno in tako dosežemo, da se ta vozlišča ne procesirajo več.

Osnovna zanka ima časovno zahtevnost $O(n)$, ker se sprehodi čez vsa vozlišča v grafu. Pri iskanju vozlišča z najnižjo vrednostjo Score moramo preveriti vsa vozlišča v grafu. Za popravljanje polj $Score$ in $CovCnt$ v vsakem koraku pa $O(d)$, če je d število sosednjih vozlišč. Ker vedno velja da $d < n$, je časovna zahtevnost znotraj osnovne zanke $O(n)$. Torej je časovna zahtevnost tega algoritma $O(n^2)$.

Algoritem 4: Izločevalni algoritem iskanja dominantne množice

Vhod: $G(V, E)$
Izhod: D
for $v \in V(G)$ **do**
    $CovCnt[v] \leftarrow deg(v) + 1$
$Score \leftarrow CovCnt$
$D \leftarrow \emptyset$

repeat $V(G)$ **times**
    $v \leftarrow$ vozlišče z najnižjo vrednostjo v polju Score
    if $\exists u \in \mathcal{N}(v) : CovCnt[u] = 1$ **then**
        $D \leftarrow u$
        for $u \in N(u)$ **do**
            $CovCnt[u] \leftarrow 0$
    else
        for $u \in N(v)$ **do**
            if $CovCnt[v] > 0$ **then**
                $CovCnt[u] - -$
                $Score[u] + +$
    $Score[v] \leftarrow \infty$

2.4 Izvajanje meritev

V tem poglavju bomo opisali, kako smo izvajali primerjave algoritmov. Opisali bomo testno okolje, ki smo ga pri tem uporabljali in kakšne testne primere smo uporabili.

2.4.1 Testno okolje

Vse algoritme smo implementirali v programskem jeziku Java. Testirali smo na osebnem računalniku HP ProBook 470 z operacijskim sistemom Windows 10. Procesor tega računalnika je Intel® Core™ i5-4210U CPU s frekvenco 1,70 - 2,40 GHz. Velikost delovnega pomnilnika pa je 8 GB. Velikost predpomnilnika L2 je 265 kB, L3 pa 3 MB.

2.4.2 Generiranje vhodnih podatkov

Za namene primerjanja algoritmov smo potrebovali testne grafe. Naredili smo generator naključnih grafov. Generator sprejme število vozlišč n in povezav m. Nato generator zgenerira n vozlišč in m povezav med naključno izbranimi vozlišči. Povezave naključno razporedi tako, da izbere dve naključni med seboj različni vozlišči u in v. Nato preveri, če povezava med tema vozliščema še ne obstaja, in jo v tem primeru doda. Ta postopek ponavlja, dokler ne vstavi m povezav.

Pri vsakem problemu je generator zgeneriral več različnih grafov. Nato smo za vsak graf G pognali požrešni in izločevalni algoritem za izbrani problem. Pri tem smo graf G uporabili kot vhod algoritma. Pri problemu dominantne množice je zgeneriral grafe s 1000 vozlišči. Nato je pri vsakem naslednjem poskusu generiral graf s 1000 več vozlišči, dokler ni prišel do 10.000 vozlišč. Vsi grafi so gostote 0,05. To pomeni, da je v grafu 5% od vseh možnih povezav v grafu.

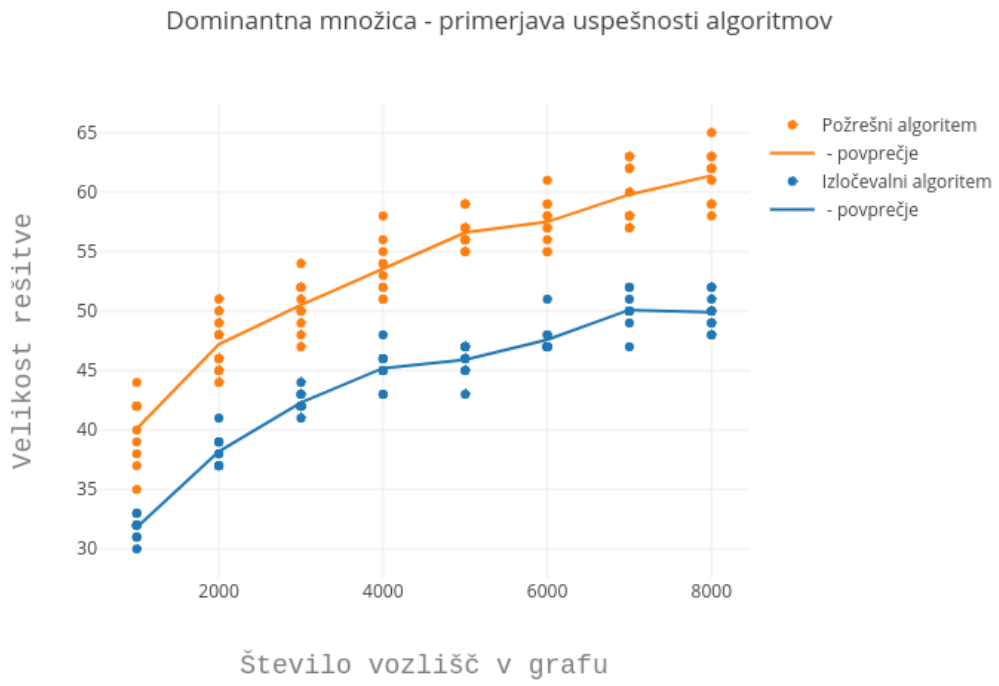
Pri tem je pri vsakem poizkusu grafa generator zgeneriral 10 različnih grafov s tako velikostjo. Na vsakem smo preizkusili oba algoritma. V grafičnem prikazu v eksperimentalni primerjavi algoritmov vsaka pika predstavlja meritev enega od teh grafov.

