

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ  
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ  
імені ІГОРЯ СІКОРСЬКОГО»

ФАКУЛЬТЕТ ПРИКЛАДНОЇ МАТЕМАТИКИ

КАФЕДРА СИСТЕМОГО ПРОГРАМУВАННЯ І  
СПЕЦІАЛІЗОВАНИХ КОМП'ЮТЕРНИХ СИСТЕМ

«На правах рукопису»  
УДК 004.4'4

«До захисту допущено»

Завідувач кафедри СПСКС

\_\_\_\_\_ В.П.Тарасенко  
(підпис) (ініціали, прізвище)

“ \_\_\_ ” \_\_\_\_\_ 2018р.

## Магістерська дисертація

на здобуття ступеня магістра

зі спеціальності 123 Комп'ютерна інженерія  
Системне програмування

на тему: Методи оптимізації при трансляції на графі залежності станів та значень

Виконав: студент II курсу, групи КВ-62м

(шифр групи)

Подзе Олександр Сергійович

(прізвище, ім'я, по батькові)

(підпис)

Науковий керівник доцент, к.т.н, Марченко О.І.

(посада, науковий ступінь, вчене звання, прізвище та ініціали)

(підпис)

Рецензент \_\_\_\_\_

(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали)

(підпис)

Засвідчую, що у цій магістерській  
дисертації немає запозичень з праць  
інших авторів без відповідних  
посилань.

Студент \_\_\_\_\_  
(підпис)

Київ – 2018 року

## РЕФЕРАТ

**Актуальність теми.** Сучасні мови програмування надають все більше можливостей для абстрагування створюваного програмного забезпечення від деталей оточення, в якому воно буде працювати. Разом з цим вимоги до швидкодії цих програм не зменшуються, що є мотивацією для дослідження нових шляхів оптимізації програм у процесі їх трансляції у машинні інструкції. Транслятори є одною з найважливіших ланок у сучасній розробці програмного забезпечення, і вже сьогодні здатні у деяких випадках покращити швидкість програм на порядки без втручання зі сторони розробника. Важливим фактором при виборі сучасного транслятору для розробників програмного забезпечення є швидкість компіляції вихідного коду, тому дослідження таких методів оптимізації, які дозволять прискорити процес трансляції, при цьому отримувати на виході програми, що не поступаються по швидкодії програмам, отриманим стандартними методами, є актуальною темою.

**Об'єктом дослідження** є процеси оптимізації коду, що генерується, в трансляторах.

**Предметом дослідження** є методи оптимізації в трансляторах з використанням графу залежності станів та значень у якості внутрішнього подання програми.

**Мета роботи:** прискорення процесів оптимізації у трансляторах, розробка більш швидких методів оптимізації у трансляторах на основі графу залежності станів та значень у якості внутрішнього подання програми.

### **Наукова новизна:**

1. Запропоновано модифікований метод оптимізації видалення мертвого коду, який відрізняється від стандартного тим, що у якості проміжного подання програми використовується граф залежності станів та значень, і має меншу алгоритмічну складність, ніж існуючі методи, та використовує менший розмір пам'яті.

2. Запропоновано модифікований метод оптимізації видалення спільних виразів, який відрізняється від стандартного тим, що у якості проміжного подання програми використовується граф залежності станів та значень, і має меншу алгоритмічну складність, ніж існуючі методи.
3. Запропоновано модифікований метод оптимізації згортки констант, який відрізняється від стандартного тим, що у якості проміжного подання програми використовується граф залежності станів та значень, і має меншу алгоритмічну складність, ніж існуючі методи.
4. Запропоновано модифікований метод оптимізації визначення інваріантів для циклів, який відрізняється від стандартного тим, що у якості проміжного подання програми використовується граф залежності станів та значень, і має меншу алгоритмічну складність, ніж існуючі методи, та використовує менший розмір пам'яті.
5. Запропоновано модифікований метод оптимізації згортки умовних конструкцій, який відрізняється від стандартного тим, що у якості проміжного подання програми використовується граф залежності станів та значень, і має меншу алгоритмічну складність, ніж існуючі методи.

**Практична цінність** отриманих в роботі результатів полягає в тому, що запропоновані методи використовують граф залежності станів та значень як більш універсальне внутрішнє подання програми, що дозволяє розроблювати в трансляторах підсистеми оптимізації програмного коду, які потребують менше часу та менше використаної пам'яті, і можуть бути застосовані як при розробці нових, так і для покращення існуючих трансляторів.

**Апробація роботи.** Основні положення і результати представлені та обговорювалися на науковій конференції магістрантів та аспірантів «Прикладна математика та комп'ютинг» ПМК-2018 (Київ, 21-23 березня 2018 р.). Модифікований метод видалення спільних виразів опублікований

у фаховому виданні «Інтернаука» №9 за 2018 рік, що індексується в міжнародних базах даних Index Copernicus, Open Academic Journals Index, ResearchBib, Scientific Indexing Services, Electronic Journals Library, InfoBase Index, CiteFactor.

**Структура та обсяг роботи.** Магістерська дисертація складається з вступу, трьох розділів та висновків.

*У вступі* подано загальну характеристику роботи, зроблено оцінку сучасного стану проблеми, обґрунтовано актуальність напрямку досліджень, сформульовано мету і задачі досліджень, показано наукову новизну отриманих результатів і практичну цінність роботи, наведено відомості про апробацію результатів і їхнє впровадження.

*У першому розділі* розглянуто загальні положення методів оптимізації, традиційні форми проміжного подання, їх переваги і недоліки.

*У другому розділі* розглядається граф залежності станів та значень, та реалізація методів оптимізації за його допомогою.

*У третьому розділі* наведено особливості реалізації запропонованих алгоритмів, проаналізовано результати перевірки ефективності запропонованих способів у порівнянні із реалізацією цих задач за допомогою класичних засобів.

*У висновках* представлені результати проведеної роботи.

Робота представлена на 80 аркушах, містить посилання на список використаних літературних джерел.

**Ключові слова:** граф залежності станів та значень, транслятори, оптимізація у трансляторах, компіляція.

## РЕФЕРАТ

**Актуальность темы.** Современные языки программирования предоставляют все больше возможностей для абстрагирования создаваемого программного обеспечения от деталей окружения, в котором оно будет работать. Вместе с этим требования к быстродействию этих программ не уменьшаются, что является мотивацией для исследования новых путей оптимизации программ в процессе их трансляции в машинные инструкции. Трансляторы являются одним из важнейших звеньев в современной разработке программного обеспечения, и уже сегодня способны в некоторых случаях улучшить быстродействие программ на порядки без вмешательства со стороны разработчика. Важным фактором при выборе современного транслятора для разработчиков программного обеспечения является скорость компиляции исходного кода, поэтому исследование таких методов оптимизации, которые позволят ускорить процесс трансляции, при этом получать на выходе программы, не уступающие по быстродействию программам, полученным стандартными методами, является актуальной темой.

**Объектом исследования** являются процессы оптимизации кода, генерируемого в трансляторах.

**Предметом исследования** являются методы оптимизации в трансляторах с использованием графу зависимости состояний и значений в качестве внутреннего представления программы.

**Цель работы:** ускорение процессов оптимизации в трансляторах, разработка более быстрых методов оптимизации в трансляторах на основе графу зависимости состояний и значений в качестве внутреннего представления программы.

**Научная новизна:**

1. Предложен модифицированный метод оптимизации удаления мертвого кода, который отличается от стандартного тем, что в качестве промежуточного представления программы используется граф зависимости

состояний и значений, имеет меньшую алгоритмический сложность, чем существующие методы и использует меньший размер памяти.

2. Предложен модифицированный метод оптимизации удаления общих выражений, который отличается от стандартного тем, что в качестве промежуточного представления программы используется граф зависимости состояний и значений, и имеет меньшую алгоритмический сложность, чем существующие методы.

3. Предложен модифицированный метод оптимизации свертки констант, который отличается от стандартного тем, что в качестве промежуточного представления программы используется граф зависимости состояний и значений, и имеет меньшую алгоритмический сложность, чем существующие методы.

4. Предложен модифицированный метод оптимизации определения инвариантов для циклов, который отличается от стандартного тем, что в качестве промежуточного представления программы используется граф зависимости состояний и значений, и имеет меньшую алгоритмический сложность, чем существующие методы и использует меньший размер памяти.

5. Предложен модифицированный метод оптимизации свертки условных конструкций, который отличается от стандартного тем, что в качестве промежуточного представления программы используется граф зависимости состояний и значений, и имеет меньшую алгоритмический сложность, чем существующие методы.

**Практическая ценность** полученных в работе результатов заключается в том, что предложенные методы используют граф зависимости состояний и значений как более универсальное внутреннее представление программы, позволяющей разрабатывать в трансляторах подсистемы оптимизации программного кода, которые требуют меньше времени и меньше использованной памяти, и могут быть применены как при разработке новых, так и для улучшения существующих трансляторов.

**Апробация работы.** Основные положения и результаты представлены и обсуждались на научной конференции магистрантов и аспирантов «Прикладная математика и компьютеринг» ПМК-2018 (Киев, 21-23 марта 2018). Модифицированный метод удаления общих выражений опубликован в специализированном издании «Интернаука» №9 за 2018, который индексируется в международных базах данных Index Copernicus, Open Academic Journals Index, ResearchBib, Scientific Indexing Services, Electronic Journals Library, InfoBase Index, CiteFactor.

**Структура и объем работы.** Магистерская диссертация состоит из введения, трёх и выводов.

Во *введении* представлена общая характеристика работы, произведена оценка современного состояния проблемы, обоснована актуальность направления исследований, сформулированы цели и задачи исследований, показано научную новизну полученных результатов и практическую ценность работы, приведены сведения об апробации результатов и их внедрение.

В *первом разделе* рассмотрены общие положения методов оптимизации, традиционные формы промежуточного представления, их преимущества и недостатки.

Во *втором разделе* рассматривается граф зависимости состояний и значений, и реализация методов оптимизации с его помощью.

В *третьем разделе* приведены особенности реализации предложенных алгоритмов, проанализированы результаты проверки эффективности предложенных способов по сравнению с реализацией этих задач с помощью классических средств.

В *выводах* представлены результаты проведенной работы.

Работа представлена на 80 листах, содержит ссылки на список использованных литературных источников.

**Ключевые слова:** граф зависимости состояний и значений, трансляторы, оптимизация в трансляторах, компиляция.

## ABSTRACT

**Topicality.** Modern programming languages provide more and more opportunities for abstracting the generated software from the details of the environment in which it will work. Along with this, the requirements for the speed of these programs do not decrease, which is a motivation for exploring new ways of optimizing programs in the process of their translation into machine instructions. Translators are one of the most important links in modern software development, and already today in some cases are able to improve the speed of programs on orders without the intervention of the developer. An important factor in choosing a modern translator for software developers is the speed of compiling the source code, so the study of such optimization methods that will speed up the translation process, while getting programs that are not inferior in speed to programs obtained by standard methods is an actual topic.

**The object** of the study are the processes of optimizing the code generated in translators.

**The study examines** methods of optimization in translators using the value state dependence graph as an internal representation of the program.

**Objective:** acceleration of optimization processes in translators, development of faster optimization methods in translators based on the graph of the dependence of states and values as an internal representation of the program.

**Scientific innovation** is as follows:

1. A modified method for dead code elimination is proposed, which differs from the standard one in that the value state dependence graph is used as an intermediate representation of the program, has less algorithmic complexity than existing methods and has less space complexity.

2. A modified method for common subexpression elimination is proposed, which differs from the standard one in that the value state dependence graph is used as an intermediate representation of the program, and has less algorithmic complexity than existing methods.



3. A modified method for constant folding is proposed, which differs from the standard one in that the value state dependence graph is used as an intermediate representation of the program, and has less algorithmic complexity than existing methods.

4. A modified method for loop invariant code motion is proposed, which differs from the standard one in that the value state dependence graph is used as an intermediate representation of the program, and has less algorithmic complexity than existing methods and uses a smaller memory size.

5. A modified method for branch folding is proposed, which differs from the standard one in that the value state dependence graph is used as an intermediate representation of the program, and has less algorithmic complexity than existing methods.

**Practical value** obtained in the results is that the proposed methods use the value state dependence graph as a more useful internal representation of the program that allows developers to implement code optimization subsystems in translators that require less time and less memory, and can be used both to develop new translators, as well as improving already existing ones.

