УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Немања Живановић

ГЕНЕТСКИ АЛГОРИТАМ ЗА РЕШАВАЊЕ ПРОБЛЕМА МИНИМАЛНОГ ДОМИНИРАЈУЋЕГ СКУПА

мастер рад

Београд, 2023.

Ментор:

др Нина Радојичић Матић, доцент
Универзитет у Београду, Математички факултет

Чланови комисије:

др Милена Вујошевић Јаничић, ванредни професор
Универзитет у Београду, Математички факултет

др Весна Маринковић, доцент
Универзитет у Београду, Математички факултет

Датум одбране: _____

Наслов мастер рада: Генетски алгоритам за решавање проблема минималног доминирајућег скупа

Резиме: У овом раду је приказан генетски алгоритам за решавање проблема минималног доминирајућег скупа, проналаском конкретног минималног доминирајућег скупа и рачунањем његове кардиналности. Посебан акценат стављен је на кодирање графа и јединки коришћењем низа природних бројева, специфично генерисање почетне популације јединки, побољшање начина поправке невалидних јединки, као и на представљање најбољег начина за рачунање броја јединица у бинарном запису јединке. Алгоритам је имплементиран у програмском језику Пајтон, заснован је на раду са битским операторима и користи локалну претрагу за побољшање решења. Вредности параметара генетског алгоритма су прилагођене за различите улазе и подешене користећи пакет irace и интерполацију помоћу радијалне базне функције. На крају рада је извршено поређење имплементираног генетског алгоритма са алгоритмом грубе силе и једним од најбољих алгоритама из литературе за решавање овог проблема.

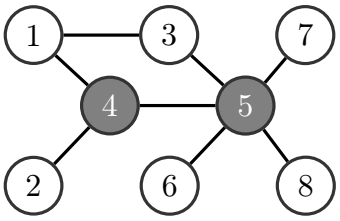
Кључне речи: генетски алгоритам, минимални доминирајући скуп, листа улазних суседа чворова, листа потенцијално недоминираних чворова

Садржај

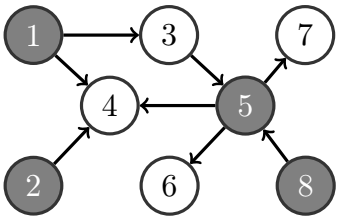
Глава 1

Увод

За произвољан прост неусмерен граф[1] $G = (V, E)$, подскуп чворова $D \subseteq V$ се назива **доминирајућим скупом** за граф G ако за сваки чвор из V важи да се или налази у D, или је суседан неком чвору из D [51]. Овај проблем постоји и у усмереној варијанти, где је D доминирајући скуп ако за сваки чвор ван D постоји грана усмерена према том чвору од неког чвора из D [42]. **Кардиналност** неког скупа представља број елемената у том скупу. Минимални доминирајући скуп је скуп најмање кардиналности међу свим доминирајућим скуповима графа G. Примери таква два скупа приказани су на сликама 1.1 и 1.2, где су чворови који припадају решењу обојени сивом бојом.



Слика 1.1: Пример минималног доминирајућег скупа на неусмереном графу



Слика 1.2: Пример минималног доминирајућег скупа на усмереном графу

Проблем минималног доминирајућег скупа за граф G је проблем

[1]Граф G је уређени пар (V, E), где скуп V означава чворове графа, а скуп E гране и важи $E \subseteq V \times V$. Уколико су елементи скупа E уређени парови чворова, тада се граф назива усмереним, док је у супротном неусмерен. Ако је E заправо мултискуп, односно ако је могуће постојање више грана (у истом смеру) између два иста чвора, такве гране се називају паралелним. Грана чији је почетак и крај у истом чвору назива се петљом. За граф који не садржи ни паралелне гране ни петље каже се да је прост [57].

одређивања кардиналности[2] минималног доминирајућег скупа графа G и то је проблем из класе $\mathcal{NP}$-тешких проблема. Овај проблем има широку примену у пракси и то, у неусмереној варијанти, у рачунарским [58, 43] и друштвеним [29, 53] мрежама, биоинформатици [54], роботици [46], алокацији ресурса [11], а у усмереној варијанти у рутирању [59] и базама података [31].

Уколико у графу G постоји грана $(u, v) \in E$, каже се да је чвор v доминиран чвором u. На слици 1.1 чворови 1 и 5 су доминирани чвором 3 (важи и обратно), док је на слици 1.2 чвор 4 доминиран чворовима 1, 2 и 5 (не важи обратно). Уколико за чвор v постоји чвор из доминирајућег скупа који га доминира или уколико је v део доминирајућег скупа, каже се да је чвор v доминиран тим доминирајућим скупом (краће - **доминирани чвор**), док у супротном важи да није доминиран (краће - **недоминирани чвор**). На основу тога, за сваки доминирајући скуп важи да у графу не смеју да постоје недоминирани чворови.

Најбољи познати алгоритми за егзактно решавање овог проблема за произвољан граф приказани су у табели 1.1. Поред егзактних, постоје и генетски алгоритми који решавају овај проблем [24, 20].

Аутори	Полиномски простор[3]	Експоненцијални простор
Фомин и др. [19]	$\mathcal{O}(1.9379^n)$	
Ширмајер [48]	$\mathcal{O}(1.8899^n)$	
Грандони [23]	$\mathcal{O}(1.9053^n)$	$\mathcal{O}(1.8021^n)$
Фомин и др. [17]	$\mathcal{O}(1.5263^n)$	$\mathcal{O}(1.5137^n)$
Ван Рој и др. [52]		$\mathcal{O}(1.5048^n)$
Ван Рој и Бодландер [51]	$\mathcal{O}(1.4969^n)$	
Јоичи [27]	$\mathcal{O}(1.4864^n)$	$\mathcal{O}(1.4689^n)$

Табела 1.1: Најбољи познати алгоритми за егзактно решавање проблема минималног доминирајућег скупа и њихове временске сложености

У раду су прво описани основни елементи општег генетског алгоритма и уведена неопходна терминологија, а затим је описан имплементирани генетски алгоритам који решава проблем минималног доминирајућег скупа. На крају, приказано је поређење генетског алгоритма са алгоритмом грубе силе и Ван Рој-Бодландеровим алгоритмом.

[2]Кардиналност минималног доминирајућег скупа се назива и доминирајућим бројем [18].

[3]За алгоритам се каже да захтева полиномски простор уколико неопходна меморија за његово извршавање расте ограничена (од горе) неком полиномском функцијом у односу на улаз, а слично важи за експоненцијални простор.

Глава 2

Генетски алгоритам

Један од основних циљева при писању квалитетних програма јесте да исправно решавају проблем за који су написани, односно да проналазе оптимално решење. Поред тога, програм би требало да буде и што је могуће ефикаснији, али те две ствари су неретко у колизији. Класу $\mathcal{NP}$-тешких проблема, неформално, чине проблеми који су сувише тешки за ефикасно решавање на великим улазима, односно за које не постоји алгоритам који ће их у разумном времену решити. У том случају се поставља питање да ли је заиста неопходно да алгоритам врати тачно решење или, прецизније, да ли је неопходно да алгоритам *гарантује* да ће то учинити? Некада је то можда услов без ког се не може, али, насупрот томе, некада је решење које је довољно добро[1] можда сасвим задовољавајуће.

Хеуристички алгоритми су управо алгоритми који за релативно кратко време (под тим се углавном подразумева полиномско време) враћају решења за која не постоји гаранција да су оптимална, па чак ни да су близу њих, али која су се често у пракси показала као довољно добра. Овакви алгоритми припадају групи апроксимативних метода и користе неку хеуристику, односно неку претпоставку о проблему која може да одведе алгоритам ка довољно добрим решењима [30]. Главни недостатак хеуристичких алгоритама јесте што су преприлагођени конкретном проблему и најчешће их није могуће уопштити и применити за решавање неког другог проблема. Апроксимативним методама припадају и метахеуристички алгоритми, који немају поменути не-

[1]Шта се сматра довољно добрим решењем зависи од конкретног проблема и конкретне примене, али на примеру минималног доминирајућег скупа, уколико је задат граф од 10 000 чворова, врло је вероватно да неке значајније разлике између доминирајућег скупа од нпр. 50 (као замишљеног оптималног решења) и другог од 55 чворова заправо и нема.

достатак пошто се не базирају ни на једном конкретном проблему. Они представљају апстрактне оквире који, уз нека додатна специфична подешавања за проблем који се решава, служе за креирање конкретних алгоритама [1]. Један од најпознатијих метахеуристичких алгоритама јесте **генетски алгоритам**, чији је аутор Џон Холанд (енг. *John Holland*).

Пре описивања компоненти општег генетског алгоритма, прво се морају увести термини којима се барата током разних фаза извршавања. Будући да је сам алгоритам уско повезан са биологијом, преовладава биолошка терминологија, али је интерпретација термина у генетском алгоритму поприлично поједностављена у односу на реалан случај. Једна од основних дефиниција биологије је да је ћелија основна јединица грађе и функције свих живих бића. Свака од тих ћелија садржи скуп хромозома[2], а сваки хромозом садржи део целокупне ДНК, која се у (скоро) истом облику налази у хромозомима свих ћелија. ДНК (скраћено од дезоксирибонуклеинска киселина) представља шаблон за раст, развој и функционисање организма, односно она садржи целокупну генетску информацију (која се назива геномом) организма. ДНК може да се посматра као веома дугачка ниска која садржи четири елемента (слова) А (аденин), Ц (цитозин), Г (гуанин) и Т (тимин) који се групно зову нуклеотиди. Ген представља сегмент ДНК (тзв. нуклеотидну секвенцу) који одређује неку особину организма (нпр. боју косе или величину шаке), док је алел конкретна варијација тог сегмента, односно конкретна особина (нпр. црна или зелена боја очију)[3]. Приликом репродукције, дете наслеђује једну гарнитуру хромозома од оца и једну од мајке (код човека, у питању су гарнитуре од по 23 хромозома колико се налази у полним ћелијама) које се посматрају у паровима. Хромозоми које ће отац проследити детету настају рекомбинацијом гена одговарајућег пара хромозома самог оца (које је он наследио од свог оца и мајке), а ово исто важи и за мајку. Рекомбинација подразумева да два хромозома буду измењена тако што се неки њихови делови међусобно размене. Оно што важи за једну популацију неке врсте јесте да се јединке рађају, потенцијално остављају потомство и умиру. Претпоставка је, додатно, да већу шансу за преживљавањем, па самим тим и остављањем потомства, имају је-

[2]Човек има 46 хромозома у соматским (телесним), односно 23 хромозома у полним ћелијама. Други организми имају различит број.

[3]Прецизније, један ген одређује једну особину само ако су у питању тзв. моногенске особине. Особине које су наведене као примери су заправо полигенске, код којих више гена утиче на особину, али је овако остављено зарад једноставнијег разумевања.

динке које су у неком смислу боље или способније за преживљавање, а да се разноликост популације, поред рекомбинације родитељских гена, постиже и мутацијама, које случајно мењају неке од нуклеотида у неке друге [39].

2.1 Повезивање проблема и генетског алгоритма

Генетски алгоритам опонаша претходно описане принципе развоја популације. Веза између биологије и генетског алгоритма је следећа. Свако (теоретско) решење проблема (енг. *candidate solution*) посматра се као јединка популације у генетском алгоритму. Саме јединке најчешће имају само један хромозом, па се између ова два појма у генетском алгоритму углавном ставља знак једнакости. Одговарајућа веза између могућих решења и јединке остварује се пресликавањем решења у хромозом (кодирање) и пресликавањем хромозома назад у решење (декодирање). Хромозом може имати различите репрезентације, а једна од најучесталијих је низ бинарних цифара фиксне дужине. За такву репрезентацију сваки од битова ће представљати један ген, а вредност тог бита биће алел. Шта ће бити функција која представља пресликавање зависиће и од проблема и од репрезентације јединке.

Квалитет јединке описује **функција прилагођености** (енг. *fitness function*). Њена интерпретација у генетском алгоритму треба да одговара интерпретацији квалитета решења у реалном проблему. Будући да се рачуна за сваку јединку у алгоритму, идеално је ако се функција прилагођености може рачунати ефикасно, али то није увек могуће. Још једна функција која је важна је **функција циља** (енг. *objective function*). Будући да су проблеми који се решавају генетским алгоритмом проблеми оптимизације[4] (или се на њих могу свести) функција циља је управо функција коју треба минимизовати или максимизовати. Природно је претпоставити да је управо функција прилагођености исто што и функција циља и то углавном јесте случај, али постоје ситуације када то не важи. Један такав пример је када у популацији постоје неке јединке које не представљају валидна решења проблема и које треба обележити како би ускоро биле уклоњене. Ако је циљ пронаћи јединку која максимизује неку функцију циља (и испуњава све постављене услове), не-

[4]Неформално, то су проблеми код којих из скупа свих могућих решења треба пронаћи оно за које нека посматрана функција има максималну (или минималну) вредност.

валидним јединкама се може доделити нула (ако функција прилагођености додељује само позитивне вредности валидним јединкама) [3]. Надаље ће се, зарад краћег записа, под вредношћу јединке подразумевати вредност функције прилагођености за ту јединку.

2.2 Почетна популација и генетски оператори

Да би генетски алгоритам уопште могао да започне са радом, неопходно је да постоји **почетна популација** јединки (енг. *initial population*). Начин на који ће она бити креирана зависи пре свега од репрезентације, али углавном је то, на неки начин, псеудослучајно генерисање јединки. На примеру бинарне репрезентације јединки, почетна популација може бити конструисана тако што сваки бит сваке јединке има једнаку вероватноћу да буде нула или један. Потребно је и пажљиво одабрати величину популације. Над таквим јединкама се, у циљу развоја популације, примењују два типа генетских оператора, оператор селекције и оператори варијације.

Селекција

Селекција је процес бирања скупа[5] јединки за репродукцију[6], на основу чијих гена ће настати нова популација. Квалитетна селекција треба да доведе до побољшања просечног квалитета читаве популације, тако што ће боље јединке имати већу шансу да буду убачене у скуп за репродукцију и тиме имати шансу да прошире своје гене кроз популацију. Један начин би био да се јединке бирају потпуно насумично из тренутне популације. Овакав одабир, међутим, може да штети перформансама, пошто алгоритам у таквој ситуацији не преферира квалитетније јединке. С друге стране, није добро узети само неки фиксирани број најбољих јединки, зато што је онда реална могућност успостављања пребрзе конвергенције и многа далеко квалитетнија решења потенцијално неће бити истражена. Због тога постоји неколико начина за избор јединки у скуп за репродукцију. Такви начини се деле у групе пропорционалних и ординалних селекција. У пропорционалним селекцијама јединке

[5]Прецизније је рећи да је у питању мултискуп будући да иста јединка може бити више пута одабрана у скуп за репродукцију.

[6]Процес креирања нових јединки се назива репродукција и обухвата укрштање и мутацију.

се бирају на основу њихових вредности релативно у односу на вредности осталих јединки. Најпознатији пример оваквог типа селекције је **рулетска селекција** (енг. *roulette wheel selection*), а постоје и други попут стохастичке универзалне селекције (енг. *stochastic universal selection*) или сигма скалирајуће селекције (енг. *sigma scaling selection*). Код ординалних селекција, вредност јединке има улогу искључиво у одређивању њеног ранга у популацији, а јединке са бољим вредностима имаће боље рангове од оних са лошијим вредностима. На основу рангова се затим бирају јединке. Најпознатији представник овог типа селекције је **турнирска селекција** (енг. *tournament selection*), а постоје и други попут селекције линеарног рангирања (енг. *linear ranking selection*) или селекције скраћивањем (енг. *truncation selection*) [3]. Све ове селекције могу бити имплементиране у варијантама са понављањем (енг. *with replacement*) и без понављања (енг. *without replacement*). У наставку ће бити описане само рулетска и турнирска селекција. У намери да се избегне селекција само најбољих јединки, може се нехотично створити други проблем, а то је да се доста најбољих или чак нека истакнута најбоља јединка не одабере у скуп за репродукцију. Тада постоји могућност да квалитетан генетски материјал буде трајно изгубљен, па би ваљало имати механизам да се то спречи. **Елитизам** је стратегија која то обезбеђује и може бити имплементиран тако да неки број најбољих јединки буде одмах, пре селекције, убачен у скуп за репродукцију. Друга варијанта је да неки број најбољих јединки буде сачуван у популацији и да не буде уопште замењен. Са величином елитизма треба бити веома пажљив, будући да ће превелики број сачуваних најбољих јединки довести до пребрзе конвергенције.

