УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Димитрије Петровић

ВИЗУАЛИЗАЦИЈА КНУТОВОГ АЛГОРИТМА ЗА ПРОФАЈЛИРАЊЕ ГРАНА ГРАФА КОНТРОЛЕ ТОКА

мастер рад

Београд, 2026.

Ментор:

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

Чланови комисије:

др Мирко Спасић, доцент
Универзитет у Београду, Математички факултет

др Иван Ристовић, асистент
Универзитет у Београду, Математички факултет

Датум одбране: _____

Наслов мастер рада: Визуализација Кнутовог алгоритма за профајлирање грана графа контроле тока

Резиме: Подаци о учесталости извршавања појединих делова програма представљају важан извор информација за оптимизацију програма и прикупљају се применом алата за профајлирање. Приступ који подразумева постављање бројача на сваку грану графа контроле тока може довести до непотребног успоравања извршавања програма, будући да се вредности великог броја бројача могу извести из осталих измерених вредности. Кнутов алгоритам смањује потребан број мерења на гране које не припадају једном разапињућем стаблу графа, док се преостале вредности одређују применом закона одржања протока. На овом принципу заснована је инструментација у савременим преводиоцима, попут *GCC*-а и *LLVM*-а, при чему су кораци алгоритма скривени од корисника.

У оквиру овог рада развијена је веб апликација *Knuth Profiler*, намењена визуелизацији и интерактивном праћењу примене Кнутовог алгоритма. Апликација омогућава извођење алгоритма корак по корак над графовима контроле тока, како над уграђеним примерима, тако и над кодом који корисник унесе на једном од четири подржана програмска језика: *C/C++*, *Java*, *Python* или *JavaScript*. Граф контроле тока за унети програм добија се применом алата за статичку анализу *Joern*. Исправност имплементације проверена је над насумично генерисаним графовима, верификацијом шест особина: величине и повезаности разапињућег стабла, броја инструментованих грана, поделе свих грана између стабла и мереног скупа, доделе тежина, тачности реконструисаних бројача на свим гранама и закона одржања протока у сваком чвору. Експериментална евалуација на 10.000 графова показала је да се у просеку инструментује мање од четвртине грана, чиме се остварује уштеда од приближно 85% операција увећавања бројача у односу на приступ који инструментује све гране графа.

Кључне речи: профајлирање грана, граф контроле тока, Кнутов алгоритам, оптимално разапињуће стабло, инструментација, визуелизација, *Joern*

Садржај

1 Увод **1**

2 Кнутов алгоритам за минималну инструментацију у графу контроле тока **3**

2.1 Граф контроле тока 3

2.1.1 Формална дефиниција 4

2.1.2 Путање и циклуси 5

2.2 Проблем минималне инструментације 7

2.2.1 Инструментација и услов реконструкције 7

2.2.2 Формализација циља минимизације 8

2.3 Минималан скуп инструментованих грана 9

2.3.1 Конструкција преко разапињућег стабла 9

2.3.2 Доказ минималности 11

2.4 Реконструкција бројача неинструментованих грана 13

2.4.1 Једначине одржања протока 13

2.4.2 Поступак решавања 13

2.5 Избор разапињућег стабла 15

2.5.1 Додела тежина гранама 16

2.5.2 Оптимално разапињуће стабло 18

3 Технологије веб апликације и статичке анализе **20**

3.1 Клијентске технологије 20

3.2 Серверске технологије 22

3.3 Алат за статичку анализу кода *Joern* 23

3.4 Контејнеризација и инфраструктура 23

4 Алат *Knuth Profiler* **25**

4.1 Преглед система 25
4.1.1 Архитектура система 26
4.1.2 Покретање и међузависност сервиса 27
4.1.3 Заједнички модел података 28
4.2 Извлачење графа контроле тока алатом *Joern* 29
4.2.1 Формат *DOT* и унапред покренут *Joern* сервис 29
4.2.2 Парсирање и нормализација графа 31
4.3 Примена алгоритма у апликацији 32
4.3.1 Додела тежина 32
4.3.2 Конструкција разапињућег стабла 33
4.3.3 Избор инструментованих грана 34
4.3.4 Симулација извршавања 34
4.3.5 Реконструкција бројача 35
4.4 Корисничко искуство и едукативни ток 36
4.4.1 Кораци визуелизације 36
4.4.2 Рад над примерима и увоз сопственог кода 37

5 Евалуација **41**
5.1 Оквир за проверу исправности 41
5.1.1 Генератор графова контроле тока 41
5.1.2 Особине које се проверавају 42
5.1.3 Однос према постојећим тестовима 44
5.2 Резултати над већим скупом графова 44
5.3 Примери 46
5.3.1 Пример: линеарни ток и једноставно гранање 46
5.3.2 Пример: програм са петљом 48
5.3.3 Увоз корисничког кода кроз цео ток 50
5.4 Дискусија резултата и ограничења 53

6 Закључак **55**

Литература **57**

Глава 1

Увод

Разумевање понашања програма током извршавања темељ је многих поступака у развоју софтвера. Подаци о томе који се делови програма извршавају и колико често усмеравају оптимизацију, откривају никада извршени код и мере темељност тестирања. Поступак којим се такви подаци прикупљају назива се профајлирање (енгл. *profiling*) и спада у динамичке технике верификације софтвера [1].

Програм се при томе посматра преко графа контроле тока. Његови чворови су основни блокови, праволинијски низови наредби без гранања, а гране могући прелази између блокова. Профајлирање грана утврђује колико је пута свака грана тог графа извршена. Из тих вредности се сабирањем добија и број извршавања сваког блока, па оне носе потпуну слику тока кроз програм [1].

Најједноставнији приступ профајлирању грана уграђује бројач у сваку грану. Такав приступ непотребно успорава програм, пошто су вредности многих бројача међусобно зависне. Број улазака у блок једнак је броју излазака из њега, па се вредност једне гране може израчунати из вредности осталих. Доналд Кнут (енгл. *Donald Knuth*) показао је да је довољно инструментовати гране које не припадају неком разапињућем стаблу графа, а вредности преосталих грана једнозначно израчунати из тог односа [2]. Тих грана има тачно $e - n + 1$, где су n и e редом број чворова и број грана графа. Ако се мери и једна грана мање, вредности преосталих више се не могу одредити. Избор разапињућег стабла не мења тај број, него које се гране мере. Стварне учесталости грана нису познате пре мерења, па се процењују из облика самог графа. Мерење се затим усмерава на гране са најмањом проценом, па је укупан трошак профајлирања мањи.

Бал и Ларус су Кнутов поступак разрадили и уградили у ширу теорију оптималног профајлирања и праћења програма [3], а данас на њему почива инструментација у самим преводиоцима. Преводилац *GCC* за сваку функцију гради граф контроле тока, налази разапињуће стабло и уграђује бројаче само у гране изван њега [4]. Инфраструктура *LLVM* исти поступак изводи над међукодом, а њена имплементација се у изворном коду позива управо на Кнутов и Стивенсонов рад [5].

Постоје платформе за визуелизацију алгоритама, које корак по корак приказују класичне структуре података и алгоритме над улазом који корисник сам зада [6]. Међу алгоритмима које оне обухватају нема оних које изводе преводиоци. Циљ овог рада је да се поступак који у преводиоцима тече невидљиво изведе пред корисником, над графом који он сам бира, ради разумевања а не ради мерења. У оквиру рада развијена је веб апликација *Knuth Profiler*, која не описује Кнутов алгоритам, него га изводи. Корисник кроз његове кораке пролази један по један, над графом који има пред собом, при чему на сваком кораку види шта се променило и због чега. Апликација ради над уграђеним примерима карактеристичних облика тока, али и над кодом који корисник сам унесе, у четири програмска језика: *C/C++*, *Java*, *Python* и *JavaScript*. Из тог кода граф контроле тока извлачи алат за статичку анализу *Joern* [7]. Рад одговара на три питања: како се Кнутов алгоритам спроводи над стварним графовима контроле тока, како се показује да је то спровођење исправно и колику уштеду поступак доноси у пракси.

Рад је организован на следећи начин. Најпре су изложене формалне основе алгоритма и доказ минималности (глава 2). Затим су представљене технологије на којима апликација почива (глава 3). Средишњи део рада описује архитектуру и имплементацију апликације (глава 4). Исправност и учинак решења испитани су над насумично генерисаним графовима и на конкретним примерима (глава 5). На крају су изведени закључци и назначени правци будућег рада (глава 6).

Глава 2

Кнутов алгоритам за минималну инструментацију у графу контроле тока

Профајлирање је техника прикупљања података о понашању програма током извршавања, попут учесталости извршавања делова кода или потрошње ресурса. Разматра се у контексту динамичких техника верификације и анализе софтвера [1].

Један од приступа профајлирању заснива се на инструментацији. Инструментација представља уметање додатних инструкција у програм, које прикупљају податке о његовом извршавању. Те додатне инструкције извршавају се заједно са програмом, па инструментовани програм троши више времена и меморије од полазног. Услед тога је неопходно свести број уметнутих инструкција на минимум. Захтев је посебно изражен у системима са строгим временским и меморијским ограничењима. У таквим системима инструментација може прекорачити задате границе и тиме онемогућити прикупљање података. Кнутов алгоритам решава овај проблем минимизације конструкцијом оптималног разапињућег стабла графа контроле тока.

2.1 Граф контроле тока

Граф контроле тока представља полазни модел над којим се формулишу и проблем минималне инструментације и алгоритам који га решава. У наставку овог одељка дата је формална дефиниција графа контроле тока, а затим су

уведени појмови путање и циклуса неопходни за касније разматрање Кнутовог алгоритма.

2.1.1 Формална дефиниција

Граф контроле тока (енгл. *control flow graph*) представља стандардну апстракцију програма. Она омогућава анализу могућих редоследа извршавања наредби, независно од конкретних улазних вредности са којима се програм извршава [8]. Уместо да се посматра свака појединачна наредба, наредбе се групишу у *основне блокове* (енгл. *basic block*). Основни блок је максималан низ наредби такав да се, ако ток извршавања уђе у његову прву наредбу, све наредбе блока извршавају редом. Прелаз контроле између два основна блока, на пример условни или безусловни скок, у графу је представљен граном.

У овом раду се посматра граф контроле тока једне, унапред одабране функције програма. У наставку се та функција означава са P[1].

Граф контроле тока функције P формално се дефинише као усмерен граф $G = (V, E)$, где је:

- V скуп чворова, при чему сваки чвор $v \in V$ представља један основни блок функције P;
- E скуп усмерених грана, при чему грана $(u, v) \in E$ постоји ако и само ако се, након извршавања основног блока u, контрола извршавања може директно пренети на основни блок v.

Број чворова графа G означава се са $n = |V|$, а број грана са $e = |E|$.

За чвор $v \in V$ уводи се скуп његових улазних грана $\mathrm{In}(v) = \{(u, v) \in E\}$ и скуп његових излазних грана $\mathrm{Out}(v) = \{(v, w) \in E\}$. Грана је *инцидентна* чвору v ако у том чвору почиње или се у њему завршава. Скуп инцидентних грана чвора v је зато $\mathrm{In}(v) \cup \mathrm{Out}(v)$.

Граф G садржи два истакнута чвора. Чвор *ENTRY* представља почетну тачку извршавања функције P, а чвор *EXIT* завршетак његовог извршавања. По уобичајеној конвенцији [8], то су вештачки чворови. Они не представљају стваран основни блок програма, већ само обележавају почетак и крај извршавања. Чвор *ENTRY* нема улазних грана, а чвор *EXIT* нема излазних.

[1] Такав граф назива се интрапроцедурални (енгл. *intraprocedural*) граф контроле тока. Позив друге функције у њему је обична наредба унутар основног блока, а не грана ка графу позване функције.

Да би се сви чворови разматрали на исти начин, граф G се проширује до графа G^+. Он се дефинише као $G^+ = (V \cup \{s, t\}, E \cup \{(s, ENTRY), (EXIT, t)\})$, где су s и t нови, помоћни чворови. Чвор $ENTRY$ тиме добија улазну грану из чвора s, а чвор $EXIT$ излазну грану ка чвору t. Гране $(s, ENTRY)$ и $(EXIT, t)$ називају се *граничним гранама*.

Помоћни чворови s и t не представљају ниједан део програма и имају тачно по једну инцидентну грану. Граничне гране носе укупан број позива функције P. Помоћни чворови не улазе у број чворова n графа G, а граничне гране не улазе у број грана e нити се икада бирају за мерење. У графу G^+ сваки чвор графа G има и улазне и излазне гране.

Без губитка општости, претпоставља се да је сваки чвор графа G достижан из чвора $ENTRY$ и да из сваког чвора постоји пут до чвора $EXIT$. Чвор недостижан из чвора $ENTRY$ одговара делу кода који се никада не извршава. Чвор из кога не постоји пут до чвора $EXIT$ одговара делу кода у коме извршавање остаје заувек, попут намерне бесконачне петље. Профајлирање, међутим, бројаче очитава по завршетку извршавања. Извршавање које се заврши не пролази ни кроз једну од ове две врсте чворова, па су сви њихови бројачи унапред познате нуле. Мерење на таквим чворовима зато не би носило никакву информацију. Граф који претпоставку не испуњава своди се на граф који је испуњава једноставним претпроцесирањем. Претрагом од чвора $ENTRY$ у смеру грана и претрагом од чвора $EXIT$ супротно смеру грана утврђује се које чворове свака од њих обилази. Чворови које бар једна претрага не обиђе уклањају се заједно са својим инцидентним гранама.

Листинг 2.1 приказује кратку функцију са гранањем и петљом, а слика 2.1 њен граф контроле тока. Овај пример се користи кроз цело поглавље, ради илустрације уведених појмова.

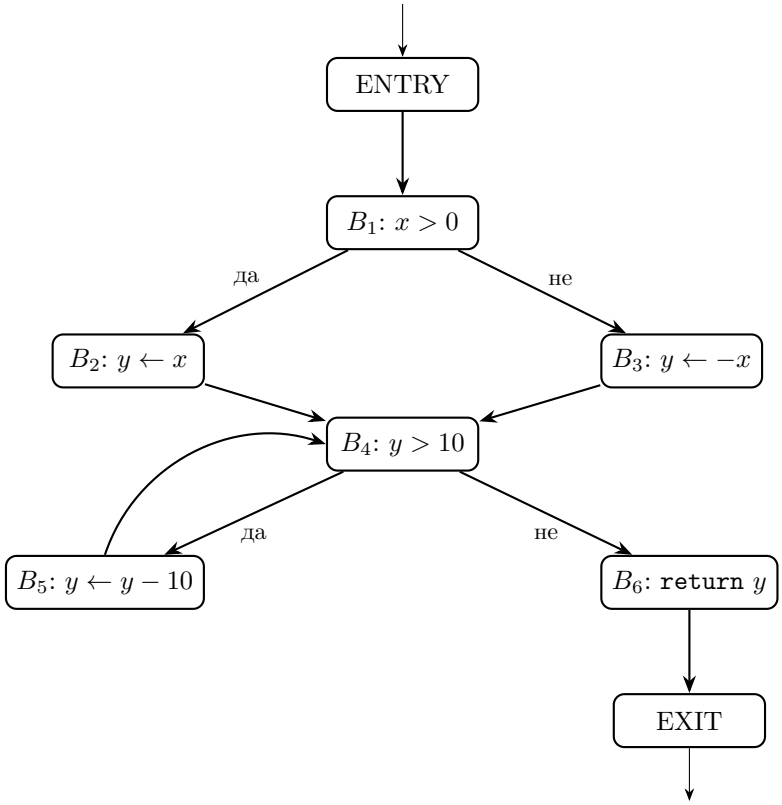
2.1.2 Путање и циклуси

Путања у графу контроле тока $G = (V, E)$ је низ чворова $v_0, v_1, \ldots, v_k$ такав да $(v_i, v_{i+1}) \in E$ за свако $i = 0, 1, \ldots, k-1$. Путања за коју важи $v_0 = ENTRY$ и $v_k = EXIT$ одговара једном могућем потпуном току извршавања функције P, посматраном на нивоу основних блокова.

Циклус је путања $v_0, v_1, \ldots, v_k$ код које је $v_0 = v_k$ и $k \geq 1$. Циклуси у графу контроле тока одговарају наредбама контроле тока као што су `while` и `for`. Граф контроле тока не описује услове под којима се петља прекида, него само

```
int f(int x) {
    int y;
    if (x > 0)
        y = x;
    else
        y = -x;
    while (y > 10)
        y = y - 10;
    return y;
}
```

Листинг 2.1: Функција `f` у програмском језику *C*, са гранањем и петљом.