**Testing of work.** The main provisions and results were presented and discussed at the scientific conference of undergraduates and postgraduates "Applied Mathematics and Computing" PMK-2018 (Kiev, March 21-23, 2018). The modified method for deleting common expressions is published in the specialized publication "Internauka" No.9 for 2018, which is indexed in the international databases Index Copernicus, Open Academic Journals Index, ResearchBib, Scientific Indexing Services, Electronic Journals Library, InfoBase Index, CiteFactor.

**The structure and scope of work.** The master's thesis consists of an introduction, three conclusions and conclusions.

*The introduction* presents the general characteristics of the work done assessment of the current state of the problem, the urgency towards research, formulated the

purpose and objectives of research, the scientific novelty of the results and practical value of work, provides information on testing results and their implementation.

*The first section* deals with general provisions of reinforcement learning methods, their features, advantages and disadvantages, also different implementation.

*The second section* describes suggested techniques of combining evolutionary approach with the temporal-difference learning, provides examples of algorithms that implement these techniques.

*The third section* shows implementation of the proposed algorithms, analyzes the results of testing the effectiveness of the proposed methods in comparison with the implementation of these tasks using classical methods.

*In conclusion*, the findings of the work.

Work submitted 80 pages, containing list of used literature sources.

**Keywords:** value state dependence graph, translators, translator optimization, compilation.

## Зміст

Перелік умовних позначень, скорочень і термінів .....	4
ВСТУП .....	5
1. АНАЛІЗ ІСНУЮЧИХ ФОРМ ВНУТРІШНЬОГО ПОДАННЯ ПРОГРАМИ ТА МЕТОДІВ ОПТИМІЗАЦІЇ, ЯКІ ЇХ ВИКОРИСТОВУЮТЬ .....	6
1.1. Класичні форми внутрішнього подання програми .....	7
1.2 Видалення мертвого коду.....	11
1.3 Визначення інваріантів для циклів.....	15
1.3.1. Відношення домінування .....	16
1.3.2. Множини досяжних визначень .....	18
1.3.3. Визначення інваріантів циклічних конструкцій .....	19
1.4 Видалення спільних виразів.....	19
1.5. Граф залежності станів та значень .....	21
1.5.1 Визначення графу залежності станів та значень .....	21
1.5.2 Поняття домінування і пост-домінування .....	24
1.5.3 Попередники та спадкоємці .....	26
1.5.4 Нормалізація графу залежності станів та значень.....	27
2. РОЗРОБКА МЕТОДІВ ОПТИМІЗАЦІЇ НА ГРАФІ ЗАЛЕЖНОСТЕЙ СТАНІВ ТА ЗНАЧЕНЬ.....	29
2.1 Видалення мертвого коду у графі залежності станів та значень .....	29
2.1.1 Опис запропонованого методу .....	29
2.1.2 Порівняння запропонованого методу із стандартним.....	31
2.2 Видалення спільних виразів .....	33
2.2.1 Опис запропонованого методу .....	33

2.2.2	Порівняння запропонованого методу із стандартним.....	38
2.3	Згортка констант.....	40
2.3.1	Опис запропонованого методу .....	40
2.3.2	Порівняння запропонованого методу із стандартним.....	42
2.4	Визначення інваріантів для циклів .....	45
2.4.1	Опис запропонованого методу .....	45
2.4.2	Порівняння запропонованого методу із стандартним.....	48
2.5	Згортка умовних конструкцій.....	49
2.5.1	Опис запропонованого методу .....	49
2.5.2	Порівняння запропонованих методів із стандартним .....	54
3.	РЕАЛІЗАЦІЯ МЕТОДІВ ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ ..	57
3.1	Розробка оптимізуючих трансляторів для порівняння .....	57
3.2	Структура розробленого програмного забезпечення .....	59
3.3	Порівняння результатів роботи окремих методів оптимізації.....	60
3.3.1	Порівняння запропонованого методу видалення мертвого коду із стандартним .....	61
3.3.2	Порівняння запропонованого методу визначення інваріантів циклів із стандартним .....	63
3.3.3	Порівняння запропонованого методу видалення спільних виразів із стандартним .....	66
3.3.4	Порівняння запропонованого методу згортки констант із стандартним.....	68
3.3.5	Порівняння запропонованого методу згортки умовних конструкцій із стандартним .....	69
3.4	Порівняння роботи транслятору, що використовує запропоновані методи із транслятором, який використовує стандартні методи .....	71

ВИСНОВКИ.....	77
СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ .....	80

ДОДАТОК 1. Копія графічного матеріалу: «Граф потоку команд»

ДОДАТОК 2. Копія графічного матеріалу: «Граф залежності станів та значень»

ДОДАТОК 3. Копія графічного матеріалу: «Порівняння алгоритмічної складності розроблених методів із стандартними»

ДОДАТОК 4. Копія графічного матеріалу: «Дерево пост-домінаторів»

ДОДАТОК 5. Копія графічного матеріалу: «Дерево домінаторів»

ДОДАТОК 6. Копія графічного матеріалу: «Порівняльний графік швидкодії запропонованих методів із стандартним»

ДОДАТОК 7. Граматика використаної мови програмування.

## Перелік умовних позначень, скорочень і термінів

CFG – Control Flow Graph, граф потоку команд.

CLOS – Common Lisp Object System.

LISP – LISt Processor, сімейство мов програмування загального призначення.

SSA – Static Single Assignment, форма програми з єдиними записами до змінних.

VSDG – Value State Dependence Graph, граф залежності станів та значень.

Вершина-домінатор – вершина у наведеному графі, яка присутня в усіх шляхах від вершини входу графу до заданої вершини.

Порт – іменоване ребро у графі залежності станів і значень.

Транслятор – програма, що виконує перетворення інших програм на одній мові програмування в програми на іншій мові програмування.

## ВСТУП

Сучасні мови програмування надають все більше можливостей для абстрагування створюваного програмного забезпечення від деталей оточення, в якому воно буде працювати. Разом з цим вимоги до швидкодії цих програм не зменшуються, що є мотивацією для створення програмних комплексів, для яких створення автоматичних перетворень вхідних програм є пріоритетним напрямком розвитку транслятору.

На даний момент стандартні структури, які використовуються в трансляторах – достатньо примітивні, оскільки зазвичай призначені лише для одного перетворення внутрішнього подання програми, або мають приховані зв'язки між собою – тобто, після застосування одного методу оптимізації вимагають додаткових обчислювальних витрат на підтримку в коректній формі. Складність розробки трансляторів в умовах використання подібних форм внутрішнього подання є мотивацією для дослідження нових структур даних для відображення проміжного подання програми при трансляції, а також трансформації програми у цьому представленні.

В роботі запропоновано декілька модифікованих методів оптимізації вхідних програм для трансляторів, які відрізняються від стандартних тим, що використовують граф залежності станів та значень у якості проміжного подання програми. Це надає можливість розробки ефективних методів оптимізації програм, при цьому не витрачаючи ресурси системи на підтримку додаткових структур даних.

# 1. АНАЛІЗ ІСНУЮЧИХ ФОРМ ВНУТРІШНЬОГО ПОДАННЯ ПРОГРАМИ ТА МЕТОДІВ ОПТИМІЗАЦІЇ, ЯКІ ЇХ ВИКОРИСТОВУЮТЬ

У сфері програмного забезпечення транслятор – програма, що перетворює інші програми з однієї форми в іншу – зазвичай, таку, що може використовувати комп'ютер, наприклад, машинні інструкції. Мова програми, яка подається на вхід транслятора, називається вхідною мовою, а мова результату роботи транслятора – цільовою мовою [1].

З самого початку транслятори були досить простими програмами – були здатні лише на пряму інтерпретацію вхідного тексту, де відповідність між текстом програми і результатом роботи була лише у форматі запису. З часом розвиток теорії мов програмування призвів до створення трансляторів та мов, що були набагато ближче до людської мови та простіше могли бути адаптовані до потрібної предметної області. Близько 1950-х почали з'являтися перші мови програмування загального призначення (COBOL, FORTRAN, LISP), які могли використовуватися у області досліджень чи підприємницької діяльності.

Не зважаючи на те, що сьогоднішній перелік мов програмування надзвичайно широкий, більшість трансляторів базуються на одних і тих самих принципах. Оскільки велика частина конструкцій мов програмування не може бути елементарно відображена у машинні інструкції, найпростіша реалізація транслятору буде створювати погано оптимізований машинний код у порівнянні із тим, який був би написаний вручну. Саме тому неможливо уявити сучасну мову програмування, транслятор якої не має низки методів оптимізації, призначених для підвищення швидкодії програм в результаті своєї роботи.



## 1.1. Класичні форми внутрішнього подання програми

Традиційна послідовність роботи транслятора зображена на Рисунку 1[1].

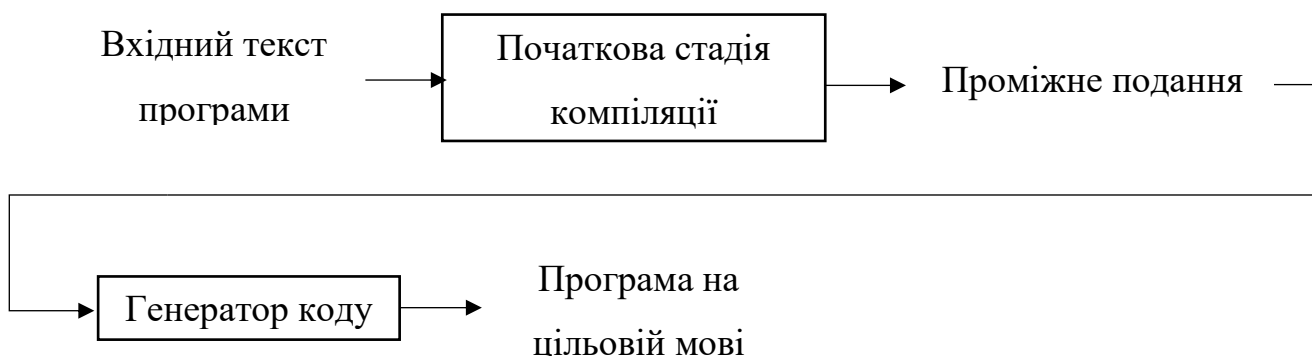


Рисунок 1 – Загальна схема роботи транслятора

Оптимізація в термінах трансляторах означає перетворення тексту програми, яке здатне зробити результат швидшим або меншим за розміром. Методи оптимізації можуть бути застосовані на будь-яких етапах трансляції, але найбільша кількість з них використовують проміжне подання програми.

Проміжне подання програми – спрощена структура, що дозволяє проводити аналіз вхідного програмного забезпечення. Деталі реалізації відрізняються від одного транслятора до іншого, але найпростішою базовою формою подання являється граф потоку керування (Control flow graph, CFG). В ньому кожна інструкція у вхідній програмі відповідає вершині графу, а ребра показують можливі переходи між вершинами. Наприклад, для наступного фрагмента псевдокоду:

```
int a = b + 4;  
if (a > 3) c++;  
else d++;  
return a;
```

Граф потоку команд має вигляд, наведений на Рисунку 2.

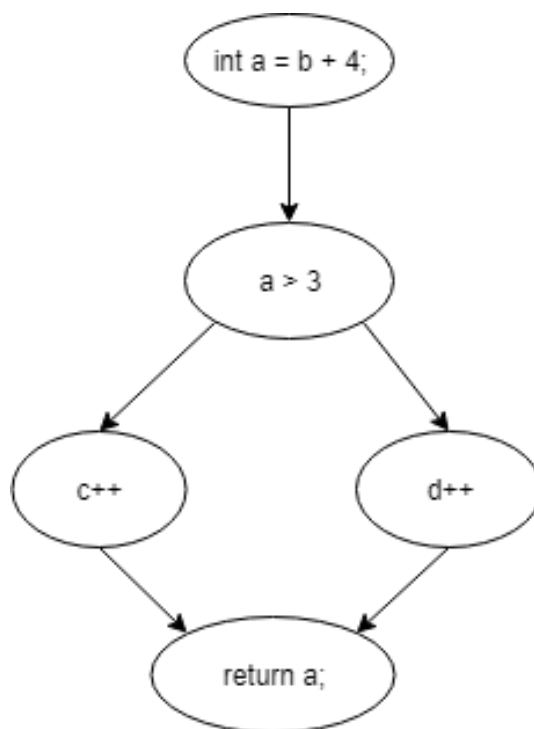


Рисунок 2 – Приклад графу потоку керування.

