

УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ



Милан Митрески

ГЕНЕРИСАЊЕ DRAT/LRAT ДОКАЗА ЗА
CDCL SAT РЕШАВАЧЕ И ЊИХОВА
ПРИМЕНА НА КОМБИНАТОРНЕ
ПРОБЛЕМЕ

мастер рад

Београд, 2026.

Ментор:

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

Чланови комисије:

др Предраг ЈАНИЧИЋ, редовни професор
Универзитет у Београду, Математички факултет

др Милан БАНКОВИЋ, доцент
Универзитет у Београду, Математички факултет

Датум одбране: 28. август 2026.

Мами – мојој највећој подршци

Наслов мастер рада: Генерисање DRAT/LRAT доказа за CDCL SAT решаваче и њихова примена на комбинаторне проблеме

Резиме: SAT решавачи имају широку употребу, услед чега је њихова верификација од кључног значаја. Решаваче је могуће верификовати статички, верификацијом њиховог изворног кода, или динамички, верификацијом добијених резултата. У случају задовољивости, верификација се може извршити валуацијом која задовољава почетну исказну формулу. Овакве валуације је једноставно генерисати и употребити за верификацију. У случају незадовољивости, верификација се може извршити помоћу доказа незадовољивости који се могу генерисати приликом извршавања SAT решавача који се заснивају на CDCL алгоритму.

Циљ овог рада је представљање различитих формата доказа незадовољивости уз имплементацију одговарајућег решавача применом CDCL алгоритма (који ће такође бити детаљно описан) и примена доказа незадовољивости на разне комбинаторне проблеме.

Кључне речи: SAT проблем, SAT решавачи, CDCL алгоритам, докази незадовољивости, DRAT докази незадовољивости, LRAT докази незадовољивости

Садржај

1	Увод	1
1.1	Исказна логика	2
1.2	SAT проблем и SAT решавачи	4
2	Алгоритми за решавање SAT проблема	6
2.1	Метод резолуције	6
2.2	DP процедура	7
2.3	DPLL процедура	7
2.4	CDCL процедура	11
3	Докази незадовољивости	17
3.1	Приступи верификацији SAT решавача	17
3.2	Резолуцијски докази незадовољивости	18
3.3	Верификација DRAT/LRAT доказа	23
3.4	Генерисање DRAT/LRAT доказа	24
4	Имплементација	28
4.1	CDCL SAT решавач	28
4.2	Генерисање DRAT доказа	46
4.3	Генерисање LRAT доказа	48
5	Примена на комбинаторне проблеме	51
5.1	Дирихлеов принцип	52
5.2	Бојење графа	55
6	Закључак	59
	Библиографија	60

Глава 1

Увод

Проблем исказне задовољивости, познатији као SAT проблем (енгл. *propositional satisfiability problem*) представља један од централних проблема теоријског рачунарства, који је своју практичну примену нашао као алат за формализацију великог броја реалних проблема укључујући верификацију хардвера и софтвера, планирање, распоређивање и комбинаторне проблеме.

Први значајан резултат који се односи на SAT проблем јесте доказ његове NP-комплетности, који је Кук¹ представио 1971. године [3]. Каснији радови и истраживања довели су до развоја проблема задовољивости ограничења (енгл. *constraint satisfiability problem* – CSP), проблема задовољивости квантификованих исказних формула као и проблема задовољивости у теоријама првог реда (енгл. *satisfiability modulo theory* – SMT).

При изради SAT решавача, софтверских алата који решавају SAT проблем, посебну пажњу треба посветити њиховој *верификацији*. Један од приступа подразумева статичку верификацију SAT решавача односно изворног кода, док други приступ подразумева генерисање одређене врсте *доказа* задовољивости односно незадовољивости дате формуле. У случају задовољивих формула, доказ је једноставно генерисати у виду валуације која представља модел дате формуле, док је у случају незадовољивих формула ситуација комплекснија – потребно је прво дефинисати шта представља доказ незадовољивости, а потом и описати процедуру за генерисање таквог доказа.

У наставку биће приказани основни појмови исказне логике и SAT проблема, у највећој мери преузети из [2, 11] где се може наћи још детаља.

¹Стивен Артур Кук (енгл. Stephen Arthur Cook), 1939, амерички информатичар.

1.1 Исказна логика

Исказна логика (енгл. *propositional logic*) изучава **исказе** односно **исказне формуле** и њихове особине. Исказна логика се састоји из три аспекта, *синтаксе исказне логики* односно језика, семантике исказне логики односно значења исказа и *дедуктивних система исказне логики* односно система за закључивање.

Синтакса исказне логики

При дефинисању синтаксе исказне логики, претпостављамо да нам је дат највише пребројив скуп Var чији су елементи *логичке променљиве*, које називамо и *исказна слова* односно *атоми*.

Азбука исказне логики у ознаци Σ_{Var} састоји се од *логичких променљивих* односно елемената скупа Var , *логичких константи* $\top$ и $\perp$, *унарној логичкој везника* $\neg$, *бинарних логичких везника* $\wedge, \vee, \Rightarrow, \Leftrightarrow$ и *помоћних симбола* (и).

Скуп **исказних формула** у ознаци $\mathcal{F}_{\text{Var}}$ је најмањи скуп (у смислу инклузије) речи над Σ_{Var} такав да:

1. логичке константе и логичке променљиве су исказне формуле
2. ако су F и G исказне формуле, тада су и $(\neg F)$, $(F \wedge G)$, $(F \vee G)$, $(F \Rightarrow G)$, $(F \Leftrightarrow G)$ такође исказне формуле.

Овако дефинисан скуп исказних формула представља **језик** односно **синтаксу исказне логики**. Формуле наведене у делу 2. претходне дефиниције редом називамо *негацијом исказне формуле*, *конјункцијом исказних формула*, *дисјункцијом исказних формула*, *импликацијом исказних формула* односно *еквиваленцијом исказних формула*. Једноставности ради уводимо и дефиницију *литерала* који представља исказну формулу која је или логичка променљива (*позитиван поларитет* литерала) или негација логичке променљиве (*негативан поларитет* литерала).

Ако другачије није наглашено, атоме и литерале означаваћемо малим латиничним словима абецеде ($p, q, r, w, l \dots$) док ћемо сложене исказне формуле означавати великим латиничним словима абецеде ($F, G, H, A, B, C \dots$).

Семантика исказне логице

Синтакса исказне логице дефинише искључиво структуру исказних формула, али не и њихово значење, односно начин како исказне формуле треба *интерпретира́ти*. Природно је исказне формуле, а првенствено исказна слова, интерпретирати *истинитосним вредностима*.

У складу са рекурзивном дефиницијом скупа исказних формула, прво је потребно доделити значење логичким променљивим и логичким константама, а потом и исказним формулама које добијамо применом логичких везника.

Валуација представља функцију $v: \text{Var} \rightarrow \{0, 1\}$ односно доделу истинитосних вредности 0 (**нетачно**) и 1 (**тачно**) логичким променљивим.

Интерпретација за валуацију v представља функцију $\text{Int}_v: \text{Var} \rightarrow \{0, 1\}$ такву да:

1. $\text{Int}_v(\top) = 1$, $\text{Int}_v(\perp) = 0$ и $\text{Int}_v(p) = v(p)$ за све атоме $p \in \text{Var}$
2. $\text{Int}_v(\neg F) = 1$ за $\text{Int}_v(F) = 0$, односно $\text{Int}_v(\neg F) = 0$ за $\text{Int}_v(F) = 1$
3. вредности интерпретација дисјункције, конјункције, импликације и еквиваленције две исказне формуле дате су истинитосном таблицом

$\text{Int}_v(F)$	$\text{Int}_v(G)$	$\text{Int}_v(F \wedge G)$	$\text{Int}_v(F \vee G)$	$\text{Int}_v(F \Rightarrow G)$	$\text{Int}_v(F \Leftrightarrow G)$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

За дату валуацију v , интерпретација Int_v свакој исказној формули додељује јединствену истинитосну вредност. За формулу F кажемо да је *тачна у валуацији* v ако је истинитосна вредност $\text{Int}_v(F) = 1$ и пишемо $v \models F$, док у супротном кажемо да је *нетачна у валуацији* v и пишемо $v \not\models F$.

Од значаја је испитати семантичке особине исказних формула које не зависе од избора валуације. За исказну формулу кажемо да је **таутологија** ако је тачна у свакој валуацији, док за исказну формулу кажемо да је **задовољива** ако је тачна у некој валуацији. Са друге стране, за исказну формулу кажемо да је **контрадикција** ако је нетачна у свакој валуацији, док за исказну формулу кажемо да је **порецива** ако је нетачна у некој валуацији.

За две исказне формуле F и G кажемо да су **логички еквивалентне** ако за сваку валуацију v важи $v \models F$ ако и само ако $v \models G$, а кажемо да су

логички еквивалентне уколико је F задовољива ако и само ако је и G задовољива.

1.2 SAT проблем и SAT решавачи

Проблем испитивања задовољивости произвољне исказне формуле назива се **SAT проблем**, док се програм који решава овај проблем назива **SAT решавач**. Лако је показати да је SAT проблем за *коначне* исказне формуле *одлучив*, односно да је у коначном времену могуће одредити да ли је нека исказна формула задовољива или не. Један од алгоритама који решава SAT проблем у коначном времену је *метод истинитосних таблица* [11], који је поприлично временски неефикасан.

Модерни SAT решавачи по правилу на улазу очекују формулу у **конјунктивној нормалној форми (CNF)**. За формулу кажемо да је у конјунктивној нормалној форми ако се може представити у облику

$$\bigwedge_{i=1}^m \bigvee_{j=1}^{n_i} \ell_{i,j}$$

где су $\ell_{i,j}$ литерали. Дисјункције литерала називамо *клаузама* и можемо рећи да је формула у CNF уколико је представљена као конјункција клауза. Испоставља се да је алгоритам трансформације произвољне исказне формуле у њој логички еквивалентну CNF исказну формулу неефикасан, односно експоненцијалне временске сложености. Међутим, узевши у обзир природу SAT проблема, довољно је при трансформацији захтевати само *еквивалентност* и испоставља се да је могуће трансформисати исказну формулу у њој еквивалентну CNF исказну формулу у полиномској временској сложености [2]. Надаље ћемо сматрати да су исказне формуле које разматрамо дате у конјунктивној нормалној форми при чему ћемо формуле представљати *скупојима клауза*, при чему клаузе представљају *скупоје литерала*. Софтверска репрезентација исказних формула у конјунктивној нормалној форми, коју SAT решавачи прихватају на улазу је DIMACS CNF формат².

С обзиром на то да SAT проблем припада класи NP-комплетних проблема, претпоставља се да не постоји алгоритам полиномске сложености који решава

²Опис DIMACS CNF формата може се наћи на <https://jix.github.io/varisat/manual/0.2.0/formats/dimacs.html>

SAT проблем. Сви до сада познати алгоритми који решавају SAT проблем, у најгорем могућем случају раде у експоненцијалној временској сложености. Зато се при конструкцији неких SAT решавача користе разне технике оптимизације (вођене хеуристикама) и правила *закључивања* односно *резоновања*. Правилима закључивања можемо одсећи више делова простора претраге, у зависности од структуре исказне формуле, односно датог скупа клауза, чиме знатно убрзавамо претрагу у одређеним случајевима. Иако се овим проширењима временска сложеност SAT решавача у најгорем случају не побољшава, показало се да овакав приступ знатно побољшава перформансе решавача, јер се у пракси ретко сусрећемо са *тешким инстанцама*, односно исказним формулама за које решавач ради у експоненцијалној временској сложености. Побољшање перформанси је уочљиво на оним инстанцама за које елементи закључивања знатно смањују простор претраге.

Глава 2

Алгоритми за решавање SAT проблема

Развијање SAT решавача који укључују елементе закључивања кренуло је од DP (енгл. *Davis¹-Putnam²*) процедуре из 1960. године, касније унапређену у DPLL (енгл. *Davis-Putnam-Logemann³-Loveland⁴*) процедуру, која представља основу за **CDCL** (енгл. *Conflict-Driven Clause Learning*)) алгоритам на ком се заснивају модерни SAT решавачи.

2.1 Метод резолуције

Метод резолуције подразумева итеративно проширивање скупа клауза новим клаузама изведеним применом *правила резолуције*:

$$\frac{C_1 \cup \{\ell\}, C_2 \cup \{\bar{\ell}\}}{C_1 \cup C_2} \text{ Res}$$

где су C_1, C_2 клаузе, ℓ је литерал, а $\bar{\ell}$ је литерал супротан литералу ℓ , а клаузу $C_1 \cup C_2$ називамо *резолвентом*. Резолвенту добијену применом правила резолуције над клаузама C' и C'' означавамо са $C' \otimes C''$. Додатно, уколико се правило резолуције спроводи над литералом ℓ , односно ако $\ell \in C'$, $\bar{\ell} \in C''$,

¹Мартин Дејвид Дејвис (енгл. Martin David Davis), 1928 - 2023, амерички математичар и информатичар.

²Хилари Вајтхол Путнам (енгл. Hilary Whitehall Putnam), 1926 - 2016, амерички филозоф, математичар и информатичар.

³Џорџ Вал Лоџман (енгл. George Wahl Logemann), 1938 - 2012, амерички математичар и информатичар.

⁴Доналд В. Лавленд (енгл. Donald W. Loveland), 1934, амерички информатичар.