Рулетска селекција

Рулетска селекција заснива се на принципу одабира бројева на рулету. Рулетски точак је подељен на 37 поља једнаке ширине и, под фер условима, бачена куглица има једнаку вероватноћу ($\frac{1}{37}$) да заврши на било ком од тих поља. У рулетској селекцији поља представљају јединке у популацији, али вероватноће избора јединки (односно ширине поља) нису једнаке, већ су сразмерне вредностима јединки. Уколико нека јединка a_i има вредност $f(a_i)$ и уколико је сума свих вредности јединки $k = \sum_{j=1}^{n} f(a_j)$ тада је вероватноћа да ће она бити одабрана $f(a_i)/k$. Бирање једне јединке рулетском селекцијом се најчешће имплементира на следећи начин:

```
k = total_fitness(population)
value = random.randrange(k)
current_sum = 0
for chromosome in population:
    current_sum += chromosome.fitness
    if current_sum > value:
        return chromosome
```

Турнирска селекција

Турнирска селекција базира се на томе да се неки број јединки из популације псеудослучајно одабере да учествује на турниру. Победник турнира биће јединка са најбољом вредношћу и њу треба убацити у скуп за репродукцију. Овакав тип селекције је једноставнији за имплементацију и временски ефикаснији од рулетске селекције, али може да доведе до пребрзе конвергенције. То следи због тога што важи да ће, што је турнир већи, бити већа и вероватноћа да је нека од најквалитетнијих јединки на турниру (те ће бити победник), па самим тим постоји опасност од тога да ће се у скупу за репродукцију наћи готово искључиво најбоље јединке, што је проблем који је већ споменут. То се може избећи коришћењем турнира у пару (енг. *pairwise tournament*) у којима учествују само две јединке. Ова величина турнира се добро показала у спречавању преране конвергенције, али може одвести у други екстрем, а то је преспора конвергенција, посебно уколико је популација доста велика [61, 3].

Варијација

Улога варијације је да изврши „мешање" генетског материјала који постоји у популацији с циљем добијања бољих јединки, а служи и за потенцијално увођење нових алела којих нема у популацији. У операторе варијације спадају **укрштање**, које се примењује над паровима јединки и **мутација**, која се примењује над појединачним јединкама.

Укрштање

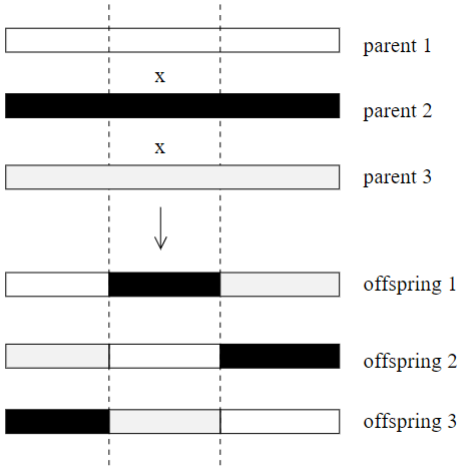
Укрштање (енг. *crossover*) или рекомбинација (енг. *recombination*) служи за креирање нових јединки, које се називају деца, тако што се из скупа за репродукцију узимају две по две јединке, које се називају родитељи и рекомбинује њихов генетски материјал како би се добиле нове јединке. Постоји

неколико начина на који деца могу наслеђивати генетски материјал. Два најчешћа приступа за бинарну репрезентацију јединки су једнопозиционо или укрштање у једној тачки (енг. *one point crossover*) и униформно укрштање (eng. *uniform crossover*). Код **једнопозиционог укрштања** насумично се бира једна позиција у бинарном запису која се назива **тачка укрштања** (енг. *crossover point*) и прво дете наслеђује све гене лево од тачке укрштања првог родитеља и све гене десно од тачке укрштања другог родитеља, а друго дете обратно. Уколико су родитељске јединке облика (**1 0** 1 1 0 0) и (0 0 **0 0 0 1**) и тачка укрштања је друга тачка слева, деца ће бити облика (**1 0 0 0 0 1**) и (0 0 1 1 0 0). Код **униформног укрштања**, прво дете наслеђује сваки бит првог родитеља с неком вероватноћом p (најчешће 0.5) и уколико га заиста наследи, друго дете наслеђује одговарајући бит другог родитеља. У супротном, прво дете ће наследити вредност тог бита од другог родитеља, а друго дете од првог. Ако су родитељске јединке облика (**1 0** 0 **0** 1 1 0 **1**) и (0 0 **1** 1 **1 0 1** 1) биће добијена деца (**1 0 1 0 1 0 1 1**) и (0 0 0 1 1 1 0 1) ако прво дете наслеђује први, други, четврти и осми бит од првог родитеља и трећи, пет и седми од другог, а друго дете супротно.

Поред оваквог укрштања, у пракси се јављају имплементације генетских алгоритама где дете може имати више од два родитеља. У такве типове укрштања спадају дијагонално и скенирајуће укрштање, која представљају уопштења једнопозиционог и униформног укрштања. **Дијагонално укрштање** се изводи над n родитеља и производи n деце тако што се коришћењем $n-1$ тачке укрштања сваки од n родитељских хромозома подели, на истим позицијама, на n делова. Након тога, прво дете наслеђује делове хромозома сваког од родитеља по замишљеној дијагонали, тако што од првог родитеља наслеђује први део хромозома (лево од прве тачке укрштања), од другог родитеља други део хромозома (између прве и друге тачке укрштања) и надаље све до последњег n-тог дела n-тог родитеља (десно од последње тачке укрштања). Друго дете наслеђује слично, само што се наслеђивање делова помера за по једног родитеља, тако да од другог родитеља наслеђује први део, од трећег родитеља други и надаље до последњег родитеља од кога наслеђује претпоследњи и првог родитеља од кога наслеђује последњи део и овакав поступак се наставља за свако дете. С друге стране, **скенирајуће укрштање** резултује креирањем једног детета, тако што се, за сваки бит, од n родитеља произвољно бира један чији ће одговарајући алел бити пренет детету и то

се назива униформно скенирање (енг. *uniform scanning*). Други начин је да се посматра који је алел најзаступљенији за дати ген међу свим родитељима и преузме та вредност и он се назива гласачко укрштање (енг. *voting crossover*) [13].

Уколико је репрезентација јединке низ реалних бројева, поред претходних могу да се користе и друга укрштања, попут линеарног (енг. *linear crossover*) [41], мешајућег (енг. *blend crossover*), унимодалног (енг. *unimodal normal distribution crossover, UNDX*) [50] или симулирајућег бинарног укрштања (енг. *simulated binary crossover*) [8]. Код пермутација, може се користити делимично (енг. *partially mapped crossover*) или уређено укрштање (енг. *ordered crossover*), а постоје и друга укрштања [2].



(a) Дијагонално укрштање три родитеља

Parent 1: 1111010111000110
Parent 2: 1100010101000010
Parent 3: 0011101010101011
Parent 4: 0101010101100100
Child : 1111010111000010

(b) Бинарно гласачко укрштање

Слика 2.1: Приказ укрштања са више родитеља [13]

Мутација

Главни разлог постојања мутације јесте да се пружи шанса проналаску решења за које, коришћењем једино оператора укрштања, не постоји никаква могућност да се пронађу. Пример у бинарној репрезентацији јесте уколико све јединке у популацији имају исту вредност неког гена (рецимо нулу) и тада, ма колико пута се вршило укрштање, не постоји могућност да се добије било која јединка која ће имати другу вредност тог гена (нпр. јединицу). За споменуту бинарну репрезентацију, најкоришћенији вид мутације је **комплементирање** (енг. *bit flip mutation*) вредности произвољног бита (или више

битова) са неком малом вероватноћом [36]. Одабрани бит се назива **тачка мутације** (енг. *mutation point*). За потребе бинарне репрезентације ово је довољно, али уколико се користе пермутације као репрезентација јединки, мутација не би смела да произвољно промени један број у неки други (на сличан начин као што се окреће бит) јер тада то више не би била валидна пермутација. У таквом случају могу се користити следећи оператори. Први је **оператор инверзије**, који окреће редослед неких узастопних гена у хромозому. На примеру пермутација целих бројева, инверзија бира подниз бројева и окреће га наопако, док све остале бројеве оставља нетакнутим. Уколико је јединка облика (1 2 **3 4 5 6** 7 8), и уколико се инверзија примењује на поднизу бројева од трећег до шестог, добија се јединка (1 2 **6 5 4 3** 7 8). Овај оператор је постојао у оригиналном генетском алгоритму Џона Холанда и представља специјалан случај оператора пермутације, који неки подскуп елемената слика у неку њихову произвољну пермутацију. Још један оператор који се користи је **оператор размене** (енг. *exchange*) која размењује позиције нека два гена. Уколико се над јединком облика (1 2 **3** 4 5 **6** 7 8) размењују трећи и шести број, добија се јединка (1 2 **6** 4 5 **3** 7 8) [36, 39]. **Премештајућа мутација** (енг. *displacement mutation*) премешта неку групу узастопних гена заједно на неку другу локацију, посматрајући је као један елемент. Уколико је јединка облика (1 **2 3 4** 5 6 7 8) и уколико треба преместити бројеве од два до четири на позицију између шестог и седмог броја, добија се јединка (1 5 6 **2 3 4** 7 8) [9]. Важно је напоменути да мутацију не треба вршити за сваку јединку јер се тиме може успорити или спречити процес конвергенције ка квалитетним решењима. Зато се задаје број између нула и један који представља вероватноћу мутације и она је најчешће веома мала (између 0.05 и 0.2).

2.3 Стратегије замене популације и услови за заустављање алгоритма

До сада се подразумевало да се једна популација креира од друге тако што се генетским операторима од јединки из скупа за репродукцију добија читава нова популација. Пошто се свака од тих популација назива генерацијом јединки, такав алгоритам се назива **генерацијским генетским алгоритмом** (енг. *generational genetic algorithm*). Број новокреираних јединки у новој у односу на стару генерацију назива се **генерацијским јазом** (енг. *generation*

gap) и он је код оваквих алгоритама једнак величини целе популације, умањен за величину елитизма. Међутим, код **генетских алгоритама стабилног стања** (енг. *steady state genetic algorithm*), генерацијски јаз је обично доста мали, односно свега неколико јединки се мења између две генерације. Код оваквог типа генетских алгоритама се неком од селекција бира мањи број родитељских јединки, чије потомство се затим убацује уместо мањег броја најлошијих јединки у популацији. Уместо најлошијих, могу се мењати и јединке из популације које су најсличније онима које су настале (тзв. *crowding* оператор). Овакав тип генетског алгоритма је погодан за проблеме код којих је битан квалитет читаве популације јединки, а не само једне најбоље [39].

Генетски алгоритам се у неком тренутку мора зауставити, али пошто се решава проблем за који најчешће нема (ефикасног) начина да се препозна оптимално решење, у општем случају не постоји јасан индикатор када треба прекинути извршавање. Један од најкоришћенијих услова за заустављање је ограничавање максималног броја итерација, након чега се прекида извршавање алгоритма и најквалитетнија јединка последње генерације враћа као решење. Недостатак овог услова јесте што није лако одабрати оптималан број итерација, а предност је једноставна имплементација. Поред овог, често се користи и прекид услед тога што се током неког броја итерација вредност најбоље јединке (или неколико најбољих јединки) није променила, што указује на то да је алгоритам пронашао неки оптимум. Са овим условом треба бити врло пажљив, јер постоји могућност да је тај оптимум заправо локални и да би се алгоритам после додатног времена из њега „ишчупао" и пронашао боље решење. Могуће је користити и оба ова услова истовремено, а постоје и други услови за заустављање [32].

2.4 Изазови при коришћењу генетских алгоритама

Одабир репрезентације јединки је вероватно и најважнији задатак при имплементирању генетског алгоритма за конкретан проблем, пошто ће од ње умногоме зависити и квалитет и брзина самог алгоритма. Низ битова не може да покрије читав спектар проблема који се могу решити генетским алгоритмом, па постоје и друге репрезентације. За неке алгоритме је могуће бирати између више репрезентација, а код неких је једна најприроднија. Могуће ре-

презентације су низови бинарних, природних или реалних бројева фиксне или произвољне дужине, стабла, пермутације бројева и друге [39, 33].

Пребрза или прерана конвергенција је главна препрека успешности генетског алгоритма. Начини да се она предупреди су у уској вези са неким од стратегија које се примењују у алгоритму. Већ је речено да је примена превелике величине елитизма нешто што може да доведе до овог проблема. Јасно је и да уколико величина турнира (ако се користи турнирска селекција) или свеукупна величина скупа јединки за репродукцију није довољно велика такође може доћи до преране конвергенције, а није свеједно ни који се тип селекције користи. Неки од механизама за сузбијање пребрзе конвергенције су превенција инцеста (енг. *incest prevention*), којом се онемогућава укрштање јединки које су исувише сличне, коришћење униформног укрштања и онемогућавање појављивања дупликата у популацији [15].

Малопре је споменут још један изазов генетских алгоритама, а то је одабир вредности параметара. Не само због преране конвергенције, већ и због временских перформанси, избор оптималних вредности параметара није нимало једноставан задатак. Најчешћи параметри генетског алгоритма су величина популације, величина скупа јединки за репродукцију, максималан број итерација, вероватноћа мутације, величина турнира и величина елитизма. Раније је једини начин за њихово подешавање било ручно тестирање, док су се у новије време појавили алати за аутоматску оптимизацију параметара [60].

Невалидне јединке (енг. *infeasible chromosomes*) су оне које крше неки услов који свако решење мора да испуњава и уколико се оне не третирају на прави начин, перформансе алгоритма могу бити драстично нарушене. Брига о таквим јединкама није увек једноставна. Једна стратегија је да се оне искључе из популације истог тренутка када се утврди да нису валидне. Друга је да се обележе као невалидне тако што ће им бити смањења (или повећана) вредност (неком казненом функцијом), како би оне убрзо (али не нужно одмах) биле елиминисане из популације. Трећи начин је поправљање таквих јединки тако да постану валидне [3].

2.5 Додатно о генетском алгоритму

Једно покретање генетског алгоритма није довољно за доношење закључка о његовом квалитету. Разлог лежи у природи генетског алгоритма који је сто-

хастички, односно заснива се на псеудослучајним бројевима, па вишеструко извршавање алгоритма за исти улаз може исходовати дијаметрално супротним резултатима. Стога је потребно више пута покретати алгоритам са различитим семеном (енг. *seed*) за генерисање псеудослучајних бројева и на основу резултата извући неке закључке. Насупрот томе, за тестирање различитих подешавања генетског алгоритма (нпр. да ли је боље користити турнирску или рулетску селекцију) неопходно је поставити исто семе, што је добро позната техника за смањење варијације под именом заједнички псеудослучајни бројеви (енг. *common random numbers*) [21].

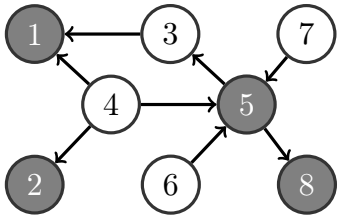
Једна стратегија која се користи у генетским алгоритмима јесте коришћење прилагодљивих параметара (енг. *adaptive parameters*). Уместо да су сви параметри унапред познати и фиксни, ова техника подразумева да они могу да мењају вредност током извршавања с циљем добијања бољих перформанси и решења. Пример употребе може бити повећање вероватноће мутације уколико је популација постала превише слична [28, 44].

Пошто је генетски алгоритам метахеуристика, не постоје никаква ограничења по питању програмске парадигме у којој га је потребно имплементирати. Најприроднија је императивна и у њој је написана већина имплементација, али могуће је написати је и у декларативној парадигми, па постоје имплементације у функционалној [45] и логичкој [37] и чак у парадигми програмирања ограничења [22]. Генетски алгоритам је могуће паралелизовати [6].

Глава 3

Опис решења

Пре описивања било чега, најпре треба увести једну важну напомену. Наиме, у овој глави ће мало другачије бити дефинисан појам доминираности чвора уведен на страни 2. Уколико у графу G постоји грана $(u, v) \in E$, тада је чвор u доминиран чвором v, што је различито у односу на ранију дефиницију када је важило обратно и за овако дефинисану доминираност тражиће се минимални доминирајући скуп. За неусмерене графове никакве разлике неће бити у односу на раније, али се разлика јавља код усмерених графова као у примеру 1.2, где ће се тражити (и добити) другачије решење. То се може решити тако што ће се, уместо на задатом графу G, радити на његовом транспонованом графу G', који се од G разликује једино по томе што ће смер сваке гране бити супротан у односу на смер гране у G. Та трансформација графа може се извршити у полиномском времену и надаље у овој глави ће бити подразумевано да је она већ извршена, а да се минимални доминирајући скуп тражи према измењеној дефиницији доминираности. На слици 3.1 се може видети да је у измењеном графу исто решење као и у примеру 1.2.