Перевагою такої форми подання є простота його побудови і використання. Також, генерація програмного коду може бути досить простою, якщо семантична різниця між вхідною і цільовою мовами невисока. Недоліком такого подання є відсутність поняття послідовності виконання операцій – ребра у графі лише показують, яка інструкція може виконатися після поточної, а також неможливість описання програм з багатьма процедурами за його допомогою – для цього використовується додатковий граф викликів процедур, який показує залежність у викликах між процедурами програми. Така проста структура дає невеликі можливості для розробки методів оптимізації – лише поверхневі аналізи, наприклад видалення мертвого коду.

Для реалізації деяких методів оптимізації використовують граф потоку даних (Data Flow Graph – DFG), який будується на основі графу потоку керування: вершини відповідають інструкціям вхідної мови програмування, але ребра показують залежність операцій від даних, що були створені або модифіковані іншими операціями.

Наприклад, для наступного фрагмента псевдокоду:

```
x = a + b;  
y = a * c;  
z = x + d;  
x = y - d;  
x = x + c;
```

Граф потоку даних має вигляд, наведений на Рисунку 3.

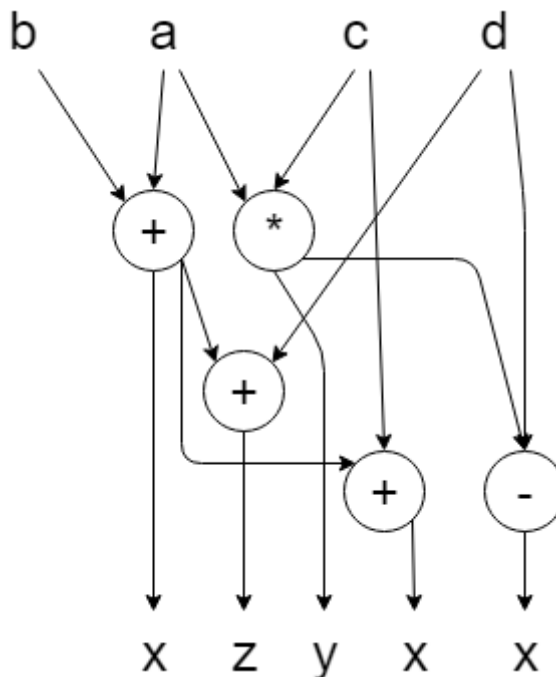


Рисунок 3 – Приклад графу потоку даних

На рисунку видно, що змінним відповідають вершини без обрамлення. Вихідні ребра від змінних означають, що їх значення використовується в операції. Операції на графі позначені вершинами з обрамленням. На даному прикладі наведені лише арифметичні операції. Вхідними ребрами для арифметичних операцій є залежності, які необхідні для отримання результату. Вихідні ребра показують, залежності в програмі від результату операції.

Варто зазначити, що у прикладі в змінну  $x$  було записано три різних значення. При цьому значення змінних залежать від стану  $x$  у різні моменти виконання програми. На графі потоку даних це помітно через те, що були створені декілька вихідних ребер із значенням  $x$ . Зазвичай для того щоб уникнути складностей з обробкою подібних ситуацій, програму в

проміжному поданні приводять до форми з єдиними записами до змінної (SSA-форма).

Наприклад, так буде виглядати програма в формі з єдиними записами до змінних для попереднього фрагмента псевдокоду:

```
 $x_1 = a + b;$   
 $y = a + c;$   
 $z = x_1 + d;$   
 $x_2 = y - d;$   
 $x_3 = x_2 + c;$ 
```

Для такої програми граф потоку даних буде мати вигляд, наведений на Рисунку 4.

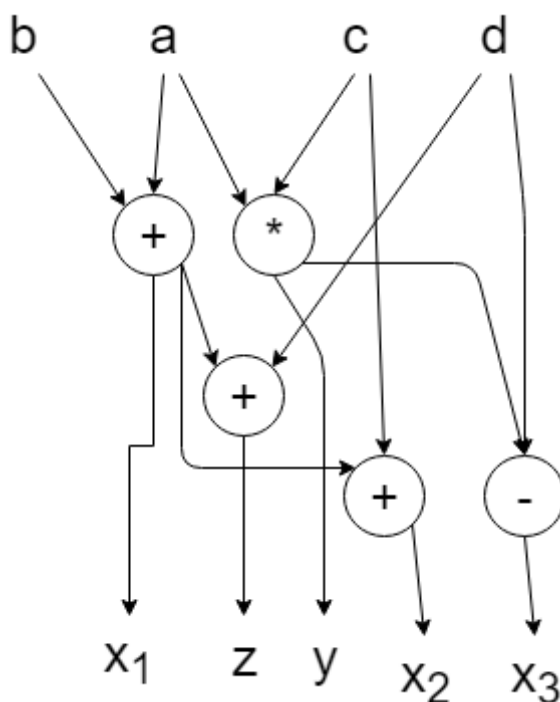


Рисунок 4 – Граф потоку даних для програми в формі одиничного запису до змінних

Приведення програми до форми з єдиними записами до змінних спрощує аналіз програми, оскільки для будь-якої змінної існує єдине місце, в якому вона набула свого поточного значення. Для багатьох методів оптимізації форма з одиничним записом до змінних є необхідною умови правильної роботи.

Наведемо приклади декількох методів оптимізації програм, які передбачають, що вхідний текст був представлений у формі з одиничним записом до змінних:

- призначення регістрів змінним;
- визначення порядку виконання арифметичних операцій;
- видалення невикористаних присвоєнь до змінної;
- згортка констант;
- глобальна нумерація значень;
- зменшення вартості операцій;

Використання комбінації з графу потоку даних і графу потоку команд дає можливість реалізації таких високорівневих методів оптимізації програми як видалення спільних виразів, більш ретельне видалення мертвого коду, тощо.

## 1.2 Видалення мертвого коду

При написанні великого програмного комплексу неминуче виникають ситуації, в яких фрагменти програми будуть збитковими. Вони можуть бути збитковими через різні причини: надмірна абстракція мови програмування від схеми, на якій ця програма виконується, необережність автора програми, тощо. «Мертвим» кодом заведено вважати конструкції наступних типів:

1. Повторне присвоєння у змінну без використання її значення між присвоєннями. Наприклад:

```
x = a + b;
x = b;
```

З фрагменту видно, що перше присвоєння у змінну **x** не змінює процес роботи програми, але призводить до зайвих операцій зчитування і запису в регістри. Видалення першого виразу не змінить результату роботи програми.

Варто зазначити, що існують винятки, при яких не можна повністю видалити подібні вирази, наприклад:

```
x = create_store_user_id();
x = a + b;
```

Видалення першого виразу в цьому випадку може призвести до зміни роботи програми (наприклад – не буде створений користувач у базі даних), що порушує одне з основних правил оптимізації. Зазвичай виклики функцій

вважаються небезпечними для видалення і залишаються у вихідному коді. Подальші застосування методів оптимізації (таких як розгортка функцій) можуть привести програму до вигляду без небезпечних викликів, і подібне присвоєння можна буде видалити.

2. Перевірка змінних на значення, яких вона не може набути в даному фрагменті програми. Наприклад:

```
x = 500;  
if (x < 200) {  
    return x * x;  
} else {  
    return x;  
}
```

З фрагменту коду видно, що умова  $x < 200$  завжди приймає значення *false*, а тому ця перевірка збиткова. Видалення перевірки зменшить загальний об'єм програми, а також дозволить уникнути умовного переходу, який являється порівняно дорогою операцією в сучасних процесорах.

3. Фрагменти програми, недосяжні при будь-яких вхідних даних.

Наприклад:

```
x = 200;  
return x;  
y = 500;  
return y;
```

Останні два вирази можна видалити з програми, при цьому потік команд не зміниться. Така оптимізація не впливає на швидкість виконання програми, лише зменшує її розмір.

Для знаходження усіх вище згаданих видів «мертвого» коду використовується комбінація з декількох проходів по внутрішньому поданню програми. Найпростішим для знаходження можна назвати третій випадок: недосяжні конструкції програми не будуть входити до одної і тої самої компоненти зв'язності графу потоку команд, що і початок програми. Для наведеного прикладу граф потоку програми буде мати вигляд, наведений на Рисунку 5.

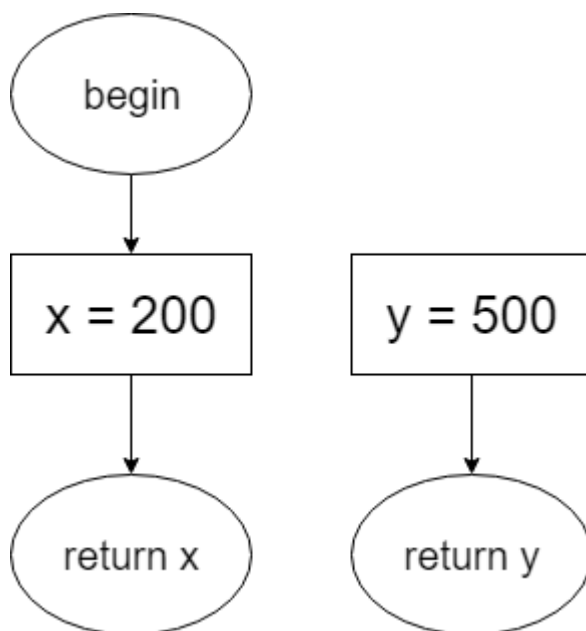


Рисунок 5 – Граф потоку команд в програмі з недосяжними інструкціями

Очевидно, що потік команд не може досягти двох останніх інструкцій, і їх можна видалити.

Інші випадки мертвого коду вимагають більш складних операцій для визначення. Зазвичай для цього використовують аналіз часу життя змінних (*liveness analysis*). Принцип такого підходу полягає в наступному:

Змінна в певній точці програми вважається «живою», якщо її значення буде використано далі в процесі виконання. Інакше, вона вважається «мертвою». Значення, що записуються у «мертві» змінні можна видалити для збереження пам'яті комп'ютера, а також для уникнення непотрібних обчислень. За допомогою графу потоку керування неможливо побудувати повний пошук мертвого коду, адже цей граф не показує залежності значень, які використовуються у інструкціях. Але такі залежності можна відстежувати, звертаючись до графу потоку даних. Дані з цих двох структур поєднуються - для кожної інструкції із графу потоку керування будуються дві множини: множина змінних, від яких дана інструкція залежить (*use*), і множина змінних, і множина змінних, які модифікуються даною інструкцією (*def*). Наприклад, для наступного фрагменту програми, наведеному на Рисунку 6:

<b>1</b>	<b><math>a := 0</math></b>	<i>use</i>	<i>def</i>
<b>2</b>	<b><math>L1: b := a + 1</math></b>		<b><math>a</math></b>
<b>3</b>	<b><math>c := c + b</math></b>	<b><math>a</math></b>	<b><math>b</math></b>
<b>4</b>	<b><math>a := b * 2</math></b>	<b><math>bc</math></b>	<b><math>c</math></b>
<b>5</b>	<b><math>if\ a &lt; 9\ goto\ L1</math></b>	<b><math>b</math></b>	<b><math>a</math></b>
<b>6</b>	<b><math>return\ c</math></b>	<b><math>a</math></b>	
		<b><math>c</math></b>	

Рисунок 6 – Множини use та def

Відповідно, для кожної вершини також будуються множини *in* та *out*. Множина *in* – змінні, що являються «живими» перед виконанням інструкції, а множина *out* – змінні, що являються «живими» після виконання інструкції. Ці зв'язки описуються рівняннями (1) та (2):

$$in'[n] = use[n] \cup (out[n] - def[n]) \quad (1)$$

$$out'[n] = \bigcup_{s \in succ[n]} in[s] \quad (2)$$

На початку роботи алгоритму обидві множини для кожної вершини вважаються пустими. Робота алгоритму завершується, коли для кожної вершини виконуються умови (3) та (4):

$$in'[n] = in[n] \quad (3)$$

$$out'[n] = out[n] \quad (4)$$

Для вказаного фрагменту коду наведемо стан множин на кожній ітерації зображений у Таблиці 1.



Таблиця 1

## Множини in та out в процесі роботи алгоритму

№	0		1		2		3		4		5		6		7	
	<i>use</i>	<i>def</i>	<i>in</i>	<i>out</i>	<i>in</i>	<i>out</i>	<i>in</i>	<i>out</i>	<i>in</i>	<i>out</i>	<i>in</i>	<i>out</i>	<i>in</i>	<i>out</i>	<i>in</i>	<i>out</i>
1		a				a		a		ac	c	ac	c	ac	c	ac
2	a	b	a		a	bc	ac	bc	ac	bc	ac	bc	ac	bc	ac	bc
3	bc	c	bc		bc	b	bc	b	bc	b	bc	b	bc	bc	bc	bc
4	b	a	b		b	a	b	a	b	ac	bc	ac	bc	ac	bc	ac
5	a		a	a	a	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac	ac
6	c		c		c		c		c		c		c		c	

Результатом роботи алгоритму можна користуватися при визначенні часу життя змінних – видаляти ті, які не містяться у множині *out* для певного виразу, або використовувати для інших методів оптимізації.