тада резолвенту означавамо са $C' \otimes_{\ell} C''$. Уколико метод резолуције изведе празну клаузу, почетни скуп клауза је незадовољив. Са друге стране, уколико након исцрпне и систематичне примене правила резолуције, метод резолуције не изведе празну клаузу, почетни скуп клауза је задовољив. Докази ових тврђења, којим се утврђује коректност метода резолуције, могу се наћи у [11].

2.2 DP процедура

DP процедура заснива се на *правилу резолуције*, које се користи за елиминацију променљивих, уз два правила закључивања: *пропагације јединичних литерала* (енгл. *unit literal propagation*) и *елиминације чистих литерала* (енгл. *pure literal elimination*).

У сваком кораку DP процедуре, једна од променљивих p_i се елиминише, или применом правила закључивања или тако што се на сваке две клаузе, од којих једна садржи p_i , а друга $\neg p_i$, примени правило резолуције. Уколико овим поступком изведемо празну клаузу, процедура пријављује незадовољивост, док са друге стране, у случају елиминације свих клауза, процедура пријављује задовољивост. С обзиром на то да DP процедура у својој основи не садржи алгоритам претраге, у случају задовољивог скупа клауза, процедура неће генерисати валуацију која задовољава дати скуп. Међутим, значај DP процедуре представљају поменута правила закључивања, која су саставни део DPLL алгоритма. Ова правила биће детаљно описана у наставку.

2.3 DPLL процедура

DPLL процедура представља алгоритам за испитивање задовољивости исказне формуле, који се састоји из два дела – **DPLL претрага** и два правила закључивања: пропагацију јединичних литерала и елиминацију чистих литерала. Улаз DPLL процедуре представља скуп клауза, док се на излазу очекује *unsatisfiable* у случају незадовољивог скупа клауза тј. *satisfiable* и одговарајућа валуација у случају задовољивог скупа клауза.

DPLL претрага

DPLL претрага представља систематичну претрагу задовољавајуће валу-

ације за дати скуп клауза и заснива се на кораку *split* [5] који подразумева испитивање обе истинитосне вредности атома који се појављују у клаузама. Улаз и излаз DPLL претраге идентични су раније описаном улазу и излазу DPLL процедуре.

Претрага индуктивно конструише *парцијалне валуације* односно непотпуне валуације, где је истинитосна вредност додељена неким атомима, док је за остале атоме истинитосна вредност непозната. На почетку претраге, крећемо од празне парцијалне валуације где ниједном атому није додељена истинитосна вредност. Извршавање алгорита претраге може се приказати следећим псеудо-кодом [2]:

```

1 FUNCTION dpll_search(clauses, partial_valuation):
2
3     IF evaluate(clauses, partial_valuation) IS TRUE:
4         RETURN satisfiable, complete_valuation(clauses, partial_valuation)
5
6     IF evaluate(clauses, partial_valuation) IS FALSE:
7         RETURN unsatisfiable, NULL
8
9     LET atom = get_unassigned_atom(clauses, partial_valuation)
10
11     set_atom_value(partial_valuation, atom, TRUE)
12     LET result, valuation = dpll_search(clauses, partial_valuation)
13     IF result IS satisfiable:
14         RETURN satisfiable, valuation
15
16     set_atom_value(partial_valuation, atom, FALSE)
17     RETURN dpll_search(clauses, partial_valuation)

```

У сваком кораку претраге, испитујемо да ли до тада изграђена парцијална валуација задовољава или не задовољава дати скуп клауза (*напомена: истинитосна вредност скупа клауза за парцијалну валуацију може бити непозната*). У случају задовољивости, допуњујемо парцијалну валуацију до потпуне валуације на произвољан начин (на пример, додељивањем истинитосне вредности тачно променљивама којима истинитосна вредност није додељена) и враћамо *satisfiable*, док у случају незадовољивости враћамо *unsatisfiable*. Ако је истинитосна вредност скупа клауза непозната, настављамо претрагу,

применом *split* правила за атом чија је истинитосна вредност непозната. Уколико једна од грана врати задовољавајућу вредност (заједно са валуацијом), ту вредност пропагирамо на горе. У супротном уколико обе гране врате незадовољивост, пропагирамо незадовољивост.

Правила закључивања

Приликом извршавања DPLL претраге, за скупове клауза са одређеним својствима је могуће на основу правила исказне логике закључити да нека од грана добијених *split* правилом враћа незадовољивост, без покретања претраге над том граном. Могућност примене оваквих правила у неком кораку, које називамо *правилима закључивања*, зависи од претходних корака, односно парцијалне валуације изграђене до тог корака – правило закључивања се можда не може применити на почетни скуп клауза, док се при одређеној парцијалној валуацији скуп клауза може трансформисати у скуп клауза над којим се неко од правила може применити. DPLL процедура се састоји од два правила закључивања: **пропагације чистих литерала** и **елиминације јединичних клауза** [5].

Пропагација чистих литерала

Пропагација чистих литерала (*eng. pure literal propagation*) може се применити уколико постоји атом који се појављује искључиво у једном поларитету у незадовољеним клаузама скупа, а да му притом није додељена истинитосна вредност, у односу на до тада изграђену парцијалну валуацију. Тада је могуће вредност таквог атома поставити на тачно уколико се појављује у позитивном поларитету, тј. нетачно у супротном. Уколико је дат скуп клауза

$$\{p, q, r\}, \{p, \neg q \neg s\}, \{q, \neg r, s\}, \{\neg q, r, \neg s\}, \{\neg p, t\}$$

при чему је атому t додељена позитивна истинитосна вредност, а атому p није додељена истинитосна вредност, тада је могуће применити пропагацију чистих литерала и поставити вредност атома p на тачно. Даља примена правила пропагације чистих литерала није могућа.

Пропагација чистих литерала не искључује у потпуности једну од грана добијених применом правила *split*, већ наставља претрагу граном којом је лакше доћи до задовољивости (применом правила пропагације чистих литерала одређени број клауза се задовољава, а притом ниједној другој клаузи

није одсечен простор претраге у ком се може доћи до задовољиве валуације за ту клаузу). Приметимо да у случају да изабрана грана врати незадовољивост, незадовољивост би вратила и одсечена грана.

Елиминација јединичних клауза

Елиминација јединичних клауза (*eng. unit clause elimination*) може се применити над скупом клауза у ком постоји незадовољена клауза која садржи тачно један литерал коме није додељена истинитосна вредност, у односу на до тада изграђену парцијалну валуацију. Тада се истинитосна вредност тог литерала **мора** поставити на тачно, јер би у супротном истинитосна вредност целе клаузе била нетачно, чиме би и цео скуп клауза постао незадовољив. Уколико је дат скуп клауза

$$\{\neg p, u\}, \{p, \neg q, r\}, \{\neg p, s, \neg t\}$$

при чему је атому u додељена негативна истинитосна вредност, а атому p није додељена истинитосна вредност, тада је могуће применити елиминацију јединичних клауза над клаузом $\{\neg p, u\}$ и поставити вредност атома p на нетачно. Даља примена правила елиминације јединичних клауза није могућа.

Приметимо да, за разлику од пропагације чистих литерала, правило елиминације јединичних клауза у потпуности искључује једну од грана добијених применом правила *split* и наставља претрагу неодсеченом граном.

Алгоритам DPLL процедуре

Алгоритам DPLL процедуре представља проширење алгоритма DPLL претраге, при чему се пре примене правила *split* над наредним атомом ком није додељена истинитосна вредност, прво испитује да ли је над тренутним скупом клауза могуће применити неко од правила закључивања. Извршавање алгоритма DPLL процедуре може се приказати наредним псеудо-кодом:

```

1 FUNCTION dpll_procedure(clauses, partial_valuation):
2
3     IF evaluate(clauses, partial_valuation) IS TRUE:
4         RETURN satisfiable, complete_valuation(clauses, partial_valuation)
5
6     IF evaluate(clauses, partial_valuation) IS FALSE:
```

```
7      RETURN unsatisfiable, NULL
8
9      LET atom, value = get_pure_literal(clauses, partial_valuation)
10     IF atom IS NOT NULL:
11         set_atom_value(partial_valuation, atom, value)
12         RETURN dpll_procedure(clauses, partial_valuation)
13
14     atom, value = get_unit_clause(clauses, partial_valuation)
15     IF atom IS NOT NULL:
16         set_atom_value(partial_valuation, atom, value)
17         RETURN dpll_procedure(clauses, partial_valuation)
18
19     atom = get_unassigned_atom(clauses, partial_valuation)
20
21     set_atom_value(partial_valuation, atom, TRUE)
22     LET result, valuation = dpll_search(clauses, partial_valuation)
23     IF result IS satisfiable:
24         RETURN satisfiable, valuation
25
26     set_atom_value(partial_valuation, atom, FALSE)
27     RETURN dpll_search(clauses, partial_valuation)
```

2.4 CDCL процедура

У односу на DPLL процедуру, CDCL процедура оптимизује процес претраге *анализирањем конфликта* односно закључивањем услед чега и у ком кораку је дати скуп клауза постао незадовољив – ово се не мора нужно десити у последњем кораку тј. приликом последње доделе истинитосне вредности неком атому већ се може десити у неком ранијем кораку, приликом неке раније доделе. Анализа конфликта подразумева извођење клаузе која је логичка последица почетног скупа клауза и која указује на разлог зашто је при тренутној парцијалној валуацији дошло до конфликта и у ком кораку се то десило. Ово нам омогућава да се вратимо на тај корак који је суштински проблематичан и наставимо претрагу из тог корака, али у другом смеру. За разлику од DPLL процедуре која подразумева враћање на претходни корак,

CDCL процедура омогућава да избегнемо претрагу у простору претраге за који се може закључити да не садржи задовољиву валуацију.

Апстрактни опис CDCL процедуре

Имплементација CDCL процедуре укључује одређени, незанемарљиви део техничких и имплементационих детаља. Један од тих детаља је и *схема два посматрана литерала* која ефикасно испитује задовољивост клауза при датој парцијалној валуацији. Такође, укључене су и разне оптимизације вођене хеуристикама.

Услед компликоване имплементације, разматрање и анализу основа CDCL процедуре је једноставније одрадити над *апстрактним системом за промену стања* [16]. *Стање* у том систему представља уређену тројку (F, M, C) где F представља скуп клауза чија се задовољивост испитује, C представља скуп литерала за анализу конфликта, док M представља до тад изграђену парцијалну валуацију у облику стека индексираних литерала чија је истинитосна вредност тачно, при чему индексе називамо *нивоима* и при томе је сваки литерал означен са d или i у зависности од тога да ли је литералу додељена истинитосна вредност применом правила *split* (тј. нашом *одлуком* (*eng. decide*)) или је истинитосна вредност одрђена применом правила закључивања (уз напомену да се испитивање друге вредности добијене применом корака *split* сматра додељивањем истинитосне вредности применом правила закључивања (рецимо логичким правилом *tertium non datur*). Индекс (ниво) литерала једнак је броју литерала означених са d који се на стеку налазе пре тог литерала, закључно са њим. *Правила за промену стања* одговарају правилу *split*, правилу елиминације јединичних клауза, правилу за детекцију конфликта, правилима за анализу конфликта и правилима за примену оптимизација вођених хеуристикама. Правила су дата у облику

$$\text{Rule name: } \frac{\text{cond}_1 \quad \dots \quad \text{cond}_n}{\text{effect}_1 \quad \dots \quad \text{effect}_m}.$$

где cond_i представљају услове које стање мора задовољити, а effect_i ефекте којим добијамо наредно стање применом правила. Скуп правила за CDCL процедуру је [12]:

$$\begin{aligned}
 \text{Decide: } & \frac{\ell \in L \quad \ell, \bar{\ell} \notin M}{M := M \ell^d}, \\
 \text{UnitProp: } & \frac{\{\ell, \ell_1, \dots, \ell_k\} \in F \quad \bar{\ell}_1, \dots, \bar{\ell}_k \in M \quad \ell, \bar{\ell} \notin M}{M := M \ell^i}, \\
 \text{Conflict: } & \frac{C = \text{no_conflict} \quad \{\bar{\ell}_1, \dots, \bar{\ell}_k\} \in F \quad \ell_1, \dots, \ell_k \in M}{C := \{\ell_1, \dots, \ell_k\}}, \\
 \text{Explain: } & \frac{\ell \in C \quad \{\ell, \bar{\ell}_1, \dots, \bar{\ell}_k\} \in F \quad \ell_1, \dots, \ell_k \prec^M \ell}{C := (C \cup \{\ell_1, \dots, \ell_k\}) \setminus \{\ell\}}, \\
 \text{Learn: } & \frac{C = \{\ell_1, \dots, \ell_k\} \quad \{\bar{\ell}_1, \dots, \bar{\ell}_k\} \notin F}{F := F \cup \{\bar{\ell}_1, \dots, \bar{\ell}_k\}}, \\
 \text{Backjump: } & \frac{C = \{\ell, \ell_1, \dots, \ell_k\} \quad \{\bar{\ell}, \bar{\ell}_1, \dots, \bar{\ell}_k\} \in F \quad \text{level}(\ell) > m \geq \text{level}(\ell_i)}{C := \text{no_conflict} \quad M := M^{[m]} \bar{\ell}^i}, \\
 \text{Forget: } & \frac{C = \text{no_conflict} \quad c \in F \quad F \setminus \{c\} \models c}{F := F \setminus \{c\}}, \\
 \text{Restart: } & \frac{C = \text{no_conflict}}{M := M^{[0]}}.
 \end{aligned}$$

уз следеће значење ознака

- $C = \text{no_conflict}$ – скуп клауза F није незадовољив при парцијалној валуацији M тј. нема конфликта,
- $\ell_1, \dots, \ell_k \prec^M \ell$ – литерали $\ell_1, \dots, \ell_k$ се појављују у парцијалној валуацији, односно на стеку M , пре литерала ℓ ,
- $\text{level}(\ell)$ – ниво (индекс) литерала ℓ у парцијалној валуацији M ,
- $M^{[m]}$ – префикс валуације M , закључно са нивоом (индексом) m .