Pri primerjavi uspešnosti na grafičnem prikazu s črto prikazujemo povprečje meritev. Pri primerjavi učinkovitosti pa smo naleteli na težavo pri meritvah. Zaradi sproščanja pomnilnika (angl. garbage collection) je kakšna izmerjena vrednost precej odstopala [4]. Ta problem smo omilili tako, da smo

namesto povprečja izračunali mediano meritev. Tako smo s tem zmanjšali vpliv prevelikih vrednosti. Tako črta na grafičnem prikazu primerjave učinkovitosti algoritmov predstavlja mediano meritev.

2.5 Eksperimentalna primerjava algoritmov

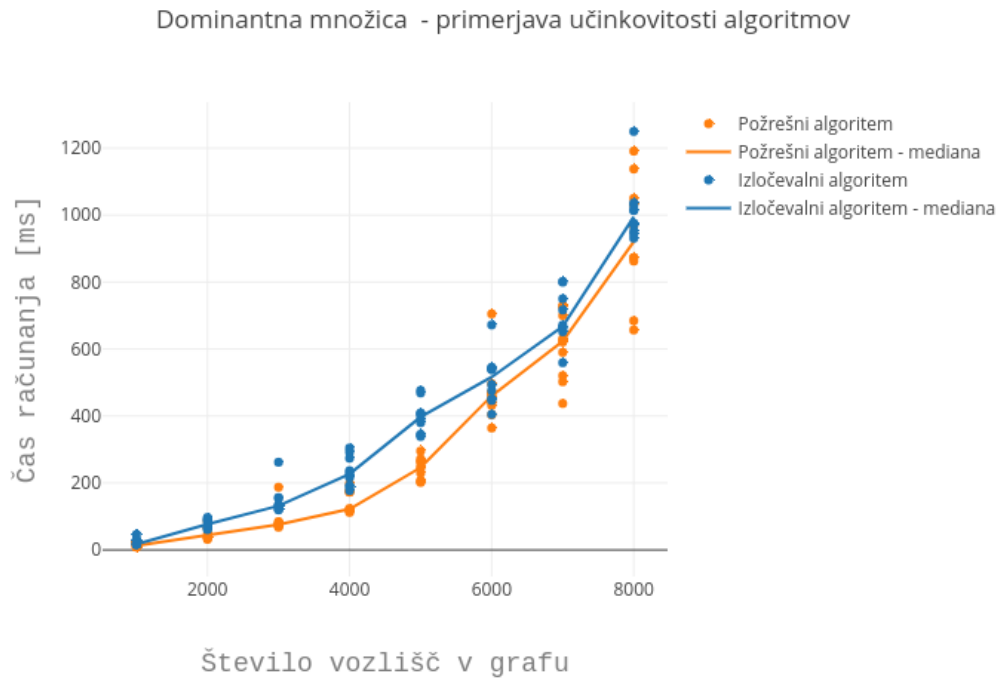
Naredili smo primerjavo obeh algoritmov. Ugotovili smo, da je izločevalni algoritem vedno vrnil boljši rezultat od požrešnega algoritma. To lahko vidimo na grafični predstavitvi rezultatov na sliki 2.2. Večji kot je graf, večja je razlika v njuni uspešnosti. Torej lahko iz rezultatov merjenja sklepamo, da je izločevalni algoritem uspešnejši od požrešnega.



Slika 2.2: Primerjava uspešnosti algoritmov

Iz slike 2.3 je razvidno, da v večini primerov izločevalni algoritem potrebuje več časa za izvajanje. Torej je po učinkovitosti požrešni algoritem boljši

od izločevalnega.



Slika 2.3: Primerjava učinkovitosti algoritmov

To pomeni, da lahko z uporabo izločevalnih algoritmov pridemo do boljšega rezultata pri iskanju dominantne množice. To lahko dosežemo le za ceno učinkovitosti. Torej, če nam je uspešnost bolj pomembna od učinkovitosti, je izbira izločevalnega algoritma primernejša od požrešnega.

Poglavje 3

Barvanje grafov

Naslednji problem, s pomočjo katerega smo primerjali požrešni in izločevalni algoritem, je problem barvanja grafov. Barvanje grafov je označevalni problem na grafu. Naš namen je označiti vsa vozlišča v grafu. Pri tem moramo upoštevati, da nobeno vozlišče ne sme imeti sosednjega vozlišča, ki bi bilo označeno z enako oznako, kot je označeno samo. Graf moramo torej označiti tako, da nobeni dve sosednji vozlišči nimata enake oznake. Tem oznakam rečemo barve, zato se problem imenuje barvanje grafov.

Kromatično število $\chi(G)$ je najmanjše število barv, s katerimi lahko pobarvamo graf G. Namen našega algoritma je, da z njegovo pomočjo minimiziramo število barv in se tako čimbolj približamo kromatičnemu številu.

Za implementacijo grafa v obeh algoritmih smo uporabili seznam sosednosti. Razlog za to je bil v tem, da algoritma pogosto dostopata do sosednjih vozlišč. Tako je bilo potrebno za boljšo učinkovitost izboljšati časovno zahtevnost te operacije.

Če je graf $G(V, E)$) predstavljen z matriko sosednosti, je pridobivanje sosednjih vozlišč manj učinkovito. Če želimo dobiti sosednja vozlišča vozlišča $v \in V$, moramo za vsako vozlišče v grafu preveriti, če je sosednje vozlišču v. Časovna zahtevnost pridobitve seznama sosednjih vozlišč je tako $O(n)$, če je n število vozlišč v grafu G. Če pa je G predstavljen z seznamom sosednosti, pa imamo za vsako vozlišče seznam vseh sosedov in lahko do seznama direktno

dostopamo. Tako lahko do seznama sosednjih vozlišč dostopamo v času $O(1)$.