## 1.3 Визначення інваріантів для циклів

Інваріантом для циклу у програмі називається значення, яке не змінюється у процесі виконання циклу. Наприклад, для наступного фрагменту коду:

```

a = 3.141;
for(i = 0; i < linkedList.length(); i++) {
    item = sqrt(sin(a));
    linkedList.putAt(i, item * i);
}

```

Для наведеного фрагменту обчислення значення змінної *item* та обчислення довжини зв'язного списку та являються інваріантними, а тому ці інструкції можна винести за межі циклу, не змінивши роботи програми:

```

a = 3.141;
item = sqrt(sin(a));
tmp = linkedList.length();
for(i = 0; i < tmp; i++) {
    linkedList.putAt(i, item * i);
}

```

Будь-які математичні обчислення, які не залежать від змінних циклу, можна безпечно перемістити за межі циклу. Деякі обчислення неможливо винести за межі циклу, не змінивши результат роботи програми, наприклад:

```
for(i = 0; i < LinkedList.length(); i++) {  
    item = random();  
    LinkedList.putAt(i, item * i);  
}
```

Винесення виклику *random()* за межі циклу призведе до того, що всі об'єкти списку будуть ініційовані одним і тим самим значенням. З цього можна зробити висновок, що необхідно забороняти переміщення інструкцій, що мають побічні ефекти.

Складність цієї оптимізації полягає в тому, що на графі потоку команд і графі потоку даних циклічні конструкції ніяк не відображаються. Це означає, що пошук інваріантів для циклів починається з пошуку циклів у графі потоку команд. Для цього зазвичай будуються множини домінування для вершин графу, а також множини досяжних визначень.

### 1.3.1. Відношення домінування

Вважається, що вершина *a* домінує вершину *b*, якщо усі можливі шляхи у графі потоку команд від вхідної вершини до *b* проходять через *a*. Таке відношення позначається *a dom b*.

Вважається, що вершина *a* безпосередньо домінує вершину *b*, якщо *a dom b*, при цьому  $a \neq b$ , та не існує такої вершини *c*, щоб *a dom c*, та *c dom b*. Таке відношення позначається *a idom b*.

Розглянемо наступний граф потоку команд з циклами, наведений на Рисунку 7:

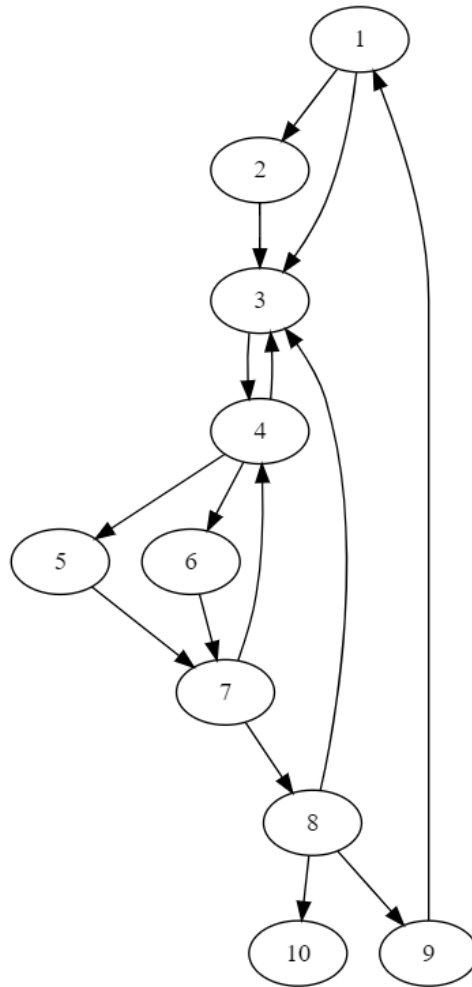


Рисунок 7 – Граф потоку команд з циклами

Відношення домінування являється відношенням часткового впорядкування: воно має властивість транзитивності, рефлексивності та антисиметрії. Це означає, що усі відношення домінування між вершинами можна відобразити на новому графі. Для попереднього прикладу він буде мати вигляд, наведений на Рисунку 8:

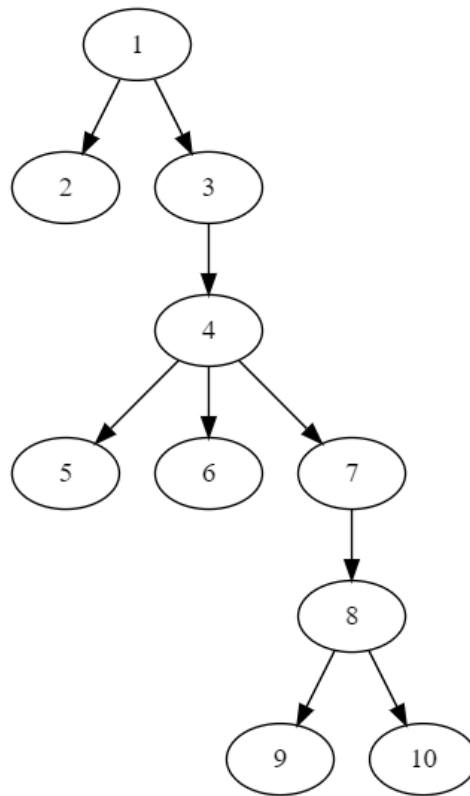


Рисунок 8 – Дерево домінаторів

На графі із визначеними множинами домінаторів для кожної вершини циклічні конструкції можна визначити наступним чином:

1. Цикли повинні мати єдину точку початку роботу, що називається заголовком.
2. Ребро між від вершини  $b$  до  $a$ , якщо  $a \text{ dom } b$  буде означати, що  $b$  являється хвостом циклу, при цьому  $a$  являється заголовком/

Цикли, визначені цими правилами, називаються природними. Ці правила не покривають усі можливі варіанти створення циклічних конструкцій у програмі, але зазвичай їх достатньо для покриття абсолютної більшості випадків у сучасних мовах програмування. У наведеному вище прикладі можна дійти висновку, що вершина 9 являється хвостом циклу, а вершина 1 – його заголовком.

### 1.3.2. Множини досяжних визначень

Визначення змінної  $x$  – інструкція у графі потоку команд, яка змінює або може змінити її значення в процесі виконання програми.

Множина досяжних значень для вершини  $v$  – множина вершин-визначень  $R$ , для яких існує шлях, що не містить повторних визначень змінної  $x$ .

Побудова цих множин зазвичай використовує визначені в попередньому розділі множини *in* та *out*. Маючи ці множини, можна сформулювати досяжні значення, слідуючи правилам:

1. Кожна вершина, що містить визначення змінної, видаляє попереднє визначення з  $R$ , та замінює її собою для всіх нащадків.
2. Якщо вершина має більше одного попередника, то її множина визначень являється об'єднанням множин усіх попередників

### 1.3.3. Визначення інваріантів циклічних конструкцій

Після визначення вершин, що належать циклічним конструкціям, та множин досяжних визначень, інваріантами циклів можна назвати наступні вершини, у яких усі досяжні визначення, від яких залежить інструкція, знаходяться за межами циклу.

$$I = \{v \mid R(v) \cap L = \emptyset\} \quad (5)$$

Де  $R(v)$  – множина досяжних значень у вершині  $v$ ,

$L$  – множина вершин, що належать циклу

### 1.4 Видалення спільних виразів

Вираз називається спільним, якщо його значення було обчислено раніше, і значення змінних, від яких цей вираз залежить, також не змінилося.

Наприклад:

```
a := b * c + g;  
d := b * c * e;
```

Для зменшення кількості виконаних обчислень доцільно ввести додаткову змінну із проміжним результатом, використавши її при обчисленні двох виразів:

```
tmp := b * c;  
a := tmp + g;  
d := tmp * e;
```

Для довільного виразу необхідне виконання декількох вимог, щоб можна було безпечно використати його як спільний у двох окремих інструкціях програми:

1. Між двома вершинами не існує інструкцій, які змінюють значення виразу.
2. Будь-який шлях між двома вершинами обчислює вираз щонайменш один раз.

Наприклад, для графу потоку команд, наведеного на Рисунку 9:

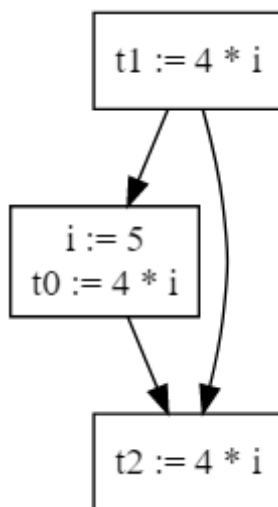


Рисунок 9 – Граф потоку команд із спільними виразами

На наведеному графі видно, що неможливо використати спільний вираз для обчислення змінних  $t_0, t_1, t_2$ . Для використання спільних виразів в даному випадку необхідно, щоб значення  $i$  було незмінним для усіх блоків.

Для визначення множин доступних виразів використовують правила, схожі на ті, що використовуються при визначенні множин досяжних визначень:

$$in[start] = \emptyset \quad (6)$$

$$in[v] = \bigcap_{p \in succ(v)} out[p] \quad (7)$$

$$out[v] = use[v] \cup (in[v] - def[v]) \quad (8)$$

Видно, що основною різницею для визначення множин є особлива поведінка для початкової вершини, та заміна операції об'єднання операцією перетину двох множин. Відповідно, після побудови множин  $in, out$  стане

відомо, чи являється для двох вершин використання спільного виразу безпечним.

## 1.5. Граф залежності станів та значень

### 1.5.1 Визначення графу залежності станів та значень

Граф залежності станів та значень (Value State Dependence Graph, VSDG), запропонований в [2] – проміжне подання програми у вигляді орієнтованого графу. Він складається з декількох видів вершин і ребер. Усі вершини у графі зображують обчислення, а ребра у графі показують залежності певного типу між вершинами. Формально цей граф описується наступним чином:

Граф залежності станів та значень – орієнтовний граф  $G = (N, E_v, E_s, l, N_0, N_\infty)$ , що складається з вершин  $N$ , ( $N_0$  – вхідна вершина,  $N_\infty$  – вершина результату), ребер залежності значень  $E_v \subseteq N \times N$ , та ребер залежності стану  $E_s \subseteq N \times N$ . Функція розмітки  $l$  визначає мітку кожної вершини.

Граф залежності станів та значень будується для однієї процедури вхідної програми – так само, як і класичний граф потоку команд. Основна ідея конструкції графу в тому, щоб поєднати можливості графу потоку команд та графу потоку даних у єдину структуру.

Ребра  $E_v$  відповідають за потік значень між вершинами. Ребра  $E_s$  відображають послідовності для обчислень, які не можуть бути розділені на незалежні блоки. Ребро завжди направлене від вершини, яка має залежність до вершини, від якої вона залежить. Вихідні ребра від вершини зазвичай називають вхідними портами, а вхідні ребра – портами виходу обчислення. Приклад таких відношень зображений на Рисунку 10.

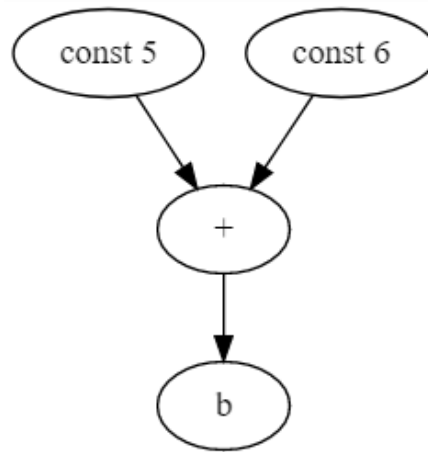


Рисунок 10 – Фрагмент графу залежності станів і значень

Існують наступні типи вершин у графі залежності станів і значень:

1. Вершини значень – елементарні арифметичні операції, а також константи.
2. Вершини станів – вершини, що мають явні побічні ефекти – наприклад, запис значення до оперативної пам’яті. Лише ці вершини можуть залежати або мати на виході ребра залежності стану. Зазвичай входами таких вершин є поточний стан  $S$ , адреса  $a$  та значення  $v$ . Виходом вершини буде ребро стану  $S'$ . Окремо виділяється також вершина циклу та вершина виклику функції, які будуть розглянуті далі.
3. Булевий селектор - відповідає умовним виразам в програмі. Він має три вхідних порти:  $T, F, C$ . В залежності від значення, яке подається на вхід  $C$ , селектор повертає значення із входу  $T$  або  $F$  до вихідного порту. Приклад вершини булевого селектора наведений на Рисунку 11.