Почетно стање за дату формулу $\mathcal{F}$ је $(\mathcal{F}, \{\}, \text{no_conflict})$. Формула је задовољива уколико формирамо потпуну валуацију без конфликта, а незадовољива је ако у фази анализе конфликта добијемо празан конфликтни скуп C .

Анализа конфликта

У описаном апстрактном систему, правилима **Conflict**, **Explain**, **Learn** и **Backjump** врши се анализа конфликта на следећи начин

1. Када правилем **Conflict** установимо да конфликт постоји, тј. да постоји клауза чија је истинитосна вредност нетачно, скуп C иницијализујемо

на скуп супротних литерала који формирају клаузу чија је истинитосна вредност нетачно – овај скуп представља скуп литерала чије се истинитосне вредности не могу истовремено поставити на тачно.

2. У скупу C могу постојати литерали чија је истинитосна вредност постављена на тачно елиминацијом јединичне клаузе, која је притом постала једничина након постављања истинитосних вредности осталих литерала те клаузе на нетачно у неким ранијим корацима претраге (ти литерали се налазе на nižем нивоу у односу на литерал из C). За овакве литерале можемо закључити зашто им је истинитосна вредност постављена на тачно па самим тим и зашто се налазе у скупу C – објашњење лежи у додељивању истинитосних вредности литералима које је довело до примене елиминације јединичне клаузе. Правилем **Explain** "мењамо" литерал ℓ из C одговарајућим литералима $\ell_1, \dots, \ell_k$ чијим се постављањем истинитосне вредности на тачно (у ранијим корацима) дошло до постављања истинитосне вредности литерала ℓ на тачно. Приметимо да скуп C и даље представља скуп литерала чије се истинитосне вредности не могу истовремено поставити на тачно, али да клауза која се састоји од супротних литерала из C не налази у скупу F – у супротном би до конфликта дошло раније.
3. Примену правила **Explain** вршимо док се не нађемо у стању које омогућава примену правила **Learn**. У таквом стању, скуп C садржи тачно један литерал са последњег нивоа одлучивања и ову схему називамо **FUIP** (енг. **F**irst **U**nique **I**mplication **P**oint) схемом [20]. Применом правила **Learn**, скупу клауза F додајемо клаузу сачињену од литерала који су супротни литералима из скупа C .
4. На крају, правилем **Backjump** се враћамо на корак у ком можемо применити елиминацију јединичне клаузе над новонаученом клаузом (и ту елиминацију применимо)

Дакле анализом конфликта долазимо до нове клаузе која је логичка последица клауза из F и која је постала незадовољива неколико корака пре него што је дошло до конфликта. Корисно је такву клаузу додати у скуп клауза како би у даљим корацима спречили формирање парцијалне валуације која би исту ту клаузу поново учинила незадовољивом. Овај исход анализе кон-

фликта назива се *учење клауза* и представља кључни корак CDCL процедуре, по ком је та процедура и добила име – *учење клауза вођено конфликтима*.

Псеудо-код CDCL процедуре

CDCL процедура може се представити наредним псеудо-кодом [2]:

```

1 FUNCTION cdcl(clauses, partial_valuation):
2
3     IF unit_propagation(clauses, partial_valuation) IS conflict:
4         RETURN unsatisfiable, NULL
5
6     LET decision_level = 0
7
8     WHILE NOT all_variables_assigned(clauses, partial_valuation) DO:
9         LET (var, val) = pick_branching_variable(clauses,
10                                                    partial_valuation)
11
12         decision_level = decision_level + 1
13         set_atom_value(partial_valuation, var, val)
14
15         IF unit_propagation(clauses, partial_valuation) IS conflict:
16             LET (conflict_clause, backjump_level) =
17                 conflict_analysis(clauses, partial_valuation)
18             add_to_set(conflict_clause, clauses)
19
20             IF backjump_level < 0:
21                 RETURN unsatisfiable, NULL
22             ELSE:
23                 backjump(clauses, partial_valuation, backjump_level)
24                 decision_level = backjump_level
25
26     RETURN satisfiable, partial_valuation
    
```

Напоменимо да се за разлику од DPLL процедуре, пропагација чистих литерала не користи у CDCL процедури. Главни разлог за то јесте некомпатибилност са схемом два посматрана литерала чији је циљ да спречи пролазак кроз све литерале клауза у сваком кораку претраге.

Оптимизација

Модерне имплементације CDCL SAT решавача, укључују и одрђене оптимизације вођене хеуристикама. Иако не доприносе и не утичу на коректност алгорита, улога ових оптимизација јесте да убрзају извршавање алгорита. Оптимизације, између осталог, укључују [2]:

- *пранање вођено хеуристикама* – избор наредне променљиве над којом се примењује правило Decide
- *заборављање научених клауза* – које клаузе ће алгоритам учити и када ће се научене клаузе избацивати из скупа клауза. Научене клаузе могу заузимати велики део меморије, док неке клаузе (поготово клаузе са већим бројем литерала) могу имати мали ефекат на процес претраге
- *отпочињање претраге изнова* – у случају тешких инстанци, отпочињање претраге изнова може довести до спровођења претраге над другим делом простора претраге

Више детаља о описаним оптимизацијама биће приказано приликом описа имплементације CDCL SAT решавача.

Глава 3

Докази незадовољивости

Примена SAT решавача обухвата широк спектар различитих домена – верификација софтвера и хардвера, проблеми планирања, SMT решавачи итд. Кључна особина модерних SAT решавача која омогућава њихову широку примену јесте њихово ефикасно извршавање. Међутим, њихова примена не би била могућа без обезбеђене коректности решавача који се користи. Примера ради, приликом развијања софтвера и хардвера који контролишу производне линије различитих прехранбених производа, потребно је бити сигуран да не може доћи до грешке као што је додавање веће количине конзерванса од дозвољеног. Због тога је *верификација* SAT решавача један од кључних изазова при њиховом развијању.

3.1 Приступи верификацији SAT решавача

Слично осталим софтверима, SAT решавачи се могу верификовати *статички* и *динамички*. Статичка верификација SAT решавача подразумева формалну верификацију изворног кода решавача. Међутим, оптимизација решавача разним техникама доводи до знатно комплекснијих имплементација, које је тешко статички верификовати. У овом раду, статичка верификација није од великог интереса, док се више о статичкој верификацији може наћи у [14, 15].

Динамичка верификација SAT решавача подразумева верификацију излаза решавача у току извршавања програма. Ово се постиже генерисањем *сертификата* тј. *доказа* од стране решавача, поред резултата SAT/UNSAT. Овакви докази се касније могу проверити независним софтвером – *верифика-*

шором којим се потврђује тачност резултата добијеног од стране решавача. Верификатори су уобичајено доста једноставнији од самих решавача и њих је једноставније формално (статички) верификовати, рецимо, користећи доказивач теорема Isabelle.

У случају да SAT решавач пријави задовољивост, као доказ задовољивости генерише се валуација која задовољава дату инстанцу. Верификатори за доказе задовољивости директно испитују да ли је истинитосна вредност дате инстанце у датој валуацији тачно или нетачно.

У случају да SAT решавач пријави незадовољивост, ситуација је компликованија. Постоји већи број формата доказа незадовољивости, а највећи број тих формата заснива се на правилу резолуције. Такви докази називају се *резолуцијски докази незадовољивости*, који представљају централни део овог рада и у наставку овог поглавља биће детаљно описани.

3.2 Резолуцијски докази незадовољивости

Генерисање доказа незадовољивости зависи од конкретне имплементације SAT решавача. Надаље сматрамо да радимо са CDCL SAT решавачем, са имплементираним заборављањем научених клауза и анализом конфликта по FUIP (енг. First Unique Implication Point) схеми, о којој ће више бити речи у наредном поглављу.

RES докази незадовољивости

RES докази незадовољивост су најједноставнији резолуцијски докази незадовољивости и у потпуности се заснивају на правилу резолуције. RES доказ незадовољивости је низ клауза $C_1, C_2, \dots, C_n$ при чему је за све $1 \leq i \leq n$, C_i или клауза из почетног скупа клауза или клауза добијена применом правила резолуције на клаузе C_j, C_k где $j, k < i$, а притом је C_n празна клауза [1]. Иако је RES доказе тешко генерисати и верификовати, они представљају теоријску основу за остале резолуцијске доказе незадовољивости.

RUP докази незадовољивости

RUP (енг. Reverse Unit Propagation) докази незадовољивости заснивају се на инкременталном проширивању почетног скупа клауза редундантним

клаузама применом RUP-извођења. Клауза $C = \{\ell_1, \dots, \ell_k\}$ је RUP-изведена из скупа клауза F уколико примена елиминације једиичних клауза на скуп $F \cup \{\{\bar{\ell}_1\}, \dots, \{\bar{\ell}_k\}\}$ резултује конфликтом. Низ клауза $C_1, C_2, \dots, C_n$ је RUP доказ незадовољивости за скуп клауза $F = F_0$ уколико је клауза C_i RUP-изведена из скупа клауза F_{i-1} за све $1 \leq i \leq n$, при чему је $F_j = F_{j-1} \cup \{C_j\}$ за све $1 \leq j \leq n$, а притом је C_n празна клауза [6].

Додатно, за RUP-изведене клаузе из неког скупа клауза F , такође кажемо да су такве клаузе и асиметричне таутологије тј. АТ-клаузе у односу на скуп клауза F . Приметимо да је скуп клауза $F \cup \{\{\bar{\ell}_1\}, \dots, \{\bar{\ell}_k\}\}$ одговара исказној формули $F \wedge \neg C$. Ако је C АТ-клауза у односу на F , тада је $F \wedge \neg C$ незадовољива. Исказна формула $\neg(F \Rightarrow C)$ је логички еквивалентна формули $F \wedge \neg C$, па је исказна формула $F \Rightarrow C$ таутологија. Дакле, C је таутологија уколико је скуп F задовољен. Ова чињеница оправдава описивање клаузе C као асиметричне таутологије у односу на скуп F .

Из претходне дискусије следи:

Теорема 1. Нека је дат скуп клауза F и нека је клауза C RUP-изведена из F . Тада су скупови клауза F и $F \cup \{C\}$ логички еквивалентни.

Доказ. Претпоставимо супротно, да скупови F и $F \cup \{C\}$ нису логички еквивалентни. Тада постоји валуација v таква да $\text{Int}_v(F) \neq \text{Int}_v(F \cup \{C\})$. Међутим, свака валуација која задовољава скуп клауза $F \cup \{C\}$, задовољава и подскуп F тог скупа клауза. Зато је $\text{Int}_v(F \cup \{C\}) = 0$ и $\text{Int}_v(F) = 1$. Међутим, како је C RUP-изведена из F , то је $F \Rightarrow C$ таутологија, па из $\text{Int}_v(F) = 1$ следи $\text{Int}_v(C) = 1$, одакле је и $\text{Int}_v(F \cup \{C\}) = 1$. Контрадикција. $\square$

Из транзитивности релације логичке еквиваленције, скупови клауза F_j , $0 \leq j \leq n$ имплицитно конструисани RUP доказом су логички еквивалентни, па су еквивалентни и скупови клауза $F = F_0$ и F_n . Међутим, скуп F_n садржи празну клаузу, па је по дефиницији незадовољив, а тиме је и почетни скуп F такође незадовољив.

RAT докази незадовољивости

RAT (енг. Resolution Asymmetric Tautology) докази незадовољивости представљају уопштење RUP доказа незадовољивости. Слично RUP доказима, RAT докази се заснивају на инкременталном проширивању почетног скупа

клауза, уз једину разлику да се при проширивању скупа клауза применом RAT-извођења добија нови скуп клауза који је логички еквивалентан као и почетни скуп клауза, а не логички еквивалентан као код RUP-извођења.

За клаузу C кажемо да је RAT-изведена из скупа клауза F уколико је C АТ-клауза у односу на F или уколико постоји литерал $\ell \in C$ такав да за све $C' \in F$ такве да $\bar{\ell} \in C'$ важи да је $C \otimes_{\ell} C'$ АТ-клауза у односу на F . У другом случају кажемо да је C RAT-изведена из F преко пивота ℓ . Низ клауза $C_1, C_2, \dots, C_n$ је RAT доказ незадовољивости за скуп клауза $F = F_0$ уколико је клауза C_i RAT-изведена из скупа клауза F_{i-1} за све $1 \leq i \leq n$, при чему је $F_j = F_{j-1} \cup \{C_j\}$ за све $1 \leq j \leq n$, а притом је C_n празна клауза [9].

Слично претходно доказаној Теорему 1, важи и:

Теорема 2. Нека је дат скуп клауза F и нека је клауза C RAT-изведена из F . Тада су скупови клауза F и $F \cup \{C\}$ логички еквивалентни.

Доказ. Уколико је C АТ-клауза у односу на F , односно RUP-изведена из F , тврђење следи из својства RUP-извођења.