3.1 Definicija problema

Pri barvanju grafov moramo vsa vozlišča v grafu pobarvati z različnimi barvami. Pri tem moramo upoštevati, da noben par sosednjih vozlišč ne sme biti enake barve.

Definicija 3.1.1 *Barvanje grafov*

Naj bo $G(V, E)$ graf. Pravilno barvanje grafa je označevanje vozlišč grafa z barvami tako, da noben par sosednjih vozlišč ni enake barve.

Pri problemu barvanja grafov moramo graf pobarvati tako, da pri tem uporabimo čim manj barv [9].

Definicija 3.1.2 *Problem barvanje grafov*

Naj bo $G(V, E)$ graf. Problem barvanja grafov je optimizacijski problem, pri katerem želimo graf pravilno pobarvati tako, da pri tem uporabimo čimmanj barv.

3.2 Požrešni algoritem

Požrešni algoritem za barvanje grafov je zelo preprost. Za vsako vozlišče $v \in V$ poišče nekonfliktno barvo in vozlišče v pobarva s to barvo. Nekonfliktna barva za vozlišče v je tista, s katero ni pobarvano nobeno sosednje vozlišče vozlišča v [3].

V naši implementaciji požrešnega algoritma so barve predstavljene s številom. Vsako število predstavlja svojo barvo.

V vsakem koraku se vozlišče označi z najmanjšo številko, s katero ni označeno nobeno sosednje vozlišče.

Za izboljšanje učinkovitosti smo algoritmu dodali hevristiko na podlagi urejanja vozlišč. Požrešni algoritem vrača boljše rezultate, če vozlišča ure-

dimo nepadajoče ali nenaraščajoče. V naši implementaciji se algoritem sprehodi čez vsa vozlišča v grafu po stopnji nepadajoče. To pomeni, da vozlišča obdeluje od vozlišča z najmanjšim številom sosedov do vozlišča z največjim številom sosedov.

Algoritem 5: Požrešni algoritem barvanja grafov

Vhod: $G(V, E)$

Izhod: $colors$

for $v \in G$ *po stopnji nepadajoče* **do**

  $color \leftarrow 1$

  while $\exists u \in \mathcal{N}(v) : colors[u] = color$ **do**

    color++;

  $color[v] \leftarrow color$

Glavna zanka se izvede n-krat, ker se algoritem sprehodi čez vsa vozlišča. V najslabšem primeru mora algoritem za iskanje proste barve iti čez vse različne barve, s katerimi so pobarvana sosednja vozlišča. Tako mora za vsako od teh barv preveriti, če je katero sosednje vozlišče pobarvano s to barvo. Število različnih barv, s katerimi so pobarvana sosednja vozlišča, je vedno manjše od števila vseh sosednjih vozlišč. Torej lahko trdimo, da je časovna zahtevnost operacije iskanja proste barve $O(d^2)$, če je d stopnja vozlišča. Torej je časovna zahtevnost tega algoritma $O(nd^2)$, če je d največja stopnja grafa. Ker je najmanjša stopnja grafa vedno manjša od števila vozlišč, velja $d < n$. Torej lahko trdimo, da je časovna zahtevnost tega algoritma $O(n^3)$

3.3 Izločevalni algoritem

Izločevalni algoritem na začetku vsako vozlišče v grafu G pobarva z drugo barvo. Množica C vsebuje vse barve, s katerimi so pobarvana vozlišča v grafu G.

Nato se algoritem sprehodi čez vse barve $c_1 \in C$ in preverja, če jih lahko odstrani. Če katero barvo lahko odstrani, jo združi z drugo ne konfliktno

barvo c_2. Barvi združi tako, da barvo c_1 izbriše. Nato pa vozlišča, ki so bila pobarvana z barvo c_1, pobarva z barvo c_2.

Barvo c_1 lahko odstrani, če obstaja kakšna taka barva c_2, da sta barvi c_1 in c_2 nekonfliktni. Barvi c_1 in c_2 sta konfliktni, če po združevanju barv barvanje ne bi bilo več pravilno. To bi se zgodilo v primeru, če bi po združitvi teh dveh barv obstajali sosednji vozlišči pobarvani z isto barvo. Torej mora algoritem preveriti, če je katero od vozlišč, pobarvano z barvo c_1, sosednje vozlišču, pobarvanemu z barvo c_2

Torej se za preverjanje konfliktnosti barv c_1 in c_2, algoritem sprehodi čez vsa sosednja vozlišča vozlišč, ki so pobarvana z barvo c_1. Za vsako tako vozlišče se preveri, če je pobarvano z barvo c_2. Če ne obstaja nobeno tako vozlišče, vozlišči c_1 in c_2 nista konfliktni in se lahko združita.

Za izboljšanje uspešnosti algoritma smo dodali hevristiko pri vrstnem redu procesiranja barv. Tako algoritem pri procesiranju po vrsti hevristično izbira vozlišča. Prioritetno izbira barve, za katere je najmanj verjetnosti, da so združljive z kakšno drugo barvo. Intuitivno smo določili, da če je vsota stopenj vozlišč barve c_1 višja od vsote stopenj vozlišč barve c_2, je za barvo c_1 manj verjetno, da bo združljiva s katerokoli drugo barvo.

