УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Роберт Т. Дожа

УГРАДЊА АЛГОРИТМА ЗА РАЗБИЈАЊЕ СИМЕТРИЈА НАД ГРАФОВИМА У РЕШАВАЧЕ ОГРАНИЧЕЊА

мастер рад

Београд, 2025.

Ментор:

др Милан Банковић, доцент
Универзитет у Београду, Математички факултет

Чланови комисије:

др Филип Марић, редовни професор
Универзитет у Београду, Математички факултет

др Весна Маринковић, ванредни професор
Универзитет у Београду, Математички факултет

Датум одбране: _____

Породици, пријатељима, папиру и морској соли

Наслов мастер рада: Уградња алгоритма за разбијање симетрија над графовима у решаваче ограничења

Резиме: Симетрије представљају значајан извор редундантности у комбинаторној претрази графова, где многобројне изоморфне репрезентације једног графа значајно увећавају простор претраге. Овај рад приказује један алгоритам за елиминацију симетрија провером лексикографске минималности графа. Приказујемо имплементитацију овог алгоритма и његову интеграцију у оквиру SMT решавача ArgoSMT. Експерименталном евалуацијом приказујемо резултате примене овог приступа при решавању проблема конструкције графова са задатим бројем чворова, грана и троуглова. Резултати сугеришу да додавање ограничења лексикографске минималности графа у одређеним случајевима може значајно смањити време претраге.

Кључне речи: графови, SAT, SMT, програмирање ограничења, CDCL, лексикографска минималност, симетрије

Садржај

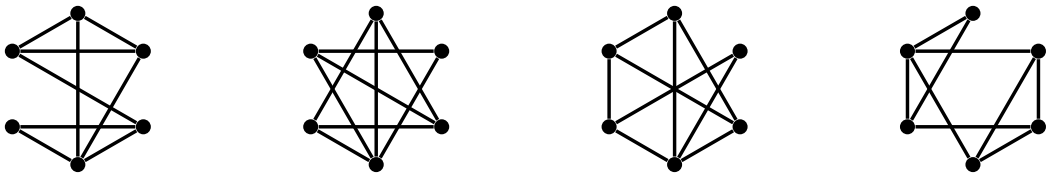
Глава 1

Увод

У многим комбинаторним проблемима, као што су бојење графова, распоређивање ресурса или генерисање структура са одређеним својствима, јављају се *симетрије*, односно одређене подударности између комбинаторних конфигурација које се разматрају. Ово значи да се разматрају суштински исти објекти више пута, што значајно увећава простор претраге. Због тога је од значаја разматрати како решити овај проблем, односно извршити тзв. *разбијање симетрија* током претраге.

У овом раду, разматраћемо како се симетрије могу разбити током трагања за графовима са одређеним својствима користећи решаваче ограничења. Код графова, симетрије настају због чињенице да један граф може имати много себи изоморфних графова, односно графова који су исти по структури, а разликују се једино по обележавању чворова. На слици 1.1 приказан је пример изоморфних графова. Циљ је, дакле, класу изоморфних графова свести на једног њеног представника. Један начин да се то уради је да се у датој класи посматра лексикографски минималан граф, односно граф који је најмањи у смислу лексикографског поретка међу матрицама суседства графова. Овај проблем је тежак, али је један алгоритам за утврђивање симетрија у овом погледу развијен и описан у [8]. У наведеном раду, алгоритам за разбијање симетрија над графовима је имплементиран у оквиру SAT решавача [3]. Проблем генерисања графа је представљен исказном формулом чију задовољивост испитује SAT решавач, док алгоритам за разбијање симетрија представља посебан модул који комуницира са SAT решавачем генерисањем одговарајућих клауза. Овакав приступ код кога се процедура за резоновање у специфичном домену комбинује са језгром које је одговорно за исказно ре-

зоновање и претрагу је карактеристично за модерне SMT решаваче [3]. Ова чињеница представља мотивацију за интеграцију алгоритма описаног у [8] у SMT решавач.



Слика 1.1: Пример изоморфних графова. Сви приказани графови међусобно су изоморфни.

Према томе, циљ овог рада је да се поменути алгоритам имплементира и угради у конкретан SMT решавач заснован на CDCL($\mathcal{T}$) архитектури [3, 11]. Конкретно, алгоритам ће бити имплементиран у оквиру решавача ArgoSMT [2] и биће отвореног кода. Приказаћемо начин на који се алгоритам уклапа у архитектуру овог SMT решавача, као и ефекте разбијања симетрија овим приступом на конкретним примерима.

Напоменимо да је у [8] коришћен *динамички приступ* елиминиацији симетрија, то јест елиминација се вршила „у ходу", током претраге. Динамички приступ је и раније коришћен у сличним применама [1, 5, 6, 7, 10]. Овај приступ ћемо и ми применити. За разлику од њега, постоји и *статички* приступ, где се унапред (пре почетка претраге) генеришу клаузе које елиминишу симетрије међу графовима [4]. Методе засноване на статичком приступу су најчешће непотпуне, тј. не обезбеђују потпуну елиминацију симетрија, с обзиром да би за тако нешто било неопходно унапред генерисати превелик број клауза. Са друге стране, у динамичком приступу се клаузе генеришу лењо, само када је то неопходно, што обично резултује знатно мањим бројем генерисаних клауза.

Остатак рада има следећу структуру. У поглављу 2 дискутујемо основне појмове и дефиниције које су релевантне за остатак рада. У поглављу 3 описујемо алгоритам MinCheck који служи за проверу минималности графа тражењем тзв. индикаторских парова. Затим, у поглављу 4 описујемо начин на који је овај алгоритам уграђен у решавач ArgoSMT. У поглављу 5 приказујемо резултате примене решавача ArgoSMT са додатим пропагатором на неким инстанцама. На крају, у поглављу 6 дајемо кратак осврт на рад.

Глава 2

Основе

У овом поглављу следе основни појмови, ознаке и дефиниције који ће бити коришћени у остатку рада.

2.1 Исказна логика

Исказним *атомом* сматрамо формулу у чију структуру не залазимо и која може бити тачна или нетачна. Формуле исказне логике градимо од исказних атома користећи *исказне везнике* ($\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$) на уобичајен начин. *Литералима* називамо атоме или њихове негације. *Клауза* је дисјункција литерала (за сваки од тих литерала кажемо да га клауза *садржи*). Формула је у *конјунктивној нормалној форми (КНФ)* уколико је конјункција клауза.

Исказна *валуација* је функција $v : X \to \{\text{true}, \text{false}\}$, где је X скуп исказних слова. Исказном валуацијом се атомима придружује истинитосна вредност *тачно* (true) или *нетачно* (false), а тачност сложених формула се дефинише индукцијом, применом исказних везника, на уобичајен начин. За исказни атом p кажемо да је *тачан* при валуацији v уколико је $v(p) = \text{true}$, а *нетачан* уколико је $v(p) = \text{false}$. Исказна валуација *задовољава* фомулу уколико је та формула тачна при тој валуацији. Специјално, исказна валуација задовољава формулу у КНФ уколико свака клауза формуле садржи литерал који је тачан у тој валуацији. Исказна формула је *задовољива* уколико постоји валуација која је задовољава.

2.2 SAT проблем

SAT проблем је проблем испитивања задовољивости исказне формуле. То је један од централних проблема рачунарства. Имплементације алгоритама за решавање SAT проблема називају се *SAT решавачи*. Модерни приступи решавању SAT проблема засновани су на CDCL (енг. *conflict driven clause learning*) алгоритму [3].

2.3 Логика првог реда

Логика првог реда представља уопштење исказне логике у којој се атоми посматрају општије, као *атомичне формуле*, чија се структура сада разматра. Иако формула првог реда дозвољава појаву *променљивих* и њихову квантификацију (квантификатори $\forall$-„за свако" и $\exists$-„постоји"), ми ћемо разматрати *базне* формуле, то јест формуле без променљивих.

Дакле, синтакса логике првог реда се, за наше потребе, састоји од *симбола константи* (које ћемо означавати a, b, c итд.), *функцијских симбола* (које ћемо означавати са f, g, h, ...) и *предикатских (релацијских) симбола* (које ћемо означавати са α, β, γ, ...). *Терм* може бити симбол константе или може бити представљен функцијским симболом примењеним над неким *подтермовима* ($f(a,b)$, $g\left(h(c),b,a\right)$, ...). Атомична формула је представљена предикатским симболом примењеним над термовима ($\alpha(f(a,b),c)$, $\beta(b,h(a),c)$, ...) Сложене формуле добијају се применом исказних везника над атомичним формулама.

Тачност (базне) формуле првог реда зависи од *структуре* у којој се та формула интерпретира. Структура првог реда се састоји од домена и интерпретација симбола константи, функцијских и релацијских симбола. Интерпретације функцијских симбола су функције над доменом, док су интерпретације релацијских симбола релације над доменом структуре. Специјално, симболи константи се интерпретирају фиксираним елементима домена, док се предикатски симболи арности нула интерпретирају као тачни или нетачни и не зависе од вредности термова (они одговарају исказним атомима). Тачност сложених формула се сада дефинише на исти начин као и у случају исказне логике. За структуру кажемо да *задовољава* дату формулу (или да је њен *модел*), ако је формула тачна у тој структури. Формула је *задовољива* у

логици првог реда ако постоји структура која је задовољава.