Даље претпостављамо да је C RAT-изведена из F преко пивота ℓ . Нека је v валуација која задовољава $F \cup \{C\}$, тј. $v \models F \cup \{C\} \equiv F \wedge C$, одакле је $v \models F$. Нека је сада $v \models F$. Уколико $v \models C$, тада важи $v \models F \cup \{C\}$.

Даље претпоставимо $v \not\models C$. Са $F_{\bar{\ell}}$ означимо скуп клауза C' таквих да $C' \in F$, $\bar{\ell} \in C'$ и нека је $F = F' \cup F_{\bar{\ell}}$. Додатно, са $C \otimes_{\ell} F_{\bar{\ell}}$ означимо скуп $\{C \otimes_{\ell} C' \mid C' \in F_{\bar{\ell}}\}$. С обзиром на то да су клаузе из $C \otimes_{\ell} F_{\bar{\ell}}$ АТ-клаузе у односу на F , односно RUP-изведене из F , то су скупови F и $F \cup (C \otimes_{\ell} F_{\bar{\ell}})$ логички еквивалентни па важи $v \models F \cup (C \otimes_{\ell} F_{\bar{\ell}})$, одакле следи $v \models C \otimes_{\ell} C'$ за све $C' \in F_{\bar{\ell}}$, односно. $v \models (C \setminus \{\ell\}) \cup (C' \setminus \{\bar{\ell}\})$ за све $C' \in F_{\bar{\ell}}$. Међутим, из $v \not\models C$ следи $v \not\models C \setminus \{\ell\}$, па важи $v \models C' \setminus \{\bar{\ell}\}$. Додатно, из $v \not\models C$ и $\ell \in C$, следи $\text{Int}_v(\ell) = 0$. Нека је сада v' валуација таква да $v(p) = v'(p)$ за све атоме p , осим за атом q за који је $\ell \in \{q, \neg q\}$ за који важи $v'(q) = 1 - v(q)$ тј. $\text{Int}_{v'}(\ell) = 1$. Тада важи $v' \models C$. С обзиром на то да $\bar{\ell} \notin C''$ ни за једно $C'' \in F'$, важи и $v' \models F'$. С обзиром на то да $v \models C' \setminus \{\bar{\ell}\}$ за све $C' \in F_{\bar{\ell}}$, тада и $v' \models C' \setminus \{\bar{\ell}\}$ за све $C' \in F_{\bar{\ell}}$, јер истинитосна вредност клаузе $C' \setminus \{\bar{\ell}\}$ не зависи од интерпретације литерала ℓ и $\bar{\ell}$. Међутим, сада из $v' \models C' \setminus \{\bar{\ell}\}$ за све $C' \in F_{\bar{\ell}}$ следи $v' \models C'$ за све $C' \in F_{\bar{\ell}}$, односно $v' \models F_{\bar{\ell}}$. На крају је $v' \models F' \cup F_{\bar{\ell}} \cup \{C\} = F \cup \{C\}$, чиме је тврђење доказано. $\square$

Интуитивно, посматрамо валуацију која задовољава F . Ако таква валуација задовољава и C , скупови F и $F \cup \{C\}$ су тривијално еквизадовољиви. Међутим, уколико таква валуација не задовољава C , потребно је изменити је тако да задовољава C тако да се не наруши задовољенист скупа F . У овом случају, валуацију мењамо на пивоту ℓ и тиме C постаје задовољена, због чега је постојање једног пивот литерала довољно. F мора остати задовољена, с обзиром на то да је $C \setminus \{\ell\}$ незадовољена, а из чињенице да су $C \otimes_{\ell} F_{\ell}$ АТ-клаузе, валуација задовољава и те клаузе, следи да су све $C' \setminus \{\bar{\ell}\}$ задовољене, па су и клаузе C' , које садрже $\bar{\ell}$ задовољене. Како валуација измењена само над литералом ℓ , једине клаузе које су могле покварити задовољеност скупа F су управо клаузе C' , чију смо задовољеност управо показали.

Из транзитивности релације логичке еквизадовољивости, скупови клауза F_j , $0 \leq j \leq n$ имплицитно конструисани RАТ доказом су логички еквивалентни, па су еквивалентни и скупови клауза $F = F_0$ и F_n . Међутим, скуп F_n садржи празну клаузу, па је по дефиницији незадовољив, а тиме је и почетни скуп F такође незадовољив.

Приметимо да је сваки RUP доказ незадовољивости уједно и RАТ доказ незадовољивост, док супротно не важи. Предност RАТ доказа незадовољивости у односу на RUP доказе је "отпорност" на већи број техника оптимизације. Конкретније, одређене технике оптимизације не задржавају својство логичке еквивалентности у односу на почетни скуп клауза који је прослеђен SAT решавачу и самим тим генерисање RUP доказа незадовољивости није подржано за решаваче који имплементирају такве технике оптимизације, за разлику од RАТ доказа незадовољивости [9]. С обзиром на то да је ефикасност SAT решавача једна од најбитнијих метрика при оцени рада самог решавача, технике оптимизације које доприносе већој ефикасности су од кључног значаја за решаваче. Услед тога, RАТ докази незадовољивости представљају значајно унапређење у односу на RUP доказе незадовољивости.

У оквиру решавача који ће бити имплементиран за потребе овог рада, неће бити имплементирани технике оптимизације које не задржавају логичку еквивалентност са почетним скупом клауза. Самим тим, генерисани докази ће најуже гледано бити RUP докази незадовољивост, али уједно и RАТ докази незадовољивости. Међутим, генерисање доказа у RАТ формату представља стандард за модерне решаваче, без обзира на то да ли је генерисани доказ и у RUP формату. Услед свега наведеног, у оквиру овог рада подразумева се

да су сви генерисани докази незадовољивости у RAT формату.

DRUP/DRAT докази незадовољивости

Главна мана претходно описаних RUP и RAT доказа незадовољивост је величина имплицитно конструисаних формула F_j . Верификатори RUP и RAT доказа незадовољивости морају конструисати сваку од формула F_j при провери валидности датог доказа. Услед временска и меморијских ограничења, извршавање верификатора при провери изузетно комплексних и великих доказа може бити изузетно захтевно, а потенцијално и немогуће.

Једно од решења за овај проблем јесте периодично смањивање величине формула F_j уклањањем клауза из тих скупова, при чему је дозвољено уклањати само оне клаузе које су логички редундантне. DRUP (енг. Deletion Reverse Unit Propagation), односно DRAT (енг. Deletion Resolution Asymmetric Tautology) докази незадовољивости су проширење RUP, односно RAT доказа незадовољивости уз додатну могућност брисања клауза [8, 19]. Формално, DRUP/DRAT доказ незадовољивости је низ наредби $\text{act}_1 C_1, \dots, \text{act}_n C_n$, при чему су C_i клаузе, а act_i представљају акције које могу бити add или del. Наредба $\text{add } C_i$ је дозвољена уколико је C_i RUP/RAT-изведена из F_{i-1} , а притом је $F_i = F_{i-1} \cup \{C_i\}$. Наредба $\text{del } C_i$ је дозвољена уколико је $C_i \in F_{i-1}$, а притом је $F_i = F_{i-1} \setminus \{C_i\}$ ¹. Додатно, $F_0 = F$, $\text{act}_n = \text{add}$ и C_n је празна клауза.

LRAT докази незадовољивости

Верификација RUP/DRUP доказа незадовољивости није захтевна. Међутим, мана RUP/DRUP доказа је ограниченост њихове примене, с обзиром на то да захтевају да свака клауза доказа буде логичка последица почетног скупа клауза, док код великог броја решавача, клаузе које се уче не морају бити логичке последице, већ се дозвољава релаксиранији услов да се њиховим додавањем задржава логичка еквивалентност. Управо због тога су RAT/DRAT докази, који раде са клаузама који не нарушавају логичку незадовољивост, распрострањенији. Међутим, као што ћемо касније видети, верификација RAT/DRAT доказа је компликованија и може бити изузетно временски захтевна. Природно се поставља питање, да ли се DRAT докази

¹Иако се теоријски то не захтева, у практичним применама клауза C_i је у смислу логичке еквивалентности редундантна у односу на скуп клауза F_{i-1} , тј. важи $F_{i-1} \equiv_s F_i$. У супротном би доказ незадовољивости био некоректан

незадовољивости могу проширити и/или прилагодити, тако да њихова верификација буде једноставнија, а да се притом не наруше остала својства.

LRAT (енг. Linear Resolution Asymmetric Tautology) докази незадовољивости дају потврдан одговор на претходно постављено питање [4]. LRAT докази представљају проширење DRAT доказа *објашњењима* како, односно употребом којих клауза се може верификовати да је дата клауза RAT-изведена. Конкретно, свакој наредби $\text{add } C_i$ додељује се низ $(\pm C'_{i,1}, \dots, \pm C'_{i,m})$ при чему су $C'_{i,j}$ клаузе из скупа F_{i-1} . Верификатор LRAT доказа помоћу овог низа верификује да је клауза C_i RAT-изведена из F_{i-1} . LRAT доказ је проширен и техничким детаљима, о којима ће више бити речи у наставку поглавља.

3.3 Верификација DRAT/LRAT доказа

Један од познатих верификатора DRAT доказа, `drat-trim` [19], при обради $\text{add } C_i$ наредби прво проверава да ли је C_i RUP-изведена клауза из F_{i-1} , директном провером. Уколико је одговор потврдан, прелази се на следећу наредбу, док се у супротном проверава да ли је C_i RAT-изведена клауза из F_{i-1} преко пивота $C_i[0]$, при чему са $C_i[0]$ означавамо први литерал наведен у клаузи C_i – ово ограничење је додато због ефикаснијег процеса верификације при чему се показало да ово не компликује процес генерисања доказа [9]. Провера такође иде директно – за сваку клаузу C' која садржи литерал супротан литералу $C_i[0]$, проверава се да ли је клауза $C_i \otimes_\ell C'$ RUP-изведена из F_{i-1} . У случају успешне провере, прелази се на следећу наредбу, док се у супротном пријављује неуспешна верификација.

Верификатор LRAT доказа проверава да ли је C_i RAT-изведена клауза из F_{i-1} преко пивота $C_i[0]$ користећи низ објашњења дат у оквиру LRAT доказа, на следећи начин:

- Нека за дати низ клауза важи $(\pm C'_{i,1}, \dots, \pm C'_{i,m}) = (+C_{0,1}, \dots, +C_{0,k_0}, -D_1, +C_{1,1}, \dots, +C_{1,k_1}, \dots, -D_j, +C_{j,1}, \dots, +C_{j,k_j})$, при чему важи $(F_{i-1})_{\bar{\ell}} = \{D_1, \dots, D_j\}$
- Верификатор прво врши јединичну пропагацију над скупом $F \cup \bar{C}_i$, при чему након јединичне пропагације над клаузама $\bar{C}_i$, све клаузе из F које постају јединичне, и над којим се извршава јединична пропагација, су $C_{0,1}, \dots, C_{0,k_0}$. У случајевима када је клауза C_i АТ-клауза, тада је $j = 0$,

а клауза C_{0,k_0} постаје конфликтна. У супротном, јединична пропагација на овако описан начин се спроводи пре директних провера над клаузама скупа $(F_{i-1})_{\bar{\ell}}$ јер су те јединичне пропагације заједничке за сваку од директних провера над тим клаузама. На тај начин, верификатор спроводи такве јединичне пропагације само једном, а не за сваку клаузу из скупа $(F_{i-1})_{\bar{\ell}}$ појединачно.

- Верификатор потом за сваку клаузу из $(F_{i-1})_{\bar{\ell}} = \{D_1, \dots, D_{i_j}\}$, редом проверава да су $C_i \otimes_{C_i[0]} D_1, \dots$ АТ-клаузе у односу на F_{i-1} , при чему током провере клаузе $C_{1,1}, \dots, C_{1,k_1-1}$ постају јединичне, а клауза C_{1,k_1} постаје конфликтна.

У оквиру доказивача теорема ACL2 имплементиран је верификатор LRAT доказа, чија је коректност доказана. Међутим, `drat-trim` алат који је првобитно био намењен верификацији DRAT доказа, је додатно проширен и верификатором LRAT доказа `lrat-check`.

3.4 Генерисање DRAT/LRAT доказа

Генерисање DRAT/LRAT доказа у многоме зависи од конкретне имплементације решавача. У овом раду биће обрађено генерисање DRAT/LRAT доказа незадовољивост за претходно описани CDCL SAT решавач.

Генерисање DRAT доказа

Свака научена клауза у оквиру анализе конфликта је RUP-изведена клауза у односу на скуп клауза проширен претходно наученим клаузама [1]. Додатно, описани CDCL SAT решавач заборавља искључиво научене клаузе, које у сваком кораку извршавања представљају редундантне клаузе у погледу логичке еквивалентности. Описана својства CDCL SAT решавача дозвољавају једноставно генерисање DRUP, а самим тим и DRAT доказа: свака научена клауза додаје се у доказ, док се свака заборављена клауза брише из доказа.

Формат DRAT доказа

DRAT докази незадовољивости се генеришу у формату сличном DIMACS CNF формату. Датотеке које садрже DRAT доказе користе екстензију `.drat`. Свака линија у оквиру датотеке одговара једној наредби самог доказа. Наредби `add C` одговара линија која садржи клаузу C , у идентичном формату како се клаузе наводе у DIMACS CNF формату. Наредби `del C` одговара линија која садржи карактер `d`, а потом клаузу C , у идентичном формату како се клаузе наводе у DIMACS CNF формату.