Слика 2.1: Граф контроле тока функције `f` из листинга 2.1. Чворови B_1 до B_6 одговарају основним блоковима те функције, а гране могућим прелазима контроле. Танке стрелице изнад чвора *ENTRY* и испод чвора *EXIT* су граничне гране; њихови помоћни чворови се не цртају.

могуће прелазе контроле. Број обилазака циклуса зато на нивоу графа није ограничен, иако у самом програму јесте кад год је петља коначна. Услед тога је број различитих путања од чвора *ENTRY* до чвора *EXIT* бесконачан чим граф садржи бар један циклус.

2.2 Проблем минималне инструментације

Када је граф контроле тока функције P дефинисан, могуће је прецизно формулисати проблем минималне инструментације за профајлирање грана. Профајлирање грана (енгл. *edge profiling*) утврђује, за дато извршавање или скуп извршавања функције P, колико је пута свака грана графа G заиста извршена. У овом одељку најпре је уведен појам инструментације и закон одржања протока, који омогућава да се вредности за неинструментоване гране одреде на основу вредности измерених за инструментоване гране. На основу тога је затим формализован циљ минимизације броја инструментованих грана.

2.2.1 Инструментација и услов реконструкције

Нека је $c\colon E \to \mathbb{N}_0$ функција која сваку грану $e \in E$ графа контроле тока $G = (V, E)$ слика у број њених извршавања (пролаза) током профајлираног извршавања функције P. Грана $e \in E$ је *инструментована* ако је програм P измењен тако да се вредност $c(e)$ директно мери током извршавања. Типичан начин за то је уметање бројача који се увећава сваки пут када се грана e изврши. Инструментација сваке гране графа G би омогућила потпуно познавање функције c. Оваква исцрпна инструментација, међутим, значајно увећава потрошњу временских и меморијских ресурса [1]. Циљ је зато одредити много мањи скуп грана чија инструментација и даље омогућава потпуно познавање функције c.

Кључно запажање које то омогућава јесте да се број извршавања блока може прочитати и са његових улазних и са његових излазних грана. Збир вредности улазних грана чвора једнак је збиру вредности његових излазних грана; то својство се назива *закон одржања протока* [9]. Исти услов важи за јачине струја у чвору електричне мреже, где је познат као први закон Густава Кирхофа (нем. *Gustav Kirchhoff*). За сваки чвор $v \in V$, посматран у графу G^+, важи

$$\sum_{e \in \mathrm{In}(v)} c(e) \;=\; \sum_{e \in \mathrm{Out}(v)} c(e).$$

Збир пролаза кроз улазне гране чвора v представља број улазака у тај чвор. Он је једнак броју напуштања чвора v, збиру пролаза кроз његове излазне

гране. Ово важи зато што свако извршавање блока v у функцији P доводи до тачно једног уласка у чвор v и тачно једног напуштања тог чвора.

Једначина се може дефинисати за сваки чвор графа G, укључујући чворове *ENTRY* и *EXIT*, чије граничне гране у њу улазе као и све остале. За помоћне чворове се она не дефинише, пошто имају по једну инцидентну грану. Сумирањем једначина одржања протока по свим чворовима графа G добија се да су вредности функције c на две граничне гране међусобно једнаке. Та заједничка вредност представља укупан број извршавања (позива) функције P. У наставку рада она се назива *гранична вредност* и означава са N. Једначине одржања протока саме не одређују вредност N. Зато се претпоставља да је она унапред позната, независно од избора скупа инструментованих грана S. У пракси јој одговара јединствен бројач позива функције P, уобичајен и независно од технике профајлирања грана којом се овај рад бави.

Закон одржања протока омогућава да се вредност $c(e)$ за неинструментовану грану e одреди и без њеног директног мерења. Ако је за чвор v позната вредност функције c на свим његовим инцидентним гранама осим на тачно једној, та једна вредност се једнозначно израчунава из једначине одржања протока за чвор v. Поступак се затим понавља, чвор по чвор, све док постоји чвор са тачно једном још непознатом инцидентном граном. За дати скуп инструментованих грана $S \subseteq E$ кажемо да задовољава *услов реконструкције* ако се овим поступком могу одредити вредности $c(e)$ за сваку грану $e \in E$. Полазне вредности поступка су измерене вредности $c(e)$ за $e \in S$ и гранична вредност N.

2.2.2 Формализација циља минимизације

Проблем минималне инструментације графа контроле тока $G = (V, E)$ може се сада формулисати као проблем одређивања скупа грана

$$S^* = \underset{S \subseteq E,\ S \text{ задовољава услов реконструкције}}{\arg\min} |S|,$$

односно скупа инструментованих грана најмање величине који задовољава услов реконструкције.

Приметимо да услов реконструкције не задовољава произвољан подскуп грана. Скуп $S = E$ тривијално задовољава услов, јер су све вредности директно измерене. Он, међутим, није минималан. Скуп $S = \emptyset$ у општем случају не

задовољава услов, јер нема ниједне измерене вредности. Штавише, испуњеност услова није одређена само величином скупа S. Два скупа исте величине могу се разликовати по томе да ли услов задовољавају.

Два таква скупа, на графу из претходног примера, приказана су на слици 2.2. Нека је $S_1 = \{(B_1, B_2),\ (B_4, B_5)\}$, са измереним вредностима c_{12} и c_{45}. Обе граничне гране носе граничну вредност N. Једначина чвора $ENTRY$ тако даје $c(ENTRY, B_1) = N$, а једначина чвора $EXIT$ даје $c(B_6, EXIT) = N$. Полазећи од њих, једначине чворова B_1, B_2, B_3, B_5 и B_4 редом одређују преосталих пет грана. Изведене вредности свих грана, изражене преко N, c_{12} и c_{45}, приказане су на слици. Скуп S_1 зато задовољава услов реконструкције.

Нека је сада $S_2 = \{(B_1, B_2),\ (B_1, B_3)\}$, са измереним вредностима c_{12} и c_{13}. Од скупа S_1 разликује се само у једној грани, па има исту величину. Једначине чворова B_2, B_3 и B_6 одређују три гране, а затим се поступак зауставља. Гране (B_4, B_5) и (B_5, B_4) остају непознате. Обе једначине које их садрже, оне чворова B_4 и B_5, садрже обе те гране, па се увећање обеју вредности за исти број у њима поништава. Број извршавања тела петље зато није једнозначно одређен, па скуп S_2 не задовољава услов реконструкције. Питање није само колико грана се инструментује, него и које.

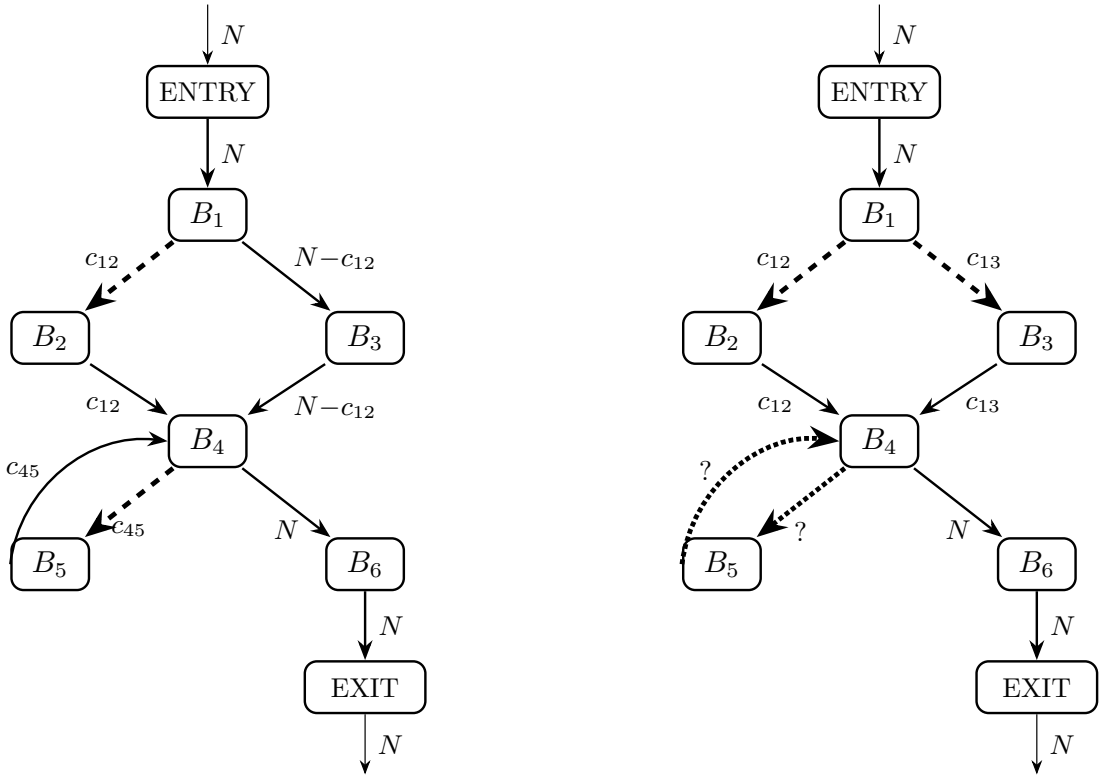
2.3 Минималан скуп инструментованих грана

У овом одељку показано је да се проблем минималне инструментације решава конструкцијом разапињућег стабла графа контроле тока [2][2]. Најпре је описана конструкција скупа инструментованих грана (одељак 2.3.1), а затим је доказано да мањи такав скуп не постоји (одељак 2.3.2).

2.3.1 Конструкција преко разапињућег стабла

Нека је $G_u = (V, E_u)$ неусмерени граф добијен од графа G занемаривањем смера његових грана. Полази се од графа G. Граничне гране, чија је вредност унапред позната, не улазе у конструкцију стабла. Свака усмерена грана $e =$

[2]Основни математички резултат на коме се заснива овај поступак, минималан број мерних места потребних да би се одредио ток у графу који задовољава закон одржања, први је доказао А. Нахапетјан (енгл. *A. Nahapetian*) [9]. Он је то учинио у општем, апстрактном контексту теорије графова, независно од примене на профајлирање програма. Кнут и Стивенсон (енгл. *Donald Knuth*, *Francis Stevenson*) су овај резултат применили на граф контроле тока програма и формулисали га преко оптималног разапињућег стабла [2].



(а) скуп S_1 (б) скуп S_2

Слика 2.2: Два скупа инструментованих грана исте величине, на графу контроле тока са слике 2.1. Испрекидане гране су мерене, а пуне су оне чија је вредност одређена. Ознака c_{ij} је измерена вредност гране од чвора B_i ка чвору B_j, а N је гранична вредност. Тачкасте гране остају неодређене и обележене су упитником.

$(u, v) \in E$ задржава свој идентитет, само се занемарује њено усмерење, тако да E_u и E имају исту кардиналност, $|E_u| = |E| = e$. Идентитет је битан кад две гране спајају исти пар чворова, као гране (B_4, B_5) и (B_5, B_4) у примеру са слике 2.1. Нека је T разапињуће стабло графа G_u.

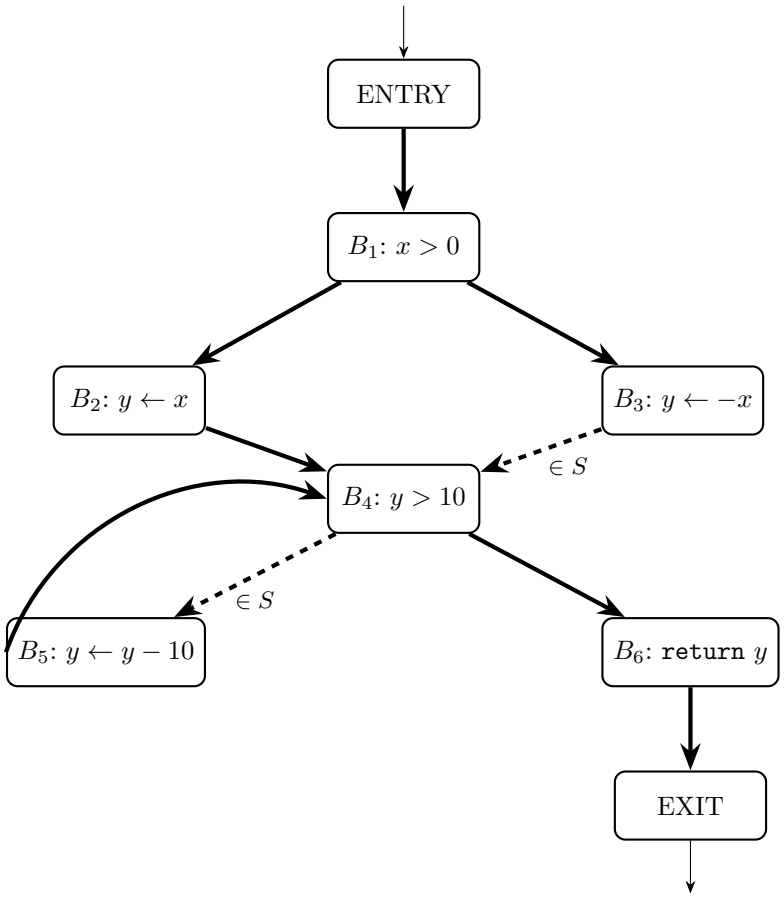
Скуп инструментованих грана дефинише се као

$$S = E \setminus T,$$

односно као скуп грана графа G које не припадају разапињућем стаблу T. Пошто разапињуће стабло повезаног графа са n чворова садржи тачно $n - 1$ грана, важи

$$|S| = e - (n - 1) = e - n + 1.$$

Слика 2.3 приказује примену овог поступка на примеру са слике 2.1. Пуним линијама су обележене гране разапињућег стабла T, а испрекиданим гране скупа S које се инструментују. За овај граф важи $n = 8$ и $e = 9$, па је $|S| = 9 - 8 + 1 = 2$.



Слика 2.3: Разапињуће стабло T (пуне линије) и инструментоване гране S (испрекидане линије) за пример са слике 2.1.

2.3.2 Доказ минималности

Тврђење. За повезан граф контроле тока $G = (V, E)$ са $n = |V|$ чворова и $e = |E|$ грана, сваки скуп $S \subseteq E$ који задовољава услов реконструкције испуњава $|S| \geq e - n + 1$. Постоји и скуп који услов задовољава и има тачно $e - n + 1$ грана, па је та граница најмања могућа [2].

Само тврђење је познато из рада Кнута и Стивенсона [2]. Доказ изложен у наставку је самосталан, прилагођен ознакама и граничној конвенцији који

су уведени у овом раду.

Доказ (доња граница). Нека $S \subseteq E$ задовољава услов реконструкције. Претпоставимо да неусмерени граф $(V, E_u \setminus S)$ (комплемент скупа S у графу G_u) садржи циклус C. Вредности $c(e)$ за гране циклуса C повезане су искључиво једначинама одржања протока чворова на C. Фиксирајмо произвољан смер обиласка циклуса C. Сваку његову грану означимо знаком $+1$ ако се њена усмереност у графу G слаже с тим смером, односно знаком -1 ако му је супротна. За произвољну константу δ, додавање вредности δ, помножене одговарајућим знаком, свакој грани циклуса C не нарушава ниједну једначину одржања протока. У сваком чвору циклуса C доприноси његове две циклусне гране, са својим знацима, увек се међусобно потиру у једначини одржања протока за тај чвор. Ово важи без обзира на граничну вредност N. Граничне гране нису део графа G_u, па не могу лежати на циклусу C; уз то, свака од њих спаја помоћни чвор, који има само једну инцидентну грану, са остатком графа. Зато гранична вредност не утиче на одређеност грана циклуса C. Отуда вредности грана циклуса C нису једнозначно одређене ни уз потпуно познавање свих преосталих вредности. То је у супротности са претпоставком да скуп S задовољава услов реконструкције. Закључујемо да $(V, E_u \setminus S)$ нема циклуса, односно да представља шуму над n чворова, са највише $n-1$ грана, одакле

$$|E_u \setminus S| \leq n-1 \quad \Longrightarrow \quad |S| \geq e-(n-1) = e-n+1.$$

Доказ (достижност). Нека је T разапињуће стабло графа G_u и $S = E \setminus T$. Тада важи $|S| = e-n+1$. Полазно су познате измерене вредности на гранама скупа S и гранична вредност N на обе граничне гране. Пошто стабло T не садржи циклус, у њему увек постоји лист, чвор чија је тачно једна инцидентна грана стабла још непозната. Све остале инцидентне гране тог чвора су познате: или директно инструментоване, или граничне, или већ реконструисане у претходном кораку. Вредност те једне непознате гране одређује се применом једначине одржања протока за дати чвор, а такву једначину има сваки чвор графа G. Чвор и његова грана се затим уклањају из даљег разматрања, а поступак понавља на преосталом, мањем стаблу. Стабло са k чворова има лист све док је $k \geq 2$, па се поступак изводи тачно $n-1$ пута и одређује вредности свих $n-1$ грана стабла T. Тиме је вредност $c(e)$ одређена за сваку грану $e \in T$. Скуп $S = E \setminus T$ зато задовољава услов реконструкције, чиме је доња граница достигнута. ■

2.4 Реконструкција бројача неинструментованих грана

Доказ минималности већ садржи поступак којим се, на основу познатих вредности $c(e)$ за инструментоване гране $e \in S$, одређују и вредности за гране разапињућег стабла T. У овом одељку тај поступак је формализован као самосталан алгоритам, на начин погодан за директну примену у имплементацији. Најпре је уопштена једначина одржања протока (одељак 2.4.1), а затим је описан поступак њеног поновљеног примењивања до потпуне реконструкције (одељак 2.4.2).