2.4 Теорије првог реда, SMT проблем

Теорија првог реда је одређена скупом структура првог реда које називамо моделима те теорије. Формула првог реда је *задовољива у теорији* ако је тачна у неком моделу те теорије. Она је *ваљана у теорији* ако је тачна у сваком моделу те теорије. Теорија је *одлучива* ако је испитивање задовољивости формула у тој теорији одлучиво, то јест ако постоји ефективна процедура која за било коју формулу у коначном времену испитује задовољивост у теорији. *SMT проблем* је проблем испитивања задовољивости формуле у некој теорији првог реда. SMT решавачи су имплементације алгоритама за одлучивање SMT проблема у некој одлучивој теорији.

Модерни SMT решавачи најчешће користе тзв. *лењи приступ* (енг. *lazy approach*) за испитивање задовољивости [3]. Идеја овог приступа је раздвајање задатка на два нивоа. Формула првог реда у датој теорији се најпре преводи у чисто исказну формулу (у тзв. *исказну апстракцију*), где се свака атомична формула првог реда посматра као један исказни атом. На овом нивоу се користи SAT решавач да би се пронашла задовољавајућа исказна валуација. Затим се врши провера задовољивости у теорији. Када SAT решавач предложи валуацију (то јест скуп атома који би требало да буду тачни), та валуација се шаље специјализованом теоријском решавачу (енг. *theory solver*) који проверава да ли је она заиста задовољива у оквиру теорије. Ако јесте, добијамо модел и цела формула је задовољива. Ако није, теоријски решавач враћа конфликт у облику клаузе који SAT решавач додаје у своју претрагу, како би елиминисао ту немогућу комбинацију и наставио да тражи нову. На овај начин, SAT решавач и теоријски решавач раде заједно - SAT решавач се бави комбинаторном претрагом, док теоријски решавач обезбеђује семантичку исправност у оквиру конкретне теорије. У пракси, SAT решавачи исказну валуацију граде инкрементално, што омогућава теоријском решавачу да детектује незадовољивост у теорији раније, пре него што парцијална исказна валуација буде комплетирана. Такође, теоријски решавач има могућност да активно учествује у изградњи задовољавајуће валуације тако што предлаже литерале које треба додати у валуацију, на основу логичких последица у теорији. Једна архитектура заснована на описаном приступу, на којој су

најчешће засновани модерни SMT решавачи, је CDCL($\mathcal{T}$) архитектура [11] коју детаљније описујемо у поглављу 4.1.

2.5 Графови

Граф је математичка структура која описује систем објеката и веза између њих. Те објекте називамо чворовима, а везе гранама. Постоји много различитих типова графова. Наш фокус у овом раду биће на неусмереним графовима (тј. графовима код којих су везе, односно гране, симетричне). У овом раду користићемо нотацију која је преузета из [8].

Формално, граф је уређени пар (V, E), где V представља коначан скуп чворова, а $E \subseteq V \times V$ скуп грана. За граф $G = (V, E)$, са $V(G)$ и $E(G)$ ћемо означавати скупове V и E. Грану $(u, v) \in E$ ћемо краће означавати са uv или $u - v$. Ако је $|V| = n$, за неки природан број $n \in \mathbb{N}$, онда ћемо сматрати да су чворови нумерисани бројевима од 1 до n, односно да је $V = \{1, \ldots, n\}$. Са $\mathcal{G}_n$ ћемо означавати скуп свих графова са чворовима $\{1, \ldots, n\}$.

Матрица суседства графа $G \in \mathcal{G}_n$, у ознаци A_G, представља квадратну матрицу димензије $n \times n$ за коју важи:

$$(A_G)_{u,v} = \begin{cases} 1, & \text{ако } u - v \in E, \\ 0, & \text{иначе} \end{cases}$$

за свако $u, v \in V$. За $u, v \in \{1, \ldots, n\}$, поље матрице A_G у u-тој врсти и v-тој колони означавамо са $A_G[u][v]$, док u-ту врсту означавамо са $A_G[u]$.

За граф $G = (V, E)$ кажемо да је *неусмерен* ако за сваки пар чворова $u, v \in V$ важи да из $(u, v) \in E$ следи да и $(v, u) \in E$. Лако се види да је граф неусмерен ако и само ако је његова матрица суседства симетрична. Као што је већ речено, у овом раду разматрамо искључиво неусмерене графове.

Пермутација коначног скупа је бијекција тог скупа на њега самог. Како ћемо посматрати пермутације скупа чворова графова, означимо са $\mathcal{S}_n$ скуп свих пермутација скупа $\{1, \ldots, n\}$.

За два графа $G, H \in \mathcal{G}_n$ кажемо да су *изоморфна*, уколико постоји пермутација чворова $\pi \in \mathcal{S}_n$ која чува гране, односно за свака два чвора $u, v \in V(G)$ важи $uv \in E(G)$ ако и само ако $\pi(u)\pi(v) \in E(H)$.

Нека су $G, H \in \mathcal{G}_n$ два графа и нека је $\pi \in \mathcal{S}_n$ пермутација. Кажемо да је граф H добијен пермутацијом π графа G ако је $E(H) = \{\pi(u)\pi(v) \mid uv \in E(G)\}$ и то означавамо са $H = \pi(G)$.

Нека су $G, H \in \mathcal{G}_n$ два графа. Кажемо да је граф G *лексикографски мањи* од графа H, у ознаци $G \prec H$, ако је низ $A_G[1]A_G[2]\dots A_G[n]$ лексикографски мањи од низа $A_H[1]A_H[2]\dots A_H[n]$. Граф G је *лексикографски мањи или једнак* графу H, у ознаци $G \preceq H$, ако важи $G \prec H$ или $G = H$. Граф $G \in \mathcal{G}_n$ је *лексикографски минималан (или $\preceq$-минималан)* уколико важи $G \preceq \pi(G)$, за сваку пермутацију $\pi \in \mathcal{S}_n$.

2.6 Проблеми задовољења ограничења

Проблем задовољења ограничења (енг. *constraint satisfaction problem* (CSP)) је уређена тројка (X, D, C), где је X (коначан) скуп *променљивих*, D скуп коначних *домена* тих променљивих и C је коначан скуп *ограничења* која је потребно да те променљиве задовоље. Решење CSP-а је додела променљивама вредности из њиховог домена таква да су сва ограничења испуњена.

Алати који имплементирају алгоритме за решавање CSP проблема називају се *CSP решавачима*. CSP решавачи су засновани на претрази и пропагацијама којима се простор претраге смањује на основу својстава ограничења датог проблема. Поред коришћења CSP решавача, CSP проблеми се могу решавати и свођењем на SAT или SMT проблем. Један приступ свођења на SAT је да се у SAT решавач угради пропагатори за специфична ограничења који генеришу клаузе којима описују своје пропагације када се за њима јави потреба. Овај приступ, познат и под називом *лењо генерисање клауза* (енг. *lazy clause generation (LCG)* [9]), врло је сличан начину рада модерних SMT решавача заснованих на CDCL($\mathcal{T}$) архитектури, као и приступу који је коришћен у раду [8] за разбијање симетрија над графовима.

Симетрије у CSP-у настају када различите доделе представљају суштински исто решење. *Симетрије променљивих* настају када пермутације променљивих дају еквивалентна решења, док *симетрије вредности* настају када пермутације вредности дају еквивалентна решења. Присуство симетричних решења није пожељна појава, јер она (у многим случајевима значајно) увећавају простор претраге, а њихово разматрање не доводи до суштински нових закључака. Као што је поменуто у уводној глави, *разбијање симетрија* је

приступ у којем се улаже труд да би се симтерије избегле и да би се уштедело на временским и просторним ресурсима.

Глава 3

Елиминација симетрија над графовима

У овом поглављу биће описан алгоритам који се користи у [8] за одређивање минималности графа. Пре тога, увешћемо концепте специфичне за овај алгоритам који су коришћени за његов опис.