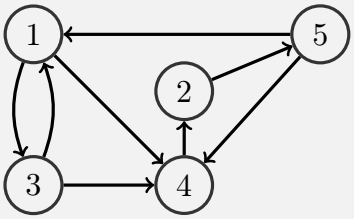
Слика 3.1: Пример минималног доминирајућег скупа на усмереном графу са измењеном дефиницијом доминираности

Прва ствар коју треба описати је начин на који је, за потребе генетског алгоритма у овом раду, граф представљен у програму. За дати позитиван цели број n, који означава број чворова, граф је одређен низом од n природних бројева. Сваки од тих бројева се, међутим, посматра у свом бинарном облику, на следећи начин — уколико i-ти број у том низу природних бројева на позицији j-тог бита (гледано слева) садржи јединицу, онда у графу постоји грана од чвора i ка чвору j, док у супротном те гране нема. То значи да i-ти број садржи информацију о томе ка којим чворовима i-ти чвор има грану, односно до којих чворова се може директно, преко једне гране, доћи из тог чвора[1]. Овим је, у суштини, креирана једна матрица повезаности, али имплицитно. Додатно, увек ће бити обезбеђено да i-ти број по реду на позицији i-тог бита садржи јединицу, јер се сматра да од неког чвора до тог истог чвора увек може да се дође. Сви бројеви који представљају граф ће се, дакле, налазити у интервалу $[1, 2^n)$ (по један бит за сваки чвор) и биће посматрани као n-битни бројеви, а први бит слева, у сваком од тих бројева, означаваће постојање гране ка првом чвору, други бит слева постојање гране ка другом чвору и тако редом, све до првог бита здесна, који ће означавати постојање гране ка n-том чвору. Овакав имплицитан запис графа је подесан за краћи запис одређених операција које ће се користити у алгоритму, а такође омогућава и лако генерисање различитих графова на којима је могуће вршити тестове.

Пример 1:
Нека је граф од пет чворова задат следећим бројевима: 22, 9, 22, 10 и 19. Посматрају се у бинарном облику:

1. чвор (број 22): (1 0 1 1 0), па постоје гране ка чворовима 3 и 4
2. чвор (број 9): (0 1 0 0 1), грана ка чвору 5
3. чвор (број 22): (1 0 1 1 0), гране ка чворовима 1 и 4
4. чвор (број 10): (0 1 0 1 0), грана ка чвору 2
5. чвор (број 19): (1 0 0 1 1), гране ка чворовима 1 и 4

[1]Пошто постоји оваква веза између чвора графа и броја који представља суседе тог чвора, надаље ће се, уколико се чвор спомиње у контексту неке нумеричке операције (битска конјункција или дисјункција), уместо чвора у њој подразумевати управо овај број.



Генерисање неког произвољног усмереног графа може да се изврши тако што се, ако граф има n чворова, псеудослучајно бира n природних бројева мањих од 2^n. Након тога, за i-ти број врши се битска дисјункција тог броја са бројем који се добије битским померањем јединице за $n-i-1$ места улево (рачуна се да индекси крећу од нуле), како би на месту i-тог бита имао јединицу и добија се коначан облик броја који означава суседе i-тог чвора.

Генерисање произвољног неусмереног графа захтева мало више посла. Поред онога што је обављено за неусмерену варијанту, треба обезбедити да ако постоји грана од чвора i ка чвору j, мора постојати грана и од чвора j ка чвору i. То ће бити обезбеђено тако што се за број на позицији j пронађу сви чворови који имају грану ка њему и онда у његовом бинарном запису додају јединице на одговарајуће позиције. Потребно је проверити да ли је битска конјункција i-тог броја и броја који се добије битским померањем јединице за $n-j-1$ места улево, различита од нуле. Уколико јесте, постоји грана од чвора i ка чвору j, па треба обезбедити да у бинарном запису j-тог броја i-ти бит има вредност један (грана од j ка i), односно треба извршити битску дисјункцију j-тог броја са помереном јединицом за $n-i-1$ места улево. Овај поступак се понавља за све парове (i, j), за $i, j \in [0, n-1]$.

Надаље, сем уколико није експлицитно наглашено другачије, када буде спомињан граф мислиће се искључиво на усмерену варијанту, будући да се неусмерени граф може посматрати као специјалан случај усмереног.

У пракси се, међутим, претходно описани вид генерисања графова ипак показао као недовољно добар. Разлог лежи у томе што се, за веће n, генеришу бројеви са великим бројем јединица у свом бинарном запису, односно густи графови, па је за $n = 5000$ у просеку излазни степен чворова 2500, а укупан број грана преко дванаест милиона. То доводи до тога да су чворови прилично исповезивани, па ће самим тим минимални доминирајући скупови бити доста мали (у просеку једноцифрени или једва двоцифрени за $n = 5000$). У таквој ситуацији је доста теже поредити различите алгоритме у погледу ква-

литета пронађених решења, будући да би и алгоритам који произвољно узима подскупове чворова могао после неког времена да пронађе прихватљиво решење.

Зато је потребно креирати граф на мало другачији начин. Уместо да се прво бирају бројеви, па онда гледа њихов бинарни запис, радиће се у супротном смеру. Наиме, прво ће, за сваки број појединачно, бити одређено колико ће највише[2] јединица имати у свом бинарном запису, а затим на којим местима ће се те јединице налазити. Оваквим начином креирања графа се може ограничити број грана по чвору, те ће самим тим граф имати мањи укупан број грана, што ће решити претходни проблем. Дакле, биће фиксиран неки мали интервал бројева и из њега биран број излазних суседа за сваки чвор. У пракси се као добра горња граница тог интервала показао квадратни корен броја чворова у графу. За $n = 5000$, то значи да ће сваки чвор имати максимално 71 чворова ка којима има грану, што доводи до просечног излазног степена од око 35. Након одређивања броја излазних суседа, псеудослучајно се бирају њихови индекси из интервала $[0, n-1]$ и врши битска дисјункција тренутног броја (на почетку ће бити једнак нули) са јединицом померeном за одговарајући број места улево. Овај поступак се затим понавља за сваки чвор у графу. Управо овако генерисани графови биће коришћени за тестирање.

За решавање проблема минималног доминирајућег скупа коришћен је генетски алгоритам имплементиран у програмском језику Пајтон. Проблем се решава проналажењем конкретног минималног доминирајућег скупа и израчунавањем његове кардиналности. У наставку следе детаљи везани за имплементацију.

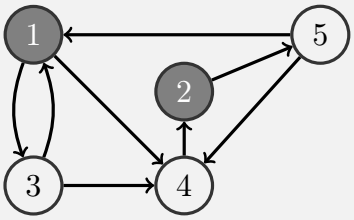
3.1 Репрезентација јединки и функција прилагођености

Прво питање које се поставља јесте шта јединке у генетском алгоритму треба да представљају. Будући да се решава проблем минималног доминирајућег скупа, односно проблем проналаска подскупова почетног скупа чворова (са неким посебним особинама), јединке представљају управо подскупове. Оне ће бити кодиране као бројеви из интервала $[1, 2^n)$ који се посматрају у

[2]Зарад бржег генерисања графова, неће бити тестирано да ли је нека јединица већ раније уписана на некој позицији.

бинарном облику на следећи начин — уколико је i-ти бит слева јединица, то одређује да је i-ти чвор укључен у подскуп, у супротном он није у подскупу[3].

Пример 2:
Нека је задат исти граф као у примеру 1. Јединка 24 = (1 1 0 0 0) означава подскуп који укључује први и други чвор. Овај подскуп јесте доминирајући скуп, а испоставља се да је и минималан.



Функција прилагођености сваке јединке је број јединица у њеном бинарном запису, а пошто је проблем проналаска минималног доминирајућег скупа проблем минимизације, јединке са мањим бројем јединица се сматрају квалитетнијим. Ово ће бити и функција циља — тражи се јединка која представља минимални доминирајући скуп, а то ће бити она са најмањим могућим бројем јединица која испуњава услов доминирајућег скупа.

Број јединица у бинарном запису броја може да се израчуна на више начина. Наиван приступ подразумева да се у петљи проверава редом један по један бит броја и ако је вредност бита један увећа бројач јединица. Пошто се врши провера за сваки бит, овај алгоритам захтева линеаран број (по броју битских цифара) битских операција и није практичан за велике бројеве. Боље решење представља алгоритам чији је аутор **Петер Вегнер**[4] (енг. Peter Wegner) [56]. Тај алгоритам у свакој итерацији петље елиминише текућу најдеснију јединицу (битском конјункцијом текућег броја и истог броја умањеног за један) док број не постане нула, па је укупан број итерација (самим тим и битских операција) сразмеран укупном броју јединица. Сем у случају када се бинарни запис броја састоји само од јединица, овај алгоритам ће увек бити бржи од наивног. Сви остали познати алгоритми су за велике бројеве непрактични (предубока рекурзија или недовољно меморије). Међутим, по-

[3]Слична репрезентација јединки, само експлицитна, користи се у [24].

[4]Чешће се као аутор наводи Брајан Керниган (енг. *Brian Kernighan*) који је овај метод споменуо 1988. у књизи *C Programming Language 2nd Ed.* чији су аутори он и Денис Ричи (енг. *Dennis Ritchie*).

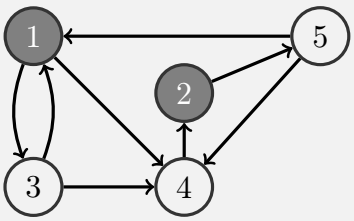
стоји начин да се још ефикасније израчуна број јединица сваке јединке, али о томе ће бити речи у секцији 3.7, након што се уведу додатне специфичности везане за имплементирани генетски алгоритам.

3.2 Поправка јединки

Уколико се не постави неко ограничење, генетски алгоритам ће, фаворизовањем јединки са мањим бројем јединица, конвергирати ка решењу које садржи све нуле, односно ка празном подскупу, а то није валидно решење проблема ни за један нетривијалан граф. Зато је неопходно обезбедити да популација садржи искључиво јединке које представљају доминирајуће скупове. Међутим, то не треба постићи одбацивањем јединки које нису доминирајући скупови јер су можда близу тога да то постану (можда је потребно укључити само неколико чворова у тренутни подскуп), стога ће све невалидне јединке бити поправљане тако да након поправке засигурно представљају доминирајуће скупове.

Јединка неће бити валидно решење, односно одговарајући подскуп неће бити доминирајући скуп, онда када постоји бар један недоминирани[5] чвор v (за чвор v се зна да је недоминирани ако је битска конјункција тог чвора и подскупа једнака нули). Тада је довољно проширити подскуп чвором v и наставити проверу за преостале чворове са тако проширеним подскупом. Када се заврши са провером последњег чвора, резултујући подскуп ће дефинитивно бити доминирајући скуп.

Пример 3:
Нека је задат исти граф као у примеру 1 и иста јединка као у примеру 2.



Утврђивање да ли је јединка 24 = (1 1 0 0 0) доминирајући скуп постиже

[5]У складу са измењеном дефиницијом доминираности с почетка поглавља, недоминирани чвор је онај који се не налази у доминирајућем скупу и чија свака излазна грана иде искључиво ка чворовима који се не налазе у доминирајућем скупу.

се тако што се за сваки чвор изврши битска конјункција чвора[a] и јединке и провери да ли је различита од нуле. За бројеве који представљају овај граф 22 = (1 0 1 1 0), 9 = (0 1 0 0 1), 22 = (1 0 1 1 0), 10 = (0 1 0 1 0), 19 = (1 0 0 1 1) и ову јединку важи следеће:

чвор 1: (1 0 1 1 0) & (1 1 0 0 0) = (1 0 0 0 0) = 16 $\neq$ 0

чвор 2: (0 1 0 0 1) & (1 1 0 0 0) = (0 1 0 0 0) = 8 $\neq$ 0

чвор 3: (1 0 1 1 0) & (1 1 0 0 0) = (1 0 0 0 0) = 16 $\neq$ 0

чвор 4: (0 1 0 1 0) & (1 1 0 0 0) = (0 1 0 0 0) = 8 $\neq$ 0

чвор 5: (1 0 0 1 1) & (1 1 0 0 0) = (1 0 0 0 0) = 16 $\neq$ 0

Може се приметити да резултат битске конјункције означава тачно оне чворове у подскупу ка којима чвор који се проверава има грану, па је тако за чвор 5 резултат (1 0 0 0 0), што значи да има грану ка чвору 1, који се налази у подскупу.
Уколико се проверава јединка која садржи трећи и пети чвор, 5 = (0 0 1 0 1), она неће представљати доминирајући скуп пошто за чвор 4 важи (0 1 0 1 0) & (0 0 1 0 1) = 0 и, заиста, тај чвор нема грану ни према 3 ни према 5. Тада је довољно у подскуп додати чвор 4, а то ће бити учињено са (0 0 1 0 1) | (0 0 0 1 0) = (0 0 1 1 1) = 7, па јединка након поправке има вредност 7.

[a] Већ је речено да се у оваквом контексту под чвором подразумева број који одређује суседе тог чвора, односно за чвор i то ће бити i-ти број у низу бројева којим је задат граф.

Остаје само одлука о томе којим редом треба проверавати чворове и по потреби проширивати подскуп. Јасно је да није потпуно свеједно којим редом се врши провера, јер, након што се подскуп прошири неким чвором, сви наредни чворови се проверавају у односу на нови подскуп, а не стари. Један приступ би био да се фиксира неки редослед чворова по којем се врши провера. Будући да ће функција поправке јединки бити позвана за сваку јединку, значило би да је функција што бржа. Приступ који се показао као задовољавајући, имена **chromosome_repair**, је да се чворови проверавају „с краја на крај", односно од последњег чвора ка првом или обратно.

Нешто бољи (али спорији) приступ, имена **better_chromosome_repair**, јесте да се распоред провере фиксира према улазном степену чворова. Они са већим улазним степеном ће, уколико буду убачени у решење, доминирати над више чворова него они са мањим улазним степеном, па ће у перспективи мање чворова бити укључено у решење. За ово је потребна сортирана листа чворова према улазном степену, а то је нешто што може да се креира на почетку алгоритма, јер остаје фиксирано за исти граф током трајања алгоритма. Овај приступ, на узорку од 100 псеудослучајно генерисаних графова са највише 2000 чворова, доводи до алгоритма који је спорији око две секунде у просеку од *chromosome_repair*, а даје просечно за пола чвора боље решење.

Још један приступ подразумева да се приликом сваке поправке генерише нека пермутација чворова и према њој бира редослед провере. Овај приступ је међутим трошио више времена (због генерисања пермутација), а није доводио до побољшања резултата, па је одбачен.

Проблем претходних начина је што за сваку јединку мора да се провери баш сваки чвор у графу. Нешто што би на први поглед могло да доведе до убрзања је да се из јединке извуче информација о чворовима који нису у доминирајућем скупу и да се само они провере (јер једино они могу да кваре решење), а да се не проверава сваки чвор. То извлачење, ипак, захтева превише времена и заправо доводи до неколико пута споријег извршавања функције поправке, па је ова идеја одбачена. Међутим, постоји начин да се ефикасно извуку само чворови који су можда недоминирани и тиме смањи број поређења, али ће, као претходно за бројање јединица, прво морати да се опишу додатни елементи генетског алгоритма.

3.3 Почетна популација

Креирање почетне популације је, наизглед, прост задатак. Најједноставнији начин генерисања почетних јединки јесте да се, без икакве стратегије или ограничења, псеудослучајно бирају из интервала $[1, 2^n)$, и након тога поправљају. Будући да се тиме у доброј мери покрива простор претраге, овај приступ би требало да доводи до задовољавајућих резултата. Међутим, овакав поступак ипак пати од бар два недостатка. Први је што је очекивано да овако генерисани бројеви имају доста јединица у свом бинарном запису, што може да успори алгоритам (брзина сваке итерације, па самим тим и брзина

целог алгоритма, поприлично зависи од тога колико јединке имају јединица). Друга мана је што ће, због великог броја јединица, бити потенцијално спорија и конвергенција резултата, те би самим тим и квалитет коначног решења могао да буде незадовољавајући (на том броју итерација), док би повећање броја итерација (зарад бољег резултата) довело до додатног успоравања алгоритма. И на крају, распоред јединица у оквиру сваког броја је одређен (имплицитно) псеудослучајно, што није нужно лоше, али и ту може да се доведе до унапређења.

Зато ће генерисање почетне популације бити мало усложњено. То ће бити учињено на два места — прво ће бити описан начин ограничења броја јединица сваке јединке, а затим како да бирамо позиције на којима ће се налазити јединице у оквиру сваке јединке.

Ограничење броја јединица

Прво што је важно нагласити јесте да, каквим год начином су јединке генерисане, оне ће морати одмах да прођу кроз функцију поправке, па самим тим неће бити егзактно ограничен број јединица, већ ће то бити учињено само до тренутка поправке. Међутим, очекивање је да ће, уз добар план, генерисане јединке и након поправке имати значајно мање јединица него што би имале псеудослучајним бирањем (и поправком тако генерисаних јединки). С друге стране, не сме се претерати у смањивању броја јединица, јер онда постоји реална опасност од конвергенције ка локалним оптимумима.