2.4.1 Једначине одржања протока

Поступак реконструкције почива на једначини одржања протока. За чвор $v \in V$, посматран у графу G^+, она гласи

$$\sum_{e \in \mathrm{In}(v)} c(e) \;=\; \sum_{e \in \mathrm{Out}(v)} c(e).$$

Ако је за чвор v позната вредност $c(e)$ за све гране из $\mathrm{In}(v) \cup \mathrm{Out}(v)$ осим за тачно једну непознату грану e_0, њена вредност се директно израчунава из ове једначине, у зависности од тога да ли је e_0 улазна или излазна грана чвора v:

$$c(e_0) = \begin{cases} \displaystyle\sum_{e \in \mathrm{Out}(v)} c(e) \;-\; \sum_{e \in \mathrm{In}(v) \setminus \{e_0\}} c(e), & e_0 \in \mathrm{In}(v), \\ \displaystyle\sum_{e \in \mathrm{In}(v)} c(e) \;-\; \sum_{e \in \mathrm{Out}(v) \setminus \{e_0\}} c(e), & e_0 \in \mathrm{Out}(v). \end{cases}$$

У оба случаја непозната вредност је разлика двају збирова. Од збира на страни једначине супротној грани e_0 одузима се збир познатих вредности на страни на којој та грана стоји.

2.4.2 Поступак решавања

Нека је T разапињуће стабло, $S = E \setminus T$ скуп инструментованих грана и $c(e)$ позната измерена вредност за свако $e \in S$. Реконструкција свих вредности $c(e)$, $e \in T$, спроводи се следећим поступком:

1. Свака грана $e \in S$ означава се као *познаша*, као и обе граничне гране, чија је вредност једнака граничној вредности N. Свака грана $e \in T$ означава се као *непознаша*.

2. Док постоји непозната грана, пронаћи лист l стабла T, чвор инцидентан тачно једној још непознатој грани стабла T. Такав чвор увек постоји док год стабло T има бар једну грану, пошто T нема циклуса. Нека је e_l та једина непозната грана инцидентна чвору l.

3. Применом једначине одржања протока на чвор l израчунати вредност $c(e_l)$, а затим грану e_l (и лист l) означити као познате и уклонити их из даљег разматрања.

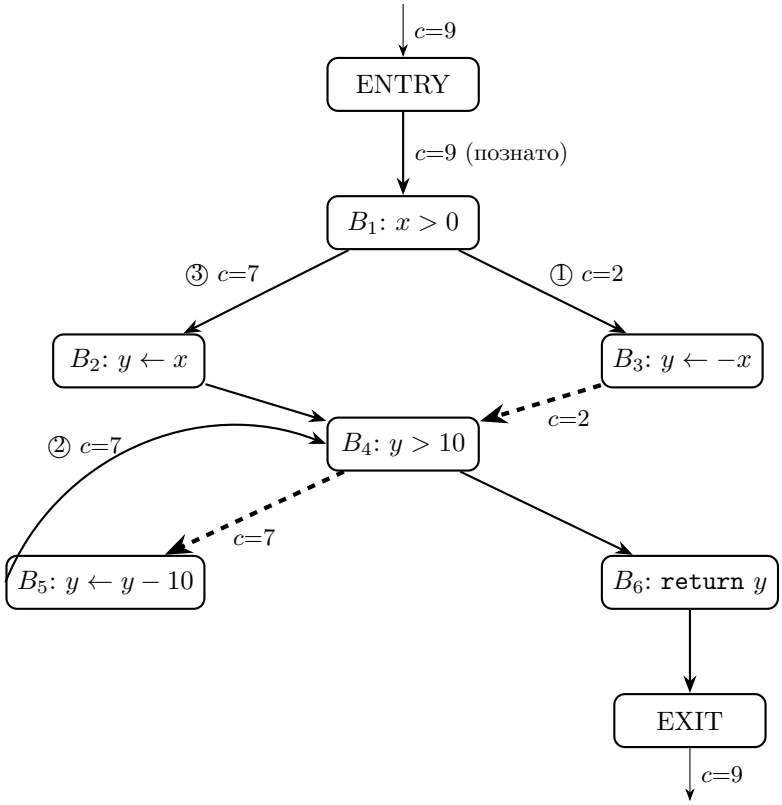
4. Поновити поступак од корака 2 на преосталом, за један чвор и једну грану мањем стаблу.

Поступак се завршава пошто стабло T садржи коначно много грана, њих $n-1$, а у сваком кораку се тачно једна грана одређује и уклања. По завршетку, позната је вредност $c(e)$ за сваку грану $e \in E$. Поредак којим се чворови обрађују одговара поретку обиласка стабла T од листова ка корену. Овај поступак је експлицитна, алгоритамска формулација конструктивног дела доказа минималности.

Слика 2.4 илуструје прва три корака овог поступка на примеру са слике 2.3, за претпостављене измерене вредности $c(B_3, B_4) = 2$ и $c(B_4, B_5) = 7$ на инструментованим гранама. Гранична вредност је $N = 9$, укупан број позива функције. Она се уноси као позната на обе граничне гране. Из једначине чвора $ENTRY$ одмах следи $c(ENTRY, B_1) = 9$, пошто је то једина излазна грана тог чвора.

Чвор B_3 има тачно једну инцидентну грану стабла T (ка чвору B_1) и једну познату инструментовану грану. Применом једначине одржања протока тако се добија $c(B_1, B_3) = 2$ (корак ①). Аналогно, чвор B_5 има једину непознату грану стабла T управо грану ка чвору B_4. Зато важи $c(B_5, B_4) = 7$ (корак ②). Прва два корака су једноставна. Свака непозната вредност у њима је преузета директно од једине познате суседне гране. Чвор B_1, међутим, у овом тренутку има *две* познате инцидентне гране. То су улазна грана из чвора $ENTRY$ ($c = 9$) и излазна грана ка чвору B_3, израчуната у кораку ① ($c = 2$). Једино још непозната излазна грана, ка чвору B_2, добија се одузимањем:

$c(B_1, B_2) = 9 - 2 = 7$ (корак ③). Поступак се на исти начин наставља на преосталом, мањем стаблу, све док се не одреде вредности за све гране.



Слика 2.4: Прва три корака реконструкције за пример са слике 2.3. Испрекидане гране су инструментоване, па су њихове вредности измерене. Заокружени бројеви означавају редослед којим се рачунају вредности преосталих грана стабла T.

2.5 Избор разапињућег стабла

Свако разапињуће стабло графа G_u даје скуп инструментованих грана исте, минималне величине. Ти скупови се ипак разликују по томе колико се често мерене гране извршавају, а тиме и по трошку мерења. У овом одељку уводи се мера тог трошка (одељак 2.5.1), а затим конструкција стабла које га своди на најмању меру (одељак 2.5.2).

2.5.1 Додела тежина гранама

Између свих скупова грана који задовољавају услов реконструкције, треба изабрати онај чија инструментација доноси најмањи практични трошак. За то је потребна мера која процењује учесталост извршавања сваке гране. Стварне учесталости извршавања грана нису познате пре него што се програм профајлира. Управо њих профајлирање и треба да измери. Зато се при избору грана за инструментацију користи процена изведена из облика графа, а не из података о извршавању [1]. Ту меру у наставку зовемо *тежина*.

Размотримо стабло претраге у дубину (енгл. *depth-first search*, *DFS*) графа G, започето из чвора *ENTRY*. Грана $(u, v) \in E$ за коју важи да је v предак чвора u у том стаблу назива се *повратна грана* (енгл. *back edge*) [8]. Повратна грана затвара циклус. Скуп повратних грана зависи од саме претраге. Полазак из другог чвора, или други редослед обиласка суседа, означиће као повратну другу грану истог циклуса. У наставку се подразумева једна фиксирана претрага, започета из чвора *ENTRY*. На слици 2.1 таква је грана (B_5, B_4), која одговара телу петље `while` из листинга 2.1.

Уклањањем свих повратних грана из графа G добија се граф $G' = (V, E \setminus E_{back})$ без циклуса, односно усмерени ациклични граф (енгл. *directed acyclic graph*, *DAG*) [8]. Сваки циклус у графу G садржи бар једну повратну грану, па уклањање свих повратних грана уклања и све циклусе. Граф G' не одговара непосредно ниједном скупу извршавања функције P. То је помоћна структура, уведена ради пребројавања путања. Пошто G' нема циклуса, број путања између било која два његова чвора је коначан.

За сваки чвор $v \in V$, број путања од чвора v до чвора *EXIT* у графу G', у ознаци $\pi(v)$, дефинише се рекурзивно:

$$\pi(v) = \begin{cases} 1, & v = EXIT, \\ \displaystyle\sum_{(v,w) \in E \setminus E_{back}} \pi(w), & v \neq EXIT. \end{cases}$$

Пошто је граф G' ацикличан, ова дефиниција је ваљана. Вредност $\pi(v)$ за сваки чвор може се израчунати у једном пролазу кроз чворове у обрнутом тополошком редоследу графа G'.

Тежина гране $e = (u, v) \in E$, у ознаци $w(e)$, дефинише се као $w(e) = \pi(v)$, број путања од њеног одредишног чвора до чвора *EXIT*. Ова величина је

структурна процена учесталости гране e, независна од конкретних улазних података функције P.

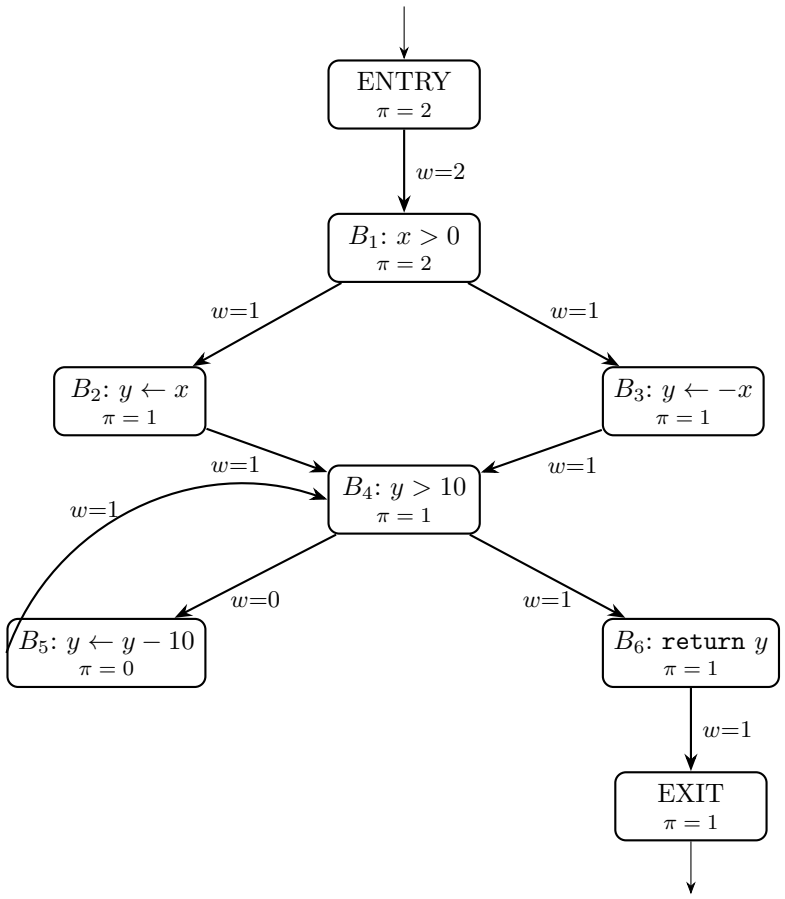
На примеру са слике 2.1 вредности $\pi(v)$ рачунају се обрнутим тополошким редоследом графа G'. Из графа је претходно уклоњена повратна грана (B_5, B_4):

- $\pi(EXIT) = 1$, по дефиницији;
- $\pi(B_6) = \pi(EXIT) = 1$, пошто чвор B_6 има једину излазну грану ка чвору $EXIT$;
- $\pi(B_5) = 0$, пошто је једина излазна грана чвора B_5 била повратна и у графу G' више не постоји, па из чвора B_5 нема ниједне путање до чвора $EXIT$;
- $\pi(B_4) = \pi(B_5) + \pi(B_6) = 0 + 1 = 1$;
- $\pi(B_2) = \pi(B_4) = 1$ и $\pi(B_3) = \pi(B_4) = 1$;
- $\pi(B_1) = \pi(B_2) + \pi(B_3) = 1 + 1 = 2$, што одговара двема путањама кроз наредбу `if`;
- $\pi(ENTRY) = \pi(B_1) = 2$.

Тежина сваке гране је вредност π њеног одредишног чвора, на пример

$$w(ENTRY, B_1) = \pi(B_1) = 2, \qquad w(B_1, B_2) = \pi(B_2) = 1.$$

Грана (B_4, B_5), која улази у тело петље, добија тежину $w(B_4, B_5) = \pi(B_5) = 0$. Повратна грана (B_5, B_4), међутим, добија тежину $w(B_5, B_4) = \pi(B_4) = 1$, пошто се тежина одређује вредношћу π одредишног, а не полазног чвора. Слика 2.5 приказује све вредности $\pi(v)$ и све тежине грана за овај граф.



Слика 2.5: Вредности $\pi(v)$ уз сваки чвор и тежине $w(e)$ уз сваку грану, за граф контроле тока са слике 2.1. Граничне гране немају тежину, пошто не улазе у конструкцију стабла.

2.5.2 Оптимално разапињуће стабло

Свакој неусмереној грани графа G_u додељена је њена тежина $w(e)$. Кнутов алгоритам конструише разапињуће стабло T графа G_u највеће укупне тежине. Такво стабло се назива *оптимално разапињуће стабло*. Може се конструисати, на пример, алгоритмом Џозефа Крускала (енгл. *Joseph Kruskal*), модификованим тако да се гране бирају у опадајућем, уместо у растућем, редоследу по тежини [10, 1].

Избор *оптималног* стабла, уместо произвољног разапињућег стабла, нема утицаја на величину $|S|$, пошто свако разапињуће стабло графа G_u има тачно $n-1$ грана. Он утиче искључиво на то *које* конкретне гране се инструментују. Гране унутар стабла T су, према дефиницији тежине, оне за које се очекује да леже на већем броју путања кроз граф. Њихов број пролаза се не мери

директно, већ се реконструише из једначина одржања протока. Преостале, мање учестале гране се инструментују, чиме се минимизира процењени трошак инструментације у односу на изабрану функцију тежине [1]. Стабло са слике 2.3 је једно оптимално разапињуће стабло примера из ове главе. Оптимално разапињуће стабло не мора бити јединствено. Гране (B_1, B_3) и (B_3, B_4) имају исту тежину 1, па замена прве другом даје стабло исте укупне тежине 8, уз скуп инструментованих грана $S = \{(B_1, B_3), (B_4, B_5)\}$. Које од оптималних стабала ће бити изабрано зависи од редоследа којим алгоритам разматра гране једнаке тежине.

Тежина $w(e) = \pi(v)$ представља чисто структурну процену учесталости, независну од семантике функције P. Уместо ње, при избору разапињућег стабла могла би се користити и прецизнија процена, добијена применом статичких хеуристика предикције гранања (на пример, на основу облика петље или поређења у условном изразу), какве су предложили Томас Бал и Џејмс Ларус (енгл. *Thomas Ball*, *James Larus*) [11]. Овакво унапређење, међутим, није предмет овог рада.