3.1 Парцијални графови

Парцијални (или парцијално дефинисани) граф је уређена тројка $G = (V, D, U)$, где је V коначан скуп, а скупови $D \subseteq V \times V$ и $U \subseteq V \times V$ су дисјунктни. За дати граф $G = (V, D, U)$, са $V(G)$, $D(G)$ и $U(G)$ означавамо скупове V, D и U, редом. D је скуп *дефинисаних грана*, а U је скуп *недефинисаних грана*. Интуитивно, дефинисане гране одговарају гранама у стандардној дефиницији графа, док су недефинисане гране оне за које, у датом тренутку током формирања графа, не знамо да ли јесу или нису у графу. За парцијалне графове ћемо (као и код „обичних" графова) сматрати да су *неусмерени*, то јест да за све $u, v \in V$ важи $(u, v) \in D$ ако и само $(v, u) \in D$, као и $(u, v) \in U$ ако и само ако $(v, u) \in U$. Јасно, „обичан граф", односно граф $G = (V, E)$ у смислу дефиниције из поглавља 2.5, можемо идентификовати са парцијално дефинисаним графом $G' = (V, E, \emptyset)$. Зато те графове зовемо и *потпуно дефинисаним графовима*. Као и раније, подразумеваћемо да, ако је $|V| = n$, онда је $V = \{1, \ldots, n\}$. Са $\mathcal{P}_n$ означавамо све парцијалне графове са n чворова. Матрицу суседства парцијално дефинисаног графа $G \in \mathcal{P}_n$

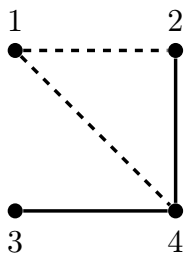
дефинишемо аналогно потпуном графу:

$$(A_G)_{u,v} = \begin{cases} 1, & \text{ако } u - v \in D, \\ \star, & \text{ако } u - v \in U, \\ 0, & \text{иначе,} \end{cases}$$

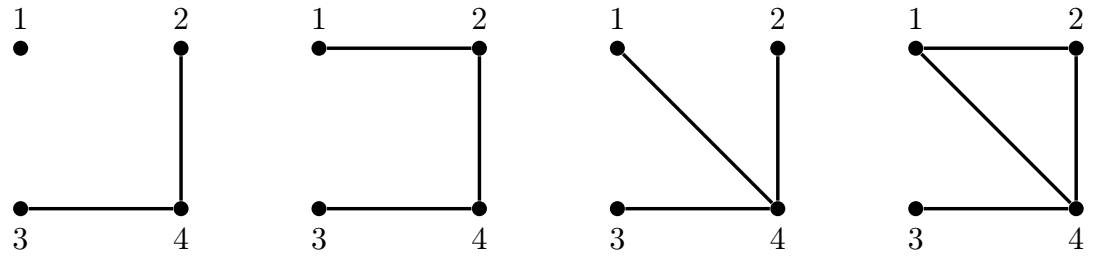
за све $u, v \in V$. Парцијални граф $G \in \mathcal{P}_n$ може се *проширити* до потпуног графа $H \in \mathcal{G}_n$ ако важи $D(G) \subseteq E(H) \subseteq D(G) \cup U(G)$. Тада за H кажемо да је *проширење* графа G. Са $\mathcal{X}(G)$ означавамо скуп свих проширења парцијално дефинисаног графа G. За парцијалне графове такође можемо дефинисати $\preceq$-минималност. За парцијални граф G кажемо да је $\preceq$-*минималан* ако $\mathcal{X}(G)$ садржи $\preceq$-минималан граф (у смислу дефиниције $\preceq$-минималности потпуних графова дате у поглављу 2.5).

Како ћемо граф $G \in \mathcal{G}_n$ конструисати инкрементално, почевши од „празног" графа (n чворова без грана између њих), постепеним додавањем грана, парцијални графови ће служити као природна репрезентација стања током те конструкције. Наиме, недефинисане гране су оне за које још увек, у датом тренутку, не знамо да ли се налазе у графу или не. Тако проширења текућег парцијалног графа представљају потпуне графове које од њега можемо добити „додавањем" грана (прецизније, претварањем неких недефинисаних грана у дефинисане и уклањањем осталих недефинисаних грана).

На слици 3.1 приказан је пример једног парцијално дефинисаног графа $G \in \mathcal{P}_4$, а на слици 3.2 његова проширења. Можемо приметити да је граф G минималан, јер има минимално проширење (на пример, најлевљи граф приказан на слици 3.2).



Слика 3.1: Пример парцијално дефинисаног графа. Дефинисане гране представљене су пуним, а недефинисане гране испрекиданим линијама.



Слика 3.2: Проширења парцијалног графа са слике 3.1

3.2 Индикаторски парови

Нека је $G \in \mathcal{P}_n$ парцијално дефинисан граф и нека је $\pi \in \mathcal{S}_n$ пермутација. За пар чворова $(i, j) \in V \times V$ кажемо да је (G, π)-*једнак* уколико за свако $H \in \mathcal{X}(G)$ важи $A_H[i][j] = A_{\pi(H)}[i][j]$. Ово важи ако и само ако важи $(i, j) \in \{(\pi(i), \pi(j)), (\pi(j), \pi(i))\}$ или $A_G[i][j] = A_{\pi(G)}[i][j] \neq \star$ [8]. За пар чворова $(i, j) \in V \times V$ кажемо да је (G, π)-*критичан* уколико важи $(A_G[i][j], A_{\pi(G)}[i][j]) \in \{(1, 0), (1, \star), (\star, 0)\}$. Пар чворова $(i, j) \in V \times V$ је (G, π)-*индикаторски пар* уколико је (G, π)-критичан и за сваки пар чворова $(i', j') < (i, j)$, при чему $i' < j'$, важи један од наредна три услова:

1. $(i', j') \in \{(\pi(i'), \pi(j')), (\pi(j'), \pi(i'))\}$;
2. $A_G[i'][j'] = 1$;
3. $A_{\pi(G)}[i'][j'] = 0$.

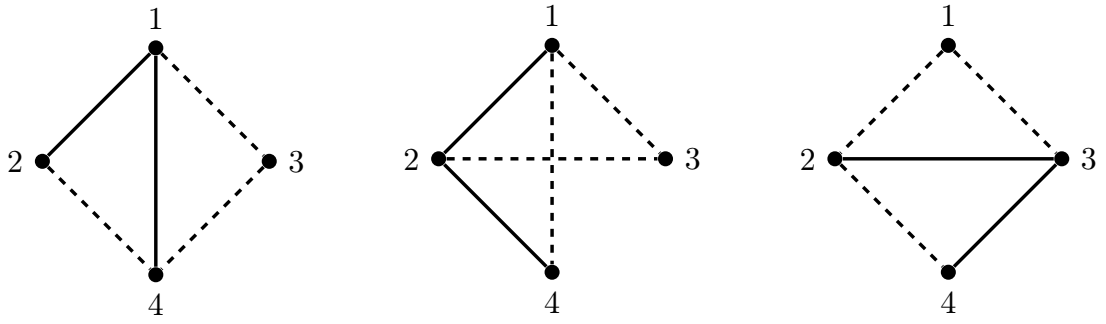
За (G, π)-индикаторски пар (i, j) кажемо да је *строг* уколико важи $A_G[i][j] = 1$ и $A_{\pi(G)}[i][j] = 0$.

Интуитивно, ако је пар чворова (i, j) (G, π)-критичан, то значи да пермутацијом π можемо потенцијално „лексикографски смањити" поље $A_G[i][j]$. Међутим, у том случају, пермутација π не мора нужно водити ка мањим графовима, јер њеном применом можемо „покварити" претходна поља матрице суседства (поља $A_G[i'][j']$, где је (i', j') лексикографски мањи пар од (i, j)). Додатни услови који дефинишу индикаторски пар обезбеђују да, уз то, парови чворова лексикографски мањи од (i, j) нису „покварени" пермутацијом, односно да је пар (i, j) индикатор неминималности графа G. Илуструјмо ове појмове кроз наредни пример. На слици 3.3 приказан је један парцијално

дефинисан граф $G \in \mathcal{P}_4$, као и графови добијени од G пермутацијама:

$$\pi_1 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 3 & 1 \end{pmatrix} \quad \text{и} \quad \pi_2 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \end{pmatrix}.$$

На истој слици су приказане и матрице суседства ових графова. Пар чворова $(1, 4)$ (на слици означен у матрицама суседства) је (G, π_1)-критичан. Приметимо да тај пар није (G, π_1)-индикаторски пар, јер за пар чворова $(1, 3)$ не важе услови 1 - 3. С друге стране, пар $(1, 4)$ јесте (G, π_2)-индикаторски пар (и то строги).



$$A_G = \begin{bmatrix} 0 & 1 & \star & \boxed{1} \\ 1 & 0 & 0 & \star \\ \star & 0 & 0 & \star \\ 1 & \star & \star & 0 \end{bmatrix} \qquad A_{\pi_1(G)} = \begin{bmatrix} 0 & 1 & \star & \boxed{\star} \\ 1 & 0 & \star & 1 \\ \star & \star & 0 & 0 \\ \star & 1 & 0 & 0 \end{bmatrix} \qquad A_{\pi_2(G)} = \begin{bmatrix} 0 & \star & \star & \boxed{0} \\ \star & 0 & 1 & \star \\ \star & 1 & 0 & 1 \\ 0 & \star & 1 & 0 \end{bmatrix}$$

Слика 3.3: Графови G, $\pi_1(G)$ и $\pi_2(G)$ и њихове матрице суседства. Пар $(1, 4)$ није индикаторски пар у односу на π_1, али јесте у односу на π_2.

Важи следећа теорема.

Теорема. *Нека је $G \in \mathcal{P}_n$. Ако постоји строги (G, π)-индикаторски пар за неко $\pi \in \mathcal{S}_n$, онда G није $\preceq$-минималан. Додатно, ако је G потпун граф и није $\preceq$-минималан, онда постоји строги (G, π)-индикаторски пар за неко $\pi \in \mathcal{S}_n$.*

Доказ видети у [8].