За исказну формулу дату у у DIMACS CNF формату:

```

1 p cnf 4 8
2 1 2 -3 0
3 -1 -2 3 0
4 2 3 -4 0
5 -2 -3 4 0
6 -1 -3 -4 0
7 1 3 4 0
8 -1 2 4 0
9 1 -2 -4 0

```

доказ у DRAT формату је:

```

1 1 0
2 d 1 -4 -2 0
3 d 1 4 3 0
4 d 1 2 -3 0
5 2 0
6 d -1 2 4 0
7 d 2 -4 3 0
8 0

```

Генерисање LRAT доказа

С обзиром на то да су све клаузе DRAT доказа RUP-изведене клаузе, низ клауза $(\pm C_{i,1}, \dots, \pm C_{i,m})$ не садржи ниједну негативну клаузу, с обзиром на то да јединичном пропагацијом долази до конфликта. Овај низ клауза је једноставно генерисати, с обзиром на то да се састоји од клауза скупа F_{i-1}

које учествују у анализи конфликта, односно у примени корака Explain, уз редослед супротан редоследу појављивања у анализи конфликта, односно од последње до прве клаузе.

Формат LRAT доказа незадовољивости

LRAT докази незадовољивости се генеришу у формати сличном формату DRAT доказа незадовољивости, при чему се за датотеке користи екстензија `.lrat`. У оквиру LRAT доказа, свака клауза има додељен позитиван целобројни идентификатор преко које се референцира у самом доказу, при чему се идентификатори додељују редом. Свака линија у оквиру датотеке на почетку садржи тренутну вредност бројача на основу ког се додељују идентификатори клаузама доказа које се додају почетном скупу. Ово омогућава компактнији запис наредбе `add  $C_i$` – након бројача, чија вредност представља идентификатор додате клаузе, наводи се сама клауза, у идентичном формату како се клаузе наводе у DIMACS CNF формату. Након навођења клаузе, наводе се идентификатори клауза из низа $(\pm C_{i,1}, \dots, \pm C_{i,m})$, при чему се идентификаторима дописује знак `-` уколико се клауза у низу појављује са истим предзнаком. Поред тога, уколико се више `del  $C$` наредби појављује узастопно, у оквиру датотеке се то представља једном линијом, у оквиру које се, након бројача и карактера `d`, наводе идентификатори клауза које се бришу.

За исказну формулу дату у у DIMACS CNF формату:

```
1 p cnf 4 8
2 1 2 -3 0
3 -1 -2 3 0
4 2 3 -4 0
5 -2 -3 4 0
6 -1 -3 -4 0
7 1 3 4 0
8 -1 2 4 0
9 1 -2 -4 0
```

доказ у LRAT формату је:

```
1 9 1 0 -2 6 8 -5 1 8 -7 6 1 0
2 9 d 8 6 1 0
3 10 2 0 9 7 5 3 0
```

ГЛАВА 3. ДОКАЗИ НЕЗАДОВОЉИВОСТИ

4	10 d 7 3 0
5	11 0 9 10 2 4 5 0

Глава 4

Имплементација

Ефикасне имплементације претходно описаних алгоритама није једноставно реализовати. Модерни SAT решавачи се ослањају на напредне временске и меморијске оптимизационе технике, које у великој мери доприносе њиховом брзом извршавању, али по цену изузетно обимног и комплексног кода.

За потребе овог рада, имплементиран је једноставнији CDCL SAT решавач, који садржи функционалност генерисања DRAT, односно LRAT доказа. У наставку ће бити приказани детаљи имплементације, при чему ће прво бити описане основне функционалности решавача, а потом и функционалности генерисања доказа.

Решавач је имплементиран у програмском језику C++²³, уз коришћење алата CMake. Изворни код се налази на Git репозиторијуму <https://github.com/milanmitreski/master-thesis>.

4.1 CDCL SAT решавач

Основна функционалност CDCL SAT решавача реализована је у оквиру класе `Solver` која енкапсулира теоријске основе CDCL SAT решавача описаних у другој глави. Функција `Solver::solve` за прослеђену исказну формулу у CNF одређује да ли је формула задовољива или не, а у случају задовољивости враћа одговарајућу валуацију. Приликом имплементације функције `Solver::solve` коришћене су одговарајуће структуре и помоћне функције, чији ће опис бити дат у наставку.

Структуре

Све описане структуре дефинисане су у оквиру заглавља `solver_types.hpp`, уз изузетка структуре `Solver::State` описане у оквиру заглавља `solver.hpp` и која представља стање решавача.

За представљање основних појмова литерала, клауза и скупа клауза коришћени су алијаси.

```
1 using Literal = int;  
2 using Clause = std::vector<Literal>;  
3 using ClauseSet = std::vector<Clause>;
```

CDCL процедура у сваком кораку и сваком стању одржава парцијалну валуацију. Парцијална валуација се може имплементирати на више начина. Међутим, како се у току извршавања алгоритма парцијална валуација интезивно мења, али се и интезивно испитује да ли је неком атому додељена истинитосна вредност, најефикаснији начин за реализацију парцијалне валуације је употребом тернарне логике.

```
1 enum class tribool : int8_t {  
2     False = -1,  
3     Undef = 0,  
4     True = 1,  
5 };  
6 using PartialValuation = std::vector<tribool>;
```

Вредност `Undef` означава да атому није додељена истинитосна вредност у тренутној парцијалној валуацији. Додатно, с обзиром на то да је потребно упамтити у ком кораку и на који начин је атому додељена истинитосна вредност. Ово је могуће реализовати стеком додела који називамо и *трај*.

```
1 struct Assignment {  
2     Literal literal;  
3     int decision_level;  
4     int unit_clause_idx;  
5 };  
6 using Trail = std::vector<Assignment>;
```

Схема два посматрана литерала

Један од кључних изазова при имплементацији решавача јесте ефикасно одржавање истинитосних вредности клауза између два суседна стања. Директно израчунавање истинитосних вредности клауза у сваком кораку и сваком стању драстично утиче на ефикасност решавача. Једно од решења за овај проблем је *схема два посматрана литерала*. За сваку клаузу C одржавамо два литерала $watch1(C)$ и $watch2(C)$. На почетку, истинитосне вредности та два литерала су недефинисане. Уколико један од литерала добије негативну истинитосну вредност, тражи се други литерал чија истинитосна вредност није негативна. Уколико такав литерал постоји, тај литерал постаје посматран уместо претходно посматраног литерала, док се у супротном посматра истинитосна вредност другог литерала. Ако је другом литералу додељена позитивна истинитосна вредност, клауза је већ задовољена и претрага се наставља даље. Са друге стране, ако је другом литералу додељена негативна истинитосна вредност, клауза постаје незадовољива и пријављује се конфликт. На крају, ако другом литералу није додељена истинитосна вредност, клауза постаје јединична и над њом ће се извршити елиминација јединичних клауза.

Овим поступком се при додели истинитосне вредности неком атому, разматрају истинитосне вредности само оних клауза које посматрају тај литерал, а не свих клауза. Ради ефикасније имплементације, поред структуре која за сваку клаузу чува два посматрана литерала, дефинишемо и структуру која за сваки литерал чува списак клауза које посматрају тај литерал.

```
1 using Watcher = std::vector<std::pair<Literal, Literal>>;
2 using WatchList = std::vector<std::vector<int>>;
```

Стање решавача

Стање решавача у теоријском смислу описано је у глави 2, у оквиру апстрактног описа CDCL процедуре. Имплементација стања у оквиру решавача је проширена додатним параметрима оптимизационих техника као и помоћним променљивим којим се постиже ефикасније извршавање алгорита. Опис параметара стања решавача биће описани у наставку, док ће употреба помоћних променљивих бити описана у оквиру описа одговарајућих функција. Овако проширено стање решавача описано је структуром **State** у оквиру

класе Solver.

```
1 struct State {
2     ClauseSet clause_set;
3     PartialValuation partial_valuation;
4     Trail trail;
5
6     Watcher watcher;
7     WatchList watch_list;
8
9     int conflict_clause_idx;
10    std::vector<int> seen;
11    int seen_counter;
12
13    std::vector<int> level;
14    int decision_level;
15
16    std::set<std::pair<double, int>, std::greater<>> variable_order;
17    std::vector<double> variable_activity;
18    std::vector<tribool> variable_polarity;
19    double variable_increment;
20    double variable_decay_factor;
21
22    int conflict_counter;
23    int restart_threshold;
24    int restart_counter;
25    double luby_multiplier;
26
27    std::vector<double> clause_activity;
28    std::vector<bool> is_learned;
29    std::vector<bool> is_deleted;
30    std::vector<bool> is_locked;
31    double clause_increment;
32    double clause_decay_factor;
33    int learned_clause_count;
34    int learned_clause_limit;
35    double learned_clause_limit_factor;
36 };
```

Класа Solver

Као што је раније наведено, класа `Solver` енкапсулира основне функционалности CDCL SAT решавача и садржи један атрибут `m_state` који представља тренутно стање решавача. У наставку је дата дефиниција класе `Solver` (декларације функција), уз изостављење дефиниције структуре `State`, која је претходно описана.

```
1 class Solver {
2 public:
3     Solver() = default;
4     std::optional<PartialValuation> solve(int atom_count, ClauseSet
        clause_set);
5 private:
6     void reset(int atom_count, ClauseSet clause_set);
7     void initialize_watcher(int clause_idx);
8     bool unit_propagation(Literal decision_literal);
9     int conflict_analysis(const Clause& conflict_clause);
10    Clause explain(const Clause& conflict_clause, int& backjump_level);
11    void learn(const Clause& clause);
12    void backjump(int backjump_level);
13    bool all_variables_assigned() const;
14    Literal pick_branching_variable() const;
15    void bump_literal_activity(Literal literal);
16    void decay_literal_activities();
17    void restart_search();
18    static double luby(double y, int i);
19    void bump_clause_activity(int clause_idx);
20    void decay_clause_activities();
21    void delete_clauses();
22    void assign_literal_value(Literal literal, int reason_clause_idx);
23    bool move_watch(int clause_idx, Literal false_literal);
24    tribool get_literal_value(Literal literal) const;
25    int get_literal_level(Literal literal) const;
26
27    State m_state;
28 };
```

Главна функционалност решавача имплементирана је у оквиру функције `solve`. Аргументи ове функције су број атома над којим је дефинисана исказна формула, као и сама исказна формула, дата у облику скупа клауза. Повратна вредност је опциона вредност типа `PartialValuation` – уколико је вредност непостојећа, тада је исказна формула незадовољива, док је у супротном задовољива, а добијена парцијална валуација задовољава исказну формулу. Имплементација функције `solve` је модуларизирана помоћу већег броја помоћних функција, што доприноси читљивости, лакшем разумевању, дебаговању и проширивању функционалности.

Иницијализација

Пре покретања самог алгорита, потребно је иницијализовати вредност атрибута `m_state` на почетно стање за прослеђену исказну формулу. Ова функционалност имплементирана је у оквиру функције `reset`. Један од корака при иницијализацији почетног стања је подешавање схеме два посматрана литерала, позивом функције `initialize_watcher`.

```
1 void Solver::initialize_watcher(const int clause_idx) {
2     const Clause &clause = m_state.clause_set[clause_idx];
3     if (clause.empty()) {
4         m_state.watcher[clause_idx] = {0, 0};
5         return;
6     }
7
8     Literal fst = clause[0];
9     Literal snd = clause.size() > 1 ? clause[1] : clause[0];
10
11     m_state.watcher[clause_idx] = std::make_pair(fst, snd);
12     m_state.watch_list[watch_list_index(fst)].push_back(clause_idx);
13     if (fst != snd) {
14         m_state.watch_list[watch_list_index(snd)].push_back(clause_idx);
15     }
16 }
```

Уколико је клауза празна, литерали који се посматрају се постављају на вредност 0. Уколико је клауза јединична, оба посматрана литерала се постављају

на једини литерал у клаузи. У супротном су литерали који се посматрају први и други литерал у клаузи.