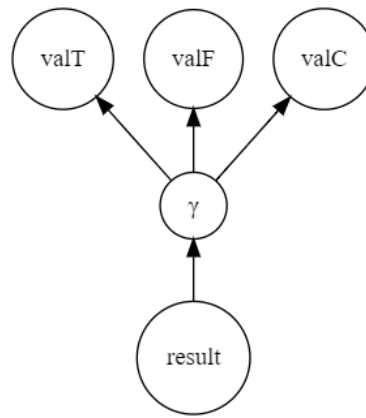


Рисунок 11 – Булевий селектор

4. Вершина циклічних конструкцій  $\theta$  – вершина, що відображає ітеративну поведінку циклічних конструкцій в типових мовах програмування. Вона має 5 портів, що відповідають за окремі етапи виконання циклів у програмі. Розглянемо наступний псевдокод:

```

j = ...
for (i = 0; i < 10; i++) --j;
... = j

```

Граф залежності станів і значень буде мати вигляд, наведений на Рисунку 12.

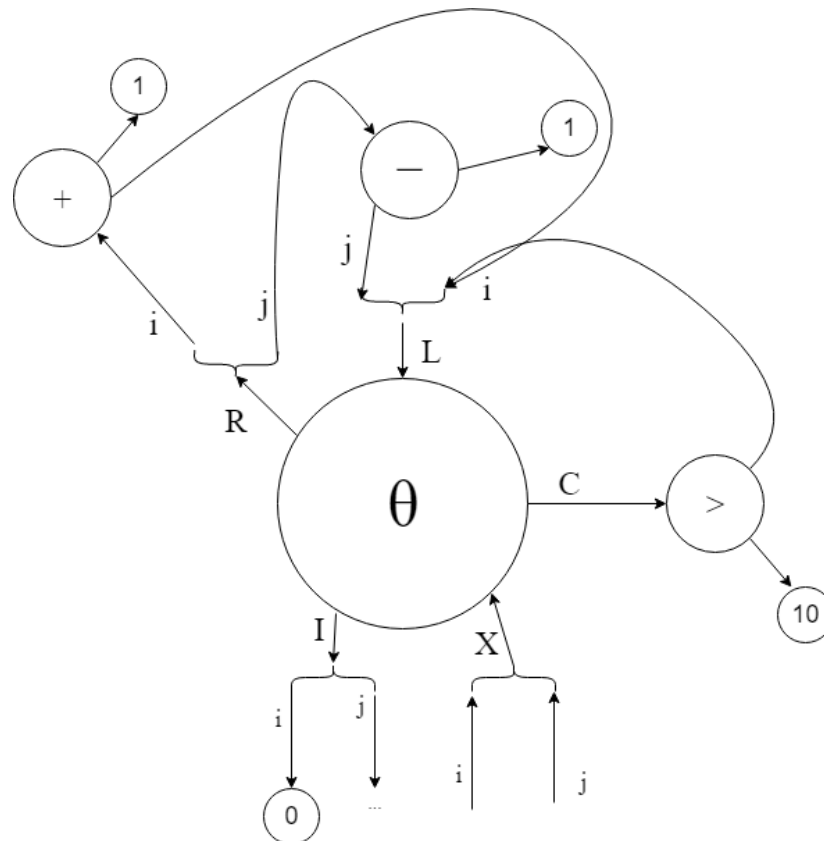


Рисунок 12 – Циклічна конструкція у графі залежності станів і значень

Вершина циклу має наступні порти:  $R, L, C, I, X$ . Вершина має свій внутрішній стан, початковим значенням являються значення, що подаються на вхід  $I$ . Вхідний порт  $C$  визначає умову завершення циклу. Доки ця умова правдива, на вхід  $L$  подається поточне значення внутрішнього стану циклу. Внутрішній стан циклу змінюється на кожній ітерації циклу відповідно до входу  $R$ . В результаті завершення циклу значення внутрішнього стану подаються через порт  $X$ . Оскільки цикл може впливати на значення декількох змінних під час роботи, то вважається, що усі порти цієї вершини складаються з кортежів.

5. Вершина результату. При побудові графу необхідно щоб була лише одна точка входу і одна точка виходу. Такою точкою виходу є вершина результату, яка містить лише одну залежність від стану програми в результаті виконання функції.
6. Вершина виклику. Приймає на вхід значення формальних параметрів, а також стан. Вихідним значенням є змінений стан та результат роботи функції.
7. Вершини формальних параметрів. Такі вершини не мають вхідних портів, на вихід видають значення, з яким була викликана функція.

#### 1.5.2 Поняття домінування і пост-домінування

По аналогії із графом потоку команд, для графу залежності станів і значень можна визначити відношення домінування. Для графу потоку команд це відношення використовується для знаходження тіла циклічних конструкцій та побудови форми з єдиним записом до змінних. В графі залежності станів і значень це відношення використовується лише для знаходження інваріантних значень при повторних обчисленнях однакових виразів.

Також можна створити обернене відношення, що називається *пост-домінуванням*:

Вершина  $p$  *пост-домінує* вершину  $q$  якщо усі шляхи від  $q$  до  $N_\infty$  містять в собі  $p$ . Таке відношення позначається як  $p \text{ pdom } q$ . Для цього відношення можна також побудувати дерево пост-домінаторів, аналогічно до дерева домінаторів. Приклади графу залежності станів і значень, а також відповідних дерев домінаторів і пост-домінаторів наведено на Рисунку 13-15.

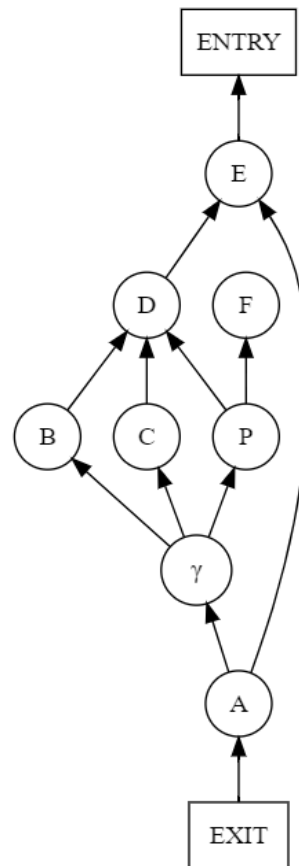


Рисунок 13 – Граф залежності станів і значень

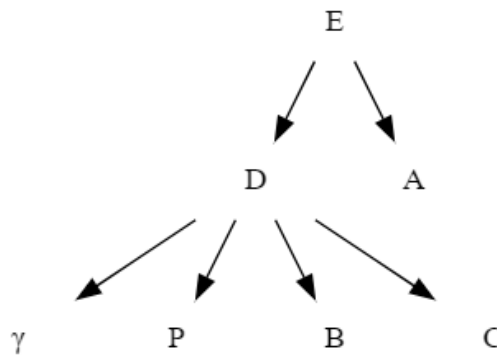


Рисунок 14 – Дерево домінаторів

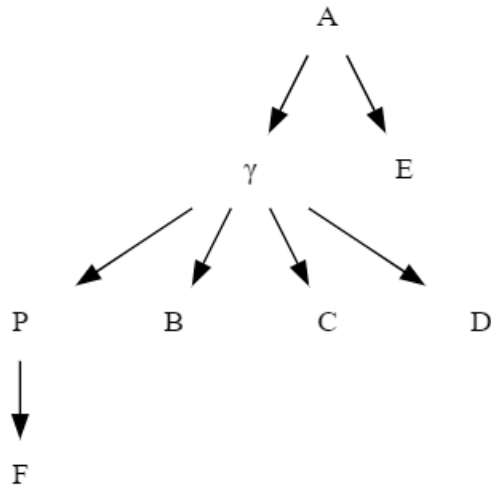


Рисунок 15 – Дерево пост-домінаторів

### 1.5.3 Попередники та спадкоємці

Відношення домінування визначені відносно початкової та кінцевої вершини у графі. В загальному випадку це не завжди потрібно, тому варто визначити також відношення попередника і спадкоємця, що залежать від конфігурації ребер залежності у графі. З цього слідує:

Якщо існує шлях від вершини  $p$  до вершини  $q$ , то  $p$  являється попередником  $q$ .

Відповідно,  $q$  являється спадкоємцем  $p$ . Множини попередників і спадкоємців позначаються відповідно як  $succ$  та  $pred$ . При необхідності можна обмежити, ребра яких множин розглядаються при пошуку шляху. Наприклад, для графу з Рисунка 13, можна знайти множину попередників і спадкоємців по ребрам значень для вершини  $D$ :

$$succ_v(D) = \{B, C, P\}$$

$$pred_v(D) = \{E\}$$

Ці множини можуть бути використані для визначення послідовності обчислень: обчислення спадкоємців повинно відбуватися після того, як значення попередників було вже знайдено.

#### 1.5.4 Нормалізація графу залежності станів та значень

Зазвичай в трансляторах розглядаються лише графи, які можна згорнути [1]. Для графу потоку команд ця умова є необхідною для коректного знаходження меж циклічних конструкцій, а тому і низки подальших методів оптимізації.

Направлений граф можна згорнути, якщо множину ребер можна поділити на дві частини:

1. Ребра, направлені вперед – ребра, що утворюють ациклічний направлений граф, і досягають усі вершини.
2. Зворотні ребра – такі, в яких вершина, куди ребро направлено, домінує над вершиною іншого кінця ребра.

Для забезпечення властивості згортки достатньо заборонити для використання в програмах оператори безумовного переходу *goto*. Використання високорівневих конструкцій управління потоком команд, таких як *if*, *while* гарантує, що граф у результаті буде можна згорнути. Існують декілька пропозицій, що надають можливість граф потоку команд, що не можна згорнути привести до коректного вигляду, додавши до графу потоку команд декілька булевих змінних [9]. Через те, що подібні алгоритми вже існують, можна вважати що на вхід транслятора подається програма у формі без операторів переходу, які формують граф потоку команд, що неможливо згорнути.

Для спрощення обробки графу, можна ще сильніше обмежити умову його коректності, заборонивши будь-які цикли. Для цього необхідно змінити визначення вершини циклу  $\theta$ , розділивши її на дві частини:  $\theta^{head}$  та  $\theta^{tail}$ . Відповідно, порти початкової вершини  $\theta$  переміщуються до нових вершин:  $I, L$  встановлюються в  $\theta^{head}$ , а  $C, R, X$  – до  $\theta^{tail}$ . Така зміна дозволяє гарантовано отримати цикли, в яких  $\theta^{tail}$  домінує усі вершини із своїх вхідних портів, а також усі вихідні порти для  $\theta^{head}$  пост-домінують її. Граф

з використанням ациклічних  $\theta$ -вершин наведених на Рисунку 16, функціонально аналогічний графу, наведеному на Рисунку 12.

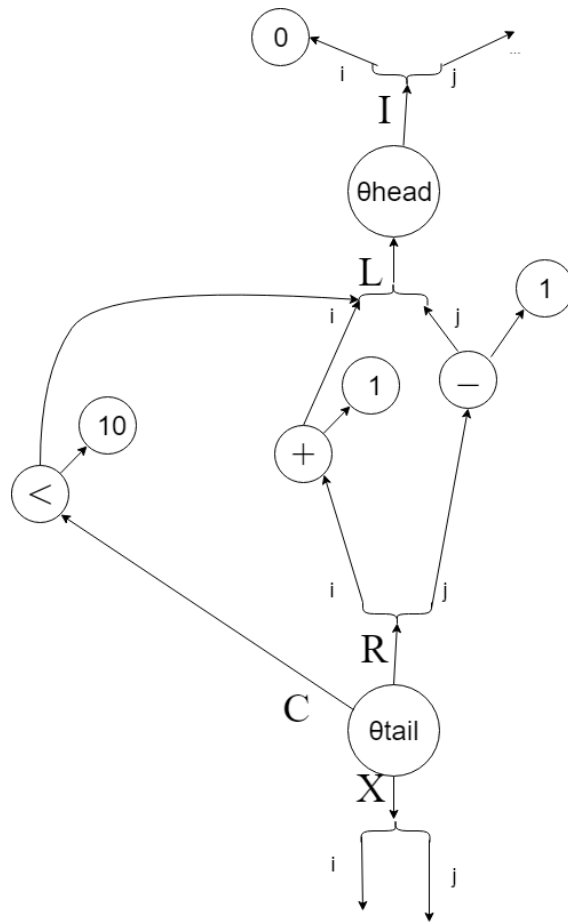


Рисунок 16 – Ациклічне представлення вершини  $\theta$

Відсутність циклів на графі спрощує роботу методів пошуку, а також допомагає однозначно формувати послідовність обчислень при генерації вихідного машинного коду[3].