3.3 Партиције

Генерализоване уређене партиције (енг. *generalized ordered partitions*) представљају помоћни објекат који служи за одређивање пермутација и индикаторских парова. Формално, то је листа $[(V_1, l_1, u_1), \ldots, (V_k, l_k, u_k)]$ где су V_i, $i \in \{1, \ldots, k\}$ дисјунктни скупови за које важи $V_1 \cup \cdots \cup V_k = V$, а l_i и u_i, $i \in \{1, \ldots, k\}$, такви да важи $l_1 = 1$ и $u_k = n$, затим $u_i + 1 = l_{i+1}$ за свако $i \in \{1, \ldots k-1\}$, као и $|V_i| = u_i - l_i + 1$, за свако $i \in \{1, \ldots, k\}$.

Генерализоване уређене партиције (које ћемо звати и само *партиције*) се користе за репрезентацију „парцијално дефинисаних пермутација" чворова графа. Прецизније, оне одређују фамилију пермутација, коју на одређени начин можемо инкрементално сужавати. На тај начин постепено дефинишемо пермутацију чворова. Формално, са $\mathrm{Perm}(P)$ означавамо све пермутације које асоцирамо са партицијом P: ако је функција $f : V \to \{1, \ldots, n\}$ таква да важи $f(v) = i$ ако и само ако $v \in V_i$, онда је

$$\mathrm{Perm}(P) = \{\pi \in \mathcal{S}_n \mid l_{f(v)} \leq \pi(v) \leq u_{f(v)}, \text{ за свако } v \in V\}.$$

На пример, ако је $P = [(\{1,5\}, 1, 2), (\{4\}, 3, 3), (\{2,3,6\}, 4, 6)]$, онда је $\mathrm{Perm}(P)$ скуп свих пермутација π за које важи $\pi(1), \pi(5) \in \{1, 2\}$, $\pi(4) = 3$ и $\pi(2), \pi(3)$, $\pi(6) \in \{4, 5, 6\}$. Приметимо да је $\mathrm{Perm}\left([(\{1, \ldots, n\}, 1, n)]\right) = \mathcal{S}_n$, за свако $n \in \mathbb{N}$.

Нека је P генерализована уређена партиција која има наредни облик:

$$[\ldots, (V_{i-1}, l_{i-1}, u_{i-1}), (V_i, l_i, u_i), (V_{i+1}, l_{i+1}, u_{i+1}), \ldots]$$

Кажемо да је партиција P' добијена од P *раздвајањем* V_i на V_i' и V_i'', где су V_i' и V_i'' дисјунктни скупови и важи $V_i' \cup V_i'' = V_i$, ако P' има облик

$$[\ldots, (V_{i-1}, l_{i-1}, u_{i-1}), (V_i', l_i, l_i + |V_i| - 1), (V_i'', l_i + |V_i|, u_i), (V_{i+1}, l_{i+1}, u_{i+1}), \ldots].$$

У овој дефиницији дозвољавамо да неки од скупова V_i' и V_i'' буде празан. У том случају дефинишемо да је $P' = P$. Ако је P партиција, где је V_i скуп у некој од њених тројки, и ако је $V_i' \subseteq V_i$, онда ћемо са $P[V_i \to V_i']$ означавати пермутацију добијену од P раздвајањем скупа V_i на V_i' и $V_i \setminus V_i'$. Јасно је да важи $\mathrm{Perm}(P[V_i \to V_i']) \subseteq \mathrm{Perm}(P)$. При томе, једнакост важи уколико је $V_i' = \emptyset$ или $V_i' = V_i$. Другим речима, раздвајањем тројки у партицији *сужавамо скуп пермутација* које она одређује. Казаћемо и да *сужавамо партицију* на овај начин.

Индикаторски парови неког графа су одређени не само тим графом, већ и пермутацијом његових чворова. Наиме, за фиксирани граф $G \in \mathcal{P}_n$, пар чворова (i, j) може бити (G, π_1)-индикаторски пар за неку пермутацију $\pi_1 \in \mathcal{S}_n$, а да није (G, π_2)-индикаторски пар, за неку другу пермутацију $\pi_2 \in \mathcal{S}_n$ (ово је илустровано на слици 3.3). Према томе, потрага за индикаторским паром у једном парцијалном графу је у неком смислу претрега скупа пермутација $\mathcal{S}_n$. Партиције и њихово сужавање управо представљају кључан механизам за инкрементално формирање пермутације у оквиру алгоритма провере минималности графа. О томе ће бити више речи у наредном поглављу.

3.4 Опис алгоритма

У овом поглављу разматрамо алгоритам за проверу минималности графа који је описан у раду [8]. Он је заснован на рекурзивној процедури MinCheck. Ова процедура користи генерализовану уређену партицију скупа чворова графа G и њеним систематским трансформацијама тражи (G, π)-индикаторске парове, при чему су валидне пермутације π одређене том партицијом. Уколико процедура не врати ниједан индикаторски пар, не можемо ништа закључити о графу (што значи да једноставно настављамо са претрагом). Уколико процедура пронађе индикаторски пар, генерише се одговарајућа клауза [8]. У оригиналном раду [8], ова клауза је прослеђивана SAT решавачу као клауза коју решавач треба да научи. У контексту SMT решавача, та клауза се користи за закључивање конфликта у теорији или за теоријску пропагацију. Процедуру MinCheck и генерисање клауза детаљније описујемо у наставку.

Процедура MinCheck

Процедура је заснована на сужавању партиција раздвајањем тројки унутар њих (поглавље 3.3). Прво ћемо дати неформалан опис процедуре. На почетку имамо партицију $[(V, 1, n)]$ која представља све могуће пермутације и почињемо претрагу од првог реда матрице суседства. Постепено пролазимо кроз редове матрице суседства, а у сваком реду матрице, текућа партиција се сужава фиксирањем чворова при индукованим пермутацијама (на основу вредности матрице суседства у текућем реду), све док не пронађемо неки индикаторски пар. Ако систематски исцрпимо све могућности и при томе не пронађемо индикаторски пар, излаз из процедуре је `nil`.

Сада ћемо формално описати алгоритам ове процедуре. Улаз у процедуру представљају парцијално дефинисани граф $G \in \mathcal{P}_n$ (то јест његова матрица суседства), генерализована уређена партиција P и текући ред r у матрици суседства ($r \in \{1, \ldots, n\}$). Излаз из процедуре је (G, π)-индикаторски пар уколико је пронађен, заједно са пермутацијом π за коју је пронађен или `nil` уколико није пронађен индикаторски пар.

Приликом позива процедуре за дато r, разматрамо r-ти скуп чворова у партицији, V_r. За свако $v \in V_r$ сужавамо партицију P тако што фиксирамо $\pi(v) = r$. Прецизније, трансформишемо P у партицију P_v тако да за све пермутације $\pi \in \mathrm{Perm}(P_v)$ важи $\pi(v) = r$. Ову трансформацију ћемо описати кроз посебну процедуру, Adapt. Уколико при тој трансформацији закључимо да имамо индикаторски пар (i, j) за неку пермутацију из $\mathrm{Perm}(P_v)$, процедура Adapt гарантује да је (i, j) (G, π)-индикаторски пар за било коју пермутацију $\pi \in \mathrm{Perm}(P_v)$ [8], па враћамо (i, j) и π и завршавамо процедуру. У супротном, ако смо добили партицију P_v за коју не можемо закључити да имамо индикаторски пар, онда за ту партицију рекурзивно позивамо MinCheck, за наредни ред у матрици (ред $r + 1$). Уколико тај рекузривни позив резултује индикаторским паром и неком пермутацијом, враћамо тај индикаторски пар и ту пермутацију, а уколико не, прелазимо на следећи чвор $v \in V_r$.

Ако ни за један чвор $v \in V_r$ не пронађемо индикаторски пар, процедура враћа `nil`. Излаз из рекурзије процедуре MinCheck представља ситуацију када смо стигли до последњег реда, односно када је $r = n$. Тада смо стигли до краја матрице и враћамо `nil`.

Псеудокод процедуре MinCheck дат је у виду алгоритма 1. У приказу алгоритма, позив процедуре GetPermutation(P_v) представља одабир произвољне пермутације одређене партицијом P_v.

Процедура Adapt

Улаз у процедуру Adapt, поред графа G (односно његове матрице суседства A_G), партиције P и текућег реда r подразумева и чвор v који желимо да фиксирамо тако да важи $\pi(v) = r$ за сваку пермутацију $\pi \in \mathrm{Perm}(P_v)$. Излаз из процедуре је партиција P_v и, евентуално, (G, π)-индикаторски пар (уколико је пронађен).