Бирање доброг горњег ограничења за број јединица није лако. Ограничавање константом може правити проблем за веће улазе, па је потребно користити неку комплекснију функцију. Треба одабрати функцију која расте спорије од n, јер је то имплицитно ограничење броја јединица када се псеудослучајно генеришу јединке. Оно што може бити идеја јесте да горње ограничење буде број јединица у доминирајућем скупу који се може брзо пронаћи, али који није претерано лоше решење (као када би био узет доминирајући скуп који обухвата све чворове графа). Овде ће у помоћ прискочити функција поправке јединки *better_chromosome_repair* која је уведена у претходној секцији. За њу је већ утврђено да након завршетка поправке гарантује да ће решење бити доминирајући скуп, па се то може искористити на следећи начин. Уколико се у функцију поправке пошаље празан скуп, тада ће она вратити скуп којем је додато минимално чворова (према тој функцији

поправке) да би постао доминирајући. Тиме је добијено решење које је боље од бар неких решења, па се кардиналност таквог скупа може узети као горње ограничење броја јединица у јединкама почетне популације. Са оваквим ограничењем, намера је да се, уз добро позиционирање јединица, бар нека јединка почетне популације гурне у део простора претраге близу неког квалитетног решења, чак и након примене функције поправке. Тестирањем на 180 псеудослучајно генерисаних графова са бројем чворова између 5000 и 15 000, оваквим начином ограничења броја јединица у јединкама почетне популације добија се за око два чвора боље решење него коришћењем псеудослучајно генерисане почетне популације, али је временска уштеда у просеку око десет секунди.

Бирање позиција јединица у оквиру једне јединке

Након одређивања границе остала је одлука о томе како ће јединице бити распоређене. Једноставан начин је да се позиције бирају псеудослучајно, одабиром бројева мањих од $n-1$ који ће означити битове на којима ће се налазити јединице. Овај начин бирања позиција јединица је најбржи.

Бољи начин се базира на идеји да приоритет имају чворови са већим улазним степеном, односно циљ је да такви чворови преовладају у почетној популацији, зато што је за њих вероватније да се налазе у минималном доминирајућем скупу. Стога, сваком чвору се додељује вероватноћа у складу са његовим улазним степеном. За добијање тих вероватноћа може се искористити **листа улазних суседа чворова**[6] која је већ имплицитно креирана за функцију поправке јединки на страни 22 (била је потребна како би се извршило сортирање чворова према улазном степену), па ту листу треба експлицитно запамтити и користити. Вероватноћа да ће чвор i бити одабран биће једнака:

$$p_i = \frac{1 + indeg(i)}{\sum_{j=0}^{n-1} 1 + indeg(j)}$$

где $indeg(i)$ означава улазни степен чвора i.

[6]То је листа која на позицији i садржи листу индекса свих чворова који имају грану према чвору i, а садржаће и сам индекс i. Листа на позицији i се у неусмереној варијанти графа назива затвореним суседством чвора i (енг. *closed neighborhood*), а у усмереној варијанти затвореним улазним суседством чвора i (енг. *closed in-neighborhood*) [7].

Бирање чвора који ће бити убачен у јединку према датим вероватноћама врши се слично као одабир јединке рулетском селекцијом на страни 8. Овај начин бирања позиција јединица је нешто спорији од претходног из разлога што није довољно само генерисање неког броја (као у претходном случају), већ је потребно пронаћи и позицију бита (чвор) коме одговара дати број. Проналазак траженог чвора може се убрзати коришћењем бинарне претраге. На узорку од 500 псеудослучајно генерисаних графова са бројем чворова између 500 и 10 000, други начин доводи до бржег алгоритма (иако је одређивање јединица спорије) просечно око једанаест секунди од потпуно псеудослучајног одабира чворова, док је број чворова у пронађеном решењу за око 17 чворова мањи него користећи претходни начин. Међутим, за велико n, уколико граф има много грана, време потребно за креирање листе улазних суседа чворова може да доведе до доста споријег извршавања него коришћењем псеудослучајног генерисања бројева. Ипак, креирана листа ће бити употребљена у наставку рада за додатно убрзавање алгоритма на другим местима, што ће и у таквим ситуацијама довести до свеукупно бржег извршавања алгоритма, па је због тога остављена.

Треба истаћи да ни у једном од ова два случаја није вршена провера да ли је претходно одабран чвор изнова одабран за исту јединку. Та провера је временски гушила оба начина, а није допринела претерано бољим резултатима, па је у потпуности изостављена из коначног решења.

3.4 Селекција

Две селекције које су узете у разматрање за генетски алгоритам у овом раду су турнирска и рулетска. Ипак, постоји један мањи проблем при коришћењу рулетске селекције који треба отклонити. Наиме, као што је познато, у овом генетском алгоритму важи да је јединка квалитетнија што има мање јединица, односно што јој је вредност функције прилагођености мања. Самим тим, не може се сматрати да је вероватноћа избора неке јединке количник њене вредности и суме свих вредности јединки као што је то наведено у делу о рулетској селекцији на страни 7, јер би онда лошије јединке имале веће вероватноће избора од бољих јединки. Зато је неопходно извршити трансформацију и направити помоћну листу која ће представљати исправне вероватноће јединки. Број k, који представља вредност неке јединке, биће пресликан у

$n - k + 1$, односно биће мапиран у број који представља удаљеност од максималне вредности јединке, што је n, уз увећање за један како би јединке за које је $k = n$ имале шансу да буду одабране (без сабирања са један би имале вероватноћу 0). Из овог пресликавања следи да, што је број k мањи, он ће бити удаљенији од n, па ће самим тим и $n - k + 1$ бити већи, односно одговарајућа јединка ће имати већу вероватноћу да буде одабрана. Надаље се са овако креираном листом ради као и у класичној рулетској селекцији.

Варијанта турнирске селекције која је прво разматрана је она код које се победником турнира сматра најквалитетнија јединка која на њему учествује, без ограничења на то колико пута се може бирати иста јединка (варијанта селекције са понављањем). У току тестирања је, међутим, у разматрање укључена и варијанта где се захтева да одабране јединке буду што је могуће више генетски различите. То значи да ће, уколико у популацији постоји m јединки са међусобно различитим генетским кодовима и уколико треба одабрати њих l, међу одабраним јединкама бити тачно $\min(m, l)$ генетски различитих јединки.

Резултати тестирања су показали да је рулетска селекција потпуно инфериорна у односу на обе турнирске (и по питању времена и по питању резултата), те је одмах искључена из даљег разматрања. С друге стране, поређење два типа турнирске селекције је било доста занимљивије. Турнирска селекција која је бринула о генетској разноликости постизала је солидно боље резултате од друге варијанте, али по цену више утрошеног времена, због неопходних провера да ли се тренутно разматрани генетски код већ јавља у скупу за репродукцију. На примеру 500 псеудослучајно генерисаних графова са 3500–7500 чворова, алгоритам који је користио тај тип турнирске селекције је проналазио, у просеку, за око 21 чвор боља (мања) решења, уз трошење двадесетак секунди више.

Закључци су другачији уколико се примени стратегија онемогућавања појављивања дупликата у популацији и уз њу користи обична турнирска селекција[7]. Поређењем овог приступа са коришћењем турнирске селекције са максималном генетском разноликошћу (без онемогућавања дупликата) уочава се да су решења сличне кардиналности за улазе до неколико хиљада чворова, али и да су овим приступом добијена солидно боља решења (неко-

[7]Уколико се онемогуће дупликати, коришћење турнирске селекције са максималном генетском разноликошћу не доприноси претерано бољим резултатима, али се троши више времена, па се у овој варијанти користила само обична турнирска селекција.

лико десетина чворова разлике) за графове веће од 10 000 чворова. Међутим, онемогућавање дупликата је временски захтевно и резултује споријим извршавањем за графове до (око) 1000 чворова, а на тим улазима може да доведе и до заглављивања алгоритма. Стога ће се у генетском алгоритму који је тема овог рада користити тунирска селекција са максималном генетском разноликошћу за графове до 1000 чворова, а за веће графове ће се користити обична турнирска селекција уз стратегију спречавања појављивања дупликата у популацији.

3.5 Укрштање, мутација и креирање нове генерације

Након што су одабране јединке за репродукцију, следи креирање нове генерације на основу њих. Из одабраних јединки извлаче се редом две по две које ће бити укрштане ради добијања два детета, која ће затим потенцијално бити мутирана, а онда убачена у скуп који представља нову генерацију и тај поступак ће се понављати док се не добије довољно нових јединки за читаву нову генерацију.

Варијанта укрштања која је одабрана за генетски алгоритам у овом раду је једнопозиционо укрштање. Псеудослучајно се бира тачка укрштања (између 1 и $n-1$) која представља број битова слева првог родитеља који се преноси првом детету, док се остатак добија од другог родитеља, копирањем дела његовог генетског кода десно од тачке укрштања. Аналогно се ради за друго дете, где тачка укрштања означава колико се битова слева копира од другог родитеља, а затим се остатак добија од првог родитеља. Креирање првог детета коришћењем битских оператора врши се на следећи начин:

```
child1 = (parent1 >> n - crossover_point) << n - crossover_point
    | parent2 & (1 << n - crossover_point) - 1
```

а аналогно овоме се прави и друго дете.

Једнопозиционо укрштање је одабрано пре свега јер се у алгоритму ради са потенцијално огромним бројевима, те је пожељно да број операција при укрштању буде што мањи. За једнопозиционо укрштање је увек потребан константан број операција по детету: три битска померања, четири одузимања и по једна битска конјункција и дисјункција. Насупрот томе, униформно укрштање захтева $\mathcal{O}(n)$ битских операција по детету.

Мутација се једноставно имплементира. Псеудослучајно се бира тачка мутације, тј. позиција бита (број између 0 и $n-1$) коме треба инвертовати вредност. Након тога се креира помоћни број битским померањем јединице за $n - mutation\_point - 1$ битова улево, односно добија се број који има јединицу само на месту мутације. Битском ексклузивном дисјункцијом почетног и помоћног броја добиће се број који ће на месту свих битова има исте вредности као почетни број, сем на месту тачке мутације, где ће бити инвертован бит у односу на почетни број. Код за мутацију се може видети испод:

```
value = random.random() # вредност између 0 и 1
if value < mutation_rate:
  mutation_point = random.randrange(n)
  helper_number = 1 << n - mutation_point - 1
  genetic_code = genetic_code ^ helper_number
```

3.6 Најбољи начин за поправку јединки

Након увођења појединости имплементираног генетског алгоритма, следи повратак на причу из поглавља 3.2. Главни проблем приступа који су раније наведени јесте што је за поправку сваке јединке увек неопходна провера свих чворова у графу. Најбољи начин за поправку јединки, назива **best_chromosome_repair**, базира се на томе да се искористи то што су јединке креиране на основу родитеља за које се зна да су доминирајући скупови. Пошто се користи једнопозиционо укрштање, дете је (скоро[8]) увек сличније једном од два родитеља. То значи да је разлика између детета и сличнијег родитеља мања од $\frac{n}{2}$ битова и ти битови су „покварили" дете које (потенцијално) није доминирајући скуп, у односу на сличнијег родитеља који то јесте. Штавише, позиције битова које су за то одговорне су оне код којих су вредности битова код детета и сличнијег родитеља различите, а, још прецизније, то су позиције битова код којих је јединица (код сличнијег родитеља) прешла у нулу (код детета) јер је тиме елиминисан чвор из решења који је пре избацивања потенцијално доминирао над неким чворовима изван доминирајућег скупа[9]. Сви чворови који су били доминирани избаченим чворовима, односно чворови који имају грану ка неком од избачених чворова

[8]Увек ће бити сличније једном родитељу, осим у случају када је n парно, а тачка укрштања тачно на пола, тада деца наслеђују исти број битова од оба родитеља.

[9]С друге стране, ако је нула прешла у јединицу то засигурно није покварило решење, већ га је само увећало.

(али и сами избачени чворови) су заправо они које треба проверити и који су потенцијално[10] постали недоминирани и тиме покварили јединку.

Позиције битова где јединица код родитеља прелази у нулу код детета могу се извући негацијом импликације[11], односно добија се број $removed\_nodes$ који има јединице само на тим позицијама. Након тога се може у петљи (која се извршава n пута) вршити битска конјункција $removed\_nodes$ са помереном јединицом за i места улево и уколико је различита од нуле забележити тај индекс (i је број текуће итерације петље). Међутим, постоји ефикаснији начин и заснива се на Вегнеровом методу за бројање јединица са стране 19. Као што је познато, овај метод пребројава јединице тако што у свакој итерацији петље уклања прву здесна и бележи колико пута се петља извршила. Овде се неће вршити пребројавање, али ће се користити то што ће се петља извршити онолико пута колико $removed\_nodes$ садржи јединица (што ће бити $\leq \frac{n}{2}$) и што ће се у свакој итерацији елиминисати по једна јединица. Остаје питање како закључити на којој позицији се налазила јединица елиминисана у текућој итерацији. Јасно је да је то прва здесна, али није познато њено место у бинарном запису. Међутим, ту може помоћи разлика добијена одузимањем броја који се добије након уклањања јединице од броја пре уклањања те јединице. Њихова разлика ће бити број који има само једну јединицу у бинарном запису (ту прву здесна која је елиминисана), па је довољно одрадити логаритам основе два над том разликом и добија се жељени индекс чвора. На примеру броја 20 = (1 0 1 0 0), уклањањем прве јединице здесна добија се број 16 = (1 0 0 0 0), разлика та два броја је број 4 = (0 0 1 0 0), а $log_2 4 = 2$, што и јесте позиција уклоњене јединице у бинарном запису броја 20 гледано здесна (почевши од нуле).

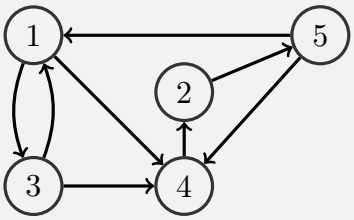
Након прикупљања избачених чворова, за сваки од њих је потребно извући информацију о свим чворовима графа који имају грану према том чвору. Структура која ће у томе помоћи је листа улазних суседа чворова, која је уведена у секцији 3.3. За сваки избачени чвор ће се из листе улазних суседа чворова прочитати који су му улазни суседи и тиме ће се имплицитно креирати листа која се назива **листа потенцијално недоминираних чворова**. Уколико је извршена мутација над неким чвором, делом те листе сматраће се и његови улазни суседи. Након тога, за дете је довољно проверити само

[10] Не зна се да ли су сигурно недоминирани јер можда неки имају грану ка неком другом чвору који је остао у доминирајућем скупу (или је у њега убачен).

[11] $\neg(p \implies q)$ је исто што и $\neg(\neg p \vee q)$, односно $p \wedge \neg q$.

чворове у тој листи.

Пример 4:
Поново је задат исти граф као у примеру 1.



Нека су родитељи јединке 25 = (1 1 0 0 1) и 23 = (1 0 1 1 1). Уколико је тачка укрштања други бит слева, тада се добијају деца 31 = (1 1 1 1 1) и 17 = (1 0 0 0 1). Посматра се друго дете (број 17). Пошто је тачка укрштања у првој половини јединке, закључује се да је јединка више битова наследила од првог родитеља.
1 1 | **0 0 1**
1 0 | 1 1 1
Затим се примењује негација импликације сличнијег родитеља и детета:
1 1 0 0 1
1 0 0 0 1
———
0 1 0 0 0 = *removed_nodes*
Уколико се примени Вегнеров метод, биће елиминисана најдеснија јединица броја (0 1 0 0 0) = 8 и добиће се 0. Разлика броја 8 и 0 је 8, а $\log_2 8 = 3$, те је чвор који је избачен четврти здесна, односно то је чвор два. Овим се добија да је једино други чвор избачен из детета у односу на родитеља, па су потребни сви чворови који имају грану усмерену према њему. То су чворови 2 (имплицитно) и 4. Након тога, довољно је само за ове чворове проверити да ли су доминираним неким чвором тренутног решења. Тиме је број провера са пет чворова, колико би било коришћењем метода *better_chromosome_repair*, смањен на два чвора.