## 2. РОЗРОБКА МЕТОДІВ ОПТИМІЗАЦІЇ НА ГРАФІ ЗАЛЕЖНОСТЕЙ СТАНІВ ТА ЗНАЧЕНЬ

### 2.1 Видалення мертвого коду у графі залежності станів та значень

#### 2.1.1 Опис запропонованого методу

Вихідні ребра з будь-якої вершини в графі залежності станів і значень показують обчислення, які є необхідними для отримання значення виразу цієї вершини. З цього можна дійти висновку, що вершини графу, які складають спільну компоненту зв'язності із вершиною виходу із процедури, являються необхідними для отримання результату цієї процедури. Розглянемо псевдокод методу видалення мертвого коду для графу залежності станів і значень:

```
function mark(v):  
  if (v була відвідана):  
    return  
  else:  
    помітити v як відвідану  
    neighbors := preds(v) ∪ predv(v)  
    for (v1 in neighbors):  
      mark(v1)  
function deadCodeElimination(graph):  
  mark(вихідна вершина graph)  
  for (v in N):  
    if (v не була відвідана):  
      delete(v)
```

Метод пошуку складається з двох етапів:

1. Визначення компоненти зв'язності з вершиною виходу, тобто виклик функції *mark*, передавши у якості параметра вершину  $N_{\infty}$ .
2. Видалення з графу усіх вершин, які не були відмічені як члени компоненти зв'язності.

Завдяки структурі графу залежності станів і значень, даний метод дозволяє безпечно видалити мертвий код наступних типів:

1. Запис значень до змінних, які не використовуються. Будь-які записи до змінних, що потім не використовуються для отримання

результату, не будуть мати вхідних ребер від вершин виходу.

Наприклад, для наступної функції:

```
int f() {  
    int a = 5;  
    int b = a + 7;  
    return a * 6;  
}
```

Граф залежності станів і значень буде мати вигляд, наведений на Рисунок 17.

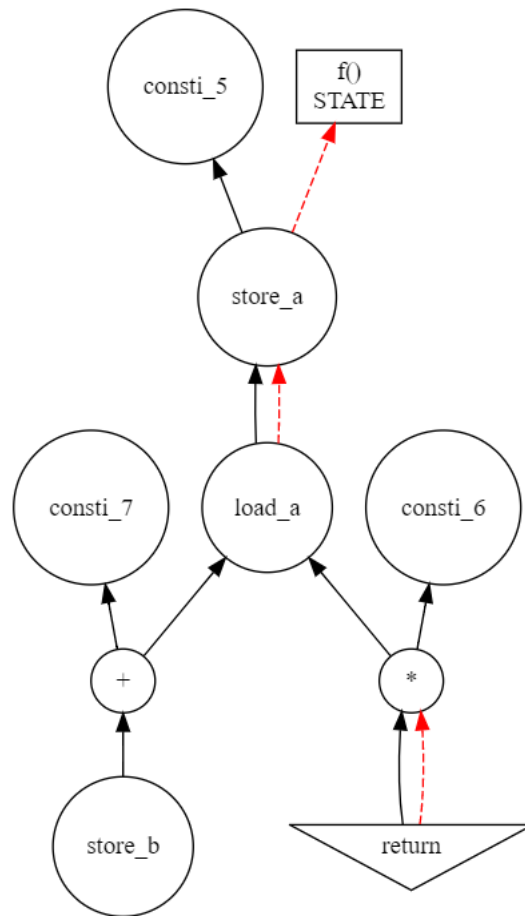


Рисунок 17 – Граф для програми із мертвим кодом.

На графі ребра залежності значень показані червоним кольором, а ребра залежності значень – чорним.

Як видно з графу, оскільки для вершин `store_b`, `+`, `consti_7` не існує вхідних ребер із вершини виходу, вони будуть видалені. З вихідного тексту програми очевидно, що така трансформація буде безпечною і не змінить результату роботи програми.



2. Фрагменти коду, які не можуть бути виконані при будь-яких вхідних параметрах, з тих самих причин, що і для першого пункту.

Порівнюючи з класичними методами, даний метод призведе до тих самих результатів, що і пошук мертвого коду за допомогою досяжних значень та пошуку досяжних конструкцій у графі потоку команд.

Для покращення результатів також необхідна додаткова попередня обробка програми – такі як згортка констант, видалення збиткових умовних конструкцій. Наприклад, для наступної функції тіло циклу не буде позначене як мертвий код:

```
int f() {
    int a = 5;
    while (a == 0) { a++; }
    return a;
}
```

### 2.1.2 Порівняння запропонованого методу із стандартним

Для графу залежності станів і значень алгоритмічна складність методу видалення мертвого коду рівна композиції із складності функцій *mark* і *deadCodeElimination*. Функція *deadCodeElimination* завжди вимагає перелік усіх вершин графу, тому її складність рівна  $O(|N|)$ . Складність функції *mark* залежить від конфігурації графу. В найгіршому випадку вона складає  $O(|k_{max} N|)$ , де  $k_{max}$  – максимальна кількість вихідних ребер для вершини графу. Відповідно, загальна складність методу видалення мертвого коду:

$$O(|N + k_{max} N|) \simeq O(|N|)$$

Найпростіший варіант видалення мертвого графу, який використовує лише граф потоку команд має таку саму складність, адже принципово метод у таких методах не відрізняється. При цьому результати роботи цих двох методів будуть сильно відрізнятися, адже граф залежності станів та значень також надає можливість видалити значення виразів, які можуть бути виконані, але не використовуються далі у програмі. Для пошуку таких конструкцій у графі потоку команд необхідно виконати додаткові обчислення – а саме побудову множин *in*, *out*, *use*, *def*.

Побудова множин  $use, def$  – вимагає одного повного переліку усіх вершин графу потоку команд, тобто складність цього етапу складає  $O(|N|)$ . Побудова множин  $in, out$  залежить від вхідної програми. В середньому випадку складність цього етапу складає  $O(|N|^2)$ , але в найгіршому випадку кількість необхідних ітерацій методу може сильно зрости, погіршивши алгоритмічну складність методу до  $O(|N|^4)$ [6].

З точки зору просторової складності, видалення мертвого коду за допомогою VSDG потребує в найгіршому випадку  $O(|N|)$  додаткової пам'яті на збереження множини вершин, помічених для збереження.

Видалення мертвого коду за допомогою CFG потребує збереження окремо чотирьох множин для кожної вершини у графі. Розмір множини  $def$  не може перевищувати 1 елемент, оскільки для виділеної підмножини мови C заборонено використовувати більше одного запису до змінної на один вираз. Розмір множини  $use$  прямо залежить від складності виразу, якому відповідає вершина, при цьому не може перевищувати кількість вершин-попередників. В середньому для вершини це означає, що її розмір лінійно залежить від розміру графу. Множини  $in, out$  залежать від кількості мертвого коду, що знаходиться у вхідній програмі, але обидві множини не можуть бути менші за розміром, ніж сумарна потужність множини  $use \cup def$ .

В результаті, просторова складність методу видалення мертвого коду за допомогою графу потоку команд складає:

$$\begin{aligned} O(N * (|use| + |def| + |in| + |out|)) &\simeq O(N * (N + 1 + N + N)) = \\ &= O(3N^2 + N) \simeq O(N^2) \end{aligned}$$

Варто зазначити, що видалення мертвого коду у графі потоку команд також може потребувати додаткових обчислень для підтримання

додаткових структур даних у актуальному стані – наприклад, може стати потрібною перебудова графу потоку даних. Для VSDG такої необхідності не існує, оскільки для більшості методів оптимізації достатньо використовувати інформацію, яка вже присутня у графі. Складність подібних операцій залежить від конкретної реалізації транслятора, і тому не буде враховуватися в даному випадку.

Отже, використання методу видалення мертвого коду за допомогою графу залежності станів та значень дозволяє скоротити часову складність алгоритму в середньому випадку з квадратичної до лінійної. В найгірших випадках для обох методів часова складність зменшується із показникової функції 4 степеню до лінійної з константним множником.

Також запропонований метод дозволяє зменшити просторову складність з квадратичної до лінійної.

## 2.2 Видалення спільних виразів

### 2.2.1 Опис запропонованого методу

Традиційно для вирішення проблеми видалення спільних виразів використовують глобальну нумерацію виразів у графі потоку команд. Складність такого підходу в тому, що в графі потоку команд вершина відповідає за цілий вираз, а не за окрему операцію. Через це перетворення графу потоку команд становиться досить складним, оскільки при виділенні виразів у окремі команди необхідно підтримати граф потоку команд у коректній формі, а також ця оптимізація потребує повторного обрахунку графу потоку даних. При використанні графу залежності станів і значень будь-які вирази відображені вершинами і ребрами графу, а тому для визначення спільних виразів достатньо знайти однакові підграфи. Спільними виразами можна назвати вершини  $p$  та  $q$ , якщо вони виконують одну і ту саму операцію, при цьому  $pred_v(p) \cup pred_s(p) = pred_v(q) \cup pred_s(p)$ .

Метод оптимізації у псевдокоді має наступний вигляд:

```
N_current = N
foreach(n in N_current)
  N_current := N_current - n;
  if (n не має мітку)
    M_current = N_current
    foreach(m in M_current)
      M_current := M_current - m
      if(n має мітку && m має мітку &&
        op(n) == op(m) && pred(n) == pred(m))
        перемістити усі вхідні ребра від m до n;
        поставити мітку на m
  foreach(n in N)
    if (n має мітку)
      видалити n
```

Метод складається з двох етапів: спочатку відбувається пошук спільних виразів за критеріями, описаними вище. Для вершин, які вважаються спільними, ребра залежності переміщуються у одну з них. На іншій вершині при цьому ставиться мітка для подальшого видалення. Другий етап методу видаляє вершини, які були позначені як спільні вирази.

Передбачається, що даний метод необхідно викликати більше одного разу. Оскільки граф залежності не містить циклів, то завжди для роботи методу можна досягти фіксованої точки – коли не буде знайдено жодного спільного виразу.

У якості прикладу розглянемо оптимізацію наступної функції:

```
int f() {
  int a = 1 + 2 + 3;
  int b = 2 + 3;
  int c = a + b;
  return c;
}
```

Початковий граф залежності станів та значень, а також результати виконання пошуку спільних виразів наведені на Рисунках 18-20.