На почетку конструкције партиције P_v раздвајамо тројку (V_r, l_r, u_r) на тројке $(\{v\}, l_r, l_r)$ и $(V_r \setminus \{v\}, l_r + 1, u_r)$. Затим се за сваку тројку (V_i, l_i, u_i) од

Алгоритам 1 Алгоритам MinCheck за проверу минималности графа

```
1: procedure CheckMinimality(G)
2:     I, π ← MinCheck(G, [(V, 1, n)], 1)
3:     if I ≠ nil then
4:         C ← GenerateClause(I, π)
5:         return C
6:     end if
7:     return nil
8: end procedure
1: procedure MinCheck(G, P = [(V_1, l_1, u_1), ..., (V_k, l_k, u_k)], r)
2:     if r = n then
3:         return nil
4:     end if
5:     for v ∈ V_r do
6:         P_v, I ← Adapt(G, P, r, v)
7:         if I ≠ nil then
8:             π ← GetPermutation(P_v)
9:             return I, π
10:        end if
11:        I, π ← MinCheck(G, P_v, r + 1)
12:        if I ≠ nil then
13:            return I, π
14:        end if
15:    end for
16:    return nil
17: end procedure
```

$(V_r \setminus \{v\}, l_r + 1, u_r)$ (укључујући и њу) до краја партиције, одређују скупови

$$V_i^0 = \{u \in V_i \mid A_G[v][u] = 0\},$$

$$V_i^\star = \{u \in V_i \mid A_G[v][u] = \star\},$$

и

$$V_i^1 = \{u \in V_i \mid A_G[v][u] = 1\},$$

и на основу тих скупова се врши трансформација партиције P_v у три корака, које описујемо у наставку.

Први корак је да се скуп V_i раздвоји на скупове V_i^0 и $V_i^\star \cup V_i^1$. У другом кораку се разматра скуп $J = \{u \mid A_G[r][u] \neq 0, l_i \leq u < l_i + |V_i^0|\}$. Уколико је овај скуп непразан, онда се процедура зауставља и, уз партицију P_v враћа индикаторски пар (r, j), где је $j = \min J$. У супротном, прелази се на трећи

корак у којем посматрамо преостале индексе $p \in \{l_i + |V_i^0|, \ldots, u_i\}$. За свако p разматрамо вредност $A_G[r][p]$.

- Уколико је $A_G[r][p] = \star$, онда:
  1. уколико $v = r$ и $p \in V_i^\star$, раздвајамо $V_i^\star \cup V_i^1$ на $\{p\}$ и $V_i^\star \cup V_i^1 \setminus \{p\}$ и избацујемо p из $V_i^\star$;
  2. уколико $v = p$ и $r \in V_i^\star$, раздвајамо $V_i^\star \cup V_i^1$ на $\{r\}$ и $V_i^\star \cup V_i^1 \setminus \{r\}$ и избацујемо r из $V_i^\star$;
  3. уколико не важи ни један ни други претходно наведени случај, завршавамо процедуру трансформације и враћамо **nil** на месту индикаторског пара.
- Уколико је $A_G[r][p] = 0$, онда завршавамо процедуру, при чему нисмо пронашли индикаторски пар, па враћамо конструисану партицију P_v и **nil** уместо индикаторског пара.
- Уколико је $A_G[r][p] = 1$, онда посматрамо скуп $V_i^\star$. Уколико тај скуп није празан, онда раздвајамо $V_i^\star \cup V_i^1$ на $V_i^\star$ и V_i^1 и завршавамо процедуру, при чему смо пронашли индикаторски пар - тај индикаторски пар је (r, p). У супротном, уколико је $V_i^\star$ празан скуп, онда разматрамо све индексе p', $p' \in \{p, \ldots, u_i\}$. Ако за неко такво p' важи $A_G[r][p'] \neq 1$, онда завршавамо процедуру, при чему нисмо пронашли индикаторски пар, а ако за све p' важи $A_G[r][p'] = 1$ онда можемо да се зауставимо са разматрањем индекса p.

Псеудокод процедуре Adapt дат је у виду алгоритма 2. За свако V_i, $i \in \{r+1, \ldots, k\}$, означена су поменута три корака трансформације.

Генерисање клаузе

У овом пододељку ћемо описати клаузу која се генерише у случају да је пронађен индикаторски пар (поступак генерисања ове клаузе се спроводи при позиву GenerateClause у алгоритму 1). Претпоставимо да смо за граф $G \in \mathcal{P}_n$ пронашли (G, π)-индикаторски пар (i, j). Уведимо ознаке за наредне скупове:

$$S = \{(i', j') \in V \times V \mid (i', j') \in \{(\pi(i'), \pi(j')), (\pi(j'), \pi(i'))\}\},$$

Алгоритам 2 Алгоритам ADAPT за адаптирање P у P_v

procedure ADAPT($G, P = [(V_1, l_1, u_1), \ldots, (V_k, l_k, u_k)], r,\ v$)
  $P_v \leftarrow P$
  раздвој (V_r, l_r, u_r) на $(\{v\}, l_r, l_r)$ и $(V_r \setminus \{v\}, l_r + 1, u_r)$
  for $i \in \{r+1, \ldots, k\}$ **do**
    $V_i^0 \leftarrow \{u \in V_i \mid A_G[v][u] = 0\}$
    $V_i^\star \leftarrow \{u \in V_i \mid A_G[v][u] = \star\}$
    $V_i^1 \leftarrow \{u \in V_i \mid A_G[v][u] = 1\}$
    раздвој (V_i, l_i, u_i) ▷ први корак
      на $(V_i^0, l_i, l_i + |V_i^0| - 1)$ и $(V_i^\star \cup V_i^1, l_i + |V_i^0|, u_i)$
    $J \leftarrow \{u \mid A_G[r][u] \neq 0, l_i \leq u < l_i + |V_i^0|\}$ ▷ други корак
    if $J \neq \emptyset$ **then**
      $j \leftarrow \min J$
      return P_v, (r, j)
    end if
    for $p \in \{l_i + |V_i^0|, \ldots, u_i\}$ **do** ▷ трећи корак
      if $A_G[r][p] = \star$ **then**
        if $v = r$ и $p \in V_i^\star$ **then**
          раздвој $(V_i^\star \cup V_i^1, p, u_i)$ на $(\{p\}, p, p)$ и $(V_i^\star \setminus \{p\} \cup V_i^1, p+1, u_i)$
          $V_i^\star \leftarrow V_i^\star \setminus \{p\}$
        else if $v = p$ и $r \in V_i^\star$ **then**
          раздвој $(V_i^\star \cup V_i^1, p, u_i)$ на $(\{r\}, p, p)$ и $(V_i^\star \setminus \{r\} \cup V_i^1, p+1, u_i)$
          $V_i^\star \leftarrow V_i^\star \setminus \{r\}$
        else
          return P_v, **nil**
        end if
      else if $A_G[r][p] = 0$ **then**
        return nil
      else if $A_G[r][p] = 1$ **then**
        if $V_i^\star \neq \emptyset$ **then**
          раздвој $(V_i^\star \cup V_i^1, p, u_i)$
            на $(V_i^\star, p, p + |V_i^\star| - 1)$ и $(V_i^1, p + |V_i^\star|, u_i)$
          return P_v, (r, p)
        else if $V_i^\star = \emptyset$ и $A_G[r][p'] \neq 1$, за неко $p' \in \{p, \ldots, u_i\}$ **then**
          return P_v, **nil**
        else
          break
        end if
      end if
    end for
  end for
  return P_v, **nil**
end procedure

$$S_1 = \{(i', j') \in V \times V \mid (i', j') < (i, j), A_G[i'][j'] = 1, (i', j') \notin S\},$$
$$S_2 = \{(i', j') \in V \times V \mid (i', j') < (i, j), A_{\pi(G)}[i'][j'] = 0, (i', j') \notin S\}.$$

Тада је клауза коју треба генерисати:

$$\bigvee_{(i',j')\in S_1} \neg e_{i',j'} \bigvee_{(i',j')\in S_1} e_{\pi^{-1}(i'),\pi^{-1}(j')} \vee \neg e_{i,j} \vee e_{\pi^{-1}(i),\pi^{-1}(j)},$$

где је $e_{u,v}$ ако и само ако $A_G[u][v] = 1$. За текући граф, сви литерали ове клаузе, осим највише једног, су нетачни [8].

Пример извршавања алгоритма

Прикажимо пример извршавања алгоритма за проверу минималности на графу G са матрицом суседства:

$$A_G = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & \star & 1 \\ 0 & 1 & 0 & 1 & \star \\ 0 & \star & 1 & 0 & 1 \\ 0 & 1 & \star & 1 & 0 \end{bmatrix}.$$

У процедуру MinCheck улазимо са партицијом $\big[(\{1,2,3,4,5)\}, 1, 5)\big]$, за вредност $r = 1$ (почињемо од првог реда матрице суседства). Важи $V_r = V_1 = \{1, 2, 3, 4, 5\}$ и разматрамо све чворове $v \in V_1$. За $v = 1$ позивамо процедуру Adapt. У тој процедури, прво раздвајамо тројку $(\{1, 2, 3, 4, 5\}, 1, 5)$ на $(\{1\}, 1, 1)$ и $(\{2, 3, 4, 5\}, 2, 5)$. Партиција постаје:

$$\big[(\{1\}, 1, 1), (\{2, 3, 4, 5\}, 2, 5)\big]$$

Потребно је размотрити све тројке почевши од $(\{2, 3, 4, 5\}, 2, 5)$ до краја партиције. Како је $V_2 = (\{2, 3, 4, 5\}, 2, 5)$ једина таква тројка, само њу разматрамо. Може се лако видети да је:

$$V_2^0 = \{2, 3, 4, 5\}, \quad V_2^\star = \emptyset, \quad V_2^1 = \emptyset.$$

Како је $V_2^\star \cup V_2^1 = \emptyset$, у првом кораку нема раздвајања тројке, односно партиција се не мења. У другом кораку, можемо видети да је $J = \emptyset$, тако да прелазимо на трећи корак. Међутим, како немамо индексе p из трећег корака, и трећи корак је тривијалан. Зато излазимо из процедуре Adapt са

измењеном партицијом и без индикаторског пара. Враћамо се у процедуру MinCheck. Резултат процедуре Adapt је трансформисана партиција

$$P_1 = \big[(\{1\}, 1, 1), (\{2, 3, 4, 5\}, 2, 5)\big].$$

Рекурзивно позивамо процедуру MinCheck за ову партицију и за $r = 2$ (односно прелазимо у наредни ред матрице суседства).