Глава 3

Технологије веб апликације и статичке анализе

Систем изграђен у овом раду је веб апликација која кораке Кнутовог алгоритма визуелизује над унапред припремљеним примерима и над изворним кодом који корисник сам унесе. Ово поглавље описује коришћене технологије и разлоге њиховог избора, не и начин њихове примене у апликацији. Свака изабрана технологија носи сопствене компромисе. Избор је готово увек зависио од природе конкретног задатка који је требало решити. Поглавље је подељено према слоју система коме технологије припадају. Најпре су описане клијентске технологије (одељак 3.1), затим серверске (одељак 3.2), потом алат за статичку анализу кода *Joern* (одељак 3.3) и на крају контејнеризација и инфраструктура (одељак 3.4).

3.1 Клијентске технологије

Клијентски слој апликације суочава се са два различита захтева. Први је управљање стањем и логиком апликације. Други је интерактиван приказ графа контроле тока.

Клијентски део апликације развијен је у веб-оквиру *Angular* [12]. То је савремен оквир заснован на компонентама и, од новијих верзија, на реактивним сигналима као основној јединици стања. Компонентни модел издваја део приказа, заједно са његовим стањем и понашањем, у засебну целину, компоненту. Иста компонента се затим употребљава на више места у апликацији, без удвостручавања кода. Систем сигнала и изведених вредности омогућава да се

више извора стања обједини декларативно, без ручног праћења када који део приказа треба поново израчунати. Поред тога, алат командне линије радног оквира *Angular* и конвенција слојевите организације пројекта по функционалним целинама погодују пројекту ове величине, са више независних целина које деле заједничку основу.

Читав клијентски код је написан у програмском језику *TypeScript* [13]. Тај језик над *JavaScript*-ом додаје статичку проверу типова. Граф, његови чворови и гране су захваљујући томе прецизно дефинисани типови. Грешку у облику података у коду саме апликације преводилац тако пријави већ при превођењу.

Граф контроле тока се приказује помоћу библиотеке за приказ графова *Cytoscape.js* [14, 15]. Она је намењена специфично приказу и интеракцији са мрежама и графовима, за разлику од општих библиотека за цртање графикона. *Cytoscape.js* подржава стилизацију чворова и грана по класама, налик класама у језику за описивање изгледа веб страница *CSS*. Изглед истог графа се зато мења придруживањем и уклањањем класа, без поновног исцртавања. Подржава и увеличавање, умањивање и померање приказа, као и појединачно премештање чворова, без додатног програмирања ових основних понашања. Приказ се гради из самих података о чворовима и гранама, а не из унапред задатог облика, па библиотека не претпоставља ни величину ни структуру графа.

Сам распоред чворова, међутим, *Cytoscape.js* препушта посебним алгоритмима за распоређивање. Апликација нуди два: *Dagre* [16] и *ELK* [17]. *Dagre* је једноставан и брз алгоритам за распоред усмерених графова у слојеве, прикладан за већину примера и увезених графова умерене величине. *ELK* (*Eclipse Layout Kernel*) нуди богатији скуп алгоритама и опција распоређивања, на рачун веће рачунске сложености. Показује се кориснијим за веће, гушће увезене графове. Понуда оба алгоритма, уз могућност да корисник сам изабере онај који даје прегледнији приказ за конкретан граф, представља практичан одговор на ограничење које ниједан појединачан алгоритам распоређивања у потпуности не решава. Оба поступка граф распоређују у слојеве, тако да гране углавном воде у истом смеру, што граф контроле тока чини прегледним.

3.2 Серверске технологије

Серверски слој се ослања на две врсте технологија. Прва је веб-оквир, а друга ред послова, који дуготрајан посао одваја од захтева који га покреће.

Серверски део система је развијен у програмском језику *Python* [18]. Разлог је делом и то што је *Python* уобичајен избор за задатке попут интеграције спољашњег алата кроз омотач који покреће посебан процес и обрађује његов излаз. Покретање процеса и обраду његовог текстуалног излаза покрива већ стандардна библиотека, без додатних зависности. За сам веб *API* користи се веб-оквир *FastAPI* [19]. *FastAPI* обрађује захтеве асинхроно. То је важно зато што *API* при обради захтева чека на ред послова, при уношењу посла или читању његовог стања, а не треба да тим чекањем блокира обраду осталих захтева. Сама, дуготрајна анализа алатом *Joern* се не извршава у оквиру *API* процеса, већ у одвојеном радном процесу (енгл. *worker*). *FastAPI*, ослањајући се на библиотеку *Pydantic* [20], такође аутоматски проверава облик сваког захтева и одговора. Грешке у размени података тако се откривају што раније, слично улози *TypeScript*-а на клијентској страни. Из истих описа оквир гради и документацију *API*-ја, која зато остаје у складу са кодом.

Стање и резултат посла обраде графа контроле тока чувају се у меморијској бази података *Redis* [21]. То је база података која податке држи у оперативној меморији, у паровима кључ-вредност. Поред читања и уписа по кључу, *Redis* нуди и листе са блокирајућим читањем. Процес који чека на податак тако не мора да понавља упите, него стоји док му *Redis* не преда ставку. Пошто су ови подаци по природи привремени, погодује им то што *Redis* подржава аутоматско истицање записа након одређеног времена (енгл. *time to leave*, скраћено *TTL*). Стари записи тако нестају сами, без посебног механизма чишћења.

Само извршавање посла, одвојено од *HTTP* захтева који га покреће, организовано је преко библиотеке за ред послова *RQ* [22] (*Redis Queue*). *RQ* је одабран јер је директно заснован на *Redis*-у, већ присутном у систему. Не захтева увођење посебног, компликованијег система за размену порука. Библиотека уз то води рокове извршавања и поновне покушаје после неуспеха, што би се иначе имплементирало ручно.

3.3 Алат за статичку анализу кода *Joern*

Подршка за више програмских језика омогућава да се рад примени на шири скуп програма. Граф контроле тока се извлачи из кода написаног у четири језика: *C/C++*, *Java*, *Python* и *JavaScript*. За тај задатак изабран је алат отвореног кода *Joern* [7, 23]. Један алат, са јединственим интерфејсом и излазним форматом, подржава сва четири језика. Статичка анализа граф изводи из самог кода, без покретања програма, па не тражи ни улазне податке ни окружење за извршавање.

Алтернатива би била посебан алат за сваки језик. Тада би сваки језик захтевао сопствену анализу изворног кода и сопствени излазни формат, па би се резултати морали накнадно уједначавати. Алат *Joern* тај посао обједињује, на исти начин за сва четири језика.

Своју представу изворног кода *Joern* назива графом својстава кода (енгл. *code property graph*) [7]. То је једна структура која обједињује више погледа на исти програм: апстрактно синтаксичко стабло, граф контроле тока и зависности међу подацима. Чворови те структуре су елементи програма, а гране односи међу њима, при чему свака грана носи ознаку врсте односа. Овом раду је од целе структуре потребан само граф контроле тока једне функције, па се остатак не користи.

Joern захтева *Java* виртуелну машину и сопствени, обимни скуп алата за анализу по језику. То га чини суштински другачијим од преосталих *Python* сервиса у систему, који заузимају мало меморије. Из тог разлога је интегрисан као засебан, самосталан сервис, покренут у сопственом контејнеру, уместо да буде директно укључен у главни *API* или радни процес. Овакво раздвајање омогућава да се окружење алата *Joern* припрема и ажурира независно од остатка кода. Уз то, терет његовог покретања не оптерећује покретање преосталих, лаких сервиса.

3.4 Контејнеризација и инфраструктура

Систем се састоји од више сервиса који раде истовремено и међусобно комуницирају. Платформа за контејнере *Docker* [24] обезбеђује да се сваки од њих покреће у изолованом, унапред дефинисаном окружењу. Алат *Docker Compose* [25] организује покретање и повезивање више сервиса одједном.

Сваки сервис се покреће у сопственом *Docker* контејнеру. Разлози за такву поделу тичу се архитектуре система. Окружење за алат *Joern*, са *Java* виртуелном машином и обимним скупом алата по језику, описује се и одржава одвојено од окружења малих *Python* сервиса, па се њихове зависности не мешају. Сервис са алатом *Joern* са остатком система комуницира кроз узак *HTTP* интерфејс. Зато се по потреби може покренути у више примерака, или заменити другим алатом за издвајање графа контроле тока, без измена у осталим сервисима. Контејнеризација притом обезбеђује и да се систем на било ком рачунару покреће у истом, унапред дефинисаном окружењу, независно од тога шта је већ инсталирано на самом рачунару. Контејнери при томе деле језгро оперативног система домаћина, па су лакши од виртуелних машина и брже се покрећу.

Ручно покретање сваког контејнера понаособ, уз ручно повезивање сервиса и постављање променљивих окружења, било би подложно грешкама и тешко поновљиво. *Docker Compose* тај посао своди на један декларативан опис. Декларативан значи да се у опису наводи жељено стање система, а не низ команди којима се оно постиже. Опис стоји у датотеци `docker-compose.yml`, уз изворни код, па се мења и прати заједно са њим.

Глава 4

Алат *Knuth Profiler*

Доказ Кнутовог алгоритма показује да алгоритам, без обзира на конкретан граф контроле тока, конструише оптимални скуп инструментованих грана. Доказ уз то даје и поступак којим се преостали бројачи реконструишу из тог скупа. Овакав резултат, иако математички потпун, тешко је сагледати искључиво на основу текста доказа. Да би се кораци алгоритма боље разумели, у оквиру овог рада развијена је веб апликација која те кораке примењује на стварним графовима контроле тока. Апликација их приказује интерактивно, корак по корак.

Ово поглавље описује имплементацију апликације *Knuth Profiler*, кроз коју су визуелизовани кораци Кнутовог алгоритма. Најпре је дат општи преглед система и заједничког модела података (одељак 4.1), а затим је описан поступак извлачења графа контроле тока из произвољног изворног кода помоћу алата *Joern* (одељак 4.2). Потом је приказана примена самог Кнутовог алгоритма у апликацији: додела тежина, конструкција разапињућег стабла, избор инструментованих грана, симулација извршавања и реконструкција бројача (одељак 4.3). Поглавље се завршава корисничким током кроз кораке визуелизације (одељак 4.4). Изворни код целог система доступан је у јавном репозиторијуму [26].

4.1 Преглед система

Графови контроле тока над којима апликација ради потичу из два извора. Први су унапред припремљени примери, уграђени у апликацију. Други су графови добијени анализом изворног кода који корисник сам уноси или

отпрема, у језицима *C/C++*, *Java*, *Python* и *JavaScript*.

Овај одељак описује структуру система. Најпре су приказане компоненте и њихова међусобна комуникација (одељак 4.1.1), затим начин на који се те компоненте покрећу (одељак 4.1.2), а на крају заједнички модел података над којим све раде (одељак 4.1.3). Дијаграми су дати у нотацији језика *UML* [27].

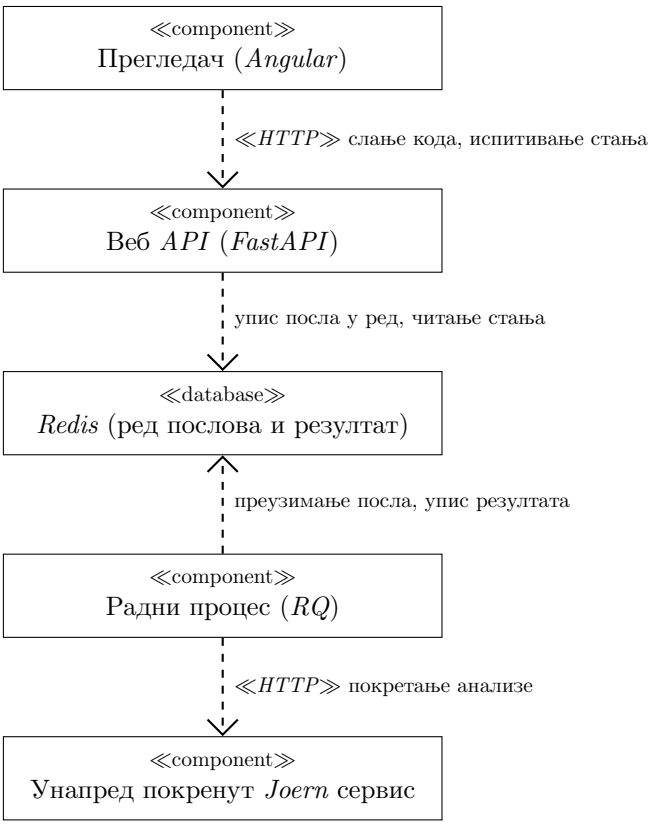
4.1.1 Архитектура система

Архитектура система подељена је у два дела, по један за сваки извор графа. Уграђени примери су граф већ сами по себи, док се из увезеног корисничког кода граф тек мора извући. Сви кораци самог алгоритма, од доделе тежина до реконструкције, представљају чисто клијентску логику. Примењују се над већ датим графом, не захтевају приступ изворном коду нити спољашње алате и извршавају се у потпуности у прегледачу, у оквиру *Angular* апликације. Извлачење графа контроле тока из произвољног изворног кода, међутим, захтева статичку анализу самог кода. За тај задатак коришћен је алат *Joern* [7]. То је знатно тежи и спорији поступак, неприкладан за извршавање у прегледачу. За ову намену развијен је посебан сервер, заснован на оквиру *FastAPI*, уз *Redis* и *RQ* за асинхрону обраду послова.

Слика 4.1 приказује ове компоненте и њихову међусобну комуникацију. Прегледач шаље изворни код (унетим текстом или отпремањем фајла) ка *API*-ју, који враћа идентификатор посла (`jobId`) и одмах ослобађа захтев, без чекања на завршетак анализе. *API* посао уписује у ред у *Redis*-у.

Радни процес библиотеке *RQ* преузима посао из реда блокирајућим читањем и извршава га. Радни процес редом пролази кроз кораке припреме изворног кода, покретања анализе, изградње и нормализације графа. У току анализе позива и посебан, унапред покренут *Joern* сервис. То је решење усвојено да би се избегло споро, поновљено покретање *Joern*-а за сваки захтев. Резултат (или, у случају грешке, дијагностичку поруку) радни процес по завршетку уписује назад у *Redis*.

Прегледач у међувремену периодично испитује (енгл. *polling*) стање посла преко *API*-ја. *API* то стање очитава из *Redis*-а, све до завршетка обраде, када прегледач преузима коначан резултат. Захтев за резултат прегледач шаље у сваком пролазу, а одговор остаје празан док обрада траје.



Слика 4.1: Дијаграм компоненти система. Испрекидане стрелице означавају зависност, у смеру од компоненте која користи ка оној која се користи.

4.1.2 Покретање и међузависност сервиса

Сваки сервис се покреће у сопственом контејнеру. Веб *API* и радни процес при томе деле исту слику контејнера и разликују се само по покренутој команди. Оба тако извршавају идентичан код, па се заједничка логика не одржава на два места. Сервиси се међусобно обраћају по имену, преко заједничке мреже, па ниједна мрежна адреса није унапред уписана у код.

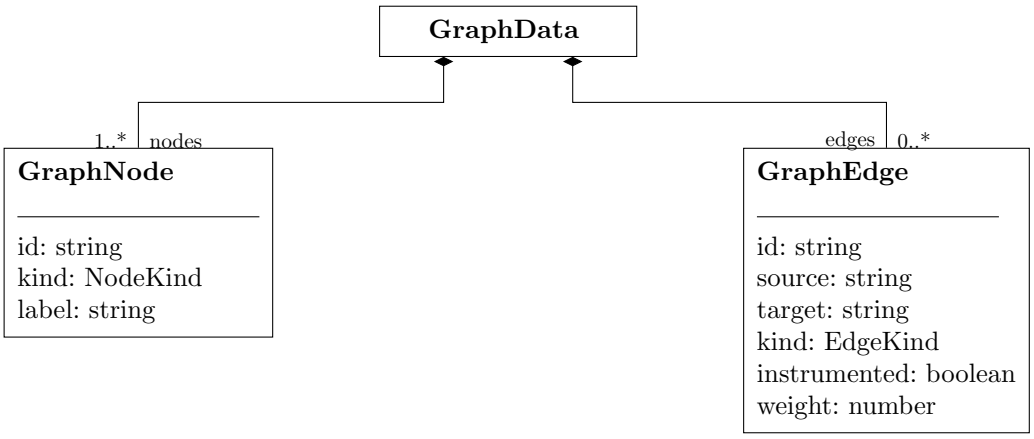
Редослед покретања је битан. Веб *API* и радни процес не смеју да почну са радом пре него што *Redis* и унапред покренут *Joern* сервис буду спремни да приме захтев. Стварање контејнера за то није довољан услов, пошто процес у њему тек треба да се подигне. Зато та два сервиса имају дефинисану проверу доступности (енгл. *health check*), а преостала два се покрећу тек пошто те провере прођу. Без тога би први захтев могао да затекне сервис који још не прима везе.