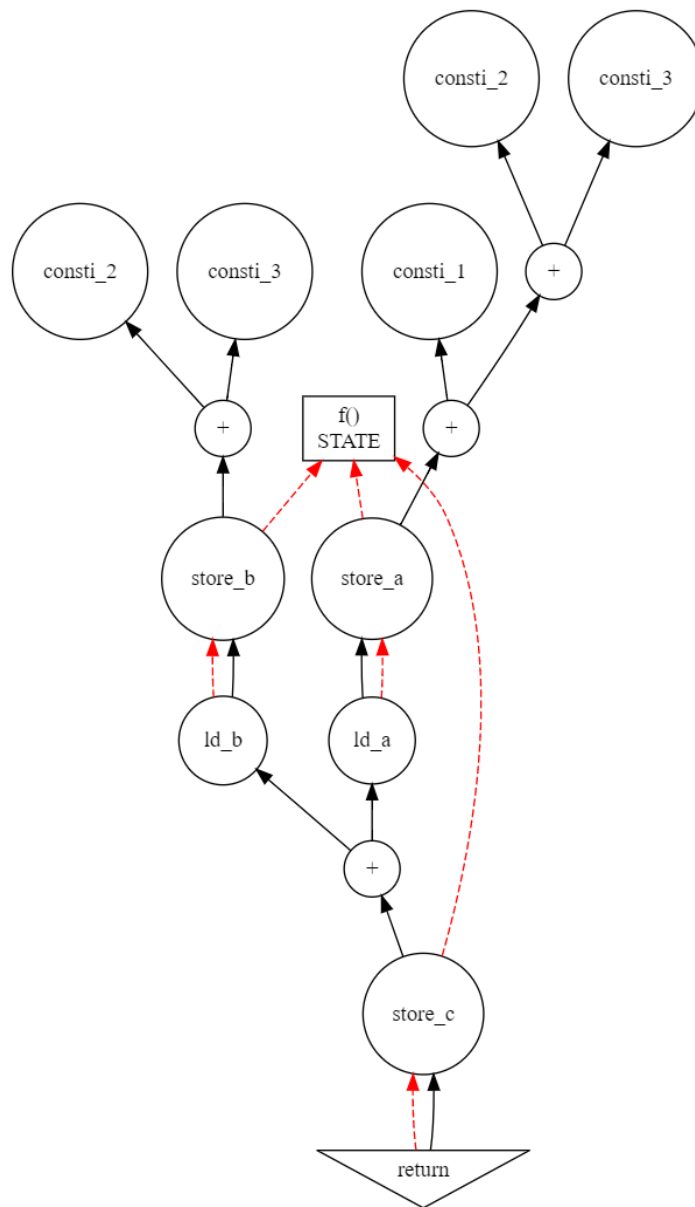


Рисунок 18 – Початковий стан графу при оптимізації згортки констант

Виконання однієї ітерації методу згортки констант призведе до результату, наведеного на Рисунку 19:

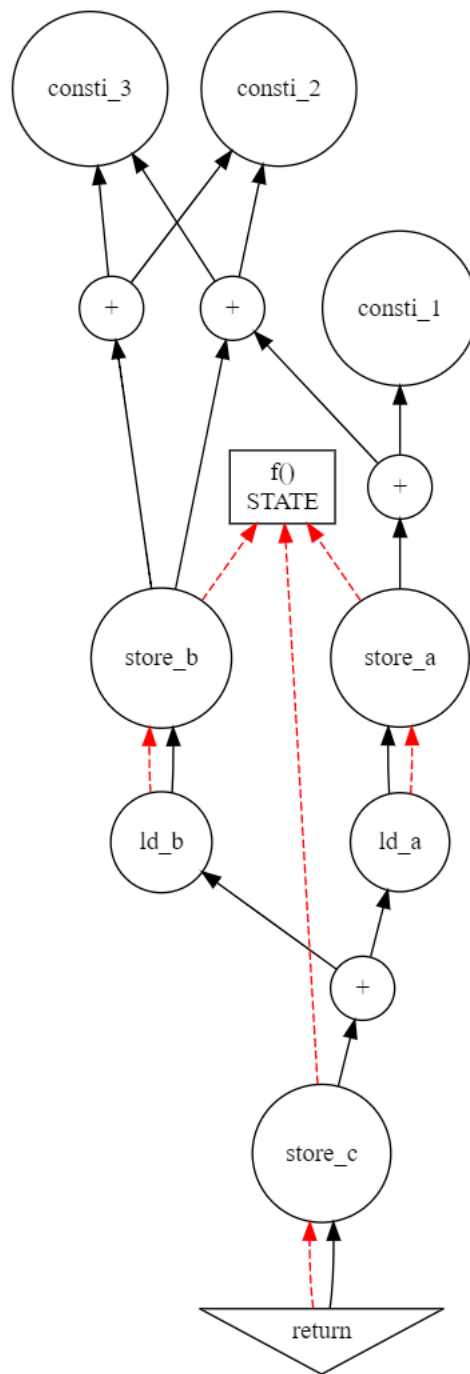


Рисунок 19 – Граф після одного обходу методу виділення спільних виразів

Як можна помітити, перший обхід методу згортки констант лише видалив вершини констант, що дублюються у графі. На даному етапі такий результат ще не представляє користі для транслятору, але при повторному обході графу вже будуть знайдені і видалені деякі арифметичні вирази. Результат другого обходу наведений на Рисунку 20.



### 2.2.2 Порівняння запропонованого методу із стандартним

Принципово пошук і видалення спільних виразів за допомогою графу потоку команд і графу залежності станів і значень відрізняється тим, що вирази вже наявні у самому внутрішньому поданні програми, і немає потреби створення окремої форми подання для виразів.

При використанні графу потоку команд зазвичай вершині відповідає синтаксичне дерево розбору конкретного виразу з вхідної програми. Першим етапом пошуку спільних виразів стає глобальна нумерація виразів [7]. Часова складність цього етапу -  $O(expr^3 * join\_points)$ , де  $expr$  – загальна кількість виразів у графі потоку команд, а  $join\_points$  – кількість вершин, які мають більше одного вхідного ребра.

Просторова складність етапу глобальної нумерації виразів складає  $O(expr^2)$ , оскільки метод рекурсивний і передбачає перелік відповідних виразів між двома вершинами.

Після глобальної нумерації виразів достатньо перевірити, яким індексам виразів відповідає більше одного значення, і обчислити для таких вершин їх множини  $in, out$  за правилами, зазначеними в розділі 1.5. Незважаючи на те, що формули для розрахунку цих множин відрізняються від тих, що використовуються для множин  $in, out$  методу пошуку мертвого коду, обчислювальна складність залишається тою самою, оскільки уся різниця між формулами полягає у заміні операції об'єднання множин операцією перетину.

З цього слідує, що складність обчислення цих множин становить  $O(|N|^2)$  в середньому випадку, і  $O(|N|^4)$  в гіршому.

Для графу залежності станів та значень подібний підхід не необхідний. Часова складність запропонованого методу складає  $O(iter * |N|^2)$ , де  $iter$  – кількість ітерацій, необхідних для досягнення фіксованої точки. Кількість ітерації напряму залежить від максимальної складності виразів у програмі.



Найгіршим випадком для роботи даного методу буде програма, що структурно відповідає намальованій на Рисунку 21.

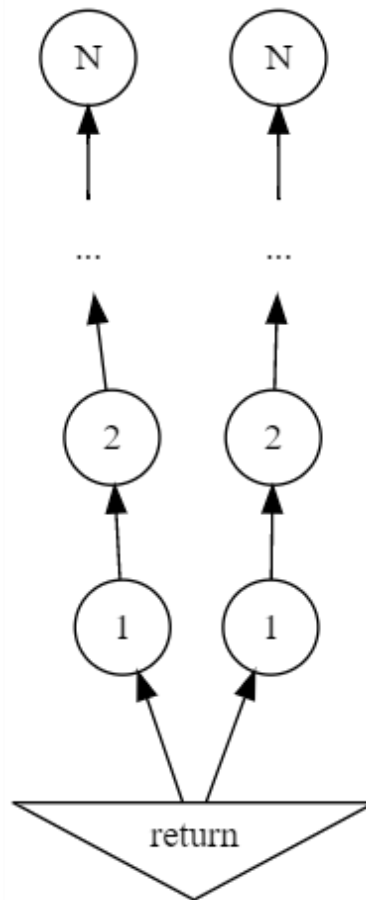


Рисунок 21 – Найгірший випадок для методу пошуку спільних виразів

Для програми в такій формі за кожну ітерацію буде видалено лише по одній вершині. Зважаючи, що загальна кількість вершин у повному графі залежності станів і значень в такому випадку рівна  $N + 2$ , для завершення роботи методу необхідно  $N$  ітерацій, тобто складність методу в найгіршому випадку становить  $O(|N|^3)$ .

Просторова складність запропонованого методу становить  $O(|N|)$ , оскільки для завершення роботи методу необхідно провести видалення мертвого коду. Сам метод пошуку не вимагає додаткових витрат пам'яті системи.

Таким чином, запропонований метод дозволяє суттєво знизити просторову складність виконання оптимізації, а також знизити витрати на пошук спільних виразів у найгіршому випадку. Також, найкращий випадок

в традиційному методі потребує  $O(expr^3 + |N|^2)$ , а в запропонованому методі лише  $O(|N|^2)$ .

## 2.3 Згортка констант

### 2.3.1 Опис запропонованого методу

Завдяки властивості графу залежності стану і значень зберігати виразі у самому графі, а не як структуру, яка зберігається у вершинах (як в графу потоку команд), згортка констант потребують обробки лише сутностей, які в ньому вже містяться.

Згортка констант в будь-якій формі залежить лише від множини правил, які були додані в транслятор. Найпростішими для реалізації є згортка булевих виразів, оскільки множина значень сильно обмежена, а також зазвичай у сучасних мовах програмування множина операцій над булевими значеннями обмежена чотирма операціями: диз'юнкція, кон'юнкція, заперечення та виключна диз'юнкція. Відповідно, для кожної з цих операцій можна підібрати набір правил, що дозволять зменшити кількість обчислень:

1.  $true \vee expr \equiv true$
2.  $false \wedge expr \equiv false$
3.  $expr \oplus expr \equiv false$

У заданих правилах  $expr$  означає, що на даному вхідному порті вершини може бути будь-яке значення. Аналогічно можна визначити деякі правила для арифметичних виразів:

1.  $0 * expr \equiv 0$
2.  $0 + expr \equiv expr$
3.  $const\_a + const\_b \equiv const(a + b)$

Сам метод згортки констант складається з пошуку підграфів, еквівалентних заданому правилу. Можна відмітити, що такий підхід нагадує процес видалення спільних виразів із графу, але основною відмінністю є те, що порівнюються не окремі вершини з множинами попередників, а цілі підграфи. Для графу залежності, наведеному як приклад у минулому розділі,

достатньо використати правило 3 для цілочисельних змінних, наведених вище. Результатом роботи буде граф залежності станів і значень, наведений на Рисунку 22.

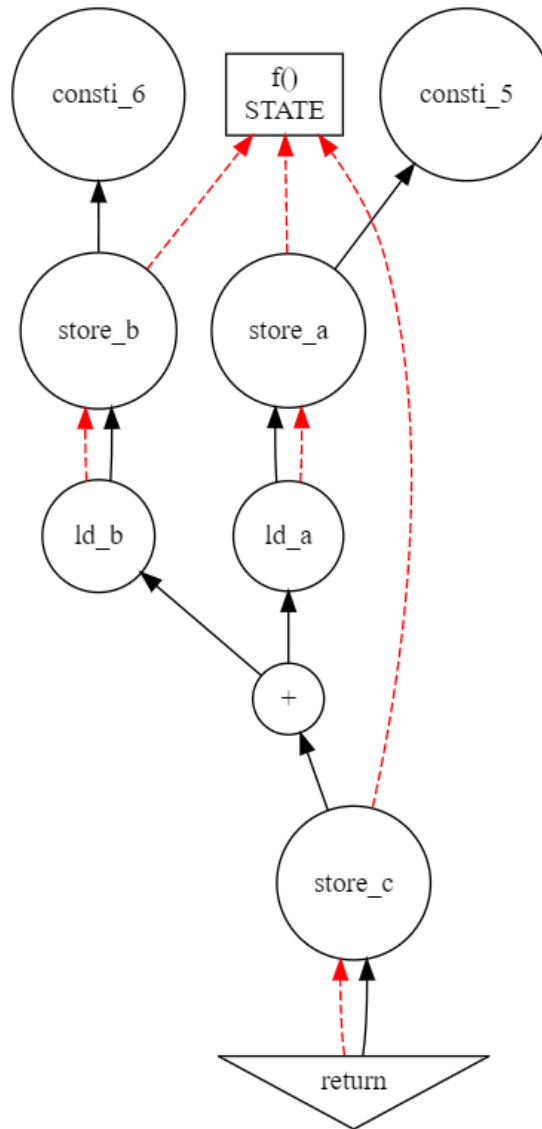


Рисунок 22 – Граф залежності станів і значень із згорнутими константами

З Рисунку 22 видно, що значення, які будуть записані у змінні  $a$ ,  $b$  будуть не результатом математичних обчислень, а звичайними константами. Виконуючи подібні трансформації варто зважати увагу на наступні фактори:

1. Значення, що підставляються під час згортки арифметичних виразів, повинні бути однаковими з результатом такого самого обчислення на цільовій архітектурі процесора, на якому буде виконуватися

програма. Для цілих значень це означає, що необхідно відповідати обмеженням розміру констант і запобігання переповнення. Для обчислення значень з рухомою комою необхідно також впевнитися у рівності довжин мантиси та показника степені.