Скуп V_r је у овом случају $V_2 = \{2, 3, 4, 5\}$. Разматрамо чвор $v = 2$ и за тај чвор зовемо процедуру Adapt. У тој процедури, прво раздвајамо $(\{2, 3, 4, 5\}, 2, 5)$ на $(\{2\}, 2, 2)$ и $(\{3, 4, 5\}, 3, 5)$, а онда примењујемо кораке 1-3 над тројком $V_3 = (\{3, 4, 5\}, 3, 5)$. Скуп V_3 се разлаже на скупове:

$$V_3^0 = \emptyset, \quad V_3^\star = \{4\}, \quad V_3^1 = \{3, 5\}.$$

Опет, у првом кораку нема раздвајања (овог пута зато што је $V_3^0 = \emptyset$) и партиција остаје непромењена. Такође, и овде је $J = \emptyset$, те је и овде други корак тривијалан. У трећем кораку, разматрамо индексе $p \in \{3, 4, 5\}$. Узмимо $p = 3$. Како је $A_G[2][3] = 1$ и $V_3^\star \neq \emptyset$, то раздвајамо тројку $(\{3, 4, 5\}, 3, 5)$ на $(\{4\}, 3, 3)$ и $(\{3, 5\}, 4, 5)$. Индикаторски пар је пронађен - то је пар $(2, 3)$. Враћамо текућа партицију:

$$\big[(\{1\}, 1, 1), (\{2\}, 2, 2), (\{4\}, 3, 3), (\{3, 5\}, 4, 5)\big]$$

и пронађени индикаторски пар и завршавамо процедуру. Како је у процедури Adapt пронађен индикаторски пар, рекурзивни позив процедуре MinCheck се такође завршава уз индикаторски пар као резултат.

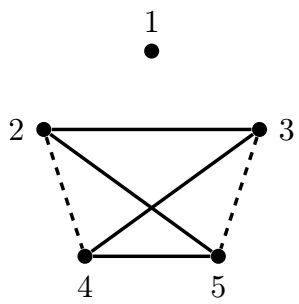
Како је рекурзивни позив вратио индикаторски пар $(2, 3)$, потребно је вратити тај индикаторски пар, заједно са произвољном пермутацијом коју одређује текућа партиција. Како је текућа партиција:

$$\big[(\{1\}, 1, 1), (\{2\}, 2, 2), (\{4\}, 3, 3), (\{3, 5\}, 4, 5)\big],$$

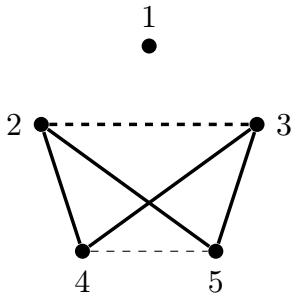
једна таква пермутација је:

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 4 & 3 & 5 \end{pmatrix}.$$

Процедура MinCheck је вратила индикаторски пар $(2, 3)$ и пермутацију π, па на основу тих информација процедуром GenerateClause конструишемо одговарајућу клаузу коју враћамо. Тиме је целокупна процедура завршена. На слици 3.4 представљени су графови G и $\pi(G)$, заједно са својим матрицама суседства.



G



$\pi(G)$

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \boxed{1} & \star & 1 \\ 0 & 1 & 0 & 1 & \star \\ 0 & \star & 1 & 0 & 1 \\ 0 & 1 & \star & 1 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \boxed{\star} & 1 & 1 \\ 0 & \star & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & \star \\ 0 & 1 & 1 & \star & 0 \end{bmatrix}$$

Слика 3.4: Графови G и $\pi(G)$ са својим матрицама суседства. Пронађени индикаторски пар $(2, 3)$ означен је у обе матрице.

Глава 4

Имплементација

У овом поглављу биће описана имплементација[1] алгоритма за проверу минималности графа и уградња истог у решавач ArgoSMT [2]. Решавач ArgoSMT и имплементација алгоритма су отвореног кода и писани су у програмском језику C++. Прво ћемо кратко описати архитектуру на којој почивају модерни SMT решавачи, а онда ћемо описати конкретне релевантне детаље архитектуре решавача ArgoSMT и приказати како се имплементација поменутог алгоритма уклапа у овај контекст.

4.1 CDCL($\mathcal{T}$) архитектура

Модерни SMT решавачи су засновани на CDCL($\mathcal{T}$)[2] архитектури [11]. Она се састоји од SAT решавача заснованог на CDCL алгоритму и теоријског решавача ($\mathcal{T}$-решавача) са којим SAT решавач комуницира.

Улога SAT решавача (зовемо га и SAT језгром) јесте да постепено гради исказну валуацију у циљу испитивања задовољивости исказне апстракције дате формуле. Он, дакле, не узима у обзир семантику теорије $\mathcal{T}$ и резонује искључиво о исказној структури формуле. С друге стране, резоновање у теорији $\mathcal{T}$ је препуштено теоријском решавачу.

Исказна валуација се гради у облику стека литерала која се обично назива *трејл* (енг. *trail*). Литерали на трејлу су подељени на *нивое одлучивања*, при чему сваки ниво одлучивања започиње *литералом одлучивања* (тј. литералом

[1] Путања до GitHub репозиторијума у оквиру којег се налази имплементација: `https://github.com/RobertDoza/argosmt.git`

[2] Овде $\mathcal{T}$ означава неку конкретну теорију првог реда.

чија је вредност произвољно одабрана као претпоставка) који је праћен *пропагираним литералима* (тј. литералима за које је резоновањем утврђено да морају да важе под датим претпоставкама). У случају да текућа парцијална валуација постане неконзистентна са формулом (тј. имамо *конфликт*), SAT решавач врши *анализу конфликта* и *враћање уназад* на неки од претходних нивоа одлучивања, поништавајући претпоставке за које се испоставило да су неконзистентне (као и литерале који су на основу њих пропагирани) [3].

Улога $\mathcal{T}$-решавача је да током конструкције задовољавајуће исказне валуације у оквиру SAT језгра испитује $\mathcal{T}$-задовољивост текуће конјункције литерала првог реда. Уколико $\mathcal{T}$-решавач закључи да је текуће стање литерала незадовољиво у теорији $\mathcal{T}$ (односно, да имамо *теоријски конфликт*), од њега се очекује да може да објасни зашто је то случај, односно да SAT језгру пружи *објашњење конфликта* (у виду конјункције литерала, односно скупа литерала који су у теоријском конфликту). Такође, веома је пожељно да $\mathcal{T}$-решавач уме и да закључује *унапред*, што значи да је у стању да изврши *теоријске пропагације*, односно да на основу до сада установљених литерала закључи да неки други литерал треба да важи. И у том случају је потребно да $\mathcal{T}$-решавач уме да *објасни* пропагацију, односно да издвоји скуп литерала из текуће парцијалне валуације на основу којих је извео тај теоријски закључак. Тај скуп литерала зовемо *објашњењем пропагације.*

Комуникација између SAT решавача и $\mathcal{T}$-решавача се изводи путем интерфејса који најчешће, у неком облику, подразумева наредне методе:

- `newLevel()` – обавештава $\mathcal{T}$-решавач да је SAT решавач започео нови ниво одлучивања;
- `backjump(m)` – обавештава $\mathcal{T}$-решавач да је SAT решавач извршио повратни скок на ниво одлучивања `m`;
- `assert(l)` – обавештава $\mathcal{T}$-решавач да је литерал `l` постављен на трејл;
- `checkConflict(E)` – од $\mathcal{T}$-решавача се тражи да провери да ли постоји конфликт у теорији; уколико постоји, враћа се објашење тог конфликта у виду скупа литерала `E` са трејла;
- `checkPropagate(L)` – $\mathcal{T}$-решавач се пита да ли се може извршити теоријска пропагација; уколико може, теоријски решавач пружа скуп `L` пропагираних литерала;

- `explainLiteral(l, E)` – уколико је литерал `l` изведен у теорији, односно резултат теоријске пропагације, $\mathcal{T}$-решавач враћа објашњење те пропагације у виду скупа литерала `E` који претходе литералу `l` на трејлу.