Tudi pri preverjanju konfliktnosti barv smo uporabili enako hevristiko. Pri tem algoritem ne preverja barv, za katera je že preveril združljivost z vsemi ostalimi barvami.

Z vrstnim redom procesiranja smo dosegli, da algoritem najprej poizkuša združiti barve, za katere je najmanj možnosti, da se bodo lahko združile.

Za boljšo učinkovitost smo za implementacijo algoritma uporabili podatkovno strukturo disjunktnih množic. Le-te omogočajo hitro združevanje množic in preverjanje, če vozlišči pripadata isti množici.

Pri požrešnem algoritmu smo za predstavitev barv uporabili polje s števili. Vsaka številka predstavlja barvo, s katero je pobarvano vozlišče. Pri izločevalnem algoritmu pa smo za predstavitev barv uporabili množico disjunktnih množic. Vsaka množica predstavlja svojo barvo in vsebuje vozlišča, ki so pobarvana s to barvo. Tako je računanje vsot stopenj vozlišč, ki pripadajo določeni barvi

bolj učinkovito.

Algoritem 6: Izločevalni algoritem barvanja grafov

Vhod: $G(V, E)$
Izhod: C
$C \leftarrow \emptyset$
for $v \in V$ **do**
  $C[v] \leftarrow \{v\}$
$C_1 \leftarrow C$
while $C_1 \neq \emptyset$ **do**
  $c_1 \leftarrow$ barva z največjo vsoto stopenj vozlišč iz C_1
  $C_1 \leftarrow C_1 - c_1$
  $C_2 \leftarrow C_1$
  while $C_2 \neq \emptyset$ **do**
    $c_2 \leftarrow$ barva z največjo vsoto stopenj vozlišč iz C_2
    $C_2 \leftarrow C_2 - c_2$
    if $\forall v_1 \in c_1, \forall v_2 \in c_2 : v_1 \notin \mathcal{N}(v2)$ **then**
      $c_2 \leftarrow c_2 \cup c_1$
      $C \leftarrow C - c_1$
      break

Algoritem se sprehodi čez vse pare barv v grafu. Zato se preverjanje in pogojno združevanje barv izvedeta $O(n^2)$-krat. Za iskanje barve z najmanjšo vsoto stopenj vozlišč, ki so pobarvana z njo, uporabljamo prioritetno vrsto. Torej ima pridobivanje te barve časovno zahtevnost $O(1)$.

Pri preverjanju, če lahko združimo dve barvi, se moramo sprehoditi skozi vsa sosednja vozliča vozlišč pobarvanih z določeno barvo. Pri tem moramo vsako vozlišče primerjati z vsemi vozlišči druge barve. Tako je časovna zahtevnost te operacije $O(n^2)$. Pri tem smo si pomagali s podatkovno strukturo disjunktnih množic. Z njeno pomočjo lahko za dve vozlišči bolj učinkovito preverimo, če se nahajata v isti množici. Amortizirana časovna zahtevnost

te operacije je $O(\alpha(n))$, če je α inverzna Ackermannova funkcija [1]. Tako ni potrebno vozlišč, ki so sosednja vozliščem, pobarvanim z določeno barvo, primerjati z vsemi vozlišči druge barve. Vozlišča, ki so sosednja vozliščem, pobarvanim z določeno barvo lahko primerjamo le z enim vozliščem druge barve in preverimo, če se nahajajo v isti množici. Tako je časovna zahtevnost preverjanja možnosti združevanja barv $O(n\alpha(n))$. Amortizirana časovna zahtevnost združevanja disjunktnih množic je $O(\alpha(n))$.

Torej je časovna zahtevnost našega izločevalnega algoritma za barvanje grafov $O(n^3\alpha(n))$.

Disjunktne množice Pri izločevalnem algoritmu smo uporabili podatkovno strukturo disjunktnih množic.

Množici sta disjunktni, kadar nimata skupnih elementov. Torej je njun presek prazna množica.

Definicija 3.3.1 *Množici M in N sta disjunktni, kadar velja $M \cap N = \emptyset$.*

Podatkovno strukturo disjunktne množice predstavlja množica množic, ki so med seboj paroma disjunktne. Podpira operacije ustvarjanja množic, unije in iskanja v skoraj konstantnem času.

Definicija 3.3.2 *Podatkovna struktura disjunktne množice je množica takih množic M, da velja $\forall M_i \in M, \forall M_j \in M, M_i \neq M_j, \nexists x : x \in M_i \wedge x \in M_j$.*

Podatkovno strukturo disjunktnih množic smo uporabili pri algoritmu barvanja grafov, da smo z njo preverili, če sta dve vozlišči iste barve.