Овако описан начин ипак доводи до функције поправљања јединки која је доста спорија од *better_chromosome_repair* на већим улазима. Разлог је у томе што се за такве улазе пречесто дешава да се више пута прове-

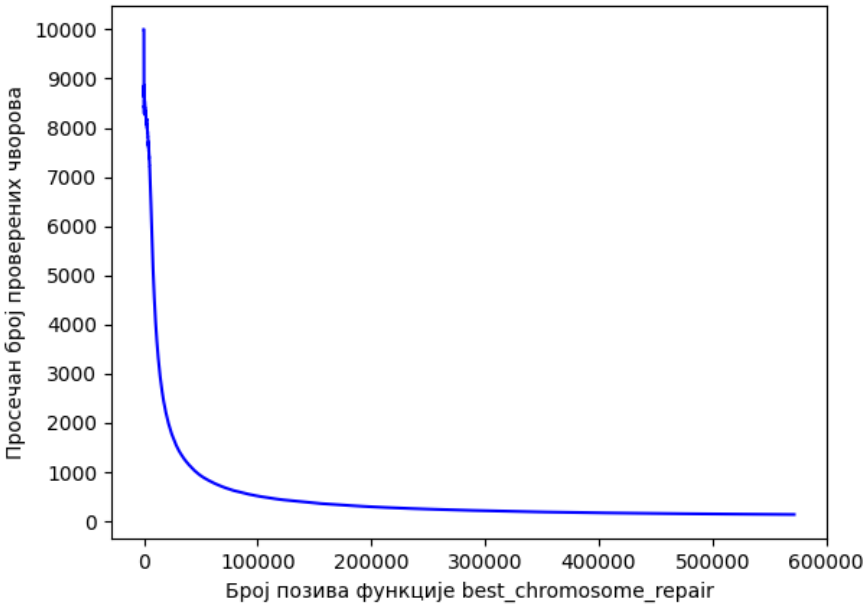
равају исти чворови[12], односно алгоритам је спорији ако је дужина листе потенцијално недоминираних чворова већа од n (број чворова који проверава *better_chromosome_repair*). Ипак, овај начин доста убрзава поправку јединки онда када је та дужина мања од n, па га не би требало олако одбацити. Зато ће *best_chromosome_repair* заправо представљати обједињавање два приступа — уколико је листа потенцијално недоминираних чворова дужа од n, биће коришћен *better_chromosome_repair*, а иначе ће се користити овај начин. Оно што се показује у пракси јесте да ће у најранијим итерацијама алгоритма чешће бити коришћен *better_chromosome_repair* јер су јединке тада доста различите, али да, како јединке конвергирају ка решењу и бивају све сличније, најпре долази до учесталије, а затим и искључиве примене другог начина, што у крајњој линији драстично убрзава алгоритам. Овде треба напоменути да се све јединке почетне популације поправљају методом *better_chromosome_repair*, будући да оне немају родитеље, па самим тим метод *best_chromosome_repair* не може да се користи.

Поређењем на 50 псеудослучајно генерисаних графова са бројем чворова између 2500 и 5000 алгоритам који користи *best_chromosome_repair* за поправљање чворова се просечно извршава око 305 секунди краће него онај који користи *better_chromosome_repair*. Тестирањем на графовима од по 10 000 чворова, утврђено је да се применом *best_chromosome_repair* у просеку за поправку једне јединке проверава око 130 чворова, уместо увек свих 10 000 колико би морало ранијим методом. На слици 3.2 се може видети пад просечног броја провера чворова током извршавања алгоритма.

3.7 Најбољи начин за бројање јединица у јединки

Пошто се у генетском алгоритму користи једнопозиционо укрштање, идеја на којој се заснива најбољи начин за бројање јединица је следећи — број јединица првог детета (пре мутације и поправке) једнак је збиру броја јединица првог родитеља до тачке укрштања и броја јединица другог родитеља од

[12]Неки чвор ће бити проверен више пута ако има гране ка више избачених чворова. Коришћење скупа овде неће бити од помоћи, пошто само креирање скупа потенцијално недоминираних чворова захтева време сразмерно величини листе потенцијално недоминираних чворова (јер се за сваки разматрани улаз из листе улазних суседа проверава да ли се већ налази у скупу), а онда је додатно потребно проверавање да ли су чворови скупа доминирани.



Слика 3.2: Приказ просечног броја провера чворова коришћењем метода *best_chromosome_repair* на примеру графа од 10 000 чворова

тачке укрштања (супротно за друго дете). За родитеље се број јединица лево и десно од тачке укрштања може ефикасно израчунати тако што се израчуна број јединица само у једном (мањем) делу Вегнеровим методом, а затим од укупног броја јединица тог родитеља (који је познат) одузме израчунати број и добије број јединица у другом (већем) делу. Коначан број јединица за прво дете биће збир броја јединица у левом делу првог родитеља и броја јединица у десном делу другог родитеља, док се за друго дете рачуна аналогно.

Пример 5:
Нека је задат произвољан граф са десет чворова и нека се у току генетског алгоритма у неком тренутку укрштају родитељи 819 = (1 1 0 0 1 1 0 0 1 1) и 405 = (0 1 1 0 0 1 0 1 0 1) и нека је тачка укрштања трећа тачка слева. Тада се добијају деца 789 = (1 1 0 0 0 1 0 1 0 1) и 435 = (0 1 1 0 1 1 0 0 1 1). Посматра се прво дете (789).
1 1 0 | 0 1 1 0 0 1 1
0 1 1 | **0 0 1 0 1 0 1**

Број јединица првог детета биће збир броја јединица првог родитеља лево од црте и броја јединица другог родитеља десно од црте. За првог родитеља издваја се број (1 1 0 0 0 0 0 0 0 0) битским померањем његовог генетског кода за $10-3=7$ места удесно па улево, а Вегнеровим методом, у две итерације петље, утврђује се да тај број има две јединице у бинарном запису, односно први родитељ има две јединице лево од тачке укрштања. Пошто он има укупно шест јединица, одузимањем двојке од шестице рачуна се број јединица десно од црте, што је четири. За другог родитеља се издваја број (0 1 1 0 0 0 0 0 0 0) и опет у две итерације петље израчунава да има две јединице у левом, а три јединице у десном делу. Самим тим, прво дете има $2+3=5$ јединица у свом бинарном запису, а друго дете $2+4=6$ јединица. Из овога следи да се петља извршила четири, уместо једанаест пута за рачунање броја јединица оба детета, а поред тога су била и додатна сабирања, одузимања и битска померања, али њихов број је увек константан.

Ако је m просечан број јединица свих родитељских јединки током извршавања алгоритма и уколико се претпостави да је расподела јединица у јединки равномерна, просечно ће се по родитељу пребројавати мање од $\frac{m}{2}$ јединица, пошто се оне броје само у једном од два дела. Будући да тачка укрштања има униформну расподелу на целом интервалу $[1, n]$, имаће ту расподелу и на подинтервалима, односно на тим деловима где ће се јединице пребројавати, те ће просечан број пребројаних јединица по родитељу бити $\frac{m}{4}$, па ће овај метод у току извршавања алгоритма пребројати око четири пута мање јединица од Вегнеровог метода.

Ово ипак није све, будући да претходно важи само за број јединица детета одмах након што је креирано. У временску анализу рачунања броја јединица овим начином морају се укључити и промене на јединки због мутација и због поправки јединке. Ако се мутација може и занемарити (једно потенцијално сабирање или одузимање), поправка јединке не може. Уколико се јединка због поправке доста пута промени додавањем новог чвора у решење, то ће морати да буде осликано и у броју јединица, који онда треба повећавати. Међутим, те јединице, које се морају додатно рачунати због поправке, су претходним методом (Вегнеровим методом је рачунат број јединица детета на крају, после завршене поправке) такође рачунате (јер се такође врши поправка која

додаје неке јединице), тако да је у односу на претходни метод и дефинитивно остварена уштеда у (скоро[13]) сваком случају. На примеру 100 псеудослучајно генерисаних графова са бројем чворова између 1000 и 3000, овим методом се током извршавања генетског алгоритма у просеку пребројавало 2,2 милиона јединица, док је претходни метод пребројавао чак 8,9 милиона, што је довело до просечне уштеде од скоро 17 секунди. Треба напоменути да се јединкама почетне популације број јединица мора рачунати Вегнеровим методом, зато што те јединке немају родитеље.

Постоји још један начин за убрзавање рачунања броја јединица, а то је коришћење кеширања помоћу речника. Због конвергенције алгоритма ка сличним решењима у каснијим итерацијама, неретко се дешава да нове јединке имају потпуно исти генетски код као нека ранија јединка. Како би се избегло поновно рачунање броја јединица за већ виђени генетски код, при првом сусрету са неким кодом он ће бити уписан као кључ у речник, а његов број јединица ће тада бити израчунат и уписан као вредност за тај кључ. Како речник не би превише растао, потребно је ограничити му величину, па је она постављена на број чворова у графу. Од неколико[14] тестираних стратегија кеширања најбоље се показала LRU (енг. *least recently used*) па је она одабрана. Тестирањем на графовима са по 10 000 чворова, кеширањем је алгоритам просечно убрзан за око 10% времена, а утрошак меморије је повећан за око 20% [4].

3.8 Локална претрага

Завршно побољшање презентованог генетског алгоритма биће примена локалне претраге (енг. *local search*). **Локална претрага** је хеуристички метод који побољшава решење тако што претражује његово непосредно окружење у циљу проналаска бољих решења [16]. У случају имплементираног генетског алгоритма, непосредно окружење неког броја (који представља решење) су сви бројеви који се добијају избацивањем тачно једне јединице из његовог бинарног записа. Другим речима, избацује се неки чвор из подскупа. До-

[13]Постоји један ивични случај када ће претходни метод бити (незнатно) бржи (разлика ће бити само у константном броју сабирања, одузимања и битских померања), док је у свим осталим случајевима нови метод бржи.

[14]Поред LRU, тестиране су и стратегије MRU (енг. *most recently used*) и LFU (енг. *least frequently used*).

бијени број ће бити боље решење од полазног само ако представља исправно решење, односно ако представља доминирајући скуп. Због тога је потребно да се, након избацивања сваког чвора, провери да ли је решење валидно и ако јесте треба га поставити за текуће и претраживати даље његову околину, а уколико није, тај чвор остаје у решењу и провера се наставља за остале чворове у доминирајућем скупу.

Испитивање да ли је решење остало валидно након избацивања неког чвора v остварује се једноставно коришћењем листе улазних суседа чворова уведене на страни 24. Будући да су чворови који имају грану према v и сам v једини који су потенцијално постали недоминирани избацивањем тог чвора, довољно је проверити једино њих, а управо се они налазе у листи улазних суседа чвора v. Уколико је j-ти чвор потенцијално недоминиран, утврђивање да ли је заиста недоминиран постиже се битском конјункцијом потенцијалног решења и j-тог броја у низу бројева којим је одређен граф. Подсећања ради, j-ти број у низу одређује који су то чворови према којима j-ти чвор има гране. Битска конјункција ће бити различита од нуле само онда када је у решењу бар један чвор ка коме j има грану, односно када је чвор j доминиран. Уколико је макар једна битска конјункција једнака нули, бар један чвор је недоминиран, па је решење невалидно и чвор v неће бити избачен.

Метод локалне претраге дефинитивно има смисла покренути на крају генетског алгоритма, како би пронађена решења била додатно побољшана. Тада се за сваку генетски различиту јединку може покренути локална претрага и проверити да ли је постала боље решење од тренутно најбољег. У пракси се добро показало покретање локалне претраге још два пута, одмах након прве итерације и након пола итерација. Премда постоји опасност од конвергенције ка локалном оптимуму (због природе локалне претраге) показало се да се у садејству са осталим техникама генетског алгоритма то не прави толики проблем, те примена овог метода у споменутим итерацијама доприноси бољим решењима, али по цену утрошеног времена. Тестирањем на око 100 графова величине од 5000 до 10 000 чворова, добијено је да генетски алгоритам применом локалне претраге у просеку налази за око 38 чворова боља решења, али је спорији за око 15 секунди.

3.9 Подешавање вредности параметара генетског алгоритма

До сада је генетски алгоритам у овом раду подразумевао следеће вредности параметара:

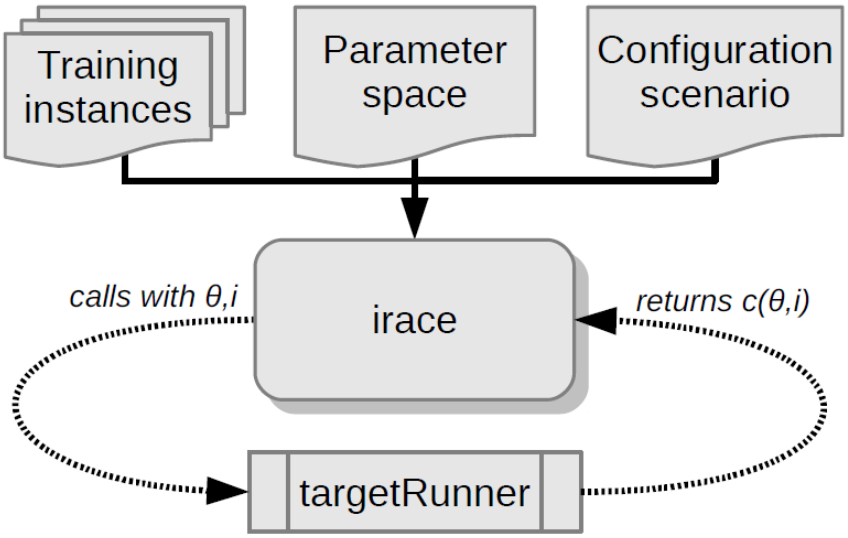
- број итерација = 1000;
- величина популације = 300;
- величина скупа за репродукцију = 100;
- величина турнира = 50;
- вероватноћа мутације = 0,3;
- величина елитизма = 14.

Ове вредности параметара су одабране ручним тестирањем и показале су се добро, али су ипак добијене тестирањем ограниченог броја графова и релативно малог броја комбинација вредности параметара и врло је могуће да постоје и боље комбинације. Такође, јасно је да исте вредности параметара немају исти утицај на прилично различите могуће улазе, јер алгоритам са малим бројем чворова и грана без потребе траје предуго, а алгоритам са доста великим бројем чворова и грана можда траје прекратко за добијање задовољавајућег решења. Први проблем биће решен коришћењем пакета irace, а други коришћењем интерполације помоћу радијалне базне функције.

Пакет irace

Пакет **irace** [35] написан је у програмском језику *R* и он имплементира метод итеративне трке (енг. *iterated race method*). Главни задатак тог метода је аутоматско подешавање параметара за различите класе алгоритама, а посебно је користан управо за генетске алгоритме. Сам метод елиминише лоша подешавања онда када „изгубе трку", односно када се утврди да је нека конфигурација параметара засигурно лошија од бар једне конфигурације. Читав процес се понавља у више итерација, а на крају резултат представљају најбоље пронађене конфигурације, према неким статистичким мерама. Више о самом методу се може прочитати у [35]. За потребе подешавања вредности

параметара имплементираног генетског алгоритма биће коришћена варијанта пакета irace прилагођена за Пајтон која се може пронаћи на [25].



Слика 3.3: Општа шема рада са пакетом irace [34]

Да би се користио овај пакет, биће потребно направити Пајтон скрипт са неколико елемената са слике 3.3. Инстанце за тренирање су различити графови над којима ће бити тестиран (са различитим параметрима) имплементирани генетски алгоритам, што значи да ће оне бити задате као низ низова бројева. Тај низ низова ће бити записан у засебној датотеци и онда учитан пре тестирања. Затим, неопходно је описати параметре генетског алгоритма који ће бити подешавани, а најважније ставке описа су име, тип и интервал у коме треба тражити најбоље вредности параметара и то чини простор параметара. Потребно је задати и сценарио конфигурације где се наводе разна подешавања самог метода итеративне трке, попут максималног броја итерација, броја цифара иза децималне запете или паралелизације. На крају је потребно дефинисање функције *targetRunner*, у којој се описује логика тестирања. Та функција ће бити позвана за сваку генерисану комбинацију конфигурације параметара θ и инстанце i и треба да врати оцену квалитета $c(\theta, i)$. У случају имплементираног генетског алгоритма, оцена ће бити кардиналност пронађеног минималног доминирајућег скупа. Поред ових, могуће је подесити и додатне ставке алгоритма, а најзначајније су почетна конфигурација параметара и временско ограничење извршавања генетског алгоритма [34].

Интерполација помоћу радијалне базне функције

Након покретања Пајтон скрипта са свим потребним елементима из претходног поглавља, пакет irace ће пронаћи конфигурације које сматра најбољим. Избором једне од тих конфигурација ту може бити крај посла, међутим, као што је већ истакнуто, циљ је да се конфигурација прилагоди улазу. То ће бити учињено на следећи начин. Прво ће, за фиксиран природни број n_1, бити генерисано m (узето је $m = 100$) графова са тачно n_1 чворова који ће бити прослеђени као инстанце методу итеративне трке пакета irace. Након завршетка метода, најбоља пронађена конфигурација ће бити забележена као конфигурација за графове величине n_1. Тај поступак се надаље понавља за величине графова $n_2, n_3, \ldots, n_k$, где се на крају добија k најбољих конфигурација за графове величине n_i, за $i \in [1, k]$. Сада може да се, за сваки подешавани параметар, изврши интерполација, односно пронађе функција која најбоље апроксимира замишљену функцију промене вредности параметра, а која пролази кроз k тачака облика *(величина графа, вредност параметра)* које су пронађене методом итеративне трке. Циљ је да се, за графове оних величина за које није тражена најбоља конфигурација, она може апроксимирати, тако што ће бити апроксимиране вредности појединачних параметара.