Рад решавача се завршава или када је парцијална валуација комплетирана без конфликта (у ком случају је формула задовољива) или када се појави конфликт на нултом нивоу одлучивања (у ком случају је формула незадовољива).

4.2 Решавач ArgoSMT

Решавач ArgoSMT [2] подржава рад са теоријом линеарне аритметике и теоријом једнакости (енг. *EUF - equality with uninterpreted functions*). Осим ових, овај решавач подржава и једну специфичну теорију која се у терминологији ArgoSMT решавача назива *CSP теорија*, а која подржава резоновање о ограничењима над коначним доменима. За сваки тип ограничења, теоријски решавач за CSP теорију садржи посебан *руковалац ограничењем* (енг. *constraint handler*) који управља тим типом ограничења. По један овакав руковалац се инстанцира за сваку инстанцу ограничења одговарајућег типа. Улога руковаоца одређеног типа ограничења је да о придруженом ограничењу резонује на одговарајући начин, узимајући у обзир семантику ограничења. Решавач ограничења комуницира са руковаоцем и доставља му ограничења домена променљивих које су релевантне за то ограничење, у виду литерала. Руковалац обрађује ова ограничења и извештава решавач о конфликтима и пропагацијама када до њих дође.

За потребе интеграције алгоритма за проверу минималности графа у решавач ArgoSMT, наша имплементација ће подразумевати увођење новог типа ограничења: *ограничења лексикографске минималности графа* (које ћемо називати `graph_lex_min` ограничење). За овај нови тип ограничења потребно је имплементирати посебан руковалац који ће бити заснован на алгоритму описаном у претходној глави. Ову имплементацију описујемо у наставку.

4.3 Руковалац `graph_lex_min` ограничења

Руковалац лексикографске минималности графа је имплементиран у виду класе

```
class graph_lex_minimal_constraint_handler;
```

Ова класа одржава свој интерни трејл литерала на коме се чувају литерали који представљају ограничења домена релевантних променљивих. Наиме, граф је представљен променљивама $e_{i,j}$ са значењем „постоји грана између чворова i и j", а које имају целобројне домене $\{0, 1\}$. Ограничења ових променљивих (попут $e_{i,j} = 1$ или $e_{i,j} = 0$) се достављају руковаоцу, а он на основу њих води евиденцију о тренутном стању графа. Најзначајније методе које ова класа имплементира су:

- `new_level()` - овом методом класа прави нови ниво одлучивања на интерном трејлу, за потребе каснијих повратних скокова;
- `assert_literal()` - овом методом се руковаоцу достављају релевантни литерали ограничења домена за његов интерни трејл, а који се касније обрађују методом `check_and_propagate`;
- `backjump()` - приликом повратног скока на трејлу теоријског решавача, овом методом се врши повратни скок на интерном трејлу руковаоца;
- `check_and_propagate()` - ова метода служи да анализира тренутно стање трејла и да закључи да ли је дошло до конфликта (и у том случају се теоријском решавачу доставља објашњење тог конфликта) или се може извршити теоријска пропагација (и у том случају се теоријском решавачу доставља резултат те пропагације);
- `explain_literal()` - овом методом руковалац објашњава литерал који је раније пропагиран, односно генерише скуп литерала који су узроковали пропагацију у овкиру руковаоца.

Класа користи инстанцу помоћне класе `GraphState` која служи да чува тренутно стање графа који се током претраге постепено конструише. Конкретно, она чува тренутно стање матрице суседства графа, а уз то памти и историју промена које су се десиле над том матрицом суседства. Матрица суседства је при иницијализацији „празна", то јест на свим позицијама су недефинисане вредности ($\star$), осим на главној дијагонали, на којој су све вредности 0. Током претраге и обраде одговарајућих ограничења, вредности се постепено постављају на 1 или 0, у зависности од тога да ли, према тренутној валуацији, у графу постоји или не постоји грана између одговарајућа два

чвора. Конкретно, одговарајућа вредност у матрици суседства се ажурира у зависности од литерала ограничења домена променљивих које пристижу на трејл, а о чему руковалац бива обавештен од стране решавача. На пример, ако решавач обавести руковалац ограничења да је на трејл постављен литерал $e_{2,3} = 0$, вредност одговарајућег поља матрице суседства $A_G[2][3]$ графа G се поставља на 0. Такође, кад год теоријски решавач обавести руковалац о томе да је на трејлу уведен нови ниво одлучивања, објекат класе `GraphState` у својој историји промена такође креира нови ниво. Сваки пут када се неко поље у матрици суседства постави на неку вредност (на претходно описан начин), то се бележи у историји промена матрице, на текућем нивоу. Приликом повратног скока, класа поништава све промене које су се десиле на свим нивоима одлучивања који се поништавају на трејлу решавача.

Процедура MinCheck имплементирана је у посебном модулу и она се позива из методе `check_and_propagate`. Ако је пронађен индикаторски пар, на основу њега се генерише одговарајућа клауза (пододељак 3.4 описа алгоритма). Та клауза, као што је поменуто, је по тренутном стању графа или поништена у целости, или има тачно један непоништен литерал. У првом случају имамо конфликт и тада се решавачу прослеђује конфликтни скуп, то јест скуп литерала у овој клаузи. У другом случају, једини непоништени литерал може бити недефинисан или је већ тачан на трејлу. Уколико је недефинисан, онда га пропагирамо, а уколико је већ тачан на трејлу, онда га не пропагирамо. При пропагацији литерала, остали литерали (они који су нетачни у клаузи) се користе за генерисање објашњења пропагације (објашњење чини конјункција негација ових литерала). Овај скуп литерала се зато памти у посебној структури `_propagation_explanations`, како би руковалац имао спремно објашњење пропагације када оно затреба решавачу. Стога, ако решавачу затреба објашњење литерала приликом анализе конфликта, он то објашњење може добити од руковаоца путем методе `explain_literal()`, чија се логика своди на једноставно читање из ове структуре и налажење одговарајућег објашњења.

Глава 5

Евалуација

Имплементацију алгоритма описану у претходној глави тестирали смо на проблему генерисања графова са задатим бројем чворова, грана и троуглова (тј. клика величине три). Ове вредности ћемо означавати са n, m и t, редом, и у даљем тексту их називати *параметрима*, односно *спецификацијом* графа.

За потребе експерименталне евалуације, на произвољан начин смо одабрали 50 спецификација за које је број чворова n између 10 и 30. За сваку од спецификација покренули смо решавач на два начина - без додавања и са додавањем ограничења лексикографске минималности, да бисмо могли да упоредимо резултате.

У табели 5.1 приказани су резултати примене решавача над спецификацијама графова за које је $10 \leq n \leq 20$. Све ове спецификације су задовољиве, односно постоје графови са тим бројем чворова, грана и троуглова, редом. Дакле, разматрамо да ли решавач успева и за које време да конструише графове тамо где је то могуће. Максимално време за извршавање које је дато решавачу (енг. *timeout*) је 10 минута по инстанци. У табели су коришћене следеће ознаке:

- n - број чворова у графу;
- m - број грана у графу;
- t - број троуглова у графу;
- t_{csp} - укупно време извршавања решавача који не разматра ограничење минималности;

- n_{confl}^{csp} - број конфликата током извршавања решавача који не разматра ограничење минималности;
- n_{decide}^{csp} - број примена правила одлучивања (енг. *decide*) током извршавања решавача који не разматра ограничење минималности;
- t_{sym} - укупно време извршавања решавача који разматра ограничење минималности графа;
- t_{check} - укупно време извршавања процедуре MinCheck;
- n_{confl}^{sym} - број конфликата током извршавања решавача који разматра ограничење минималности;
- n_{decide}^{sym} - број примена правила одлучивања током извршавања решавача који разматра ограничење минималности;
- n_{prop} - број пропагација руковаоца ограничењем минималности;
- n_{confl} - број конфликата руковаоца ограничењем минималности;
- m_{total} - укупан број позива процедуре MinCheck;
- m_{ind} - број позива процедуре MinCheck који су резултовали индикаторским паром.

Можемо приметити да постоје инстанце код којих је решавач са ограничењем минималности успео за доста мање времена да изврши конструкцију графа него без тог ограничења. Специјално, за $n = 12$, $m = 35$ и $t = 32$, примећујемо да решавач без ограничења минималности чак није ни успео да конструише граф за 10 минута, док је са тим ограничењем успео, за упоредиво много краће време. С друге стране, постоје спецификације на којима је решавач провео мање времена када није узимао у обзир ограничење минималности (на пример, спецификација $n = 12$, $m = 36$, $t = 37$).

У табели 5.2 приказани су резултати експерименталне евалуације над спецификацијама у којима је $21 \leq n \leq 30$ (ознаке колона у овој табели имају идентично значење као у претходној). Као у претходном разматрању, све спецификације одговарају задовољивим проблемима и свакој од њих је дато максимално 10 минута за извршавање. Као што је очекивано, повећавањем траженог броја чворова, добијамо више спецификација у којима није било могуће

извршити конструкцију за дато време. Приметимо да има више спецификација у којима је решавач био спорији када је разматрао ограничење минималности графа. Такође, имамо и инстанце у којима је једна варијанта решавања била значајно бржа од друге, у оба смера (инстанце $(n, m, t) = (21, 30, 15)$ и $(n, m, t) = (27, 30, 6)$). За дубље разумевање добијених резултата, потребно је обавити детаљнију анализу, што може бити предмет даљег рада.