Гранање

Гранање тј. примена корака *split* заснива се на избору литерала чија је тренутна вредност у парцијалној валуацији недефинисана и потом додељивање позитивне истинитосне вредности том литералу у парцијалној валуацији. Додела истинитосне вредности имплементирана је у оквиру функције `assign_literal_value` која се користи за доделу истинитосне вредности и у случају елиминације јединичне клаузе. Са друге стране, избор атома над којим се примењује гранање врши се применом VSIDS (енг. Variable State Independent Decaying Sum) стратегије гранања. Примена ове стратегије је приказала побољшање перформанси CDCL SAT решавача [17]. VSIDS стратегија сваком атому додељује скор, који је иницијално постављен на 0. За сваки атом који учествује у конфликту или анализи конфликта, скор се увећава (најчешће за 1). Периодично, скорови атома се умањују за одређени фактор. У тренутку када је потребно извршити гранање, бирамо онај атом који у том тренутку има највећи скор. На овај начин се бира атом који је учествовао у већем броју *скоријих* конфликта од осталих атома. Након избора атома, њему додељујемо ону истинитосну вредност која му је последњи пут била додељена. Приликом имплементација VSIDS стратегије уместо умањења свих скорова, врши се увећање вредности којом увећавамо скорове атомима који учествују у конфликту/анализи конфликта. У тренутку када нека променљива достигне изузетно висок скор, врши се скалирање свих скорова. Подешавањем параметара `variable_increment` и `variable_decay_factor` контролише се понашање VSIDS стратегије.

```
1 Literal Solver::pick_branching_variable() const {
2     if (m_state.variable_order.empty())
3         throw std::runtime_error("No unassigned variables left");
4
5     const int variable = m_state.variable_order.begin()->second;
6
7     return (m_state.variable_polarity[variable] == tribool::True) ?
8         variable : -variable;
9 }
```

```
9
10 void Solver::bump_literal_activity(const Literal literal) {
11     const int variable = abs(literal);
12     double activity = m_state.variable_activity[variable];
13     const bool unassigned = m_state.variable_order.erase({
14         activity, variable
15     });
16
17     m_state.variable_activity[variable] += m_state.variable_increment;
18     if (unassigned) {
19         m_state.variable_order.insert({
20             m_state.variable_activity[variable], variable
21         });
22     }
23
24     if (m_state.variable_activity[variable] > 1e100) {
25         m_state.variable_increment *= 1e-100;
26         m_state.variable_order.clear();
27         for (int i = 1; i <
28             static_cast<int>(m_state.variable_activity.size()); i++) {
29             m_state.variable_activity[i] *= 1e-100;
30             if (m_state.partial_valuation[i] == tribool::Undef) {
31                 m_state.variable_order.insert({
32                     m_state.variable_activity[i], i
33                 });
34             }
35         }
36     }
37 }
38
39 void Solver::decay_literal_activities() {
40     m_state.variable_increment *= 1.0 / m_state.variable_decay_factor;
41 }
```

Поновно покретање претраге

У случају великог броја достигнутих конфликта, поновним покретањем претраге, уз задржавање до тада научених клауза, могуће је побољшати перформансе решавача [17]. Поновно покретање претраге се врши након достигнутог одређеног броја конфликта од последњег поновног покретања. Граница за поновно покретање се током извршавања алгоритма увећава, чиме се осигурава заустављање алгоритма. Примена Luby технике за контролу границе за поновно покретање даје боље резултате од осталих техника [10]. Подешавањем параметара `luby_multiplier` и `restart_threshold` контролише се понашање оптимизационе технике поновног покретања претраге.

```
1 void Solver::restart_search() {
2     backjump(0);
3     m_state.restart_counter++;
4     m_state.conflict_counter = 0;
5     m_state.restart_threshold =
6         static_cast<int>(m_state.luby_multiplier *
7             luby(2, m_state.restart_counter + 1));
8 }
9
10 double Solver::luby(const double y, int i) {
11     int size, seq;
12     for (size = 1, seq = 0; size < i + 1; seq++, size = 2 * size + 1) {
13     }
14
15     while (size - 1 != i) {
16         size = (size - 1) >> 1;
17         seq--;
18         i = i % size;
19     }
20     return std::pow(y, seq);
21 }
```

Заборављање научених клауза

Код великих и тешких инстанци, могуће је доћи до изузетно великог броја научених клауза, услед чега долази до значајног меморијског оптерећења и успореног извршавања. Применом оптимизационе технике, сличној VSIDS стратегији гранања [18], можемо идентификовати које научене клаузе највише доприносе самој претрази. Скор клаузе се увећава када се клауза научи, као и сваки пут када та клауза учествује у анализи конфликта. Заборављање клауза покреће се након достизања одређеног броја научених клауза, при чему се не бришу клаузе чијом је јединичном пропагацијом дошло до доделе истинитосне вредности атому која се налази на трагу у тренутку заборављања. Број клауза који се брише износи половину укупног броја научених клауза које се могу заборавити. Ову оптимизациону технику можемо контролисати подешавањем параметара `clause_increment`, `clause_decay_factor`, `learned_clause_limit` и `learned_clause_limit_factor`.

```
1 void Solver::bump_clause_activity(const int clause_idx) {
2     if (!m_state.is_learned[clause_idx])
3         return;
4
5     m_state.clause_activity[clause_idx] += m_state.clause_increment;
6
7     if (m_state.clause_activity[clause_idx] > 1e100) {
8         for (auto &activity: m_state.clause_activity) {
9             activity *= 1e-100;
10        }
11        m_state.clause_increment *= 1e-100;
12    }
13 }
14
15 void Solver::decay_clause_activities() {
16     m_state.clause_increment *= 1.0 / m_state.clause_decay_factor;
17 }
18
19 void Solver::delete_clauses() {
20     std::vector<std::pair<double, int> > candidates;
21     for (int i = 0; i < static_cast<int>(m_state.clause_activity.size());
22         i++) {
```

```

23     if (m_state.is_learned[i] && !m_state.is_deleted[i]
24         && m_state.clause_set[i].size() > 2 && !m_state.is_locked[i])
25         candidates.emplace_back(m_state.clause_activity[i], i);
26     }
27
28     const int limit = static_cast<int>(candidates.size()) / 2;
29     std::ranges::nth_element(candidates, candidates.begin() + limit);
30     for (int i = 0; i < limit; i++) {
31         const int clause_delete_idx = candidates[i].second;
32         m_state.is_deleted[clause_delete_idx] = true;
33         m_state.learned_clause_count--;
34     }
35 }

```

Елиминација јединичних клауза

У току извршавања алгоритма, до елиминације јединичних клауза може доћи на два начина – на почетку претраге и непосредно након примене корака *split*. Елиминација јединичних клауза имплементирана је у оквиру функције `unit_propagation`. Аргумент функције је литерал над којим је претходно примењено правило *split*. У случају да елиминацију јединичних клауза покрећемо на почетку, као аргумент прослеђујемо 0. Повратна вредност функције је истинитосна вредност која указује на то да ли је дошло до конфликта или не. Функција `unit_propagation` исцрпно спроводи елиминацију јединичних клауза – уколико елиминацијом једне јединичне клаузе долази до појаве друге јединичне клаузе, исти позив функције `unit_propagation` елиминисаће и новонасталу јединичну клаузу.

```

1 bool Solver::unit_propagation(const Literal decision_literal) {
2     std::queue<Literal> literals;
3     if (decision_literal != 0)
4         literals.push(decision_literal);
5     else
6         for (int i = 0; i < static_cast<int>(m_state.clause_set.size());
7             i++) {
8             const Clause &clause = m_state.clause_set[i];
9             if (clause.empty())

```

```
10         return false;
11
12     if (clause.size() == 1) {
13         Literal literal = clause[0];
14         if (get_literal_value(literal) == tribool::False)
15             return false;
16         if (get_literal_value(literal) == tribool::Undef) {
17             assign_literal_value(literal, i);
18             literals.push(literal);
19         }
20     }
21 }
22
23 while (!literals.empty()) {
24     const Literal literal = literals.front();
25     literals.pop();
26
27     const Literal false_literal = -literal;
28     std::vector<int> &watched_clauses =
29         m_state.watch_list[watch_list_index(false_literal)];
30
31     for (int i = 0; i < static_cast<int>(watched_clauses.size());) {
32         const int clause_idx = watched_clauses[i];
33
34         if (m_state.is_deleted[clause_idx]) {
35             watched_clauses[i] = watched_clauses.back();
36             watched_clauses.pop_back();
37             continue;
38         }
39
40         if (move_watch(clause_idx, false_literal)) {
41             watched_clauses[i] = watched_clauses.back();
42             watched_clauses.pop_back();
43             continue;
44         }
45
46         const auto &[fst, snd] = m_state.watcher[clause_idx];
```

```
47         const Literal other = (fst == false_literal) ? snd : fst;
48         const tribool val = get_literal_value(other);
49
50         if (val == tribool::False) {
51             m_state.conflict_clause_idx = clause_idx;
52             return false;
53         }
54
55         if (val == tribool::Undef) {
56             assign_literal_value(other, clause_idx);
57             literals.push(other);
58         }
59
60         i++;
61     }
62 }
63
64 return true;
65 }
```

Јединичну клаузу детектујемо помоћу схеме два литерала. Сви литерали којима је додељена истинитосна вредност након последње елиминације јединичне клаузе смештају се у ред `literals`. За сваки такав литерал проверавамо које клаузе посматрају њему супротан литерал, чија је истинитосна вредност у том тренутку негативна. Уколико је могуће наћи нови посматрани литерал, клауза није јединична. Уколико није могуће наћи нови посматрани литерал, разматрамо други посматрани литерал. Ако је и њему додељена негативна истинитосна вредност, клауза је конфликтна и пријављујемо конфликт. У супротном, ако другом посматраном литералу није додељена вредност, у питању је јединична клауза и вршимо њену елиминацију.

Анализа конфликта

Уколико `unit_propagation` пријави конфликт, потребно је извршити анализу конфликта. Анализа конфликта имплементирана је у оквиру функција `conflict_analysis`, `explain` и `learn`. Након пријаве конфликта, позивом функције `conflict_analysis`, чији је аргумент конфликтна клауза, започи-

њемо анализу конфликта. Функција `explain` конструише нову клаузу коју додајемо у скуп клауза функцијом `learn`, након чега се функцијом `backjump` враћамо до нивоа на ком је новонаучена клауза постала конфликтна.

```

1 int Solver::conflict_analysis(const Clause &conflict_clause) {
2     int backjump_level = 0;
3     const Clause learned_clause =
4         explain(conflict_clause, backjump_level);
5     learn(learned_clause);
6     return backjump_level;
7 }

```

Функција `explain` конструише клаузу коју је потребно научити применом FUIP схеме. Увођењем глобалног (гледајући у односу на функцију `explain`) низа `seen` и бројача `seen_counter`, којим означавамо који атоми учествују у конфликтној клаузи, постижемо значајну временску оптимизацију.

```

1 Clause Solver::explain(const Clause &conflict_clause, int &backjump_level)
2 {
3     m_state.seen_counter++;
4     int count = 0;
5     Literal uip = 0;
6     Clause learned(1, uip);
7
8     for (const Literal &literal: conflict_clause) {
9         if (m_state.seen[abs(literal)] != m_state.seen_counter) {
10             bump_literal_activity(literal);
11             m_state.seen[abs(literal)] = m_state.seen_counter;
12
13             if (get_literal_level(literal) == m_state.decision_level) {
14                 count++;
15             } else {
16                 learned.push_back(literal);
17                 if (get_literal_level(literal) > backjump_level) {
18                     backjump_level = get_literal_level(literal);
19                 }
20             }
21         }
22     }
23 }

```

```
22     }
23
24     for (int i = static_cast<int>(m_state.trail.size()) - 1;
25         i >= 0; i--) {
26         const Assignment &assignment = m_state.trail[i];
27
28         if (m_state.seen[abs(assignment.literal)] == m_state.seen_counter
29             && count == 1) {
30             uip = assignment.literal;
31             break;
32         }
33
34         if (m_state.seen[abs(assignment.literal)] == m_state.seen_counter
35             && assignment.unit_clause_idx != -1) {
36             if (m_state.is_learned[assignment.unit_clause_idx])
37                 bump_clause_activity(assignment.unit_clause_idx);
38
39             for (const Literal &literal:
40                 m_state.clause_set[assignment.unit_clause_idx]) {
41                 if (m_state.seen[abs(literal)] != m_state.seen_counter) {
42                     bump_literal_activity(literal);
43                     m_state.seen[abs(literal)] = m_state.seen_counter;
44                     if (get_literal_level(literal) ==
45                         m_state.decision_level) {
46                         count++;
47                     } else {
48                         learned.push_back(literal);
49                         if (get_literal_level(literal) > backjump_level)
50                         {
51                             backjump_level = get_literal_level(literal);
52                         }
53                     }
54                 }
55             }
56
57             m_state.seen[abs(assignment.literal)] =
58                 m_state.seen_counter - 1;
```

```

59         count--;
60     }
61 }
62
63     learned[0] = -ui;
64
65     return learned;
66 }

```

Функција `learn` додаје научену клаузу у скуп клауза, иницијализује одговарајуће структуре и вредности за нову клаузу и разрешава конфликт.

```

1 void Solver::learn(const Clause &clause) {
2     m_state.clause_set.push_back(clause);
3
4     m_state.clause_activity.push_back(0.0);
5     m_state.is_learned.push_back(true);
6     m_state.is_deleted.push_back(false);
7     m_state.is_locked.push_back(false);
8     m_state.learned_clause_count++;
9     bump_clause_activity(
10         static_cast<int>(m_state.clause_activity.size()) - 1
11     );
12
13     initialize_watcher(static_cast<int>(m_state.clause_set.size()) - 1);
14     m_state.conflict_clause_idx = -1;
15 }

```

На крају анализе конфликта, функцијом `backjump` враћамо се на одговарајући ниво претраге.

```

1 void Solver::backjump(const int backjump_level) {
2     while (!m_state.trail.empty()
3         && m_state.trail.back().decision_level > backjump_level) {
4         m_state.partial_valuation[
5             partial_valuation_index(m_state.trail.back().literal)
6         ] = tribool::Undef;
7         m_state.variable_polarity[abs(m_state.trail.back().literal)] =

```

```

8         (m_state.trail.back().literal > 0) ? tribool::True :
9         tribool::False;
10        m_state.variable_order.insert({
11            m_state.variable_activity[
12                abs(m_state.trail.back().literal)
13            ], abs(m_state.trail.back().literal)
14        });
15        if (m_state.trail.back().unit_clause_idx != -1) {
16            m_state.is_locked[m_state.trail.back().unit_clause_idx]
17                = false;
18        }
19        m_state.trail.pop_back();
20    }
21    m_state.decision_level = backjump_level;
22 }