Интерполација која је одабрана је **интерполација помоћу радијалне базне функције**. Радијална базна функција је било која функција чија вредност у некој тачки зависи искључиво од растојања између те тачке и неке фиксиране тачке, која се назива центар[15]. Другим речима, радијалне базне функције су оне које су радијално (ротационо) симетричне у односу на центар. Идеја интерполације јесте да се пронађу коефицијенти λ_i који решавају систем једначина облика:

$$f(x_i) = \lambda_1\phi(x_i, x_1) + \lambda_2\phi(x_i, x_2) + \ldots + \lambda_k\phi(x_i, x_k),\ i \in [1, k] \tag{3.1}$$

где су $(x_i,\ f(x_i))$ тачке на основу којих се врши интерполација, а ϕ нека радијална базна функција. У свакој од ових једначина x_i представља центар, па је $\phi(x_i, x_j)$ заправо вредност у тачки x_j радијалне базне функције са центром у x_i. Након што се пронађу конкретне вредности λ_i, може се вршити процена вредности функције за неку тачку p тако што се у формули (3.1) (са постављеним конкретним коефицијентима λ_i), замени x_i за p, односно врши се апроксимација $f(p)$. Детаљније о самом методу интерполације, као и које

[15]Најпознатији пример такве функције је Гаусова функција.

је ограничење за грешку и под којим условима оно важи, може се прочитати у [5].

Имплементација ове интерполације у Пајтону може се наћи у оквиру пакета *SciPy* и потпакета interpolate где је имплементирана класа под именом *RBFInterpolator* [49]. Приликом креирања објекта те класе потребно је, у конструктору, проследити листу од k вредности x координата (то су у овом случају величине графова), као и листу од k листа, где свака од тих листа редом представља утврђене вредности сваког параметра за одговарајућу величину графа. Након тога, тај објекат треба позвати као функцију којој се прослеђује величина графа n и биће враћена процењена вредност свих параметара и те вредности ће бити коришћене у генетском алгоритму када је величина улаза n. Овде је потребно пазити на максималну утрошену количину меморије, која ће расти са порастом броја чворова у графу и броја јединки у алгоритму.

У пракси је на два места извршена „насилна" интервенција над резултатима који су добијени аутоматским подешавањем параметара пакетом irace и интерполацијом. Због пребрзог раста за веће улазе, за величине графова од 10 000 чворова одлучено је да се за њих постави број итерација на 3000, величина популације на 800, величина скупа за репродукцију на 100, величина турнира на 50 и величина елитизма на 14 и са таквим параметрима је број 10 000 као вредност x координате убачен у интерполацију. И након тога је интерполација, за још веће улазе, доводила до огромних параметара, па су они коначно ограничени, те ни за један улаз неће прећи 10 000 итерација, 4000 јединки у популацији, 1000 јединки у скупу за репродукцију, 400 јединки које учествују на турниру и 100 елитистичких јединки.

Глава 4

Поређење са егзактним алгоритмима

Алгоритми са којима ће бити поређен имплементирани генетски алгоритам су алгоритам грубе силе, као најједноставнији егзактни алгоритам и алгоритам [51] чији су аутори Јохан ван Рој (енг. *Johan van Rooij*) и Ханс Бодландер (енг. *Hans Bodlaender*), као један од најбољих алгоритама из литературе за егзактно решавање проблема минималног доминирајућег скупа. У наставку ће најпре бити описани споменути алгоритми, а затим ће бити приказано поређење перформанси.

4.1 Алгоритам грубе силе

Алгоритам грубе силе (енг. *brute force*) или алгоритам исцрпне претраге (енг. *exhaustive search*) представља основну алгоритамску стратегију за решавање оптимизационих проблема [40]. Заснива се на идеји да се решење може пронаћи тако што се провери свако теоретски могуће решење проблема и одабере оно које је најбоље. Овим начином се недвосмислено увек долази до оптималног решења проблема, али је недостатак тај што је, за веће улазе, потребно превише времена за извршавање.

На примеру проблема минималног доминирајућег скупа, алгоритам грубе силе ће за сваки подскуп чворова графа проверити да ли је такав подскуп доминирајући скуп и, уколико јесте, даље проверити да ли је његова кардиналност мања од кардиналности тренутно најбољег решења и, ако и то важи, забележити тренутни подскуп као најбољи. У складу са уведеним репрезен-

тацијама графа и решења на странама 16 и 18 и измењеном дефиницијом доминираности са стране 15, потребно је генерисати бројеве из интервала $[1, 2^n)$ који представљају подскупове чворова, сваком од њих израчунати број јединица у бинарном запису и проверити да ли је доминирајући скуп, битском конјункцијом тренутног броја са сваким чвором у графу. Време потребно за проверу исправности једног решења је $\mathcal{O}(n)$ битских операција, а провера ће се, у најгорем случају, покретати за $2^n - 1 = \mathcal{O}(2^n)$ подскупова, те је то укупно $\mathcal{O}(n2^n)$. Петља која рачуна број јединица у бинарном запису ће се, уколико се примењује Вегнеров метод, извршити тачно онолико пута колико тренутни број садржи јединица. У интервалу $[1, 2^n]$ има тачно $\binom{n}{1}$ бројева са једном јединицом, $\binom{n}{2}$ бројева са две јединице и надаље до $\binom{n}{n}$ бројева са n јединица. Самим тим, рачунање броја јединица биће сума свих биномних коефицијената за број n, а та сума има вредност[1] $2^n - 1 = \mathcal{O}(2^n)$. Укупно време оваквог алгоритма грубе силе је стога $\mathcal{O}(n2^n) + \mathcal{O}(2^n) = \mathcal{O}(n2^n)$ битских операција.

Другачији алгоритам грубе силе је онај који има паметнији редослед провере да ли је подскуп доминирајући скуп. Конкретно, будући да се тражи минимални доминирајући скуп, боље је кренути од подскупова са мањим бројем чворова и њих проверавати, а затим ићи ка већим и стати чим се пронађе прво решење, будући да свако наредно решење сигурно неће имати мање чворова од тренутног. Редослед којим ће се проверавати подскупови биће одређен комбинацијама чворова без понављања, чије ће се дужине итеративно увећавати по потреби. Наиме, прво ће бити генерисане све комбинације дужине један, односно подскупови које треба проверити биће дужине један и биће проверен сваки такав подскуп. Након тога, ако није пронађено решење, проверавају се све комбинације чворова дужине два, па три и тако даље, све до n у најгорем случају[2]. У Пајтону постоји пакет *itertools* [26], у оквиру кога се налази функција *combinations*, која прихвата произвољну колекцију елемената и дужину комбинације и враћа све комбинације елемената без понављања те дужине из задате колекције. Ипак, могуће је и ефикасније генерисати комбинације. За дужину комбинације k неопходно је генерисати почетну комбинацију, ону код које је k јединица поређано с десне стране (нпр. (0 0 1 1 1) за $n = 5$ и $k = 3$), а свака наредна комбинација те дужине се генерише на основу претходне у

[1] Следи из биномне теореме за $\sum_{k=0}^{n} \binom{n}{k} = \sum_{k=0}^{n} \binom{n}{k} 1^{n-k} 1^k = (1+1)^n = 2^n$, а јединица се одузима јер се у суму не укључује $\binom{n}{0} = 1$.

[2] Најгори случај ће бити онда када нема грана између чворова и када је минимални доминирајући скуп заправо скуп свих чворова.

времену $\mathcal{O}(1)$ битских операција коришћењем метода описаног у [55] за генерисање наредног броја са истим бројем јединица у бинарном запису. Овакав алгоритам се показао бржим у односу на претходни па ће он бити тестиран као алгоритам грубе силе.

4.2 Ван Пој-Бодландеров алгоритам

Ван Пој-Бодландеров алгоритам заснива се на алгоритамској стратегији званој гранај и редукуј (енг. *branch and reduce*) [51]. Сваки алгоритам заснован на тој стратегији је сачињен од редукционих правила, правила гранања и опционе процедуре која одређује које правило гранања (када их има више) треба применити на задатој инстанци проблема[3] који се решава. Редукциона правила служе да трансформишу текућу у еквивалентну мању инстанцу (у полиномском времену), док свако правило гранања генерише низ мањих инстанци (на основу тренутне) над којима се рекурзивно примењује исти алгоритам, а решење за текућу инстанцу се израчунава на основу решења враћених из рекурзије (опет у полиномском времену). Идеја је да се прво примењују редукциона правила све док је то могуће, а када то више не може, онда се примењује једно од правила гранања и проблем се рекурзивно решава.

Оно што важи за овакве алгоритме јесте да је временска анализа праволинијска. Користи се нека природна мера која описује величину (тежину) инстанце, као што је, за графовске проблеме, број чворова или грана у графу, и посматра се како се она, у најгорем случају, смањује током примене редукционих и правила гранања. Ипак, испоставља се да анализа са таквом мером не израчунава довољно тесну горњу границу времена извршавања, већ оставља доста „празног простора", те је због тога је настала техника **измери па владај** (енг. *measure and conquer*) за анализу потребног времена оваквих алгоритама. Основни принцип ове технике јесте коришћење неке пажљивије одабране нетривијалне мере тежине инстанце која ће боље захватити понашања алгоритма (у односу на обичну меру) и стога довести до доста боље временске анализе алгоритма у најгорем случају [18]. Додатна предност ове

[3]Инстанца неког проблема је конкретан проблем који треба решити. За пример проблема минималног доминирајућег скупа, фиксирање конкретног графа, односно задавање конкретног скупа чворова и грана, за који треба одредити кардиналност минималног доминирајућег скупа чини инстанцу проблема минималног доминирајућег скупа.

анализе је што се уз њену помоћ може проширити алгоритам неким новим редукционим правилима која можда раније нису обећавала и тиме сам алгоритам додатно убрзати.

Пре описивања мере која ће бити коришћена за Ван Роj-Бодландеров алгоритам, важно је напоменути да овај алгоритам заправо не решава проблем минималног доминирајућег скупа, већ решава сродни проблем минималног покривача скупа, на који је могуће свести изворни проблем у полиномском времену. У проблему минималног покривача скупа имамо задату колекцију скупова (мултискуп) $\mathcal{M}$ који садрже елементе из универзума $\mathcal{U}$, тако да сваки елемент из универзума мора бити садржан у бар једном скупу, па важи $\mathcal{U} = \bigcup_{S \in \mathcal{M}} S$ и двојка $(\mathcal{M}, \mathcal{U})$ чини инстанцу проблема. Пошто је универзум једнак унији свих елемената који се јављају у мултискупу, ово се може и мало другачије формулисати и каже да скупови садрже произвољне елементе, а универзум се дефинише као унија свих скупова из мултискупа. За подскуп $\mathcal{C} \subseteq \mathcal{M}$ се каже да је покривач $\mathcal{U}$ ако важи $\bigcup_{S \in \mathcal{C}} S = \mathcal{U}$, а проблем минималног покривача скупа се дефинише као одређивање покривача најмање кардиналности. На примеру мултискупа $\mathcal{M} = \{\{a, b, c\}, \{a, b\}, \{d, e\}, \{a, d\}, \{a, b\}\}$, важи да је $\mathcal{U} = \{a, b, c, d, e\}$, а минимални покривач скупа је скуп $C = \{\{a, b, c\}, \{d, e\}\}$. За елементе из универзума се дефинишу две функције, прва је скуп скупова елемента $S(e)$ која враћа све скупове у којима се јавља елемент e, а друга је фреквенција елемента $f(e)$ која представља број појављивања елемента e у мултискупу, односно представља кардиналност $S(e)$. У случају претходног примера, за елемент d важи $S(d) = \{\{d, e\}, \{a, d\}\}$, док је $f(d) = 2$. Редукција проблема минималног доминирајућег скупа на проблем минималног покривача скупа се може извршити тако што се за сваки чвор v графа креира по један скуп, а елементи тог скупа биће v и сви чворови до којих постоји грана из v [51]. Овом редукцијом добија се инстанца проблема минималног покривача скупа са мултискупом који садржи n скупова и универзумом који садржи n елемената.

Мера k која се користи у Ван Роj-Бодландеровом алгоритму је следећа:

$$k = \sum_{e \in \mathcal{U}} v(f(e)) + \sum_{S \in \mathcal{M}} w(|S|), \quad v, w : \mathbb{N} \to \mathbb{R}_+ \tag{4.1}$$

где су v и w тренутно непознате тежинске функције, $f(e)$ фреквенција елемента, а $|S|$ кардиналност скупа S.

Претходно споменута анализа оваквог типа алгоритама врши се на наредни начин. За инстанцу проблема коју је могуће директно (редукцијама) решити у полиномском времену каже се да је базна инстанца. Циљ анализе је израчунавање горњег ограничења броја базних инстанци које ће бити генерисане алгоритмом за инстанцу величине k и тај број се обележава са $P(k)$, док ће трајање алгоритма онда бити $\mathcal{O}^*(P(k))$. За функцију $f(n)$ важи $f(n) \in \mathcal{O}^*(g(n))$ ако $f(n) \in \mathcal{O}(g(n)p(n))$ за неки полином $p(n)$ и оваква нотација служи за потискивање полиномских фактора ради простијег записа. Ако је $\mathcal{P}$ базна инстанца, за њу важи $P(k) = 1$. У супротном, применом једног правила гранања добија се низ мањих инстанци $\mathcal{P}_1, \ldots, \mathcal{P}_p$ које редом имају величине $k_1, \ldots, k_p$ које су све мање од k. Тада се добија рекурентна релација:

$$P(k) \leq \sum_{i=1}^{p} P(k_i) \tag{4.2}$$

Пошто треба узети у обзир све могуће ситуације у којима настаје гранање, односно треба обухватити све инстанце величине k (за две различите инстанце величине k неће се добити исте мање инстанце, односно неће се добити исти бројеви $k_1, \ldots, k_p$), за једно правило гранања добија се низ[4] рекурентних релација облика (4.2). Додатно, ако постоји више правила гранања, овакви низови ће се добити за свако правило. Решење за овај низ рекурентних релација има облик α^k за неко $\alpha > 1$. При коришћењу мере (4.1) може се ићи још даље. Нека су $v_{\max} = \max_{n\in\mathbb{N}} v(n)$ и $w_{\max} = \max_{n\in\mathbb{N}} w(n)$ коначни бројеви. Тада важи следеће:

$$\begin{aligned} k &= \sum_{e\in\mathcal{U}} v(f(e)) + \sum_{S\in\mathcal{M}} w(|S|) \leq \sum_{e\in\mathcal{U}} v_{\max} + \sum_{S\in\mathcal{M}} w_{\max} \\ &= nv_{\max} + nw_{\max} = (v_{\max} + w_{\max})n \end{aligned}$$

Претпоследња једнакост важи због тога што је и број елемената и број скупова у мултискупу n, што следи из редукције са стране 43. Због закључка да је $k \leq (v_{\max} + w_{\max})n$, важи $\alpha^k \leq \alpha^{(v_{\max}+w_{\max})n}$, па коначно решење може да се ограничи са $\alpha^{(v_{\max}+w_{\max})n}$ и онда је сложеност $\mathcal{O}^*(\alpha^{(v_{\max}+w_{\max})n})$. Остаје проблем одређивања идеалних тежинских функција v и w које ће минимизовати $\alpha^{v_{\max}+w_{\max}}$. Под одређеним претпоставкама за те функције (једна од њих је да су обе функције монотоно неопадајуће) проблем минимизације $\alpha^{v_{\max}+w_{\max}}$

[4]Како би овај низ постао коначан, уводе се нека додатна ограничења.

постаје тзв. квазиконвексни оптимизациони проблем и он се може решити у рачунару [14]. Тиме се добија коначан облик мере (4.1) [18, 51].

Ван Рој-Бодландеров алгоритам састоји се од шест редукционих правила и једног правила гранања. Први корак алгоритма је провера да ли је мултискуп празан и, ако јесте, треба вратити празан скуп. Затим се примењују редукциона правила и то редом:

- *Правило јединственог елемента*: Уколико постоји неки елемент $e \in \mathcal{U}$ фреквенције један, то значи да је његово једино појављивање у неком скупу $R \in \mathcal{M}$ и једини начин да се тај елемент покрије јесте да се R укључи у покривач, па то треба учинити. Надаље се R више неће разматрати, као ни сви елементи који се у њему налазе, будући да су постали покривени и треба их избацити из свих осталих скупова у којима се налазе.

- *Правило подскупа*: Ако постоје скупови $R, Q \in \mathcal{M}$ такви да је $R \subseteq Q$, тада R може одмах бити искључен из даљег разматрања будући да све елементе које R покрива, покрива и Q, те R сигурно неће бити део минималног покривача. Овим правилом и правилом јединственог елемента, из инстанце се избацују сви скупови кардиналности један.