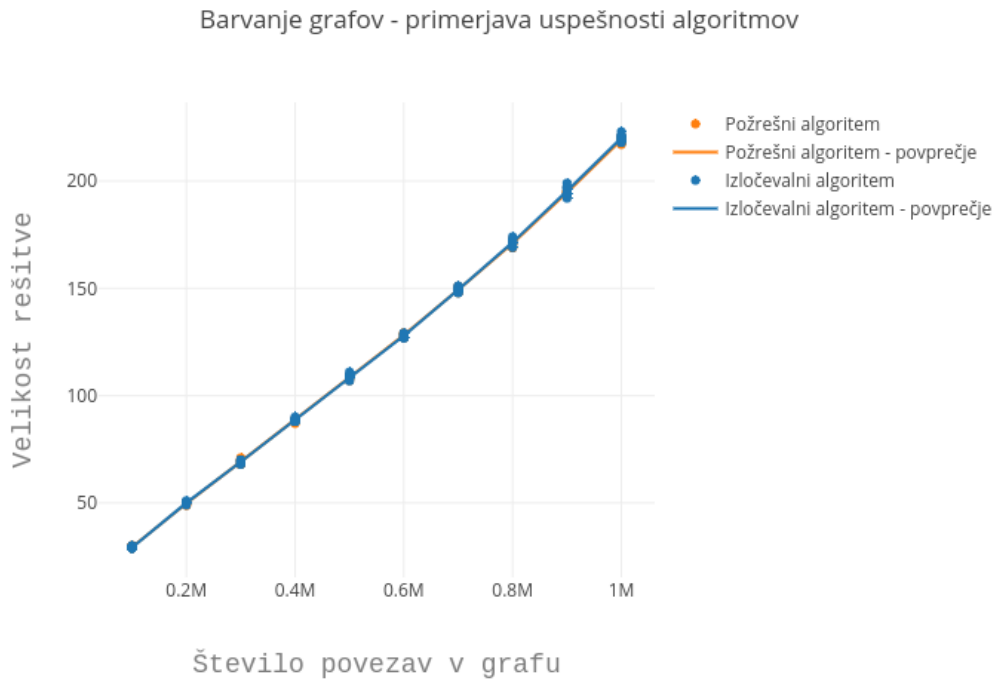
3.4 Eksperimentalna primerjava algoritmov

Naredili smo eksperimentalno primerjavo obeh algoritmov.

Pri problemu barvanja grafov smo uporabili enako testno okolje, kot pri problemu dominantne množice.

Za vhodne podatke pa smo uporabili grafe s 2000 vozlišči in 100.000 povezavami. Pri vsakem naslednjem poskusu smo uporabili graf s 100.000 dodatnimi povezavami. To smo ponavljali do grafa s 1.000.000 povezavami. Torej smo generirali grafe z gostotami 0.025, 0.05, 0.075, ..., 0.25. Testiranje pa je potekalo na enak način kot pri problemu dominantne množice.

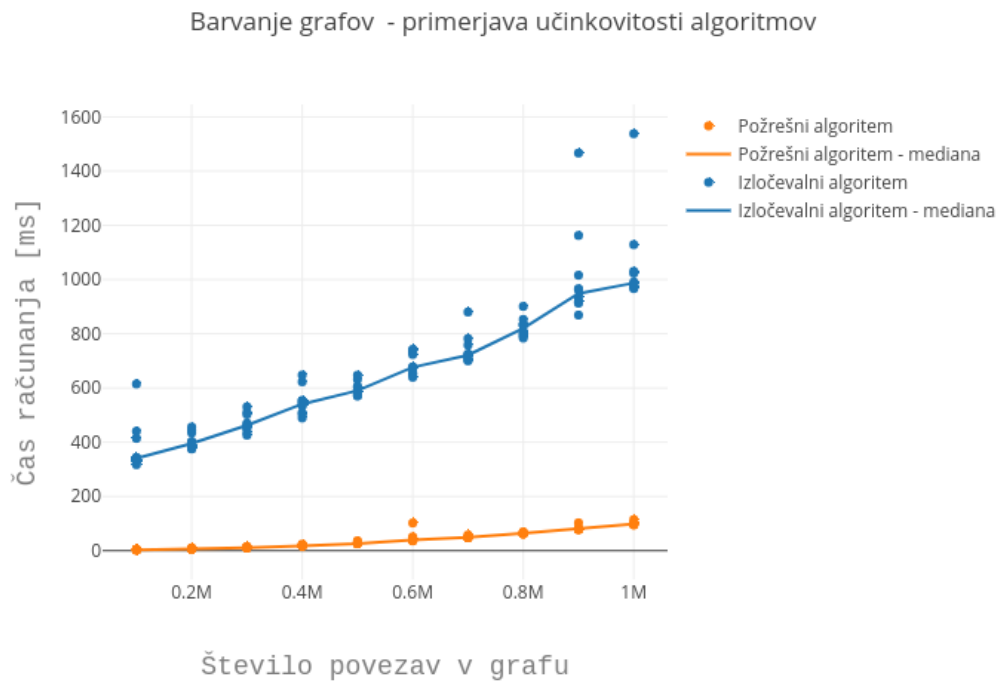
Ugotovili smo, da je uspešnost pri obeh algoritmih zelo podobna. To lahko vidimo na sliki 3.1.



Slika 3.1: Primerjava uspešnosti algoritmov

Požrešni algoritem vrne približno enako dober rezultat kot izločevalni algoritem, a je v primerjavi z njim učinkovitejši.

Iz tega lahko sklepamo, da je pri primeru barvanja grafov, bolje uporabiti klasični požrešni algoritem.



Slika 3.2: Primerjava učinkovitosti algoritmov

Poglavje 4

Najmanjše vozliščno pokritje

Problem vozliščnega pokritja je problem pokrivanja na grafu. Pri tem problemu moramo poiskati tako podmnožico vozlišč v grafu, da z njo pokrijemo vse povezave v grafu. Optimizacijski problem, ki izhaja iz problema vozliščnega pokritja, je problem najmanjšega vozliščnega pokritja. Pri tem problemu želimo minimizirati moč množice vozliščnega pokritja.

Problem najmanjšega vozliščnega pokritja je minimizacijski problem. Maksimizacija tega problema pa ne bi bila smiselna, saj je največje vozliščno pokritje v grafu enako množici vseh vozlišč v grafu.