```

Функција solve

Функција `solve` имплементира алгоритам CDCL SAT процедуре. Након иницијализације и подешавања стања решавача, покушава се елиминација јединичних клауза. У случају конфликта, одмах пријављујемо незадовољивост. У супротном, у петљи примењујемо корак *split*, а потом елиминацију јединичних клауза. Уколико дође до конфликта, анализирамо исти, осим у случају када се конфликт догодио на првом нивоу, када пријављујемо незадовољивост. Након анализе конфликта, опет покушавамо елиминацију јединичних клауза, све док елиминација доводи до конфликта. Када спроведемо елиминацију без конфликта, покрећемо још једну итерацију петље применом корака *split*, осим у случају када је свим атомима додељена истинитосна вредност, када заустављамо петљу, пријављујемо задовољивост и као повратну вредност прослеђујемо изграђену валуацију.

```

1 std::optional<PartialValuation> Solver::solve(const int atom_count,
2     ClauseSet clause_set) {
3     reset(atom_count, std::move(clause_set));
4
5     if (!unit_propagation(0))
6         return std::nullopt;

```

```
6
7   while (!all_variables_assigned()) {
8       if (m_state.conflict_counter >= m_state.restart_threshold) {
9           restart_search();
10      }
11
12      if (m_state.learned_clause_count
13          >= m_state.learned_clause_limit) {
14          delete_clauses();
15          m_state.learned_clause_limit = static_cast<int>(
16              m_state.learned_clause_limit *
17              m_state.learned_clause_limit_factor);
18      }
19
20      Literal literal = pick_branching_variable();
21      assign_literal_value(literal, -1);
22
23      while (!unit_propagation(literal)) {
24          m_state.conflict_counter++;
25          if (m_state.decision_level == 0)
26              return std::nullopt;
27          Clause conflict_clause =
28              m_state.clause_set[m_state.conflict_clause_idx];
29          const int backjump_level = conflict_analysis(conflict_clause);
30          backjump(backjump_level);
31
32          decay_literal_activities();
33          decay_clause_activities();
34
35          literal = m_state.clause_set.back()[0];
36          assign_literal_value(literal,
37              static_cast<int>(m_state.clause_set.size()) - 1);
38      }
39  }
40
41  return m_state.partial_valuation;
42 }
```

4.2 Генерисање DRAT доказа

У претходном поглављу наглашено је да су научене клаузе у оквиру анализе конфликта RUP-изведене, а самим тим и RAT-изведене клаузе. Ова кључна особина научених клауза омогућава једноставно проширивање дате имплементације решавача на функционалност генерисања DRAT доказа.

Наредбе у оквиру DRAT доказа представљаћемо структуром `DRATAct`, док DRAT доказ представљамо листом наредби и уводимо алијас `DRATProof`.

```
1 struct DRATAct {
2     enum Type : int8_t { Add, Delete } type;
3     Clause clause;
4 };
5 using DRATProof = std::vector<DRATAct>;
```

Повратна вредност функције `solve` се мења из опционе вредности типа `PartialValuation` у унију типова `PartialValuation` и `DRATProof`, имплементирани преко шаблонске класе `std::variant`.

```
1 class Solver {
2     ...
3     std::variant<PartialValuation, DRATProof>
4         solve(int atom_count, ClauseSet clause_set);
5     ...
6 };
```

Стање решавача проширује се променљивом типа `DRATProof` у оквиру које се генерише одговарајући DRAT доказ.

```
1 struct State {
2     ...
3     DRATProof drat_proof;
4 };
```

У оквиру анализе конфликта, приликом учења клаузе, у оквиру функције `learn`, научену клаузу додајемо у доказ.

```
1 void Solver::learn(const Clause &clause) {
2     m_state.clause_set.push_back(clause);
```

```

3     m_state.drat_proof.push_back(DRATAct{DRATAct::Add, clause});
4     ...
5 }

```

Са друге стране, приликом заборављања научених клауза, додајемо наредбе за брисање заборављених клауза из доказа.

```

1 void Solver::delete_clauses() {
2     ...
3     for (int i = 0; i < limit; i++) {
4         const int clause_delete_idx = candidates[i].second;
5         m_state.is_deleted[clause_delete_idx] = true;
6         m_state.drat_proof.push_back(DRATAct{DRATAct::Add,
7                                             m_state.clause_set[clause_delete_idx]});
8         m_state.learned_clause_count--;
9     }
10    ...
11 }

```

На крају, у оквиру имплементације функције `solve`, незадовољивост пријављујемо додавањем празне клаузе на крај доказа и прослеђивањем генерисаног доказа као повратне вредности.

```

1 std::variant<PartialValuation, DRATProof> Solver::solve(
2     const int atom_count, ClauseSet clause_set) {
3     ...
4     if (!unit_propagation(0)) {
5         m_state.drat_proof.push_back(DRATAct{DRATAct::Delete, Clause{}});
6         return m_state.drat_proof;
7     }
8
9     while (!all_variables_assigned()) {
10        ...
11        while (!unit_propagation(literal)) {
12            m_state.conflict_counter++;
13            if (m_state.decision_level == 0) {
14                m_state.drat_proof.push_back(DRATAct{DRATAct::Delete,
15                                                        Clause{}});

```

```

16         return m_state.drat_proof;
17     }
18     ...
19 }
20 }
21 ...
22 }

```

4.3 Генерисање LRAT доказа

У односу на DRAT доказе, при генерисању LRAT доказа потребно је уз сваку клаузу доказа дати и одговарајући низ клауза. Тај низ клауза је могуће генерисати на основу клауза које учествују у самој анализи конфликта.

Слично као код генерисања DRAT доказа, уводимо структуру `LRATAct` за представљање наредби LRAT доказа као и алијас `LRATProof` за представљање LRAT доказа као листе наредби.

```

1 struct LRATAct {
2     enum Type : int8_t { Add, Delete } type;
3     int counter;
4     Clause added_clause;
5     std::vector<int> clause_indices;
6 };
7 using LRATProof = std::vector<LRATAct>;

```

Клауза `added_clause` се у случају наредбе брисања поставља на празну клаузу и занемарује. Низ индекса клауза `clause_indices` у случају наредбе додавања клаузе представља индексе клауза у низу објашњења, док у случају наредбе брисања представља индексе клауза које се бришу.

Функција `solve` и стање решавача се модификују аналогно модификацијама код генерисања DRAT доказа. Са друге стране, пре додавања научених клауза у доказ, потребно је за сваку научену клаузу генерисати одговарајући низ објашњења. Пре почетка анализе конфликта, у оквиру функције `solve` потребно је иницијализовати наредбу додавања клаузе.

```

1 m_state.lrat_proof.push_back(LRATAct{

```

```

2   LRATAct::Add,
3   static_cast<int>(m_state.clause_set.size()) + 1,
4   Clause{},
5   std::vector<int>{}),
6 });

```

Сваку клаузу која учествује у анализи конфликта додајемо у низ објашњења.

```

1 Clause Solver::explain(const Clause &conflict_clause, int &backjump_level)
2 {
3   ...
4   m_state.lrat_proof.back().clause_indices.push_back(m_state.
5   conflict_clause_idx+1);
6   ...
7   if (m_state.seen[abs(assignment.literal)] == m_state.seen_counter
8       && assignment.unit_clause_idx != -1) {
9       m_state.lrat_proof.back().clause_indices
10      .push_back(assignment.unit_clause_idx+1);
11   }

```

У оквиру функције `learn`, слично као код генерисања DRAT доказа, научену клаузу додајемо у доказ.

```

1 void Solver::learn(const Clause &clause) {
2   ...
3   m_state.lrat_proof.back().added_clause = clause;
4   ...
5 }

```

Приликом брисања клауза, све клаузе које бришемо додајемо у једну наредбу брисања у LRAT доказу.

```

1 void Solver::delete_clauses() {
2   ...
3   m_state.lrat_proof.push_back(LRATAct{
4       LRATAct::Delete,
5       static_cast<int>(m_state.clause_set.size()),
6       Clause{}),

```

```
7         std::vector<int>{}  
8     });  
9     ...  
10    m_state.lrat_proof.back().clause_indices  
11                                   .push_back(clause_delete_idx+1);  
12    ...  
13 }
```

На крају, уколико решавач пријави незадовољивост, због генерисања низа објашњења је потребно спровести анализу конфликта. Међутим, ову анализу није потребно експлицитно спровести, већ је довољно за низ објашњења узети јединичне клаузе које се налазе на трагу, с обзиром на то да се у тренутку пријављивања незадовољивост, на трагу налазе искључиво доделе на основу елиминације јединичних клауза.

```
1 m_state.lrat_proof.push_back(LRATAct{  
2     LRATAct::Add,  
3     static_cast<int>(m_state.clause_set.size()) + 1,  
4     Clause{},  
5     std::vector<int>{},  
6 });  
7 m_state.lrat_proof.back().clause_indices.push_back(m_state.  
8     conflict_clause_idx+1);  
9 for (auto & it : std::ranges::reverse_view(m_state.trail)) {  
10     m_state.lrat_proof.back().clause_indices  
11                                   .push_back(it.unit_clause_idx+1);  
12 }
```

Глава 5

Примена на комбинаторне проблеме

Разни комбинаторни проблеми се математички могу свести на испитивање исказне задовољивости одговарајуће исказне формуле, односно на SAT проблем. Ефикасност модерних SAT решавача омогућило је њихову примену на решавање тешких и гломазних комбинаторних проблема. Један од познатијих комбинаторних проблема који је решен применом SAT решавача је проблем бојења природних бројева тако да ниједна Питагорина тројка није обојена истом бојом [13]. Велику примену на комбинаторне проблеме имају и решавачи за исказну логику првог реда, као и SMT решавачи, који користе SAT решаваче при свом извршавању. Њихова примена није од значаја за овај рад и даље неће бити разматрана.

Резолуцијски докази задовољивости се код комбинаторних проблема могу употребити као формалан математички доказ којим се оправдава добијени резултат. Уобичајено је да је величина ових доказа огромна, као што је случај са претходно поменутиим проблемом бојења природних бројева, где датотека која садржи одговарајући резолуцијски доказ заузима 200 терабајта меморије. Стога примена SAT решавача на тешке комбинаторне проблеме често захтева велику количину хардверских ресурса.

Друга примена резолуцијских доказа задовољивости код комбинаторних проблема је семантичке природе. На основу резолуцијског доказа незадовољивости, могуће је одредити *минимално језгро незадовољивости*, односно минимални скуп клауза који је незадовољив. Такво минимално језгро незадовољивости прецизније указује на кључне конфликтне клаузе услед којих је

дошло до незадовољивости. Код комбинаторних проблема, минимално језгро незадовољивости може дати семантичко значење доказу. Један од проблема при овом приступу јесу хеуристике које се користе при имплементацији SAT решавача. Хеуристике са једне стране доприносе ефикасности, док са друге стране доводе до непредвидљивог процеса претраге. С обзиром на то да минимално језгро незадовољивости не мора бити јединствено, могуће је да такво језгро не указује на јасно семантичко значење доказа. Вероватноћа да језгро указује на семантичко значење доказа је већа, што је величина самог доказа мања.

У наставку ће кроз два примера бити илустрована примена резолуцијских доказа незадовољивости при решавању комбинаторних проблема, при чему ће семантичко тумачење минималног језгра незадовољивости бити од посебног интереса.

5.1 Дирихлеов принцип

Дирихлеов принцип је позната математичка теорема која тврди да се при распоређивању m објеката у n кутија, при чему је $m > n$, у једној кутији морају наћи барем 2 објекта, односно да није могуће објекте распоредити у кутије тако да се у свакој кутији налази највише 1 објекат. Конкретне инстанце (фиксирани вредности m и n) Дирихлеовог принципа се могу представити као проблем исказне задовољивости. Обележимо објекте са $p_1, \dots, p_m$, а кутије са $h_1, \dots, h_n$ ¹. За све $i \in \{1, \dots, m\}$, $j \in \{1, \dots, n\}$, атом $q_{i,j}$ представља исказ "објекат p_i је распоређен у кутију h_j ". Исказна формула

$$F = \bigwedge_{1 \leq i \leq m} (q_{i,1} \vee \dots \vee q_{i,n}) \wedge \bigwedge_{\substack{1 \leq i \leq m \\ 1 \leq j' < j'' \leq n}} (\neg q_{i,j'} \vee \neg q_{i,j''}) \wedge \bigwedge_{\substack{1 \leq i' < i'' \leq m \\ 1 \leq j \leq n}} (\neg q_{i',j} \vee \neg q_{i'',j})$$

је задовољива ако и само ако је m објеката могуће распоредити у n кутија, тако да се у свакој кутији налази највише један објекат. У случају да је формула задовољива, одговарајући распоред може се одредити помоћу валуације која задовољава исказну формулу. У супротном, доказ незадовољивости можемо искористити при одређивању минималног језгра незадовољивости. Напоменимо да инстанце Дирихлеовог принципа представљају тешке инстанце за SAT решавач [7].