Продукционо окружење се од развојног разликује у два аспекта. Први

је опоравак. Сваки сервис се поново покреће сам, осим ако је изричито заустављен. Други је приступ споља. Једино веб *API* објављује свој порт на рачунару домаћину, и то везан за локалну адресу тог рачунара. Порт зато није доступан са јавне мреже, него само веб серверу на самом домаћину. Тај сервер кориснику испоручује статички део апликације, а захтеве ка *API*-ју прослеђује на тај порт. Преостала три сервиса не објављују ниједан порт и доступна су само унутар заједничке мреже сервиса.

4.1.3 Заједнички модел података

Без обзира на порекло графа, сви кораци у апликацији раде над једним истим моделом података. Тај модел је структура `GraphData`. Она садржи листу чворова и листу грана. Слика 4.2 приказује та три типа као дијаграм класа.



Слика 4.2: Дијаграм класа заједничког модела података. Структура `GraphData` садржи листу чворова и листу грана, а са њима је повезана двема засебним везама. Испуњени ромб на њеном крају сваке везе означава композицију, пошто чворови и гране постоје само као део те структуре. Уз супротни крај наведени су име својства и кардиналност.

Чвор (`GraphNode`) носи идентификатор, врсту и натпис који се приказује на графу. Грана (`GraphEdge`) носи идентификатор, изворни и одредишни чвор, врсту, ознаку да ли је инструментована и тежину. Врста чвора (`NodeKind`) разликује улаз, излаз, гранање, обичну наредбу и помоћни чвор. Врста гране (`EdgeKind`) разликује обичну, повратну и граничну грану.

Граф добијен анализом изворног кода се своди на овај исти модел. Чим се то догоди, обе путање, готов пример и увезени код, даље се обрађују иден-

тично. Исти код примењује Кнутов алгоритам и изводи симулацију и реконструкцију.

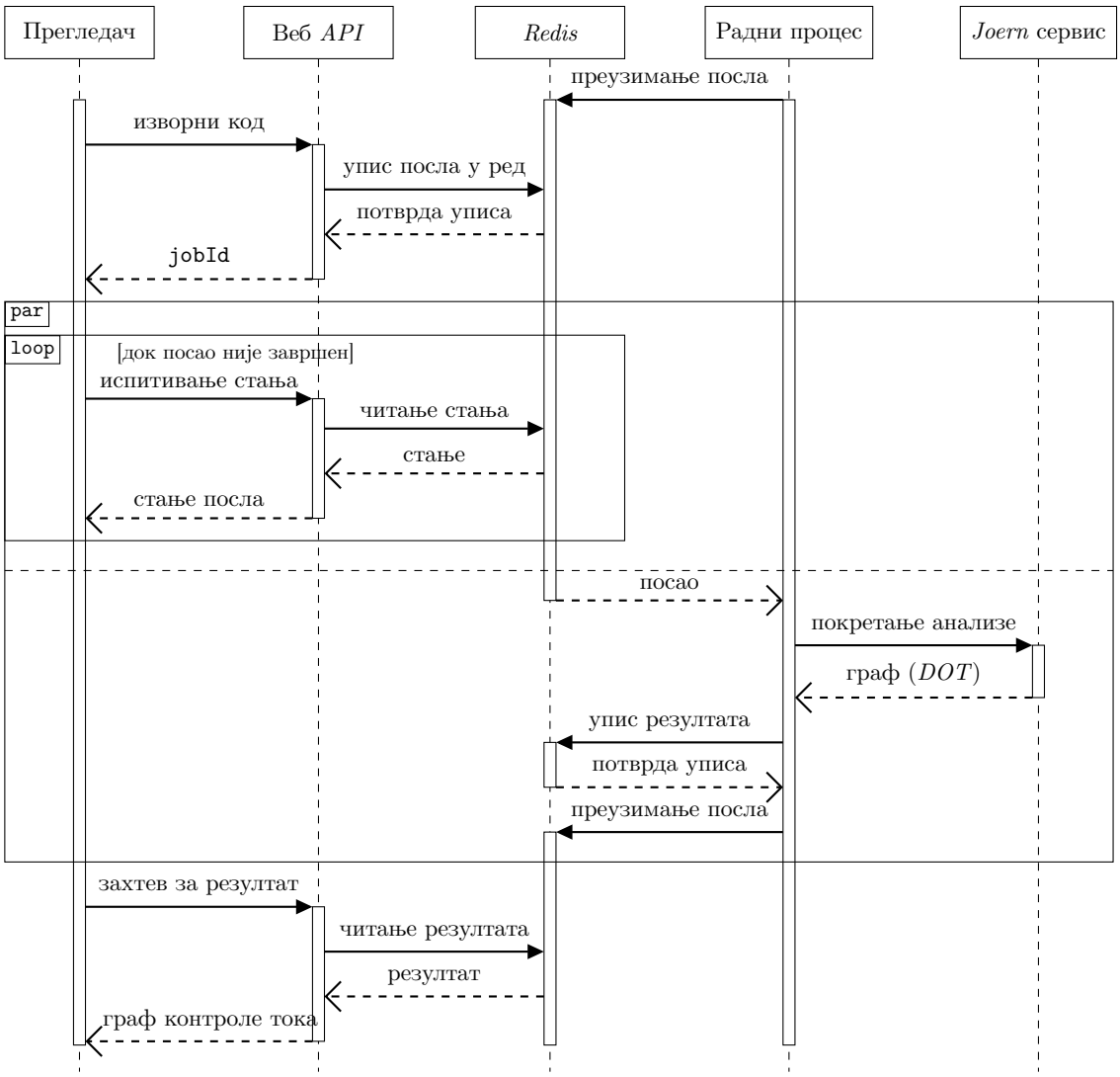
4.2 Извлачење графа контроле тока алатом *Joern*

За унапред припремљене примере граф контроле тока је уграђен у апликацију и не захтева никакву обраду. За изворни код који корисник сам унесе, међутим, граф контроле тока није унапред познат. Њега треба извући статичком анализом тог кода. Тај задатак, на серверској страни, обавља алат *Joern*. Кориснички код се при томе никада не извршава, ни на серверу ни у прегледачу. Слика 4.3 приказује тај ток у времену, од корисниковог захтева до графа враћеног прегледачу. У наставку одељка најпре је описан начин на који се *Joern* позива и у којем формату се добија резултат његове анализе (одељак 4.2.1), а затим како се тај резултат обрађује и своди на заједнички модел података коришћен у апликацији (одељак 4.2.2).

4.2.1 Формат *DOT* и унапред покренут *Joern* сервис

Joern [7] произвољан изворни код своди на граф својстава кода, из кога се затим издваја граф контроле тока анализиране функције. Радни процес покреће *Joern* над предатим изворним кодом посредством наменске скрипте на језику *Scala*. Скрипта прво учитава код и бира функцију над којом се рачуна граф контроле тока: функцију `main`, ако постоји, иначе прву пронађену функцију. Над том функцијом затим позива уграђену *Joern* функцију за извоз графа контроле тока у формат *DOT* [28], текстуални формат за описивање графова који подржава широк скуп алата, попут алата *Graphviz* [29]. Овакав излаз се, окружен договореним почетним и завршним маркером, исписује на стандардни излаз, одакле га радни процес издваја.

Одредница „унапред покренут“ односи се на то да је сам алат *Joern*, заједно са окружењем потребним за његово извршавање (*Java* виртуелна машина, распаковано издање алата *Joern* за командну линију), унапред инсталиран и спреман у посебном, непрекидно покренутом контејнеру. То је другачије од приступа у којем би се ово окружење изнова припремало за сваки појединачни захтев, што би било знатно спорије. И поред тога, свако извршавање анализе и



Слика 4.3: Дијаграм секвенце за увоз изворног кода. Пуне стрелице су позиви, а испрекидане одговори. Уски правоугаоници на животним линијама означавају оквир активности. Радни процес чека посао блокирајућим читањем реда. На дијаграму зато почиње да чека пре него што посао постоји. На чекање се враћа после сваке обраде. Оквир `par` означава да испитивање стања и обрада посла теку упоредо.

даље покреће нов *Joern* потпроцес. Овакав сервис тако не елиминише трошак самог покретања алата *Joern* по захтеву, већ искључиво трошак припреме окружења у којем се то покретање дешава. Пошто *Joern*, осим резултата, на стандардни излаз исписује и сопствене дијагностичке поруке, издвајање *DOT* садржаја мора бити отпорно на њихово присуство. Ако договорени маркери нису пронађени, резервни поступак претраживањем регуларним изразом по-

кушава да пронађе *DOT* блок директно у излазу. Овај резервни поступак препознаје карактеристичан облик описа графа у формату алата *Graphviz*: кључну реч `digraph`, праћену именом графа и телом графа затвореним у витичасте заграде.

4.2.2 Парсирање и нормализација графа

Добијени *DOT* текст се на серверу даље обрађује регуларним изразима, без потпуног синтаксичког анализатора *DOT* формата. Овакав приступ је довољан, пошто *Joern* граф контроле тока увек извози у препознатљивом, уском подскупу *DOT* синтаксе. Засебно се издвајају редови који описују чворове и редови који описују гране, према присуству ознаке `->`. Из обележја сваког чвора и гране затим се издваја натпис. *Joern* натпис некада записује као обичан, наводницима ограничен текст, а некада као запис сличан језику *HTML*, ограничен угластим заградама уместо наводника, са *HTML* ентитетима и ознакама прелома реда (`<BR/>`). Оба облика се на крају своде на чист текст.

На основу садржаја натписа, сваком чвору се хеуристички додељује врста: улаз, излаз, гранање, позив функције, наредба повратка или обична наредба. На пример, натпис који почиње ознаком `METHOD,` препознаје се као улазни чвор, ознака `METHOD_RETURN` као излазни, а присуство кључних речи `if` и `while` или знака `?` као гранање. Улазни и излазни чвор графа бирају се на основу ове класификације, са резервним правилом (први, односно последњи чвор у низу) ако класификација не одреди врсту ни једном чвору.

Осим за одређивање улазног и излазног чвора, ова класификација служи искључиво визуелном приказу. Облике гранања које хеуристика не препознаје, попут наредбе `switch`, приказ третира као обичне наредбе. Њихово гранање у графу ипак остаје, пошто такав чвор задржава све своје излазне гране. Сами кораци Кнутовог алгоритма не зависе од врсте чвора, већ искључиво од структуре графа и тежина грана.

Овако добијен граф, заједно са идентификатором улазног и излазног чвора, чини договорени резултат посла. Радни процес га уписује у *Redis*, а *API* враћа прегледачу.

У прегледачу се над овим резултатом извршава још један, завршни корак нормализације. Њиме се произвољан увезени граф своди на исти, канонски облик у коме су унапред припремљени примери. Чвор који *Joern* означава

као `METHOD` (односно `METHOD_RETURN`, ако постоји) пресликава се на фиксиран идентификатор чвора *ENTRY* (односно *EXIT*). Уобичајено опширан натпис чвора у *Joern*-у (на пример, `addition, 14 a + b`) скраћује се на део након прве запете (`14 a + b`).

Чвору *ENTRY* додаје се улазна гранична грана (енгл. *sentinel*) из помоћног, скривеног чвора, а чвору *EXIT* излазна гранична грана ка другом таквом чвору. Тиме се у прегледачу гради проширени граф G^+. Вредност граничне гране је укупан број позива анализиране функције, дакле унапред позната гранична вредност. Закон одржања протока важи и у чворовима *ENTRY* и *EXIT*, па се при реконструкцији они не разликују од осталих чворова. Увезена функција зато сме имати и више од једне наредбе повратка. Свака наредба повратка даје по једну улазну грану чвора *EXIT*, док је у примеру из теоријског дела рада та грана једна. Једначина одржања протока за чвор *EXIT* тада изједначава збир вредности свих његових улазних грана са граничном вредношћу, па се поступак реконструкције не мења.

4.3 Примена алгоритма у апликацији

Претходни одељак описује на који начин апликација долази до графа контроле тока. Овај одељак описује на који начин се над тако добијеним графом примењују кораци Кнутовог алгоритма. Потпоглавља редом обрађују доделу тежина (одељак 4.3.1), конструкцију разапињућег стабла (одељак 4.3.2), избор инструментованих грана (одељак 4.3.3), симулацију извршавања (одељак 4.3.4) и реконструкцију бројача (одељак 4.3.5).

Сваки корак изводи по једна функција, издвојена од приказа и од стања корисничког окружења. Ниједна од тих функција није део неке компоненте, него стоји у засебном модулу који компоненте само позивају. Исте те функције позива и оквир за проверу исправности, без покретања приказа.

4.3.1 Додела тежина

Тежине рачуна функција `assignKnuthWeights` из датотеке `knuth-weights.ts`. Она прима листе чворова и грана, а резултат уписује у поље `weight` сваке гране. Непосредно примењује дефиницију $w(e) = \pi(v)$ (одељак

2.5.1), по којој је тежина гране једнака броју путања од њеног одредишног чвора до чвора *EXIT*.

Функција најпре гради мапу која сваком идентификатору чвора придружује листу његових излазних грана. Над том мапом открива повратне гране претрагом у дубину, започетом из чвора *ENTRY*. Полазни чвор је при томе битан, пошто је повратна грана дефинисана преко стабла те претраге. Полазак из неког другог чвора истог циклуса означио би као повратну другу грану тог циклуса. На основу откривених повратних грана одређује се *DAG* поглед графа и тополошки поредак његових чворова. У обрнутом тополошком поретку, за сваки чвор се израчунава вредност $\pi(v)$. Свакој грани се на крају додељује тежина једнака вредности π њеног одредишног чвора.

Функција граничне гране препознаје по њиховом идентификатору, а не по врсти. Врста гране у апликацији одређује и њен изглед на графу, па исте вредности добијају и неке стварне гране. У унапред припремљеним примерима то важи за гране уз чворове *ENTRY* и *EXIT*, које се цртају бојом граничних грана. Идентификатор те двосмислености нема, пошто га граничне гране добијају при самом стварању.

4.3.2 Конструкција разапињућег стабла

Стабло рачуна функција `computeMaxWeightSpanningTree` из датотеке `graph-analysis.ts`. Она прима граф у заједничком моделу података, а враћа листу идентификатора грана које чине стабло. Примењен је Крускалов алгоритам [10], над свим гранама графа осим граничних. Граничне гране се из разматрања унапред искључују, у складу са граничном конвенцијом. Крускалов алгоритам је одабран зато што ради непосредно над листом грана, каквом је граф у заједничком моделу и представљен. Уз две помоћне структуре, сортирану листу грана и структуру за дисјунктне скупове, он обезбеђује детерминизам који визуелизација захтева. Гране се разматрају по једном фиксираном потпуном уређењу, по тежини, а затим по идентификатору, чиме су стабло, скуп инструментованих грана и ток реконструкције једнозначно одређени улазним графом. Временска сложеност је $O(e \log e)$ и одређена је сортирањем грана, што је асимптотски упоредиво са осталим алгоритмима за разапињуће стабло, попут алгоритма Роберта Прима (енгл. *Robert Prim*) [30, 31]. Примов алгоритам би, међутим, за исту функционалност

и исти степен детерминизма захтевао листе суседства, ред са приоритетом, фиксиран почетни чвор и конвенцију о разрешавању једнаких тежина.

Гране се прво сортирају опадајуће по тежини. Гране једнаке тежине се додатно уређују растуће по идентификатору. Гране се затим редом разматрају, а свака се прихвата само ако њена два краја још нису у истој компоненти повезаности. Припадност компонентама прати класа `DisjointSet`. Њена операција `union` истовремено проверава услов и спаја компоненте, а враћа податак о томе да ли је до спајања дошло. Операција `find` при сваком налажењу скраћује путању до корена, па поновљени упити над истим чвором постају јефтинији. Поступак се завршава чим стабло достигне $n - 1$ грана.

4.3.3 Избор инструментованих грана

Скуп S рачуна функција `computeInstrumentedEdgeIds` из исте датотеке. Она прима граф и листу идентификатора грана стабла, а враћа идентификаторе грана које се инструментују. Скуп се добија као комплемент стабла у скупу свих стварних грана, дакле свих грана осим граничних. Листа грана стабла се при томе преводи у скуп, ради брже провере припадности.