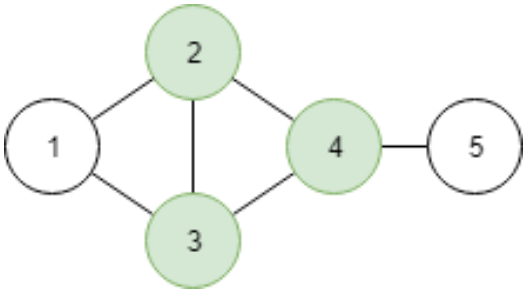
4.1 Definicija problema

Pri problemu vozliščnega pokritja moramo najti taka vozlišča, da pokrijemo vse povezave v grafu. Z vozliščem v lahko pokrijemo njene sosednje povezave. Sosednje povezave grafa so tiste, ki vozlišče v povezujejo z sosednjimi vozlišči.

Definicija 4.1.1 *Naj bo $G(V, E)$ neusmerjen graf. Vozliščno pokritje $S \subseteq V$ grafa $G(V, E)$ je taka množica vozlišč, da za vse povezave $(u, v) \in E$ velja $u \in S \vee v \in S$.*

Pri problemu najmanjšega vozliščnega pokritja iščemo najmanjše vozliščno pokritje v grafu.

Na enem grafu imamo lahko več vozliščnih pokritij. Prav tako imamo lahko tudi več najmanjših vozliščnih pokritij. Na sliki 4.1 vidimo graf, na katerem je z zeleno barvo označeno vozliščno pokritje. To je eno od možnih vozliščnih pokritij, ki je hkrati tudi najmanjše. Na tem grafu imamo tudi vozliščno pokritje {1, 2, 4}, ki je tudi najmanjše. Prav tako je vozliščno pokritje tega grafa tudi {1, 2, 3, 4}. V tem primeru pa to ni najmanjše vozliščno pokritje.



Slika 4.1: Primer vozliščnega pokritja

4.2 Požrešni algoritem

Požrešni algoritem obdeluje vozlišča od vozlišča z največjim številom sosedov do tistega z najmanjšim. V vsakem koraku se za vozlišče v preveri, če ima kakšno sosednje vozlišče in če ga ima, se vozlišče v doda v rešitev. Nato vozlišče v izbriše iz grafa.

Vozlišča obdelujemo po stopnji nenaraščajoče, ker lahko tako v vsakem koraku algoritma pokrijemo največ povezav. S to hevristiko želimo doseči, da s čim manj vozlišči pokrijemo celoten graf.

Algoritem 7: Požrešni algoritem iskanja najmanjšega vozliščnega pokritja

Vhod: $G(V, E)$
Izhod: C
$C \leftarrow \emptyset$
while $V \neq \emptyset$ **do**
    $v \leftarrow$ vozlišče z največjim številom sosedov
    if $deg(v) \neq 0$ **then**
        $C \leftarrow C \cup \{v\}$
    $G = G - v$

Osnovna zanka se izvede v n korakih. Za iskanje vozlišča z največjim številom sosedov, bi morali iti čez vsa vozlišča v grafu. Časovna zahtevnost te operacije je $O(n)$. Zato smo izboljšali učinkovitost tako, da na začetku vozlišča uredimo. Časovna zahtevnost urejanja je $O(n \log n)$ [6]. Torej je časovna zahtevnost požrešnega algoritma za problem vozliščnega pokritja $O(n \log n)$.

4.3 Izločevalni algoritem

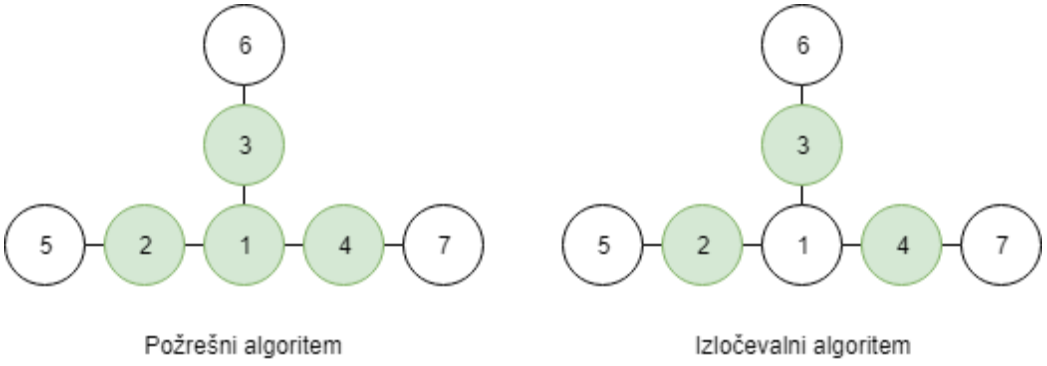
Sedaj si bomo ogledali izločevalni algoritem za problem najmanjšega vozliščnega pokritja. Algoritem najprej začne z vozliščem z najmanjšo stopnjo in ga izloči iz rešitve. Nato pa vsa njegova sosednja vozlišča doda v rešitev, saj so edina, ki lahko še pokrijejo njegove sosednje povezave.

Na sliki 4.2 vidimo primer vozliščnega pokritja v primeru uporabe požrešnega in izločevalnega algoritma. Na izbranem grafu se dobro vidi prednost izločevalnega algoritma.

Požrešni algoritem najprej obdela vozlišča 1, 2, 3 in 4 ter vsa doda v rešitev. Ostalih vozlišč pa ne doda, ker imajo vse povezave že pokrite.

Izločevalni algoritem pa najprej obdela vozlišča 5, 6, in 7, ker imajo naj-

manjšo stopnjo. Njihova sosednja vozlišča 2, 3 in 4 doda v rešitev. Vozlišča 2, 3 in 4 tako sama pokrivajo svoje sosednje povezave. Na koncu ostane še vozlišče 1, ki pa ima že vse sosednje povezave pokrite.