- *Правило подређивања* (енг. *subsumption rule*): За елемент $e_1 \in \mathcal{U}$ се каже да је подређен елементу $e_2 \in \mathcal{U}$ ако се у сваком скупу у ком се налази e_1 налази и e_2. То ће бити испуњено ако важи $S(e_1) \subseteq S(e_2)$, где је $S(e)$ скуп скупова елемента e. Пошто e_1 мора бити покривен минималним покривачем, укључивањем било ког скупа у ком се он налази биће покривен и e_2, па самим тим e_2 нема значај и може да се избаци из свих скупова у којима се налази.

- *Правило бројања*: Нека у неком скупу $R \in \mathcal{M}$ постоји r_2 елемената фреквенције два. Тада, због правила подређивања, сваки од тих елемената своје друго појављивање има у различитом скупу[5], те постоји r_2 скупова који садрже тачно један елемент из R и неке друге елементе који

[5]Уколико би нека два елемента e_1 и e_2 фреквенције два из R била заједно у неком другом скупу R', онда би за њих морало да важи $S(e_1) = \{R, R'\}$, $S(e_2) = \{R, R'\}$, односно важи $S(e_1) \subseteq S(e_2)$ и e_2 би био елиминисан према правилу подређивања. Исти закључак важи и уколико је e_1 фреквенције два, а e_2 фреквенције веће од два.

нису у R. Нека је q укупан број елемената који се налазе у тих r_2 скупова, не рачунајући оне који се налазе у R. Даље се посматрају два случаја. Уколико се R узме у покривач, биће покривено $|R|$ елемената коришћењем једног скупа, а додатних q елемената (из тих r_2 скупова) се покрива убацивањем једног по једног скупа који садржи неки од тих q елемената у покривач, па ће свеукупно бити покривено $q+|R|$ елемената са највише $q+1$ скупова[6]. С друге стране, уколико се R одбаци, споменутих r_2 скупова ће по правилу јединственог елемента бити убачено у покривач и биће покривено $q+|R|$ елемената са тих r_2 скупова. Из ова два случаја следи да, уколико је $q+1 \leq r_2$, односно уколико је $q < r_2$, тада укључивање R у покривач сигурно није лошији избор него неукључивање (јер се користи мање или једнако скупова за покривање истог броја елемената), па се R убацује у покривач. Пример када би ово правило било примењено је ако би постојао скуп $R = \{e_1, e_2, e_3\}$ и скупови $\{e_1, e_4\}, \{e_2, e_4\}, \{e_3, e_4\}$, а важило $f(e_1) = f(e_2) = f(e_3) = 2$ и $f(e_4) = 3$. Ако би R био додат у покривач, били би покривени елементи e_1, e_2 и e_3, док би за покривање e_4 било потребно укључивање једног од преостала три скупа, што значи да би покривач за ове елементе у првом случају имао кардиналност два. Ако R не би био додат у покривач, било би потребно узети сва три преостала скупа за покривање свих елемената, што је више него малопре. Заиста, $q = 1$ (само елемент e_4) и то је мање од $r_2 = 3$, те дефинитивно треба убацити R у покривач.

- *Правило елиминисања скупова величине два сачињених само од елемената фреквенције два*: Нека постоји скуп $R \in \mathcal{M}$, $R = \{e_1, e_2\}$ тако да важи $f(e_1) = f(e_2) = 2$. Због правила подређивања, друго појављивање елемената e_1 и e_2 неће бити у истом скупу, па постоје скупови $R_1, R_2 \in \mathcal{M}$ који редом садрже e_1 и e_2 и још неке елементе. Нека је Q скуп који садржи елементе из $R_1 \cup R_2$ који нису у R. Уколико се текућа инстанца трансформише у инстанцу у којој се из тренутног мултискупа избаце скупови R, R_1 и R_2, а убаци Q, универзум за ту инстанцу ће садржати исте елементе као пре тога, али без e_1 и e_2. Рекурзивним решавањем проблема минималног покривача на тој инстанци добиће се покривач за

[6]Убацивањем једног од тих r_2 скупова биће покривен бар један елемент, па ће за покривање q елемената, у најгорем случају, бити потребно убацивање q скупова у покривач, а додатни један скуп је скуп R који је пре тога додат.

мањи универзум, те је потребно допунити добијени покривач тако да покрива и елементе e_1 и e_2. Могућа су два случаја. Ако је Q (или неки његов подскуп) део покривача мањег универзума, он мора бити избачен (јер није постојао у изворној инстанци), а уместо њега убачени R_1 и R_2[7]. У другом случају, ако Q није део покривача, сви његови елементи су покривени неким другим скуповима у покривачу, па је довољно убацити само R, како би се покрили и e_1 и e_2.

- *Правило завршавања када сви скупови имају кардиналност два*: Нека је $R \in \mathcal{M}$ скуп највеће кардиналности у мултискупу. Ако је кардиналност R два, због правила јединственог елемента и правила подскупа следи да ће сви скупови имати кардиналност два. Тада је могуће извршити редукцију проблема минималног покривача скупа на проблем оптималног упаривања у графу (енг. *maximum matching problem*). За задат неусмерен граф $G = (V, E)$ упаривање је скуп грана без заједничких чворова[8]. Оптимално упаривање је упаривање са максималним бројем грана, односно то је упаривање од кога не постоји упаривање са већим бројем грана [62]. Редукција се врши на следећи начин. За сваки елемент $u \in \mathcal{U}$ уводи се по један чвор графа, а за сваки скуп $S = \{u, v\} \in \mathcal{M}$ додаје се грана између чворова u и v и над таквим графом се решава проблем оптималног упаривања, за чије решавање постоји алгоритам полиномске[9] сложености чији је аутор Џек Едмондс (енг. *Jack Edmonds*) [12]. Након што се пронађе решење проблема оптималног упаривања, покривач се гради тако што се у њега укључују скупови који одговарају гранама из пронађеног оптималног упаривања. Уколико су неки чворови остали неупарени[10], одговарајући елементи ће остати непокривени (и биће сви у засебним скуповима) и за такве елементе је довољно редом додавати у покривач било који скуп у ком се они налазе. Након што се

[7]Уколико је Q део покривача, то значи да је бар један елемент покривен само захваљујући њему, иначе би Q могао бити избачен из покривача, па тај покривач не би био минималан. Пошто је Q настао на основу R_1 и R_2, потребно их је убацити у покривач јер они садрже те елементе које једино Q покрива. Није могуће убацити само један од скупова R_1 и R_2 у покривач јер тада не би био покривен или e_1 или e_2. С друге стране, у овом случају није потребно убацити и R зато што R_1 и R_2 садрже e_1 и e_2 који чине R.

[8]Другим речима, то је скуп грана такав да никоје две гране не садрже заједнички чвор.

[9]У питању је алгоритам под називом Алгоритам цветања (енг. *Blossom algorithm*) чија је оригинална временска сложеност $\mathcal{O}(|V|^4)$, а могуће ју је спустити на $\mathcal{O}(\sqrt{|V|}|E|)$ [38].

[10]За неко упаривање у графу, чвор се сматра упареним ако постоји грана која из њега излази и део је упаривања, иначе се каже да је чвор неупарен.

сви елементи покрију, добијени покривач је оптималан за дату инстанцу и враћа се као решење.

Уколико није могуће применити ниједно од претходних редукционих правила, примењује се правило гранања. Из мултискупа се узима скуп R максималне кардиналности и креирају два потпроблема која се рекурзивно решавају. Први потпроблем обухвата случај када је R убачен у решење, па је из даљег разматрања (у том потпроблему) потребно избацити и R и све елементе који се налазе у R из свих осталих скупова, зато што су ти елементи већ покривени. Други потпроблем је разматрање случаја када R није део покривача и тада је довољно искључити R из мултискупа за тај потпроблем. Рекурзивним решавањем оба потпроблема добиће се два решења C_1 и C_2, а решење за текућу инстанцу биће мањи од два покривача $\{R\} \cup C_1$ и C_2.

Анализом описаном на страни 43 и коришћењем мере (4.1) испоставља се да Ван Рој-Бодландеров алгоритам има сложеност $\mathcal{O}(1.4969^n)$.

4.3 Резултати поређења

Будући да се имплементирани генетски алгоритам пореди са неефикасним егзактним алгоритмима, поређење ће бити вршено за графове мањих величина. Експерименталним резултатима биће утврђено до које величине графова има смисла користити који од алгоритама, а граница прихватљиве дужине извршавања је 120 секунди и односи се на просечну дужину извршавања на 100 графова (за сваку величину графа). Заједно са тим биће праћено и просечно пронађено решење генетским алгоритмом и посматрано у односу на просечно решење пронађено егзактним алгоритмом, а пратиће се и просечна стопа грешке, односно колико чворова више у просеку има решење генетског алгоритма онда када јесте начињена грешка, као и максимална грешка. Графови над којима су тестирани алгоритми су генерисани на начин који је описан на страни 18.

Сви алгоритми су ручно имплементирани, уз изузетак неколико делова. У имплементацији Ван Рој-Бодландеровог алгоритма је за потребе последњег редукционог правила употребљена јавно доступна имплементација Алгоритма цветања [47]. За кеширање у генетском алгоритму коришћена је корисничка библиотека *cacheout* [10], док је за потребе интерполације употребљен пакет *SciPy* на начин описан на страни 39. Мимо тога, коришћени су

само системски модули *time*, *copy* и *random*. Кодови свих алгоритама се могу наћи на [63]. Сви тестови извршени су на рачунару са процесором *Intel Core i7-4790* и *8GB DDR3* радне меморије брзине *1600 MHz*.

Алгоритам грубе силе је већ за графове са двадесетак чворова пребацио једну секунду извршавања, док је за графове са тридесетак чворова у просеку трајао преко границе од 120 секунди. Ван Рој-Бодландеров алгоритам се боље држао и просечно извршавање тог алгоритма је прешло једну секунду на графовима са око 70 чворова, а границу од 120 секунди је пробио на графовима са око 90 чворова. Генетски алгоритам је просечно трајао краће од једне секунде за графове величине до 115 чворова, границу од 120 секунди прескакао је тек за графове са око 1600 чворова, за графове величине 5000 чворова извршавао се око 520 секунди, док је за графове величине 15 000 чворова време извршавања у просеку било око 1040 секунди. Што се стабилности времена извршавања тиче, алгоритам грубе силе и Ван Рој-Бодландеров алгоритам су имали најгоре случајеве када је алгоритам трајао и преко десет пута дуже него просечна дужина извршавања за графове те величине, док је генетски алгоритам био много стабилнији и разлика просека и најгорег извршавања није била толико изражена. Генетски алгоритам је постао бржи од алгоритма грубе силе за графове са око 17 чворова, а од Ван Рој-Бодландеровог за графове са око 60 чворова.

Генетски алгоритам се показао као прилично добро решење за улазе до око 50 чворова, где му је проценат поготка био готово увек изнад 90%, а просечна грешка свега један чвор. За графове величине око 90 чворова проценат поготка је, ипак, пао на свега 25%. Међутим, пратећи просечну грешку у случајевима када је грешка уследила, долази се до тога да је генетски алгоритам за те улазе у просеку грешио највише за 1,34 чвора, а да је максимална грешка била три чвора у односу на егзактно решење. Рачунајући и ситуације када је погодио тачно решење, генетски алгоритам је у просеку проналазио решења која су лошија за око један чвор у односу на егзактно решење (за највећи улаз где је тестиран и егзактни алгоритам). Детаљан приказ резултата може се видети у табелама 4.1 и 4.2, а визуелни приказ на сликама 4.3 и 4.4.

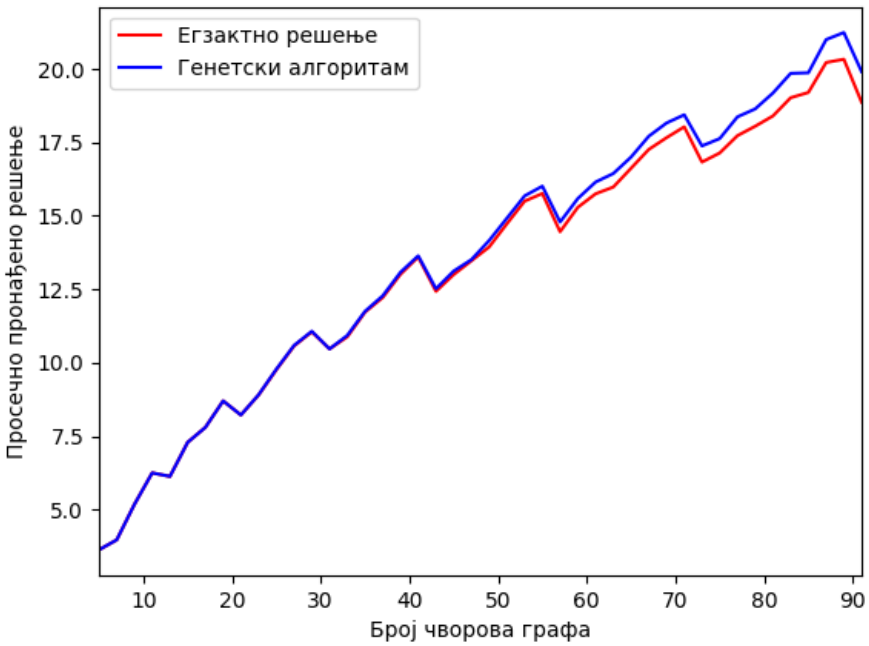
Величина графа	Груба сила просечно	ВРБ просечно	ГА просечно	Груба сила најгоре	ВРБ најгоре	ГА најгоре
5	0.00007	0.00004	0.00238	0.00044	0.00022	0.01122
9	0.00040	0.00005	0.00575	0.00392	0.00022	0.01919
11	0.00141	0.00007	0.00892	0.00553	0.00030	0.02769
15	0.01720	0.00017	0.02305	0.05880	0.00225	0.06806
17	0.06060	0.00020	0.0338	0.17932	0.00086	0.09229
19	0.22631	0.00022	0.03661	0.61071	0.00066	0.07245
21	0.44215	0.00041	0.04425	1.84692	0.00182	0.0808
23	1.81774	0.00055	0.04926	7.46353	0.00298	0.06859
25	6.79558	0.00072	0.06107	21.99393	0.00390	0.10117
27	25.85067	0.00065	0.06854	80.18162	0.00389	0.0971
29	75.21295	0.00090	0.078	403.94476	0.00471	0.10392
31	169.93808	0.00149	0.08264	1587.90356	0.01234	0.10452
35	/	0.00301	0.10351	/	0.01551	0.17786
45	/	0.01816	0.14956	/	0.11950	0.20175
55	/	0.06162	0.21825	/	0.61844	0.2517
59	/	0.23078	0.2464	/	2.38735	0.28911
61	/	0.28990	0.27035	/	2.28765	0.32986
65	/	0.45579	0.311	/	5.21362	0.35536
71	/	1.00511	0.37748	/	12.89916	0.44517
75	/	4.17409	0.41813	/	36.33619	0.48287
81	/	18.22302	0.48909	/	181.65879	0.55222
85	/	20.83658	0.53227	/	231.32704	0.60947
87	/	17.14871	0.5561	/	299.92003	0.60427
89	/	29.11526	0.61037	/	300.34816	0.73157
91	/	136.92274	0.67509	/	1960.52344	1.21885

Табела 4.1: Просечна и најгора времена извршавања сва три алгоритма за различите величине графова

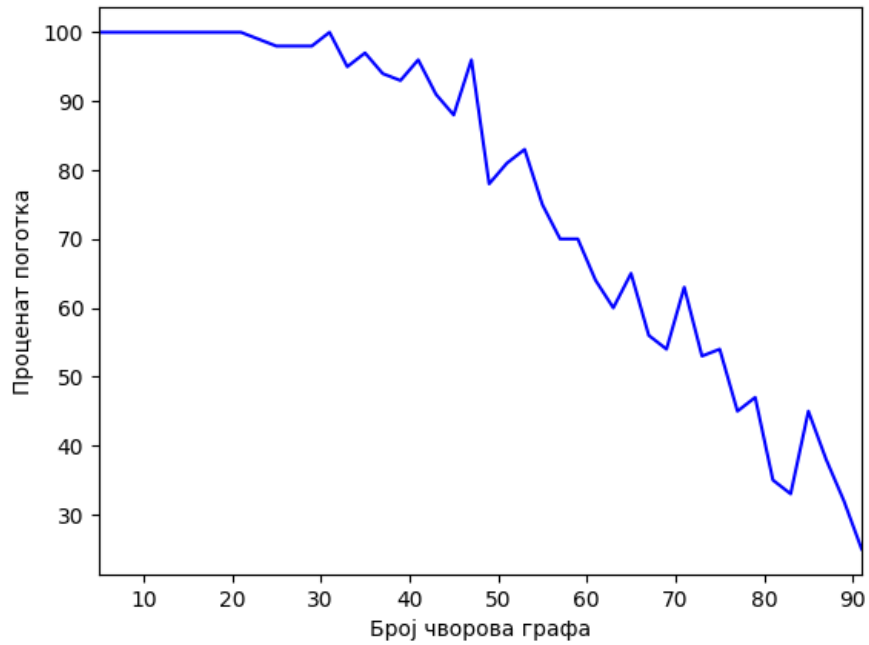
Величина графа	Просечно егзактно решење	Просечно пронађено решење	Проценат поготка	Просечна стопа грешке	Максимална грешка
5	3.63	3.63	100%	0	0
7	3.97	3.97	100%	0	0
9	5.18	5.18	100%	0	0
11	6.25	6.25	100%	0	0
13	6.13	6.13	100%	0	0
15	7.29	7.29	100%	0	0
17	7.8	7.8	100%	0	0
19	8.7	8.7	100%	0	0
21	8.22	8.22	100%	0	0
23	8.9	8.91	99%	1.0	1
25	9.76	9.78	98%	1.0	1
27	10.57	10.59	98%	1.0	1
29	11.05	11.07	98%	1.0	1
31	10.47	10.47	100%	0	0
33	10.88	10.93	95%	1.0	1
35	11.72	11.75	97%	1.0	1
37	12.22	12.28	94%	1.0	1
39	13.01	13.08	93%	1.0	1
41	13.6	13.64	96%	1.0	1
43	12.43	12.52	91%	1.0	1
45	13.0	13.12	88%	1.0	1
47	13.47	13.51	96%	1.0	1