Граничне гране се затим третирају усклађено са граничном конвенцијом. Гранична грана на улазу се увек накнадно укључује у скуп S, чиме се представља чињеница да је њена вредност унапред позната, а не стварно инструментована. Гранична грана на излазу се увек искључује, пошто њена вредност увек одговара вредности улазне граничне гране, на основу закона одржања протока.

4.3.4 Симулација извршавања

Симулација опонаша извршавање програма насумичним обиласком графа од чвора *ENTRY* до чвора *EXIT*. На сваком чвору v са више излазних грана следећу грану бира функција `pickRandomEdge`. Грана $e \in \mathrm{Out}(v)$ бира се са вероватноћом

$$P(e) = \frac{w(e) + 1}{\sum_{e' \in \mathrm{Out}(v)} \big(w(e') + 1\big)}.$$

Вероватноће свих излазних грана чвора v тако дају збир један.

Овакав избор користи исту тежину $w(e) = \pi(v)$ која служи и за конструкцију разапињућег стабла, као практичну замену за стварну учесталост извр-

шавања. Увећање за један чува поредак значајности међу гранама, а ниједној грани не одузима могућност да буде изабрана. Без њега грана тежине нула, каква улази у тело петље, никада не би била пређена, па се тело петље у симулацији не би ни извршило.

Приликом сваког пролаза кроз инструментовану грану увећава се одговарајући бројач $c(e)$. Бројачи грана ван скупа S остају непознати све до поступка реконструкције.

Два режима рада одговарају двема улазним тачкама датотеке `simulation.engine.ts`. Режим без анимације изводи функција `runFastSimulation`, која све тражене пролазе одради одједном. Анимирани режим се води споља, позивом функције `tickAnimatedTraversal` по једном кораку, над стањем типа `AnimatedTraversalState`. Оба режима користе исто правило избора гране, па се разликују само по томе када се пролази изводе. Режим без анимације при томе прима извор насумичности као аргумент, па се исти низ пролаза може поновити истим семеном (енгл. *seed*).

4.3.5 Реконструкција бројача

Поступак реконструкције изведен је у датотеци `reconstruction.helpers.ts`. Скуп познатих вредности се пред сваки корак изнова гради функцијом `buildKnownCounts`. У првом кораку то су само бројачи измерени у симулацији, на гранама скупа S. Сваки наредни корак затиче у том скупу и вредности које је реконструкција дотле сама израчунала. Чвор инцидентан тачно једној још непознатој грани разапињућег стабла тражи функција `findSolvableNodeAndEdge`. Вредност те гране рачуна функција `solveBalanceAtNode`, применом једначине одржања протока за пронађени чвор.

Поступак се извршава корак по корак, на захтев корисника. Сваки корак решава по једну грану разапињућег стабла и пролази кроз све три наведене функције. Стање чува класа `ReconstructionStateService`, која држи већ реконструисане бројаче и уређен низ корака са објашњењима. Функција `solveBalanceAtNode` уз саму вредност враћа и списак познатих сабирака са предзнацима. Из њега се саставља објашњење примењене једначине, приказано уз сваки корак.

4.4 Корисничко искуство и едукативни ток

Претходна два одељка описују унутрашњу логику апликације: извлачење графа и примену алгоритма. Овај одељак описује на који начин је та логика изложена кориснику кроз корисничко окружење апликације. Најпре су описани кораци визуелизације (одељак 4.4.1), а затим два различита начина рада: над готовим примерима и над увезеним кодом (одељак 4.4.2).

4.4.1 Кораци визуелизације

Апликација алгоритам кориснику представља као вођени низ од седам корака: почетни приказ графа, додела тежина, конструкција разапињућег стабла, избор инструментованих грана, симулација извршавања (мерења), реконструкција бројача и, на крају, извештај. Пет средњих корака су кораци самог алгоритма. Прва и последња ставка нису део алгоритма. Почетни приказ служи упознавању са графом, а извештај сажима добијени резултат. Сваки корак има наслов, кратак опис и одговарајући визуелни приказ на самом графу, дефинисане на јединствен начин за читаву апликацију, тако да исти текст и иста логика важе и за готове примере и за увезени код.

Изглед гране зависи од корака у коме се граф приказује. У кораку конструкције разапињућег стабла, гране стабла T истичу се подебљаном тамном линијом, а преостале гране остају танке и светлије. Од корака инструментације надаље, гране скупа S се цртају испрекиданом линијом, као и у илустрацијама уз теоријски приказ алгоритма. У кораку доделе тежина, уз сваку грану се приказује вредност њене тежине. У кораку симулације, тренутни чвор и тренутна грана се истичу посебном бојом, а бројачи инструментованих грана се ажурирају уживо. У кораку реконструкције, бројачи се појављују и за гране стабла T, оним редом којим се рачунају.

Чворови у унапред припремљеним примерима носе кратке ознаке. Ознака `S` стоји уз наредбу пре гранања, а `D` уз чвор гранања. Ознаке `T` и `F` стоје уз чворове на грани тачног, односно нетачног исхода гранања, а `M` уз чвор у коме се те две гране поново спајају. У примеру са петљом, `I` означава чвор иницијализације, а `B` чвор тела петље. Изнад чвора $ENTRY$ и испод чвора $EXIT$ види се по једна стрелица без другог краја. То су граничне гране, чији се помоћни чворови у приказу не цртају.

Површ за приказ графа (енгл. *canvas*) је интерактивна. Дозвољено је зуми-

рање точкићем миша и померање превлачењем, а сваки чвор се појединачно може превући на жељену позицију. Поред тога, кориснику су на располагању и дугмићи за поново распоређивање чворова и прилагођавање приказа целом графу, као и избор између два различита алгоритма аутоматског распоређивања (*Dagre* и *ELK*, одељак 3.1). Ова интерактивност је нарочито корисна за веће, увезене графове, код којих је аутоматски распоред понекад тешко прегледан без додатног подешавања.

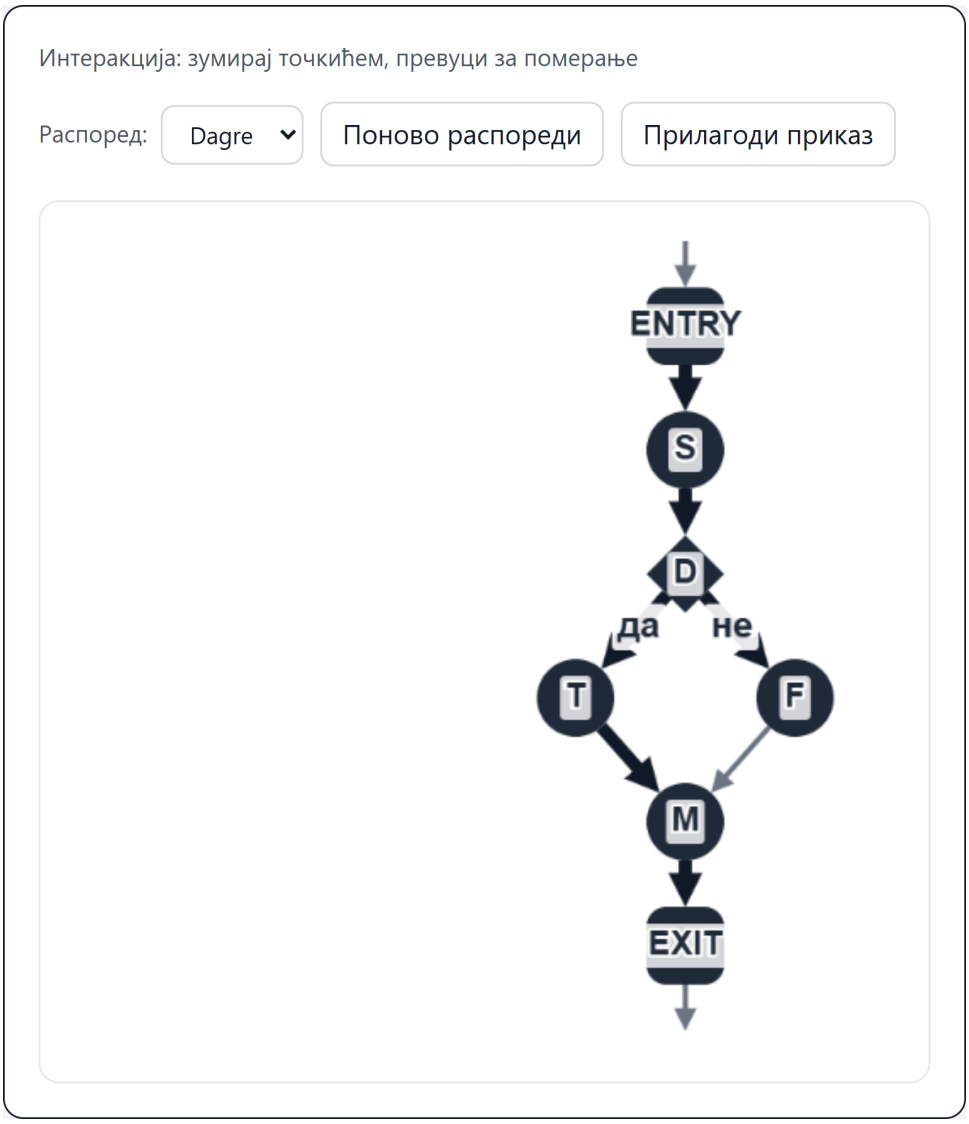
Апликација је, захваљујући прилагодљивом приказу корисничког интерфејса (енгл. *responsive design*), употребљива и на мањим екранима. Слике 4.4 и 4.5 приказују исти корак исте визуелизације на екрану рачунара и на екрану мобилног телефона. Распоред елемената се прилагођава ширини екрана, док сама површ за приказ графа остаје подједнако функционална.

Прелазак на следећи корак није увек слободан. Пре прелаза на реконструкцију, на пример, морају се прво завршити сви захтевани пролази симулације, иначе бројачи инструментованих грана не би били потпуни. Апликација за тренутни корак одређује да ли се сме напред и, ако не сме, која порука се приказује кориснику. Стање графа, симулације и реконструкције се обједињује у јединствен преглед за приказ на површи, при чему апликација одлучује која се стања чувају, а која се ресетују при сваком прелазу између корака.

4.4.2 Рад над примерима и увоз сопственог кода

Апликација кориснику нуди два начина да дође до графа контроле тока. Њима одговарају странице „Примери“ и „Увоз графа контроле тока“. Први начин је избор једног од шест унапред припремљених примера, међу којима су линеарни ток, наредба `if-else` за гранање, једноставна петља, две угнежђене петље, наредба `switch` за вишеструко одлучивање и петља са гранањем у телу. Други начин је увоз сопственог кода, уносом текста или отпремањем фајла, у било ком од четири подржана језика.

Без обзира на одабрани начин, сви наредни кораци визуелизације су потпуно исти. Обе странице користе исти централни модул за вођење кроз кораке и исту компоненту за приказ графа. Једина стварна разлика је у тренутку добијања графа. Готов пример се бира одмах из списка. Увезени код, са друге стране, прво пролази кроз поступак извлачења графа: слање кода, чекање на обраду и преузимање резултата. Чим увезени граф буде нормализован на



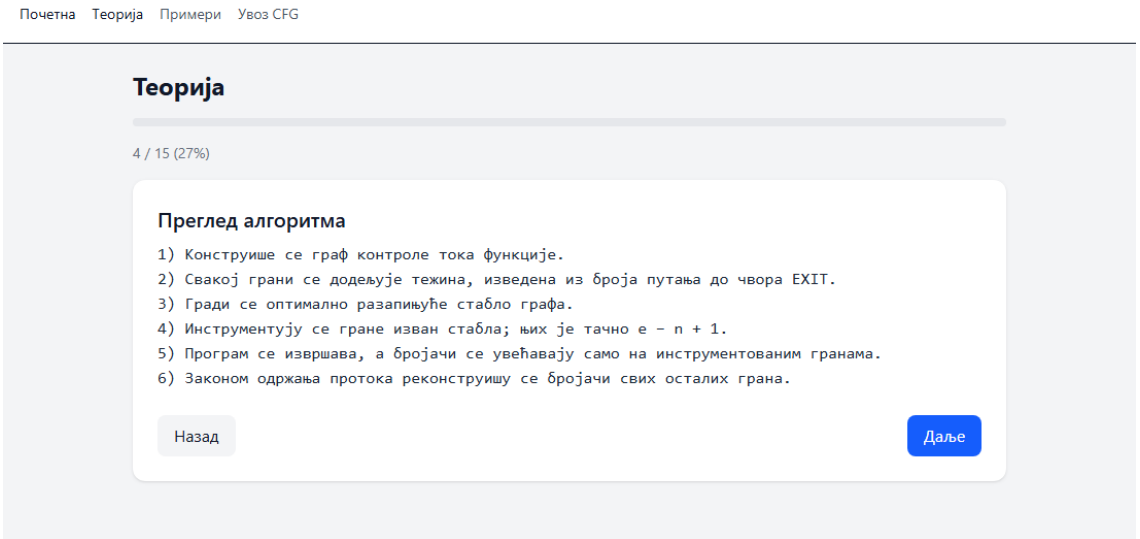
Слика 4.4: Корак конструкције разапињућег стабла на екрану рачунара.

заједнички модел података, апликација га третира потпуно исто као и било који готов пример.

Поред две странице за рад над примерима и за увоз кода, апликација садржи и трећу, потпуно одвојену, са теоријом. За разлику од претходне две странице, она не приказује граф нити рачуна било шта над конкретним примером, већ кориснику корак по корак, кроз низ кратких текстуалних целина, износи основне појмове: граф контроле тока, додела тежина, разапињуће стабло, инструментација и реконструкција. Намењена је као увод пре преласка на интерактивни део апликације. Не зависи од тока визуелизације описаног у

Слика 4.5: Корак конструкције разапињућег стабла на екрану мобилног телефона, за исти пример као на слици 4.4.

овом одељку. Слика 4.6 приказује једну од картица те странице, са прегледом корака алгоритма.



Слика 4.6: Страница „Теорија“, на картици са прегледом корака алгоритма. Кроз картице се пролази редом, дугмићима за напред и назад.

Глава 5

Евалуација

Претходна глава описује како је алат изграђен. Ова глава испитује да ли он ради оно што дефиниција Кнутовог алгоритма налаже. Најпре је описан оквир за проверу исправности, заснован на насумично генерисаним графовима контроле тока (одељак 5.1). Затим су приказани резултати мерења над великим скупом графова (одељак 5.2). Потом је рад алата приказан на конкретним примерима (одељак 5.3). Глава се завршава дискусијом резултата и уочених ограничења (одељак 5.4).

5.1 Оквир за проверу исправности

Сви кораци Кнутовог алгоритма у апликацији написани су за потребе овог рада, без ослањања на готове библиотеке. Теорија при томе прецизно прописује шта резултат сваког корака мора да испуни, од величине разапињућег стабла до тачности реконструисаних бројача. Такве тврдње највише вреде када се провере над широким скупом графова, укључујући и облике које нико није унапред одабрао. Зато је направљен оквир који их проверава над великим бројем насумично генерисаних графова. Оквир позива управо оне функције које користи и сама апликација: доделу тежина, конструкцију разапињућег стабла, избор инструментованих грана, симулацију и реконструкцију.

5.1.1 Генератор графова контроле тока

Граф који није структурно исправан чини мерење бесмисленим. Зато се графови не састављају насумичним спајањем чворова, већ надовезивањем сег-

мената. Сваки сегмент има тачно један улаз и тачно један излаз, па то важи и за граф као целину. Структурна исправност је тако последица конструкције, а не накнадне провере.

Облици сегмената су они које апликација већ нуди међу унапред припремљеним примерима: линеаран ток, гранање, петља, угнежђена петља, вишеструко одлучивање и петља са гранањем у телу. Вишеструко одлучивање је сегмент чији чвор гранања има три до пет излазних грана. По облицима сегмената именовано је пет класа графова: линеарни ток, гранање, петља, угнежђена петља и вишеструко одлучивање. Шеста је мешовита класа, у којој се сегменти свих облика, укључујући петљу са гранањем у телу, нижу насумичним редоследом.

Генератор поштује исте границе које важе и у самој апликацији. Граф има тачно један улазни и тачно један излазни чвор. Сваки чвор је достижан из чвора *ENTRY*, а из сваког чвора постоји пут до чвора *EXIT*. Улазном и излазном чвору придружују се граничне гране, на исти начин као при увозу. Ове особине се, уз то, и проверавају за сваки генерисани граф пре него што уђе у мерење. Извор насумичности у генератору прима семе, па исти улазни број даје исти скуп графова. Без тога ниједан број наведен у овој глави не би могао да се понови.