Slika 4.2: Prikaz boljšega rezultata izločevalnega algoritma

Izločevalni algoritem obdeluje vozlišča nepadajoče. To pomeni, da najprej začne z vozlišči, ki pokrivajo najmanj vozlišč. Tako vozlišča po vrstnem redu odstranjuje iz rešitve. Ob vsaki odstranitvi vozlišča v pa mora nato v rešitev dodati vsa njegova sosednja vozlišča. Razlog za to je v tem, da so to edina vozlišča, ki lahko poleg vozlišča v pokrijejo sosednje povezave vozlišča v.

Algoritem 8: Izločavalni algoritem iskanja dominantne množice

Vhod: $G(V, E)$
Izhod: C
$C \leftarrow \emptyset$
for $v \in V$ *po stopnji nepadajoče* **do**
  if $v \notin C$ **then**
    $C \leftarrow C \cup \mathcal{N}(v)$

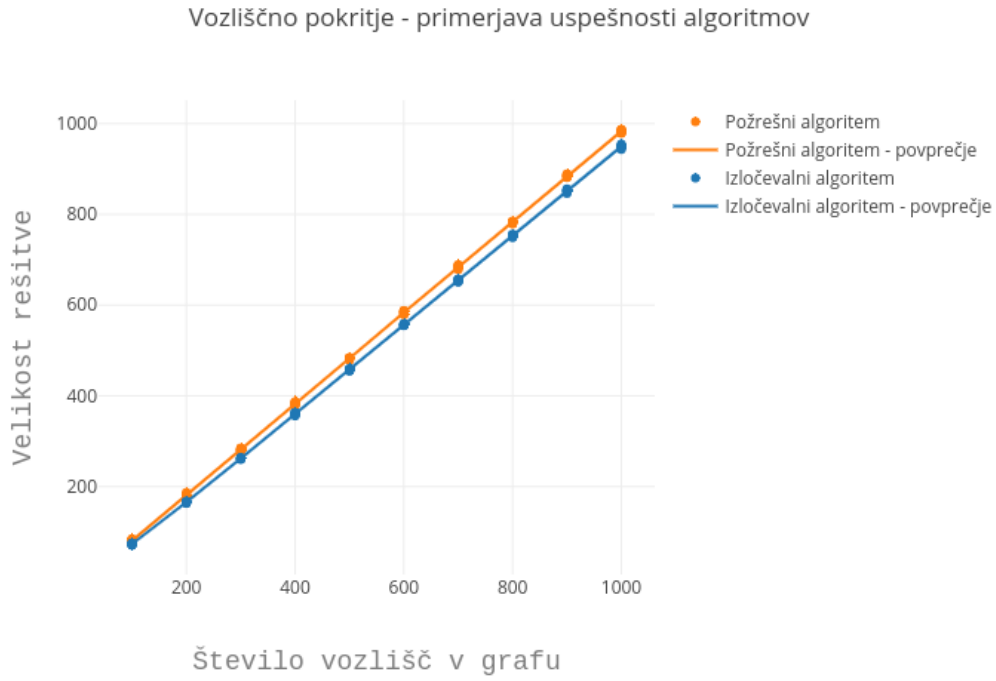
Časovna zahtevnost tega algoritma je prav tako $O(n)$, saj se algoritem samo enkrat sprehodi čez vsa vozlišča.

4.4 Eksperimentalna primerjava algoritmov

Za testiranje in meritve smo uporabili isto testno okolje kot pri ostalih problemih.

Vhodne podatke smo generirali na enak način kot pri problemu dominantne množice. Prav tako je merjenje uspešnosti in učinkovitosti potekalo na enak način kot pri ostalih problemih. Za primerjavo uspešnosti smo uporabili manjše grafe kot pri ostalih problemih. Uporabili smo grafe s 100 do 1000 vozlišči, ker je v tem primeru na grafičnem prikazu razlika bolje vidna.

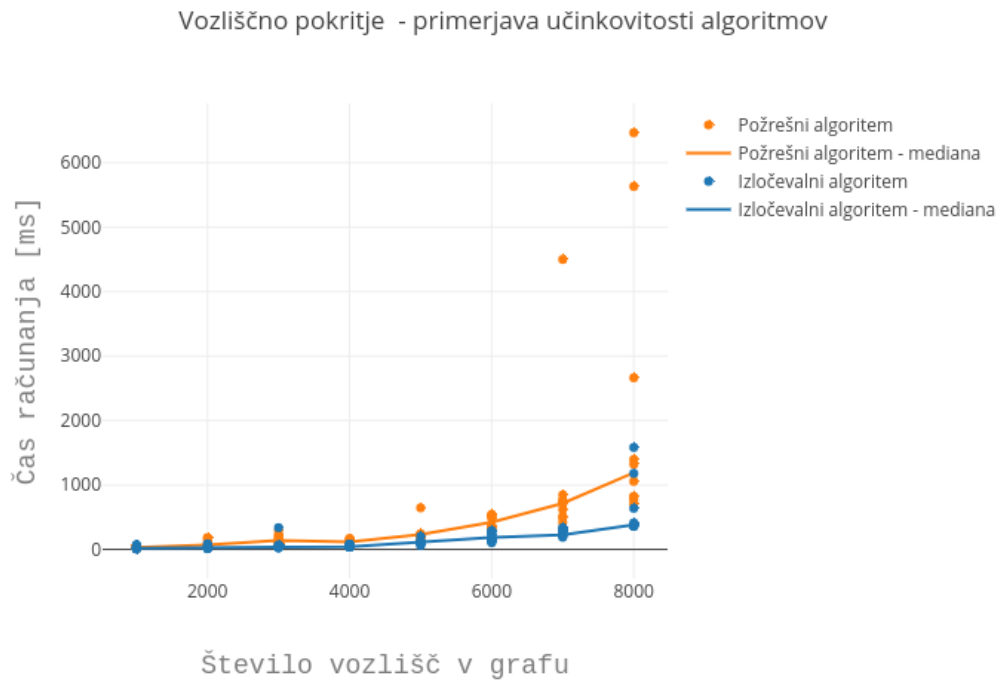
Pri primerjavi obeh algoritmov smo ugotovili, da izločevalni algoritem vrne boljšo rešitev. To lahko vidimo na sliki 4.3.