Табела 5.1: Резултати евалуације за $10 \leq n \leq 20$ (времена су дата у секундама)

Параметри			CSP без MinCheck			CSP са MinCheck							
n	m	t	t_{csp}	n_{confl}^{csp}	n_{decide}^{csp}	t_{sym}	t_{check}	n_{confl}^{sym}	n_{decide}^{sym}	n_{prop}	n_{confl}	m_{total}	m_{ind}
10	15	6	**0.033**	7	47	0.076	0.001	25	78	1	0	189	187
10	16	4	**0.070**	101	161	0.181	0.015	560	824	6	3	2612	2610
12	18	3	**0.076**	106	205	0.338	0.017	606	1067	7	4	3176	3174
12	18	4	**0.094**	103	271	0.492	0.022	880	1337	9	6	4209	4207
12	24	2	**0.225**	430	742	1.606	0.103	4746	5654	38	10	21507	21505
12	28	18	**1.462**	3982	12076	5.536	0.472	12707	30642	17	0	80816	80806
12	35	32	-	-	-	**4.124**	0.377	10698	32847	17	1	79379	79377
12	36	37	**6.519**	16127	53952	16.967	1.759	31249	94478	17	0	227576	227574
14	21	1	**0.053**	0	21	0.370	0.014	319	718	4	11	1927	1923
15	14	0	**0.053**	0	14	0.080	0.000	2	21	1	2	48	46
15	15	1	**0.065**	0	67	0.077	0.001	2	52	1	1	83	81
15	16	2	0.091	0	88	**0.090**	0.001	1	124	1	1	209	207
15	17	3	0.086	0	107	**0.062**	0.000	3	39	1	1	55	53
15	18	4	0.086	0	106	**0.085**	0.001	2	44	1	1	68	66
15	28	14	**0.130**	101	227	0.161	0.005	108	297	4	1	746	744
16	15	0	**0.083**	0	15	0.088	0.001	4	33	1	4	74	72
16	16	1	0.090	1	69	**0.075**	0.001	2	50	1	1	76	74
16	17	2	**0.075**	1	70	0.085	0.001	7	69	1	4	103	101
16	18	3	0.096	2	148	**0.083**	0.001	7	66	1	4	105	103
16	19	4	0.098	1	124	**0.084**	0.001	5	87	1	3	123	121
16	24	0	**0.066**	0	24	0.094	0.003	48	124	6	13	327	323
16	30	15	**0.161**	101	243	0.176	0.005	108	248	5	2	656	654
16	32	0	**0.079**	0	32	0.206	0.021	662	833	7	23	2974	2968
20	19	0	**0.124**	0	19	0.143	0.001	3	57	1	3	121	119
20	20	1	**0.149**	0	25	0.166	0.001	1	54	1	0	75	73
20	21	2	**0.141**	1	45	0.152	0.001	1	55	1	0	77	75
20	22	3	1.854	127	2178	**0.157**	0.001	2	72	1	0	105	103
20	23	4	37.042	921	55661	**0.221**	0.002	3	108	1	0	177	175
20	25	4	**0.144**	0	54	3.766	0.025	306	865	4	0	2221	2219
20	28	12	-	-	-	**0.796**	0.014	136	628	3	0	1445	1443
20	38	19	60.649	2414	85419	**0.548**	0.017	106	517	4	3	1174	1132
20	69	0	**0.139**	1	72	0.261	0.018	361	592	9	27	1808	1804

Табела 5.2: Резултати евалуације за $21 \leq n \leq 30$ (времена су дата у секундама)

Параметри			CSP без MinCheck			CSP са MinCheck							
n	m	t	t_{csp}	n^{csp}_{confl}	n^{csp}_{decide}	t_{sym}	t_{check}	n^{sym}_{confl}	n^{sym}_{decide}	n_{prop}	n_{confl}	m_{total}	m_{ind}
21	22	7	-	-	-	-	-	-	-	-	-	-	-
21	30	15	-	-	-	**0.383**	0.005	30	207	1	0	431	429
22	24	0	**0.167**	0	24	0.195	0.005	33	216	4	19	463	459
22	33	9	**1.338**	111	402	8.871	0.074	444	2408	5	1	5293	5291
23	26	2	**0.211**	0	36	6.419	0.013	208	311	2	1	973	971
23	31	14	-	-	-	-	-	-	-	-	-	-	-
24	28	5	**0.473**	2	118	17.512	0.108	674	3214	9	1	7125	7123
24	35	20	-	-	-	-	-	-	-	-	-	-	-
25	29	8	-	-	-	-	-	-	-	-	-	-	-
25	32	17	-	-	-	-	-	-	-	-	-	-	-
26	27	1	**0.329**	0	34	24.820	0.048	478	950	9	23	2710	2702
26	36	11	11.691	214	659	**7.556**	0.018	179	464	5	1	1204	1202
27	30	6	**0.462**	1	66	-	-	-	-	-	-	-	-
27	33	19	-	-	-	-	-	-	-	-	-	-	-
28	32	3	**0.455**	0	47	94.586	0.500	2247	10521	9	22	22968	22966
28	38	12	**0.489**	1	58	5.207	0.024	114	530	3	0	1173	1171
29	34	10	-	-	-	-	-	-	-	-	-	-	-
30	39	18	-	-	-	-	-	-	-	-	-	-	-

Глава 6

Закључци и даљи рад

У овом раду размотрен је алгоритам MinCheck за проверу лексикографске минималности графа, као и његова имплементација у оквиру CDCL($\mathcal{T}$) заснованог решавача ArgoSMT. Овај алгоритам омогућава детектовање индикаторских парова који указују на неминималност графа и омогућава сужавање простора претраге елиминацијом симетрија између изоморфних графова.

Примена разбијања симетрија овим алгоритмом илустрована је на проблему конструкције графова са одређеним бројем чворова, грана и троуглова. Евалуација је показала обећавајуће резултате на одређеним инстанцама овог проблема, али је потребна дубља анализа ових резултата у циљу бољег разумевања предности коришћења овог приступа, као и даљег унапређења и евентуалних оптимизација алоритма MinCheck.

У даљем раду, могао би бити размотрен проблем *енумерације* свих графова који задовољавају ова ограничења, то јест тражење свих графова са датим бројем чворова, грана и троуглова. Елиминација симетрија код овог проблема би била од изузетног значаја јер би се њом, осим повећања ефикасности, обезбедила енумерација искључиво неизоморфних (то јест суштински различитих) графова. Такође бисмо могли разматрати проблеме конструкције графова са другачијим својствима. На пример, како су троуглови уједно и циклуси са 3 чвора и клике са 3 чвора, ограничење броја троуглова у графу би могло бити уопштено - ограничењем броја циклуса одређене дужине или клика одређене величине.

Библиографија

[1] Rolf Backofen and Sebastian Will. Excluding symmetries in constraint-based search. *Constraints*, 7(3):333–349, Jul 2002.

[2] Milan M Banković. Унапређивање SMT решавача коришћењем CSP техника и техника паралелизације. *Универзитет у Београду*, 2016.

[3] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, second edition, 2021.

[4] Michael Codish, Alice Miller, Patrick Prosser, and Peter J. Stuckey. Constraints for symmetry breaking in graph representation. *Constraints*, 24(1):1–24, Jan 2019.

[5] Jo Devriendt, Bart Bogaerts, Broes De Cat, Marc Denecker, and Christopher Mears. Symmetry propagation: Improved dynamic symmetry breaking in sat. In *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*, volume 1, pages 49–56, 2012.

[6] Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. In Toby Walsh, editor, *Principles and Practice of Constraint Programming — CP 2001*, pages 93–107, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[7] Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Chapter 10 - symmetry in constraint programming. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 329–376. Elsevier, 2006.

[8] Markus Kirchweger and Stefan Szeider. SAT Modulo Symmetries for Graph Generation. In Laurent D. Michel, editor, *27th International Conference*

on Principles and Practice of Constraint Programming (CP 2021), volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 34:1–34:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[9] Olga Ohrimenko, Peter J Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

[10] Jean-François Puget. Symmetry breaking using stabilizers. In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003*, pages 585–599, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[11] Cesare Tinelli and Clark Barrett. Smt solvers: Techniques and applications. In Edmund M. Clarke, Thomas A. Henzinger, Helmut Veith, and Roderick Bloem, editors, *Handbook of Model Checking*, pages 305–343. Springer, 2016.

Биографија аутора

Роберт Дожа, рођен 15. новембра 1998. године у Београду, је мастер студент студија на Математичком факултету Универзитета у Београду. Основне студије завршио је 2023. године. Те године је био демонстратор на предмету *Програмирање 1*. Исте године је био учесник студентске летње истраживачке праксе Математичког института Српске академије наука и уметности. Роберт је од 2023. године запослен као сарадник у настави у оквиру Катедре за рачунарство и информатику Математичког факултета, где је изводио наставу вежби на предметима *Рачунарски системи* и *Програмирање база података*.