Повратне гране генератор обележава у тренутку када их гради. Оне су тако позната чињеница о графу, а не резултат накнадног препознавања. Тиме постаје могуће проверити и да ли их сама апликација препознаје тачно, што је предуслов за исправну доделу тежина.

5.1.2 Особине које се проверавају

Над сваким генерисаним графом проверава се шест особина:

1. разапињуће стабло има тачно $n - 1$ грана и повезује све чворове;
2. скуп инструментованих грана има тачно $e - n + 1$ елемената;
3. скупови T и S немају заједничких грана, а заједно дају све гране графа;
4. реконструисани бројачи једнаки су стварнима на *свим* гранама, не само на инструментованим;

5. тежина сваке гране једнака је броју путања од њеног одредишног чвора до чвора *EXIT*, при чему су повратне гране искључене из тог бројања;

6. закон одржања протока важи у сваком чвору графа.

Прве три особине су тврдње о структури решења. Четврта је језгро тврдње о исправности. Мерење на малом скупу грана мора дати тачне вредности за све гране, иначе цео поступак нема сврху. Пета проверава доделу тежина наспрам независног пребројавања путања, при чему су повратне гране узете онакве какве их је генератор направио. Шеста проверава да добијене вредности међусобно нису у сукобу.

Уз ових шест, проверава се и понашање на границама графа. Гранична грана на улазу увек припада скупу мерених вредности, а гранична грана на излазу никада. Проверава се и да се разапињуће стабло бира према идентитету граничних грана, а не према вредности њихове тежине. Када све гране добију тежину нула, стабло и даље мора имати $n - 1$ грана.

Особине се проверавају над два скупа генерисаних графова. Први скуп чини двеста графова. Над њим се проверавају прве три особине, пета особина и понашање на границама графа, пошто се све оне читају непосредно из графа и не захтевају извршавање. Други скуп чини сто педесет графова и служи за проверу четврте и шесте особине. Обе захтевају стварне вредности бројача, па се свака функција из другог скупа симулира двадесет пет пута, што одговара двадесет пет позива функције. Бројачи се сабирају преко свих позива, а реконструисане вредности се затим упоређују са стварним. Други скуп је мањи зато што је симулација знатно скупља од провера над самим графом, а двадесет пет позива обезбеђује да се петље прођу различит број пута. Такав обим даје широку покривеност, а редовно покретање тестова остаје брзо. Мерење над знатно већим скупом обавља се засебном алатком.

Оквир је испитан и намерно уведеним грешкама, како би се проверило да свака тврдња заиста уме да падне. Уклањање једне гране из разапињућег стабла обара прву и другу особину, као и проверу избора стабла по идентитету граничних грана. Обртање знака у једначини одржања протока обара четврту и шесту особину. Изостављање улазне граничне гране из скупа мерених обара четврту и шесту особину, као и тврдњу да се улазна гранична грана увек инструментује. Необележена повратна грана у генератору обара пету особину, јер независно пребројавање путања пријави циклус. У свим случајевима испис

именује граф на коме је тврдња пала, заједно са његовом класом и редним бројем, па се супротан пример може поновити истим семеном.

5.1.3 Однос према постојећим тестовима

Оквир описан у овом одељку није једини вид провере у пројекту. Клијентски део апликације има двадесет три датотеке тестова са укупно сто педесет четири теста, писаних помоћу оквира *Jasmine* [32] и покретаних алатом *Karma* [33]. Серверски део има пет датотека са четрдесет девет тестова, писаних за оквир *pytest* [34].

Ове две врсте провере имају различите улоге. Постојећи тестови проверавају појединачне случајеве: да ли одређена функција над одређеним улазом даје очекивани излаз, да ли се стање корисничког окружења мења очекивано, да ли сервер враћа договорени облик одговора. Провере особина, са друге стране, не наводе очекивани излаз. Оне наводе тврдњу која мора важити за сваки граф, а генератор тражи граф на коме та тврдња не важи.

5.2 Резултати над већим скупом графова

Провере особина говоре да ли је поступак исправан. Оне не говоре колико он штеди. За то је направљена засебна алатка, која исти генератор пушта над скупом од десет хиљада графова. Скуп је описан истим семеном као и у тестовима, па су сви наведени бројеви поновљиви. Ниједан од десет хиљада графова није прекршио структурне особине.

За сваки граф се бележе број чворова n, број грана e, величина скупа инструментованих грана $|S|$ и удео $|S|/e$. Уз то се, симулацијом од педесет пролаза по графу, мери и број операција увећавања бројача, па се пореди са бројем који би исте те симулације захтевале уз инструментацију свих грана.

Један просек над целим скупом не би много значио, пошто се класе графова међусобно знатно разликују. Табела 5.1 зато даје резултате по класи графа, у облику просека и стандардне девијације.

Величине n и e знатно варирају, јер генератор гради и мале и велике графове. Удео $|S|/e$ при томе унутар сваке класе одступа за највише неколико процентних поена. Он, дакле, зависи од облика графа, а не од његове величи-

Класа графа	n	e	$\|S\|$	$\|S\|/e$ (%)	Уштеда (%)
линеарни ток	$14{,}4 \pm 6{,}9$	$13{,}4 \pm 6{,}9$	$0{,}0 \pm 0{,}0$	$0{,}0 \pm 0{,}0$	$100{,}0 \pm 0{,}0$
гранање	$46{,}9 \pm 24{,}6$	$58{,}7 \pm 31{,}5$	$12{,}8 \pm 7{,}0$	$21{,}6 \pm 1{,}1$	$82{,}7 \pm 1{,}3$
петља	$39{,}1 \pm 21{,}1$	$50{,}5 \pm 28{,}1$	$12{,}4 \pm 7{,}0$	$24{,}1 \pm 1{,}1$	$84{,}2 \pm 1{,}4$
угњеждена петља	$52{,}2 \pm 27{,}5$	$76{,}4 \pm 41{,}2$	$25{,}1 \pm 13{,}7$	$32{,}5 \pm 1{,}0$	$79{,}5 \pm 1{,}7$
вишеструко одлучивање	$76{,}9 \pm 41{,}7$	$113{,}3 \pm 62{,}7$	$37{,}4 \pm 21{,}0$	$32{,}6 \pm 1{,}6$	$76{,}5 \pm 1{,}6$
мешовити	$49{,}6 \pm 27{,}9$	$66{,}3 \pm 38{,}5$	$17{,}7 \pm 10{,}7$	$25{,}8 \pm 3{,}5$	$86{,}6 \pm 2{,}7$
укупно	$46{,}5 \pm 32{,}7$	$63{,}1 \pm 48{,}8$	$17{,}6 \pm 16{,}6$	$22{,}8 \pm 11{,}1$	$84{,}9 \pm 7{,}7$

Табела 5.1: Резултати по класи графа. Свака вредност је просек по графовима класе, а иза знака $\pm$ стоји стандардна девијација. Прве четири класе броје по 1667 графова, последње две по 1666, укупно 10000. Уштеда је смањење броја операција увећавања бројача у односу на инструментацију свих грана.

не. Велика девијација у збирном реду потиче од разлика међу самим класама, пре свега од линеарног тока.

Линеаран ток не тражи ниједну инструментовану грану. За граф без гранања важи $e = n - 1$, па је $|S| = e - n + 1 = 0$. Мери се једино гранична грана на улазу, чија вредност представља број позива функције. Из ње се, законом одржања протока, добијају бројачи свих осталих грана. То је најоштрији облик уштеде који алгоритам може да да.

Класа вишеструког одлучивања има највећи удео мерених грана, а ипак већу уштеду у броју операција од угнежђене петље. Разлог је у томе што се гране гранања извршавају највише једном по пролазу кроз граф, док се тело петље извршава више пута. Удео мерених грана и удео мерених операција тако нису иста мера, па је вредно навести обе.

Табела 5.2 исте податке групише по величини графа.

Величина	Графова	n	e	$\|S\|$	$\|S\|/e$	Уштеда
$n \leq 10$	1208	6,9	7,0	1,1	12,4%	92,1%
$n \leq 25$	2300	17,8	20,1	3,3	14,2%	90,6%
$n \leq 50$	2389	37,7	50,2	13,5	26,2%	82,6%
$n \leq 100$	3510	72,0	99,1	28,1	27,9%	81,7%
$n > 100$	593	123,7	183,0	60,4	32,9%	76,3%

Табела 5.2: Резултати по величини графа, мереној бројем чворова.

Удео мерених грана расте са величином графа, од 12% на близу 33%, али остаје испод једне трећине. Величина $|S| = e - n + 1$ мери колико грана граф има изнад минимума потребног за повезаност. Већи графови у корпусу су и гушћи, са више гранања по чвору, па је такав раст очекиван.

Ове две мере нису подједнако поуздане. Удео $|S|/e$ следи непосредно из структуре графа и не зависи ни од чега другог. Уштеда у броју операција, међутим, зависи и од претпоставке о томе колико се пута пролази кроз петљу током извршавања. Симулација ту претпоставку прави на основу тежина грана, што је процена изведена из облика графа, а не из стварног извршавања програма. Наведене вредности уштеде треба читати са том оградом.

5.3 Примери

Претходна два одељка говоре о понашању алата над великим бројем графова. Овај одељак приказује његов рад на три конкретна примера, кроз цео ток кроз кораке. Најпре је приказан једноставан пример без циклуса (одељак 5.3.1), затим пример са петљом (одељак 5.3.2), а потом и кориснички увезен код, од изворног кода до извештаја (одељак 5.3.3).

5.3.1 Пример: линеарни ток и једноставно гранање

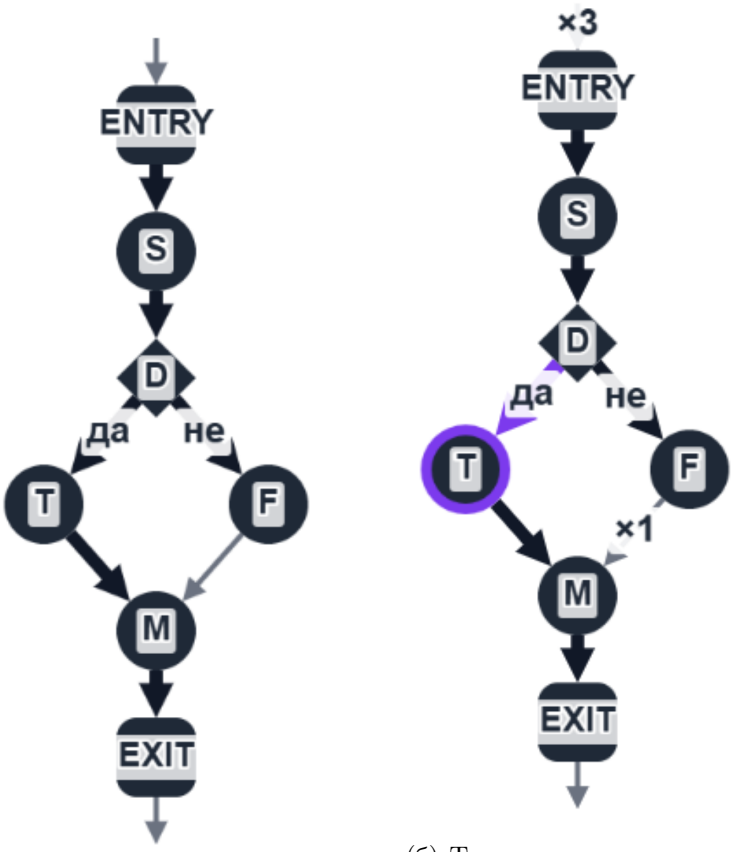
Најједноставнији пример кроз који се овде прати рад апликације је функција са једним гранањем и без циклуса. Улазни параметар се испитује, а затим се у зависности од резултата извршава један од два могућа исхода. Граф контроле тока ове функције има седам видљивих чворова и једно гранање. Листинг 5.1 приказује функцију којој тај граф одговара.

```
1 int f(int x) {
2     int y = x;
3     if (y > 0)
4         y = y + 1;
5     else
6         y = y - 1;
7     return y;
8 }
```

Листинг 5.1: Функција са једним гранањем, чијем графу контроле тока одговара први пример.

Слика 5.1(а) приказује конструисано разапињуће стабло. Подебљаном тамном линијом су означене гране разапињућег стабла T, а тањом и светлијом линијом једина грана из скупа S која се инструментује.

Слика 5.1(б) приказује тренутак током симулације извршавања. Тренутна грана је истакнута бојом, а бројачи већ пређених инструментованих грана се исписују уживо уз одговарајуће гране.



(а) Разапињуће стабло

(б) Тренутак током симулације извршавања

Слика 5.1: Пример са једним гранањем. Танке стрелице изнад чвора *ENTRY* и испод чвора *EXIT* су граничне гране. Бројач уз граничну грану изнад чвора *ENTRY* показује број до тада завршених пролаза.

Слика 5.2 приказује завршни извештај, након двадесет симулираних пролаза кроз граф. Бројач једине инструментоване гране показује 8 пролаза. Из те једне вредности, применом једначина одржања протока, реконструисан је број пролаза кроз сваки преостали чвор. Овде треба разликовати две различите мере уштеде. Број мерних места је смањен са седам на једно, односно за шест мерних места (85,7%). Број стварно извршених операција увећавања бројача током ове симулације износи 8. Уз инструментацију свих седам грана, на истих двадесет пролаза било би их 100, па је уштеда 92%. Ове две

вредности се разликују. Статичка уштеда зависи само од структуре графа, а динамичка и од тога којим путањама извршавање заиста пролази. За неку другу расподелу пролаза кроз исти граф, динамичка уштеда би испала другачија.

Број чворова
7

Број грана
7

Инструментоване гране
1
14,29%

Уштеђене операције над бројачима
92
92,0%

(а) Сажете метрике

Извршавање наредби (чворова), опадајуће по броју пролазака

Анимација (извршено / подешено): 20 / 20 пролазака кроз граф.

Чвор	Ознака	Број извршавања	% по проласку
D	D	20	100,0%
M	M	20	100,0%
S	S	20	100,0%
T	T	12	60,0%
F	F	8	40,0%

(б) Реконструисан број извршавања по чворовима

Слика 5.2: Извештај за пример са једним гранањем.

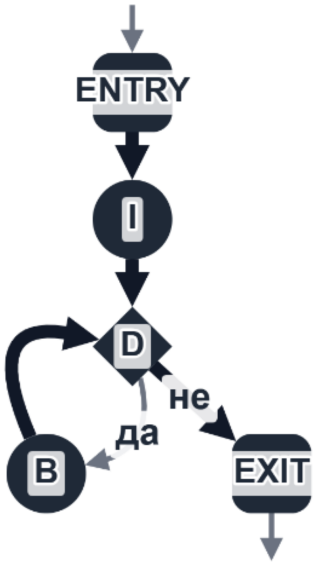
5.3.2 Пример: програм са петљом

Други пример садржи једну петљу. Тело петље се извршава све док је испуњен услов гранања, а излаз из петље наступа у супротном случају. Граф контроле тока овог примера садржи повратну грану. Листинг 5.2 приказује функцију којој тај граф одговара.

```
1 void f(int n) {
2     int i = 0;
3     while (i < n)
4         i = i + 1;
5 }
```

Листинг 5.2: Функција са једном петљом, чијем графу контроле тока одговара други пример.

Слика 5.3 приказује корак конструкције разапињућег стабла за овај пример. Из чвора тела петље, означеног као B, у ацикличном погледу графа нема ниједне путање до чвора $EXIT$, пошто из њега води једино повратна грана. Због тога је $\pi(B) = 0$, па и грана која у тело петље улази има тежину нула. Она је најмање значајна од свих грана овог графа и остаје у скупу S, где се инструментује. Повратна грана води у чвор гранања, чија је вредност π већа од нуле, па она улази у разапињуће стабло.



Слика 5.3: Корак конструкције разапињућег стабла за пример са петљом. Грана која улази у тело петље има тежину нула и инструментује се, док повратна грана припада стаблу. Танке стрелице изнад чвора $ENTRY$ и испод чвора $EXIT$ су граничне гране.

5.3.3 Увоз корисничког кода кроз цео ток

Трећи пример прати увоз кориснички унетог кода кроз цео ток. Унета је кратка функција на језику *C*, која враћа знак свог целобројног параметра: нулу, један или минус један, у зависности од три могућа случаја. Листинг 5.3 приказује тај код.