Slika 4.3: Primerjava uspešnosti algoritmov

Na sliki 4.4 pa lahko vidimo, da je izločevalni algoritem bolj učinkovit.

Tako smo ugotovili, da je za vozliščno pokritje izločevalni algoritem boljši



Slika 4.4: Primerjava učinkovitosti algoritmov

od požrešnega.

Poglavje 5

Zaključek

V diplomski nalogi smo predstavili požrešne algoritme in idejo o možnostih izboljšave le-teh. Izboljšavo smo želeli doseči s tem, da bi za reševanje problema uporabili algoritme, ki v vsakem koraku izločijo najslabše možnosti. Tovrstne algoritme smo poimenovali izločevalni algoritmi. Nato smo oba algoritma, požrešnega in izločevalnega, implementirali na izbranih NP-težkih problemih na grafu. Na koncu pa smo ju med seboj primerjali po njuni uspešnosti in učinkovitosti. Izbrali smo probleme najmanjše dominantne množice, barvanja grafov in najmanjšega vozliščnega pokritja.

Za vse probleme smo naredili primerjavo obeh algoritmov. Izkazalo se je, da je pri vseh problemih izločevalni algoritem dal boljše ali enako dobre rezultate kot požrešni algoritem. Po učinkovitosti pa je bil boljši požrešni algoritem, razen pri vozliščnem pokritju.

Izkazalo se je, da lahko z izločevalnim algoritmom pri problemu najmanjše dominantne množice dosežemo boljše rezultate. To pa lahko dosežemo le za ceno učinkovitosti.

Izločevalni algoritem je bil v primeru dominantne množice in vozliščnega pokritja uspešnejši. Vračal je očitno boljše rezultate. Iz tega lahko vidimo, da se izločevalni algoritem očitno dobro obnese pri problemih pokrivanja na grafu, ki smo jih implementirali v tej nalogi. Raziskavo bi bilo smiselno nadaljevati na takih problemih. Pri problemu barvanja grafov pa sta oba

algoritma dala približno enake rezultate. Ker je bila učinkovitost v tem primeru slabša pri izločevalnem algoritmu, se nam uporaba le-tega ne zdi smiselna.

Na podlagi rezultatov sklepamo, da je pri problemu vozliščnega pokritja uporaba izločevalnega algoritma boljša. Prav tako je uporaba smiselna pri problemu dominantne množice, a v ta namen zmanjšamo učinkovitost reševanja problema. Uporaba pri problemu barvanja grafov pa ni smiselna, saj z njim niti ne izboljšamo rezultata niti ne pospešimo reševanja problema.

Obstaja še veliko možnosti za raziskovanje na tem področju. Raziskavo bi lahko razširili še z drugimi problemi na grafih, npr. kliko ali neodvisno množico. Izločevalne algoritme bi lahko mogoče uporabili tudi na drugih optimizacijskih problemih, ki niso na grafih. Ena izmed možnosti nadaljevanja bi bila še izboljšava napisanih algoritmov, ki so nastali pri izdelavi te diplomske naloge. Lahko bi na primer izboljšali učinkovitost algoritmov s podatkovno strukturo, ki omogoča hitrejše operacije.

Literatura

[1] Stephen Alstrup, Inge Li Gørtz, Theis Rauhe, Mikkel Thorup, and Uri Zwick. Union-find with constant time deletions. In *International Colloquium on Automata, Languages, and Programming*, pages 78–89. Springer, 2005.

[2] Alina Campan, Traian Marius Truta, and Matthew Beckerich. Fast dominating set algorithms for social networks. In *MAICS*, pages 55–62, 2015.

[3] Joseph Culberson. Iterated greedy graph coloring and the difficulty landscape. In *Technical Report TR*. DEPARTMENT OF COMPUTING SCIENCE, The University of Alberta Edmonton, Alberta, Cana, 1992.

[4] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *ACM SIGPLAN Notices*, volume 42, pages 57–76. ACM, 2007.

[5] Harvey J Greenberg. Greedy algorithms for minimum spanning tree. *University of Colorado at Denver*, 1998.

[6] Volodymyr Korniichuk. Timsort sorting algorithm, 2015. Dostopno na `http://www.infopulse.com/blog/timsort-sorting-algorithm/`. [Obiskano 2019-03-05].

[7] Jurij Mihelič. Hevristično reševanje problemov pokrivanja in razmeščanja. Magistrsko delo, Fakulteta za računalništvo in informatiko, Univerza v Ljubljani, Ljubljana, 2004.

[8] Jurij Mihelič and Borut Robič. Solving the k-center problem efficiently with a dominating set algorithm. *Journal of computing and information technology*, 13(3):225–234, 2005.

[9] Ehsan Salari and Kourosh Eshghi. An ACO algorithm for graph coloring problem. In *2005 ICSC Congress on computational intelligence methods and applications*, pages 5–pp. IEEE, 2005.

[10] Yiyuan Wang, Shaowei Cai, Jiejiang Chen, and Minghao Yin. A fast local search algorithm for minimum weight dominating set problem on massive graphs. In *IJCAI*, pages 1514–1522, 2018.