Величина графа	Просечно егзактно решење	Просечно пронађено решење	Проценат поготка	Просечна стопа грешке	Максимална грешка
49	13.94	14.16	78%	1.0	1
51	14.73	14.92	81%	1.0	1
53	15.5	15.68	83%	1.06	2
55	15.76	16.01	75%	1.0	1
57	14.46	14.79	70%	1.1	2
59	15.29	15.59	70%	1.0	1
61	15.75	16.15	64%	1.11	2
63	15.98	16.44	60%	1.15	2
65	16.62	17	65%	1.09	2
67	17.26	17.71	56%	1.02	2
69	17.66	18.16	54%	1.09	2
71	18.03	18.44	63%	1.11	2
73	16.83	17.38	53%	1.17	2
75	17.14	17.63	54%	1.06	2
77	17.73	18.37	45%	1.16	3
79	18.05	18.64	47%	1.11	2
81	18.4	19.18	35%	1.2	3
83	19.02	19.85	33%	1.2	3
85	19.2	19.87	45%	1.22	2
87	20.22	21	38%	1.26	3
89	20.33	21.24	32%	1.34	3
91	18.87	19.81	25%	1.25	3

Табела 4.2: Просечно егзактно и пронађено решење, проценат колико пута је генетски алгоритам погодио тачно решење, просечна и максимална стопа грешке онда када је начињена грешка за различите величине графова



Слика 4.3: Поређење просечне кардиналности пронађених решења егзактним и генетским алгоритмом за различите величине графова



Слика 4.4: Приказ пада процента поготка генетског алгоритма

Глава 5

Закључак

У овом раду приказан је генетски алгоритам за решавање добро познатог проблема минималног доминирајућег скупа. Будући да егзактни алгоритми већ за стотинак чворова доводе до времена извршавања које се мери десетинама минута, јасно је да је неки апроксимативни метод једини начин за проналазак иоле квалитетнијих решења за улазе од неколико хиљада чворова у разумном времену. Генетски алгоритам је једна од најпопуларнијих метахеуристика за решавање оптимизационих проблема и погодан је за примену код графовских проблема и због тога је одабран за коришћење. У раду је описана репрезентација графа коришћењем низа природних бројева, али који се посматрају као низови битова, чиме се креира имплицитна матрица повезаности и тиме остварује максимална меморијска уштеда при коришћењу таквог типа репрезентације. Као две најозбиљније препреке временској ефикасности овог алгоритма показале су се функција прилагођености и функција поправке јединки које су, у својим основним облицима, трошиле превише времена и самим тим успоравале генетски алгоритам, али се испоставило да постоје начини за њихово драстично убрзавање. Бирање квалитетне почетне популације се показало као нетривијалан задатак, па је и томе посвећена посебна пажња у овом раду. Подешавање параметара је један од комплекснијих проблема при имплементацији генетског алгоритма, а у овом раду је покушано са проналажњем најбољих конфигурација за сваку величину улаза, са одређеним успехом.

Унапређење презентованог алгоритма може се постићи применом паралелизације, будући да је рачунање вредности јединке или њено поправљање суштински независно од других јединки. Бољи параметри се могу пронаћи

покретањем метода итеративне трке на већем узорку, као и коришћењем већег броја конфигурација у интерполацији, а напредак се може остварити и тиме што би се параметри подешавали не само према броју чворова графа, већ и према броју грана. Може се истражити и утицај примене прилагодљивих параметара на резултате алгоритма. Једна од стратегија која се користи у генетским алгоритмима јесте да се за исти улаз алгоритам покреће више пута, али да се неке од најбољих јединки последње генерације претходног извршавања користе као део почетне популације при наредном покретању, те би се то могло применити овде, уз смањење параметара за веће улазе или њихово потпуно фиксирање.

Библиографија

[1] Mohamed Abdel-Basset, Laila Abdel-Fatah, and Arun Kumar Sangaiah. Metaheuristic algorithms: A comprehensive review. *Computational intelligence for multimedia big data on the cloud with engineering applications*, pages 185–231, 2018.

[2] Otman Abdoun and Jaafar Abouchabaka. A comparative study of adaptive crossover operators for genetic algorithms to resolve the traveling salesman problem. *arXiv preprint arXiv:1203.3097*, 2012.

[3] Chang Wook Ahn. *Advances in evolutionary algorithms*. Springer, 2006.

[4] Sarwan Ali. Cache replacement algorithm. *arXiv preprint arXiv:2107.14646*, 2021.

[5] Martin D. Buhmann. *Radial Basis Functions: Theory and Implementations*, volume 12. Cambridge University Press, 2003.

[6] Erick Cantú-Paz et al. A survey of parallel genetic algorithms. *Calculateurs paralleles, reseaux et systems repartis*, 10(2):141–171, 1998.

[7] Yair Caro and Michael A. Henning. Directed domination in oriented graphs. *Discrete Applied Mathematics*, 160(7-8):1053–1063, 2012.

[8] Kalyanmoy Deb, Ram Bhushan Agrawal, et al. Simulated binary crossover for continuous search space. *Complex systems*, 9(2):115–148, 1995.

[9] Kusum Deep and Hadush Mebrahtu. Combined mutation operators of genetic algorithm for the travelling salesman problem. *International Journal of Combinatorial Optimization Problems and Informatics*, 2(3):1–23, 2011.

[10] Derrick Gilland. A caching library for Python. `https://github.com/dgilland/cacheout`. accessed: 11/07/2023.

[11] Pablo Echenique, Jesús Gómez-Gardeñes, Yamir Moreno, and Alexei Vázquez. Distance-d covering problems in scale-free networks with degree correlations. *Physical Review E*, 71(3):035102, 2005.

[12] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of mathematics*, 17:449–467, 1965.

[13] Agoston E. Eiben and Cees H. M. van Kemenade. *Performance of multi-parent crossover operators on numerical function optimization problems.* Rijksuniversiteit Leiden. Vakgroep Informatica, 1995.

[14] David Eppstein. Quasiconvex analysis of backtracking algorithms. *arXiv preprint cs/0304018*, 2003.

[15] Larry J Eshelman. Preventing premature convergence in genetic algorithms by preventing incest. In *Proceedings of Fourth International Conference on Genetic Algorithms, 1991*, 1991.

[16] Michael R. Fellows, Fedor V. Fomin, Daniel Lokshtanov, Frances Rosamond, Saket Saurabh, and Yngve Villanger. Local search: Is brute-force avoidable? *Journal of Computer and System Sciences*, 78(3):707–719, 2012.

[17] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. Some new techniques in design and analysis of exact (exponential) algorithms. *Bull. EATCS*, 87:47–77, 2005.

[18] Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch. A measure & conquer approach for the analysis of exact algorithms. *Journal of the ACM (JACM)*, 56(5):1–32, 2009.

[19] Fedor V. Fomin, Dieter Kratsch, and Gerhard J Woeginger. Exact (exponential) algorithms for the dominating set problem. In *Graph-Theoretic Concepts in Computer Science: 30th International Workshop, WG 2004, Bad Honnef, Germany, June 21-23, 2004. Revised Papers 30*, pages 245–256. Springer, 2005.

[20] Cu Nguyen Giap and Dinh Thi Ha. Parallel genetic algorithm for minimum dominating set problem. In *2014 International Conference on Computing, Management and Telecommunications (ComManTel)*, pages 165–169. IEEE, 2014.

[21] Paul Glasserman and David D Yao. Some guidelines and guarantees for common random numbers. *Management Science*, 38(6):884–908, 1992.

[22] Kate Ean Nee Goh, Jeng Feng Chin, Wei Ping Loh, and Melissa Chea-Ling Tan. A constraint programming-based genetic algorithm (CPGA) for capacity output optimization. *Journal of Industrial Engineering and Management*, 7(5):1222–1249, 2014.

[23] Fabrizio Grandoni. A note on the complexity of minimum dominating set. *Journal of Discrete Algorithms*, 4(2):209–214, 2006.

[24] Abdel-Rahman Hedar and Rashad Ismail. Hybrid genetic algorithm for minimum dominating set problem. In *Computational Science and Its Applications–ICCSA 2010: International Conference, Fukuoka, Japan, March 23-26, 2010, Proceedings, Part IV 10*, pages 457–467. Springer, 2010.

[25] iracepy. Example of using irace from Python. `https://github.com/auto-optimization/iracepy`. accessed: 11/04/2023.

[26] itertools. Functions creating iterators for efficient looping. `https://docs.python.org/3/library/itertools.html`. accessed: 26/05/2023.

[27] Yoichi Iwata. A faster algorithm for dominating set analyzed by the potential method. In *Parameterized and Exact Computation: 6th International Symposium, IPEC 2011, Saarbrücken, Germany, September 6-8, 2011. Revised Selected Papers 6*, pages 41–54. Springer, 2012.

[28] Eric Kee, Sarah Airey, and Walling Cyre. An adaptive genetic algorithm. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation*, pages 391–397, 2001.

[29] Laura L. Kelleher and Margaret B. Cozzens. Dominating sets in social network graphs. *Mathematical Social Sciences*, 16(3):267–279, 1988.

[30] Natallia Kokash. An introduction to heuristic algorithms. *Department of Informatics and Telecommunications*, pages 1–8, 2005.

[31] Vladlen Koltun and Christos H. Papadimitriou. Approximately dominating representatives. *Theoretical Computer Science*, 371(3):148–154, 2007.

[32] Oliver Kramer. *Genetic Algorithm Essentials*. Springer, 2017.

[33] Pedro Larranaga, Cindy M. H. Kuijpers, Roberto H. Murga, Inaki Inza, and Sejla Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial intelligence review*, 13:129–170, 1999.

[34] Manuel López-Ibáñez, Leslie Pérez Cáceres, Jérémie Dubois-Lacoste, Thomas G Stützle, and Mauro Birattari. The irace Package: User Guide, version 3.5, October 2022. online at: `https://cran.r-project.org/web/packages/irace/vignettes/irace-package.pdf`, accessed: 11/04/2023.

[35] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016.

[36] George F. Luger. *Artificial Intelligence. Structures and Strategies for Complex Problem Solving.* Pearson, 6th edition, 2009.

[37] Carl Medsker and Il-Yeol Song. ProloGA: a Prolog implementation of a genetic algorithm. In *[1993] Proceedings IEEE International Conference on Developing and Managing Intelligent System Projects*, pages 77–84. IEEE, 1993.

[38] Silvio Micali and Vijay V. Vazirani. An $\mathcal{O}(\sqrt{|v|}|e|)$ algorithm for finding maximum matching in general graphs. In *21st Annual Symposium on Foundations of Computer Science (sfcs 1980)*, pages 17–27. IEEE, 1980.

[39] Melanie Mitchell. *An introduction to genetic algorithms.* MIT press, 1998.

[40] Jürg Nievergelt. Exhaustive search, combinatorial optimization and enumeration: Exploring the potential of raw computing power. In *Sofsem*, page 23. Springer, 2000.

[41] Tatsuya Nomura. An analysis on linear crossover for real number chromosomes in an infinite population size. In *Proceedings of 1997 IEEE International Conference on Evolutionary Computation (ICEC'97)*, pages 111–114. IEEE, 1997.

[42] Chaoyi Pang, Rui Zhang, Qing Zhang, and Junhu Wang. Dominating sets in directed graphs. *Information sciences*, 180(19):3647–3652, 2010.

[43] N. Parthiban, Indra Rajasingh, and R. Sundara Rajan. Minimum connected dominating set for certain circulant networks. *Procedia Computer Science*, 57:587–591, 2015.

[44] Eric Pellerin, Luc Pigeon, and Sylvain Delisle. Self-adaptive parameters in genetic algorithms. In *Data mining and knowledge discovery: theory, tools, and technology VI*, volume 5433, pages 53–64. SPIE, 2004.

[45] Fethi A. Rabhi, Guy Lapalme, and Albert Y. Zomaya. A functional design framework for genetic algorithms. In *Scottish Functional Programming Workshop*, pages 116–125, 1999.

[46] Cristanel Razafimandimby, Valeria Loscrí, Anna Maria Vegni, and Abderrahim Benslimane. Neuro-dominating set scheme for a fast and efficient robot deployment in internet of robotic things. *Ad Hoc Networks*, 86:36–45, 2019.

[47] Robert Durfee. An implementation of the blossom algorithm for constructing maximum matchings on graphs. `https://github.com/RobertDurfee/Blossom`. accessed: 11/07/2023.

[48] Ingo Schiermeyer. Efficiency in exponential time for domination-type problems. *Discrete Applied Mathematics*, 156(17):3291–3297, 2008.

[49] SciPy. Documentation for radial basis function interpolation in Python. `https://docs.scipy.org/doc/scipy/reference/generated/scipy.interpolate.RBFInterpolator.html`. accessed: 14/04/2023.

[50] Masato Takahashi and Hajime Kita. A crossover operator using independent component analysis for real-coded genetic algorithms. In *Proceedings of the 2001 congress on evolutionary computation (ieee cat. no. 01th8546)*, volume 1, pages 643–649. IEEE, 2001.

[51] Johan M. M. van Rooij and Hans L Bodlaender. Exact algorithms for dominating set. *Discrete Applied Mathematics*, 159(17):2147–2164, 2011.

[52] Johan M. M. van Rooij, Jesper Nederlof, and Thomas C. van Dijk. Inclusion/exclusion meets measure and conquer: Exact algorithms for counting dominating sets. In *Algorithms-ESA 2009: 17th Annual European*

Symposium, Copenhagen, Denmark, September 7-9, 2009. Proceedings 17, pages 554–565. Springer, 2009.

[53] Feng Wang, Hongwei Du, Erika Camacho, Kuai Xu, Wonjun Lee, Yan Shi, and Shan Shan. On positive influence dominating sets in social networks. *Theoretical Computer Science*, 412(3):265–269, 2011.

[54] Haiying Wang, Huiru Zheng, Fiona Browne, and Chaoyang Wang. Minimum dominating sets in cell cycle specific protein interaction networks. In *2014 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 25–30. IEEE, 2014.

[55] Henry S. Warren. *Hacker's delight*. Pearson Education, 2013.

[56] Peter Wegner. A technique for counting ones in a binary computer. *Communications of the ACM*, 3(5):322, 1960.

[57] Robin J Wilson. *Introduction to graph theory*. Pearson Education India, 1979.

[58] Renata Wong, Weng-Long Chang, Wen-Yu Chung, and Athanasios V. Vasilakos. Biomolecular and quantum algorithms for the dominating set problem in arbitrary networks. *Scientific Reports*, 13(1):4205, 2023.

[59] Jie Wu. Extended dominating-set-based routing in ad hoc wireless networks with unidirectional links. *IEEE transactions on parallel and distributed systems*, 13(9):866–881, 2002.

[60] Furong Ye, Carola Doerr, Hao Wang, and Thomas Bäck. Automated configuration of genetic algorithms by tuning for anytime performance. *IEEE Transactions on Evolutionary Computation*, 26(6):1526–1538, 2022.

[61] Jinghui Zhong, Xiaomin Hu, Jun Zhang, and Min Gu. Comparison of performance between different selection strategies on simple genetic algorithms. In *International conference on computational intelligence for modelling, control and automation and international conference on intelligent agents, web technologies and internet commerce (CIMCA-IAWTIC'06)*, volume 2, pages 1115–1121. IEEE, 2005.

[62] Миодраг Живковић. *Алгоритми*. Математички факултет, 2000.

[63] Немања Живановић. Генетски алгоритам за решавање проблема минималног доминирајућег скупа. `https://github.com/NemanjaZivanovic/master-rad`. accessed: 16/07/2023.