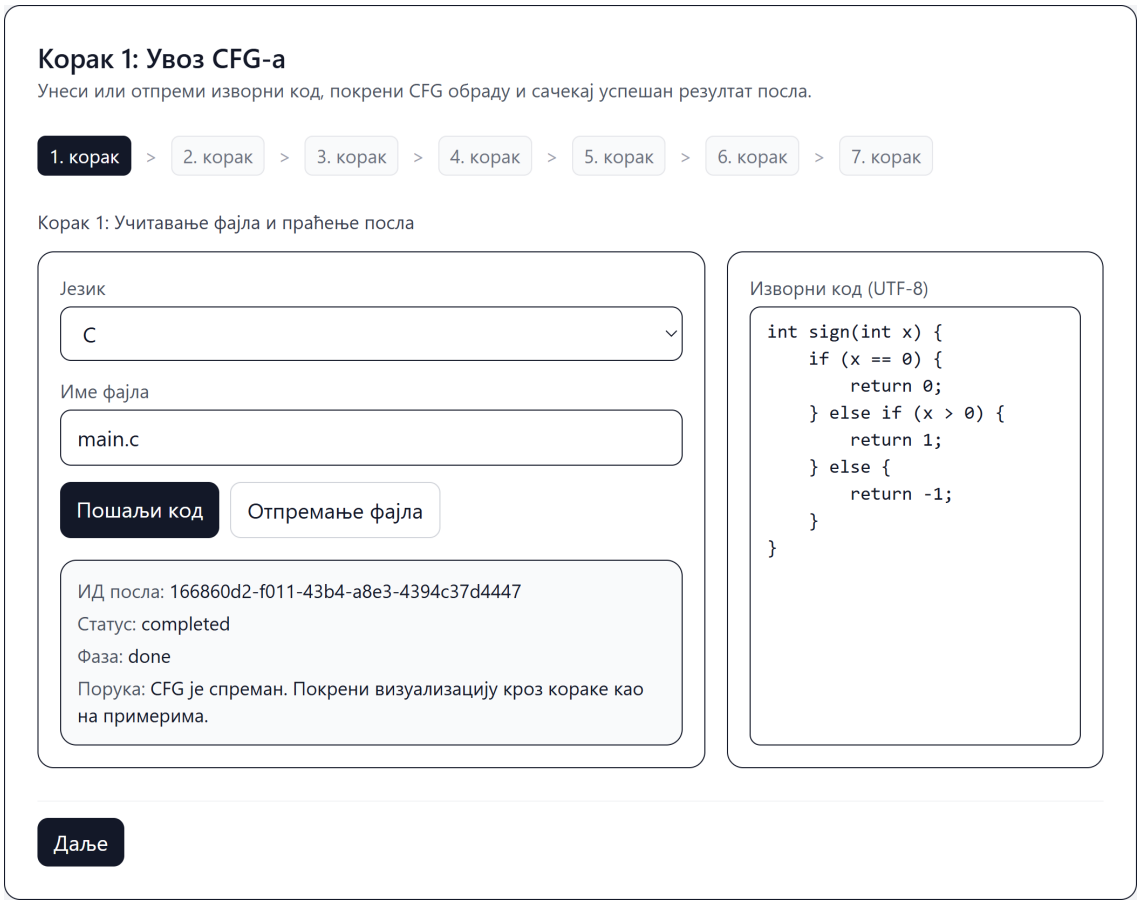
```
int sign(int x) {
    if (x == 0) {
        return 0;
    } else if (x > 0) {
        return 1;
    } else {
        return -1;
    }
}
```

Листинг 5.3: Функција `sign` у програмском језику *C*, унета кроз страницу за увоз.

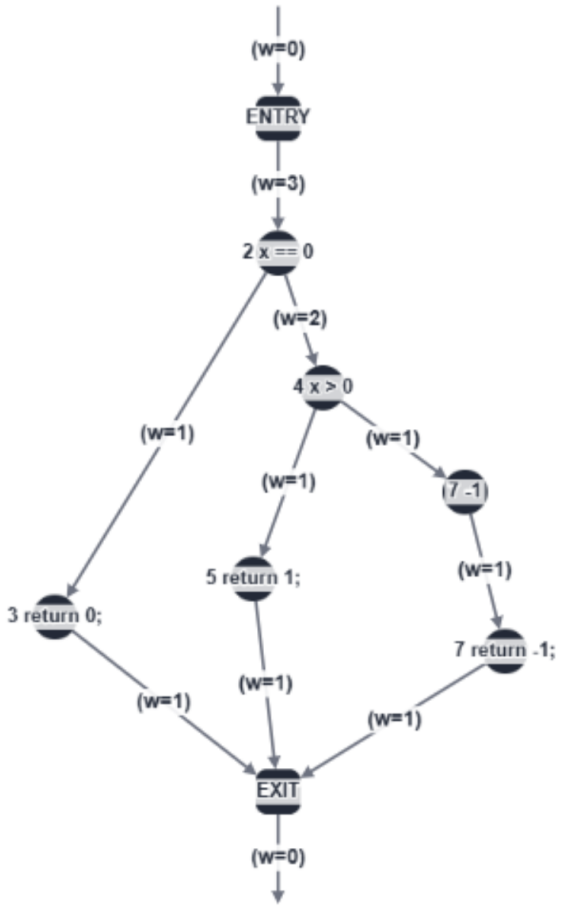
Слика 5.4 приказује страницу за увоз, са унетим кодом, непосредно након успешне обраде на серверу.

Слика 5.5 приказује добијени граф контроле тока, у кораку доделе тежина. Граф има облик сличан првом примеру, са једним додатним гранањем услед трећег могућег случаја. Без икакве ручне обраде спреман је за исте наредне кораке: конструкцију разапињућег стабла, избор инструментованих грана, симулацију и реконструкцију.

Увоз не мора да успе. Очигледно неисправан захтев, попут празног уноса или превелике датотеке, сервер одбија одмах, пре покретања посла. Синтаксне грешке у коду обраду по правилу не прекидају, пошто *Joern* граф гради из оног дела кода који успе да рашчлани. Резултат тада може бити и осиромашен граф, у крајњем случају сведен на чворове *ENTRY* и *EXIT*. Ако сама анализа не успе или прекорачи временско ограничење, посао добија статус „неуспешно“. Страница за увоз тада уместо графа приказује разлог неуспеха и фазу обраде у којој је посао стао.



Слика 5.4: Страница за увоз графа контроле тока, са унетим кодом и успешно завршеним послом обраде.



Слика 5.5: Увезени граф контроле тока, у кораку доделе тежина гранама. Граничне гране изнад чвора $ENTRY$ и испод чвора $EXIT$ носе тежину 0 и не учествују у конструкцији разапињућег стабла.

5.4 Дискусија резултата и ограничења

Табела 5.3 сумира основне бројеве за сва три коришћена примера: број чворова n, број грана e, величину скупа инструментованих грана $|S|$, теоријску доњу границу $e-n+1$ и исход поступка реконструкције. На сва три примера, $|S|$ се поклапа са доњом границом $e-n+1$, а реконструкција у потпуности тачно одређује бројаче свих неинструментованих грана.

Пример	n	e	$\|S\|$	$e-n+1$	Реконструкција
Линеарни ток и гранање	7	7	1	1	успешна
Петља	5	5	1	1	успешна
Увезени код	8	9	2	2	успешна

Табела 5.3: Сажетак резултата по примеру коришћеном у овој глави.

Први пример показује да приступ изложен у овом раду у пракси даје осетну уштеду. Измерена је само једна од седам грана, а укупан број операција увећавања бројача је смањен за 92% у односу на инструментацију свих грана. На примеру са петљом иста идеја се проширује и на графове са циклусима, без измене у поступку. Увоз произвољног кода показује да цео ток подједнако добро ради и над графом који апликација сама не познаје унапред.

Оквир описан у овој глави има и границе које треба навести. Друга особина, $|S| = e-n+1$, аритметичка је последица прве, пошто се скуп S рачуна као комплемент стабла. Она не проверава независну чињеницу, већ то да се комплемент узима над правим скупом грана. Уз то, проверава се да је добијено стабло разапињуће, али не и да је оптимално. Провера оптималности захтевала би поређење са свим разапињућим стаблима графа, што је изводљиво само за врло мале графове. Генератор, најзад, гради графове који одговарају структурираном коду. Он не производи вишеструке гране између истог пара чворова, петље над истим чвором, нити нередуцибилне петље, какве настају скоком у средину тела петље.

Увоз произвољног кода открива и ограничење које није видљиво на унапред припремљеним примерима. Формална дефиниција чвор графа контроле тока изједначава са основним блоком: максималним праволинијским низом наредби. Граф својстава кода алата *Joern*, међутим, не мора пратити ову конвенцију. На пример, потпуно линеарна функција са пет узастопних наредби, без икаквог гранања (три доделе вредности, једно сабирање три променљиве и

једна наредба `return`), кроз увоз даје граф са девет чворова, уместо очекиваног једног. Разлог је што *Joern* сваки подизраз сложенијег израза представља посебним чвором, на пример одвојено $a+b$ и потом збир тог резултата са c. То је финије од нивоа целе наредбе. Ова ситнија подела не утиче на исправност алгоритма, већ искључиво на прегледност добијеног графа. Доказ минималности не зависи од тога шта тачно један чвор представља. Увезени графови за код са сложенијим изразима су, ипак, видно гушћи и тежи за праћење него одговарајући ручно моделовани примери.

Могуће унапређење би било постпроцесирање увезеног графа које би спајало низове чворова међусобно повезане тачно једном граном (чвор u са тачно једним излазним суседом v и чвор v са тачно једним улазним суседом u) у један чвор, чиме би се поново успоставила сагласност са формалном дефиницијом основног блока. Ово унапређење није спроведено у оквиру овог рада и представља могући правац за будући рад.

Слично, при избору разапињућег стабла, уместо чисто структурне тежине $w(e) = \pi(v)$, могла би се применити и прецизнија процена на основу статичких хеуристика предикције гранања, какве су предложили Бал и Ларус [11]. И ово унапређење је остало неспроведено, из истих разлога: обим рада и ризик од грешке при увођењу нове логике касно у изради пројекта. Остаје, попут спајања чворова у основне блокове, као правац за будући рад.

Глава 6

Закључак

Основни допринос овог рада је веб апликација *Knuth Profiler*. Она сваки корак Кнутовог алгоритма приказује на самом графу контроле тока и пушта корисника да кроз кораке пролази сопственим темпом. Апликација ради над два извора графова: над уграђеним примерима карактеристичних облика тока, од линеарног низа до угнежђених петљи, и над произвољним корисничким кодом, из кога се граф извлачи статичком анализом. Од доделе тежина до реконструкције бројача, апликација оба извора обрађује истим поступком. Сваки корак изводе исте функције, над уграђеним примерима и над увезеним кодом.

Исправност имплементације потврђена је оквиром који над сваким насумично генерисаним графом проверава шест особина, од величине разапињућег стабла до тачности реконструисаних бројача на свим гранама. Свих шест особина важи на сваком од генерисаних графова. Над десет хиљада графова просечно се инструментује 22,8% грана, уз уштеду од 84,9% операција увећавања бројача у односу на инструментацију свих грана. Уштеда расте што је граф ближи чистом линеарном току. На самом линеарном току ниједна грана графа се не инструментује, па све вредности следе из броја позива функције.

Кнутов резултат данас постоји у два облика. У литератури је то теорема о минималном броју мерних места, са потпуним и конструктивним доказом. У програмским преводиоцима исти резултат је основ профајлирања грана [3]: корисник добија готове бројеве, а кораци којима се до њих долази остају унутар преводиоца. Главни допринос овог рада је платформа у којој се сваки корак тог поступка може видети, поновити и проверити, како на припремљеним примерима тако и на произвољном корисничком коду.

Решење оставља и правце за будући рад. Алат *Joern* и подизразе представља посебним чворовима, па увезени граф има више чворова него што има основних блокова. Постпроцесирање које би узастопне чворове без гранања спајало у један свело би увезени граф на ниво основних блокова. Додела тежина приказана је као један корак, иако се вредности $\pi(v)$ рачунају чвор по чвор, обрнутим тополошким редоследом; тај рачун би се могао приказивати корак по корак, као што се већ приказује реконструкција бројача. При избору разапињућег стабла могле би се, уместо чисто структурних тежина, применити статичке хеуристике предикције гранања. Најзад, иста апликација могла би да прикаже и профајлирање путања Бала и Ларуса [35], које над истим графом контроле тока решава сложенији задатак и надовезује се на приказани поступак.

Литература

[1] Milena Vujošević Janičić. *Verifikacija softvera*. Matematički fakultet Univerziteta u Beogradu, Beograd, 1 edition, 2026.

[2] Donald E. Knuth and Francis R. Stevenson. Optimal measurement points for program frequency counts. *BIT Numerical Mathematics*, 13(3):313--322, 1973.

[3] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319--1360, 1994.

[4] GNU Project. Using the GNU compiler collection (GCC) – instrumentation options. Online, 2026. `https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html`.

[5] LLVM Project. LLVM – `PGOInstrumentation.cpp`, MST-based PGO instrumentation. Online, 2026. `https://llvm.org/doxygen/PGOInstrumentation_8cpp_source.html`.

[6] Steven Halim. VisuAlgo – visualising data structures and algorithms through animation. *Olympiads in Informatics*, 9:243--245, 2015.

[7] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590--604, 2014.

[8] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2 edition, 2006.

[9] A. Nahapetian. Node flows in graphs with conservative flow. *Acta Informatica*, 3(1):37--41, 1973.

[10] Joseph B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48--50, 1956.

[11] Thomas Ball and James R. Larus. Branch prediction for free. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)*, pages 300--313, 1993.

[12] Google. Angular – documentation. Online, 2026. `https://angular.dev/`.

[13] Microsoft. TypeScript – documentation. Online, 2026. `https://www.typescriptlang.org/`.

[14] Cytoscape.js contributors. Cytoscape.js – documentation. Online, 2026. `https://js.cytoscape.org/`.

[15] Max Franz, Christian T. Lopes, Gerardo Huck, Yue Dong, Onur Sumer, and Gary D. Bader. Cytoscape.js: a graph theory library for visualisation and analysis. *Bioinformatics*, 32(2):309--311, 2016.

[16] dagrejs contributors. Dagre: Directed graph layout for JavaScript. GitHub repozitorijum, 2026. `https://github.com/dagrejs/dagre`.

[17] Eclipse Foundation. Eclipse Layout Kernel (ELK). Online, 2026. `https://eclipse.dev/elk/`.

[18] Python Software Foundation. Python – documentation. Online, 2026. `https://www.python.org/`.

[19] Ramírez, Sebastián. FastAPI – documentation. Online, 2026. `https://fastapi.tiangolo.com/`.

[20] Colvin, Samuel. Pydantic – documentation. Online, 2026. `https://docs.pydantic.dev/`.

[21] Redis Ltd. Redis – documentation. Online, 2026. `https://redis.io/`.

[22] RQ contributors. RQ: Simple job queues for Python. Online, 2026. `https://python-rq.org/`.

[23] Joern contributors. Joern – documentation. Online, 2026. `https://docs.joern.io/`.

[24] Docker Inc. Docker – documentation. Online, 2026. `https://docs.docker.com/`.

[25] Docker Inc. Docker Compose – documentation. Online, 2026. `https://docs.docker.com/compose/`.

[26] Dimitrije Petrović. Knuth profiler – изворни код. GitHub repozitorijum, 2026. `https://github.com/DimePetrovic/knuth-profiler-monorepo`.

[27] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 3 edition, 2003.

[28] Graphviz contributors. Graphviz – DOT language documentation. Online, 2026. `https://graphviz.org/doc/info/lang.html`.

[29] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software: Practice and Experience*, 30(11):1203--1233, 2000.

[30] Robert C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36(6):1389--1401, 1957.

[31] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 3 edition, 2009.

[32] Jasmine contributors. Jasmine – documentation. Online, 2026. `https://jasmine.github.io/`.

[33] Karma contributors. Karma – documentation. Online, 2026. `https://karma-runner.github.io/`.

[34] pytest developers. pytest – documentation. Online, 2026. `https://docs.pytest.org/`.

[35] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-29)*, pages 46--57, 1996.

Биографија аутора

Димитрије Петровић рођен је 1. августа 2000. године у Краљеву. Завршио је Девету гимназију „Михаило Петровић Алас“ у Београду, на природно-математичком смеру. Основне студије на смеру Информатика Математичког факултета Универзитета у Београду уписао је 2019. године и завршио 2024. године, када је уписао и мастер студије на истом факултету. Од краја 2022. године ради на развоју веб апликација, како на клијентској тако и на серверској страни, а искуство је стекао и у инжењерству података. Посебно га занимају пројектовање и развој информационих система.