¹Ознаке p и h потичу од енглеског назива *pigeonhole principle*

Посматрајмо инстанцу Дирихлеовог принципа када $m > n + 1$, конкретно није $m = 4$, $n = 2$. Јасно је да се незадовољивост постиже у тренутку када трећи објекат покушамо да распоредимо у једну од кутија, јер је пре тога у свакој кутији већ распоређен један од објеката. Самим тим, четврти објекат је редундантан у смислу незадовољивости исказне формуле. То можемо приметити и одређивањем минималног језгра незадовољивости након покретања SAT решавача и генерисања одговарајућег доказа. Уколико атоме $q_{i,j}$ кодирамо формулом $(i - 1) \cdot n + j$, можемо генерисати одговарајућу формулу F у DIMACS CNF формату:

```

1 p cnf 8 20
2 1 2 0
3 3 4 0
4 5 6 0
5 7 8 0
6 -1 -2 0
7 -3 -4 0
8 -5 -6 0
9 -7 -8 0
10 -1 -3 0
11 -1 -5 0
12 -1 -7 0
13 -3 -5 0
14 -3 -7 0
15 -5 -7 0
16 -2 -4 0
17 -2 -6 0
18 -2 -8 0
19 -4 -6 0
20 -4 -8 0
21 -6 -8 0

```

Применом имплементираног решавача, уз генерисање одговарајућег доказа незадовољивости и употребом **drat-trim** алата за генерисање минималног језгра незадовољивости долазимо до језгра

```

1 p cnf 8 9
2 2 1 0

```

3	4	3	0
4	8	7	0
5	-3	-1	0
6	-1	-7	0
7	-3	-7	0
8	-4	-2	0
9	-2	-8	0
10	-4	-8	0

Овакво језгро јасно указује да 4 објекта није могуће распоредити у 2 кутије због тога што ни 3 објекта није могуће распоредити у 2 кутије. односно указује да је, у смислу незадовољивости, четврти објекат редундантан.

Генерисана исказна формула састоји се од 20 клауза и 8 атома, време решавања² развијеним решавачем је 0.09s, доказ у DRAT/LRAT формату садржи 2 корака, док незадовољиво језгро добијено помоћу тог доказа садржи 9 клауза и 6 атома (алат `drat-trim` пријављује постојање 8 атома у језгру, међутим само 6 атома се појављује у клаузама које чине језгро).

Скалирање доказа

Инстанце Дирихлеовог принципа је могуће једноставно генерисати за дате m и n . Самим тим, на овом примеру можемо приказати како се докази незадовољивости скалирају у односу на величину проблема тј. улазну исказну формулу. Испитивање како се докази скалирају је од посебног интереса за решавача ради утврђивања ефикасности као и њихових реалних ограничења. Ради компактнијег записа уводимо ознаке:

- `VarNo` – ознака за број променљивих у исказној формули
- `ClauseNo` – ознака за број клауза у исказној формули
- `SolveTime` – ознака за време решавања (развијеним решавачем)
- `ProofSize` – ознака за величину доказа у LRAT формату
- `VerifyTime` – ознака за време верификације доказа (алатом `lrat-check`)

²Време извршавања мерено је командом `time`, на конфигурацији: MacBook Pro 2021, M1 Pro Chip, 16GB RAM, оперативни систем macOS Tahoe 26.5.2

У наредној табели дати су резултати добијени при генерисању и верификовању доказа за Дирихлеов принцип за различите вредности бројева n , при чему за m узимамо вредност $n + 1$. Време решавања и време верификације² за свако n једнаки су просеку времена добијених у 5 независних покретања решавача и верификатора над одговарајућом исказном формулом, односно доказом.

n	VarNo	ClauseNo	SolveTime (s)	ProofSize	VerifyTime (s)
4	20	75	0.0868	28	0.0136
5	30	141	0.0963	162	0.0115
6	42	238	0.2113	971	0.0137
7	56	372	1.5339	6734	0.0297
8	72	549	21.1981	51932	0.1858
9	90	775	427.0561	301732	1.0576

5.2 Бојење графа

Проблем бојења чворова графа G са n боја, означених са $c_1, \dots, c_n$, можемо једноставно представити као проблем исказне задовољивости. Нека су чворови графа G означени са $v_1, \dots, v_m$. За све $i \in \{1, \dots, m\}$, $j \in \{1, \dots, n\}$, атом $p_{i,j}$ представља исказ "чвор v_i је обојен бојом c_j ". Исказна формула

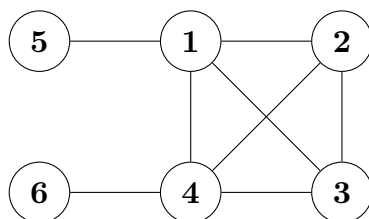
$$F = \bigwedge_{1 \leq i \leq m} (p_{i,1} \vee \dots \vee p_{i,n}) \wedge \bigwedge_{\substack{1 \leq i \leq m \\ 1 \leq j' < j'' \leq n}} (\neg p_{i,j'} \vee \neg p_{i,j''}) \wedge \bigwedge_{\substack{(v_i, v_k) \in E \\ 1 \leq j \leq n}} (\neg p_{i,j} \vee \neg p_{k,j})$$

је задовољива ако и само ако се чворови графа G може обојити са n боја тако да су суседни чворови обојени различитим бојама. У случају да је формула задовољива, одговарајуће бојење чворова можемо одредити помоћу валуације која задовољава исказну формулу. У супротном, доказ незадовољивости можемо искористити при одређивању минималног језгра незадовољивости.

Размотримо произвољни граф $G = (V, E)$ који садржи подграф $G' = (V', E')$ изоморфан комплетном графу K_{n+1} за неко $n \in \mathbb{N}$. Јасно је да се чворови графа G' не могу обојити помоћу n боја тако да свака два суседна чвора буду обојена различитом бојом. Самим тим, такво бојење није могуће ни за граф G , па ће одговарајућа исказна формула бити незадовољива. Очекујемо да ће одговарајуће минимално језгро незадовољивости јасно указати

да се граф G не може обојити са n боја зато што постоји његов подграф изоморфан са K_{n+1} .

Следећи граф



садржи комплетан граф K_4 као свој подграф, па се његови чворови не могу обојити са 3 боје. Уколико атоме $p_{i,j}$ кодирамо формулом $(i-1) \cdot n + j$, можемо генерисати одговарајућу формулу F у DIMACS CNF формату:

```

1 p cnf 18 48
2 1 2 3 0
3 4 5 6 0
4 7 8 9 0
5 10 11 12 0
6 13 14 15 0
7 16 17 18 0
8 -1 -2 0
9 -1 -3 0
10 -2 -3 0
11 -4 -5 0
12 -4 -6 0
13 -5 -6 0
14 -7 -8 0
15 -7 -9 0
16 -8 -9 0
17 -10 -11 0
18 -10 -12 0
19 -11 -12 0
20 -13 -14 0
21 -13 -15 0
22 -14 -15 0
23 -16 -17 0
24 -16 -18 0
25 -17 -18 0

```

```

26 -1 -4 0
27 -2 -5 0
28 -3 -6 0
29 -1 -7 0
30 -2 -8 0
31 -3 -9 0
32 -1 -10 0
33 -2 -11 0
34 -3 -12 0
35 -4 -7 0
36 -5 -8 0
37 -6 -9 0
38 -4 -10 0
39 -5 -11 0
40 -6 -12 0
41 -7 -10 0
42 -8 -11 0
43 -9 -12 0
44 -1 -13 0
45 -2 -14 0
46 -3 -15 0
47 -10 -16 0
48 -11 -17 0
49 -12 -18 0

```

Применом имплементираног решавача, уз генерисање одговарајућег доказа незадовољивости и употребом **drat-trim** алата за генерисање минималног језгра незадовољивости долазимо до језгра

```

1 p cnf 18 22
2 3 2 1 0
3 4 6 5 0
4 7 9 8 0
5 12 11 10 0
6 -4 -1 0
7 -5 -2 0
8 -6 -3 0
9 -7 -1 0

```

10	-8	-2	0
11	-9	-3	0
12	-1	-10	0
13	-2	-11	0
14	-3	-12	0
15	-7	-4	0
16	-8	-5	0
17	-9	-6	0
18	-4	-10	0
19	-5	-11	0
20	-6	-12	0
21	-7	-10	0
22	-8	-11	0
23	-9	-12	0

Овакво језгро јасно указује да граф није могуће обојити помоћу 3 боје због одговарајућег подграфа који је изоморфан комплетном графу K_4 .

Генерисана исказна формула састоји се од 48 клауза и 18 атома, време решавања² развијеним решавачем је 0.09s, доказ у DRAT/LRAT формату садржи 6 корака, док незадовољиво језгро добијено помоћу тог доказа садржи 22 клаузе и 12 атома (алат **drat-trim** пријављује постојање 18 атома у језгру, међутим само 12 атома се појављује у клаузама које чине језгро).

Глава 6

Закључак

У оквиру мастер рада детаљно су описани CDCL алгоритам за решавање SAT проблема, резолуцијски докази незадовољивости у RUP, DRUP, RAT, DRAT, LRAT форматима. Имплементиран је SAT решавач, унапређен разним техникама оптимизације, као и могућношћу генерисања DRAT/LRAT доказа незадовољивости. На крају приказане су и одговарајуће примене описаних доказа на комбинаторне проблеме.

Између осталог, DRAT/LRAT докази незадовољивости су математички формализовани, доказана је коректност RAT доказа незадовољивости, док је на мањим примерима приказана могућност семантичког тумачења доказа кроз одговарајућа минимална језгра незадовољивости. Истакнут је значај доказа незадовољивости у случају употребе SAT решавача за формално доказивање математичких теорема, односно решавање математичких проблема.

Даљи рад може укључивати примене додатних техника и оптимизација при имплементацији решавача и испитивање њихових утицаја на генерисање доказа. С обзиром на то да оптимизације вођене хеуристикама доводе до непредвидљивог процеса претраге, што даље утиче на генерисање доказа као и на одговарајуће минимално језгро незадовољивости, даљи рад се може усмерити ка превазилажењу оваквих ефеката ради генерисања језгра незадовољивости које указује на семантичко значење доказа.

Библиографија

- [1] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *J. Artif. Int. Res.*, 22(1):319–351, December 2004.
- [2] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
- [3] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.
- [4] Luís Cruz-Filipe, Marijn J. H. Heule, Warren A. Hunt, Matt Kaufmann, and Peter Schneider-Kamp. *Efficient Certified RAT Verification*, page 220–236. Springer International Publishing, 2017.
- [5] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.
- [6] Allen Van Gelder. Verifying RUP Proofs of Propositional Unsatisfiability. In *International Symposium on Artificial Intelligence and Mathematics*, 2008.
- [7] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985. Third Conference on Foundations of Software Technology and Theoretical Computer Science.
- [8] Marijn J. H. Heule. Trimming while Checking Clausal Proofs. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 181–188. IEEE, 2013.

- [9] Marijn J. H. Heule, Warren A. Hunt, and Nathan Wetzler. Verifying refutations with extended resolution. In *Proceedings of the 24th International Conference on Automated Deduction, CADE'13*, page 345–359, Berlin, Heidelberg, 2013. Springer-Verlag.
- [10] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI'07*, page 2318–2323, San Francisco, CA, USA, 2007. Morgan Kaufmann Publishers Inc.
- [11] Predrag Janičić. *Matematička logika u računarstvu*. Matematički fakultet, 2009.
- [12] Sava Krstić and Amit Goel. Architecting Solvers for SAT Modulo Theories: Nelson-Oppen with DPLL. In *Proceedings of the 6th International Symposium on Frontiers of Combining Systems, FroCoS '07*, page 1–27, Berlin, Heidelberg, 2007. Springer-Verlag.
- [13] Evelyn Lamb. Two-hundred-terabyte maths proof is largest ever. *Nature*, 534(7605):17–18, 2016.
- [14] Filip Marić. *Formalizacija, implementacija i primene SAT rešavaca*. PhD thesis, University of Belgrade, Serbia, 2009.
- [15] Filip Marić. Formalization and Implementation of Modern SAT Solvers. *Journal of Automated Reasoning*, 43:81–119, 06 2009.
- [16] Filip Marić and Predrag Janičić. Formalization of Abstract State Transition Systems for SAT. *Logical Methods in Computer Science*, Volume 7, Issue 3, September 2011.
- [17] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535. ACM, 2001.
- [18] Niklas Sörensson and Niklas Een. Minisat v1.13-a SAT solver with conflict-clause minimization. *International Conference on Theory and Applications of Satisfiability Testing*, 01 2005.

- [19] Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 422–429, Cham, 2014. Springer International Publishing.
- [20] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '01, page 279–285. IEEE Press, 2001.

Биографија аутора

Милан Митрески је рођен 23. октобра 2001. године у Београду. Завршио је основну школу "Јелена Тетковић", а потом и Шесту београдску гимназију. Основне студије на Математичком факултету, Универзитета у Београду уписао је 2020. године на студијском програму Математика, модул Рачунарство и информатика и завршио их 2024. године, са просечном оценом 10,00. Мастер студије на истом студијском програму и модулу, уписао је 2024. године. Запослен је на Математичком факултету од октобра 2024. године, на Катедри за рачунарство и информатику, у звању сарадника у настави и држао је вежбе из предмета Рачунарске мреже, Увод у релационе базе података, Дизајн и анализа алгоритама, Увод у алгоритме, Елементарна математика. Члан је Студентског парламента од 2022. године, а обавља функцију студента продекана од марта 2024. године. Члан је Државне комисије за математичка такмичења ученика средњих школа, од 2021. године. Био је члан студентске организације "Студентска унија математичара - СУМА" у периоду од 2021. до 2024. године.