2. Необхідно уникати видалення виразів, які впливають на загальний стан програми. Це означає, що при видаленні ребра залежності значення, ребра залежності стану необхідно зберігати, і оброблювати окремо.
3. Такі операції, що впливають на стан програми, як завантаження у пам'ять і збереження до пам'яті, можна спрощувати, додаючи додаткові правила до методу пошуку констант. Наприклад, для попереднього графу можна видалити послідовне збереження і завантаження змінних *a*, *b*, оскільки це локальні змінні, використовуються лише один раз, та не існує додаткових змін у додаткових адресах у пам'яті. Таким чином, вхідна програма може бути скорочена лише до одного виразу *return 11*;

### 2.3.2 Порівняння запропонованого методу із стандартним

Різниця між традиційним методом згортки констант та запропонованим збігається з методами пошуку спільних виразів – в графі потоку команд необхідно створювати окремий механізм для обробки частин виразів, які відповідають вершинам графу, в той час як у графі залежності станів та значень такої потреби немає.

В цілому, обидва методи зводяться в результаті до пошуку певних шаблонних виразів у проміжному представленні програми. Набір шаблонів може відрізнитися в залежності від критерію оптимізації програми – наприклад, якщо критерієм оптимізації є зменшення розміру програми, не варто використовувати наступне правило:

$$pow(x, 5) \equiv x * x * x * x * x$$

При цьому, обчислення лівої частини виразу може бути суттєво довшим, ніж правої – операція зведення дійсного числа до дійсного степеня значно дорожча у порівнянні із множенням на процесорах загального призначення.

Для деяких правил під час згортки констант необхідно також зберігати початкову послідовність обчислень. В графі потоку команд безпечність операцій над графом гарантується, якщо він приведений до SSA-форми[8].

Для графу залежності станів і значень немає аналогу SSA-форми, оскільки на правильно сформованому VSDG залежності до змінних будуть залежати від відповідних завантажень з пам'яті. Наприклад, для наступного фрагменту програми:

```
int f(){  
    int x = 5;  
    int b = x;  
    x = 6;  
    return x + b;  
}
```

Граф залежності станів і значень наведений на Рисунку 23.

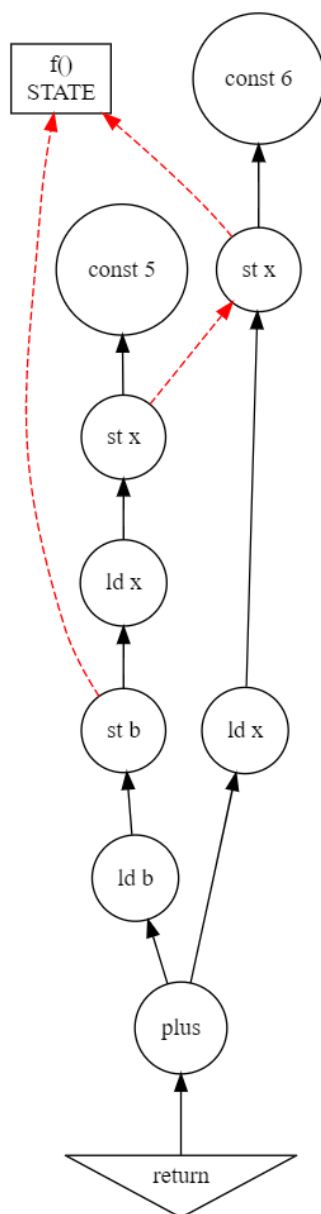


Рисунок 23 – Граф залежності станів та значень для програми з декількома записами до однієї змінної.

З Рисунок 23 видно, що залежності значень вершин *ld* направлені на відповідні записи до адрес в пам'яті, тому SSA форма не потрібна для такого проміжного представлення.

Із вказаного вище видно, що алгоритмічна складність двох методів не буде відрізнятися, якщо не враховувати процес зведення графу потоку команд до SSA-форми. Існує декілька підходів до перетворення графу до SSA-форми, один із традиційних [9] має обчислювальну складність  $O(R^3)$  в найгіршому випадку, і  $O(R)$  в середньому, при цьому  $O = \max(N, E)$ .

Пошук підграфів за шаблонами залежить від їх складності, але у загальному випадку пошук підграфу у графі – NP-повна задача[4]. Завдяки тому, що вирази у графі потоку даних та граф залежності станів та значень – направлені графи, що не мають циклів, можна застосувати метод, запропонований у [10], складність якого становить  $O(n^4)$ .

Методи згортки констант на графі залежності станів і значень у порівнянні із традиційним методом, що використовує граф потоку команд, мають однакову загальну характеристику  $O$ , оскільки  $O(n^4 + R^3) \simeq O(n^4)$ , при цьому використання запропонованого методу дозволяє зменшити складність обчислень на величину  $O(R^3)$ .

## 2.4 Визначення інваріантів для циклів

### 2.4.1 Опис запропонованого методу

Для графу потоку команд пошук інваріантів циклічних конструкцій починався із пошуку меж циклічних конструкцій. У графі залежності станів і значень ця проблема відсутня завдяки створенням окремого виду вершин для циклічних конструкцій. Відповідно, можна одразу перейти до другого етапу методу: пошуку значень, що використовуються у циклі, значення яких не залежить від процесу його виконання у програмі. Обчисливши множини пост-домінуючих вершин, можна сказати, що вершина  $n$  є інваріантом циклу  $\theta_i$ , якщо  $\theta_i^{tail}$  пост-домінує  $n$ , і при цьому  $n \notin succ(\theta_i^{head})$ .

Розглянемо методи побудови вказаних множин.

Відношення пост-домінування передбачає пошук усіх можливих шляхів між двома вершинами у графі. Оскільки нормалізований граф залежності станів і значень не містить циклів, то можна використати модифікацію пошуку в ширину[4]. Зазвичай пошук шляху у графі завершується при знаходженні першого шляху – він же й буде найкоротшим. Для доведення, що одна вершина графу пост-домінує над іншою, необхідно побудувати усі шляхи, тому умова завершення роботи алгоритму повинна бути змінена відповідно.

Псевдокод методу пошуку усіх шляхів буде мати наступний вигляд:

```
function FindAllPaths(graph, vFrom, vTo):  
  s := пустий стек  
  s.push(vFrom)  
  while s не пустий:  
    v = s.pop()  
    if v не була відвідана:  
      запам'ятати v як відвідану  
      if v = vTo:  
        зберегти отриманий шлях  
      else:  
        neighbors := succ(v)  
        for n in neighbors:  
          s.push(n)
```

Для знаходження множин спадкоємців достатньо використати звичайний пошук у глибину[4]. Варто зазначити, що методи побудови графу залежності станів та значень зазвичай містять етап побудови графу домінаторів, пост-домінаторів, а також спадкоємців і попередників[9]. В такому випадку алгоритмічна складність операції встановлення інваріантності вершини циклу  $O(1)$ . Складність методу, вказаного вище, складає  $O(\text{FindAllPaths}(p, q) + \text{FindPath}(p, q)) = O(|N| + |E|)$ .

Розглянемо роботу методу на прикладі оптимізації наступної функції:

```
int f() {  
  int a = 100;  
  double res = 0;  
  while(a > 0) {  
    double b = sin(PI);  
    res = a * b;  
    a--;  
  }  
  return res;  
}
```

Граф залежності для такої функції буде мати вигляд, наведений на Рисунку 24. З рисунку видно, що обчислення значення змінної  $b$  є інваріантом для циклу, в той час як обчислення змінної  $a$  - ні, оскільки існує шлях від вершини  $decr\_a$  до вершини  $\theta^{head}$ , тобто вона є її попередником, що порушує другу умову.



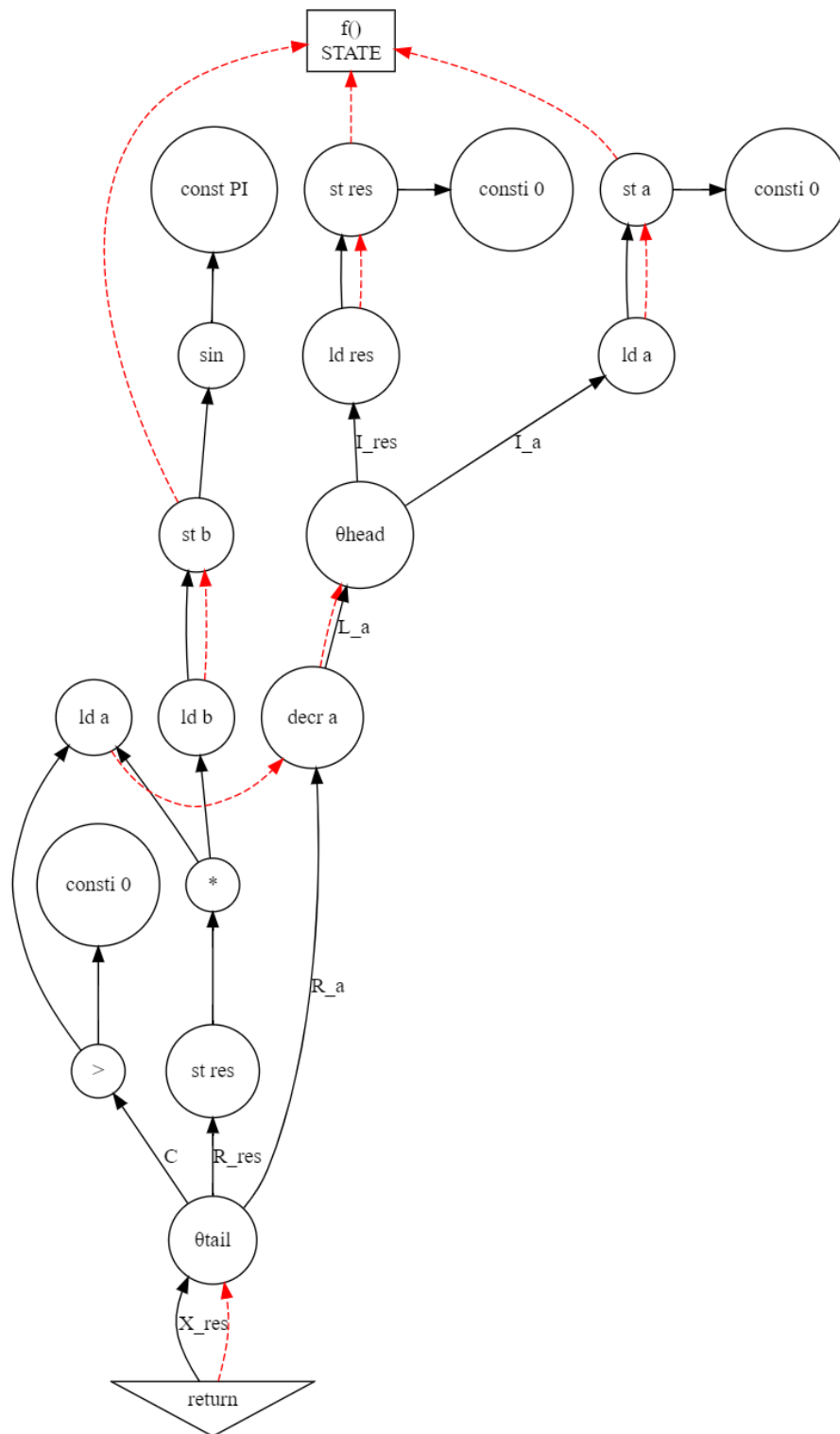


Рисунок 24 – Граф для функції з циклом і інваріантним значенням у тілі циклу

Після знаходження інваріантних конструкцій достатньо винести їх за межі циклу. Для графу залежності станів та значень це означає переміщення ребер залежності із середини циклу до вхідних портів вершини  $\theta^{head}$  – в

даному випадку це означає створення додаткового порту  $I_{res}$ , в якому ребро залежності буде направлено до вершини  $st\ b$ .

#### 2.4.2 Порівняння запропонованого методу із стандартним

Одною із значних переваг графу залежності станів і значень у порівнянні із графом потоку команд є введення додаткового типу вершин для циклічних конструкцій. Завдяки цьому методи оптимізації у трансляторах, які передбачаються для використання в циклах, можуть бути суттєво спрощені. Традиційно задача пошуку інваріантів у циклах передбачає спочатку пошук відповідних заголовків і хвостів циклу – задача, типове рішення якої має обчислювальну складність  $O(|E||N|^2)$  [4]. Другий етап роботи методу складається з пошуку інваріантів циклічних конструкцій, складність якого залежить від складності обчислення множин досяжних визначень у графі потоку даних. Типовий метод вирішення цієї проблеми використовує множини  $in, out$ , аналогічні до тих, що використовувалися при видаленні мертвого коду. Як було визначено в розділі 3.2.1, складність обчислення цих множин складає  $O(|N|^2)$  в середньому випадку. Оскільки сам пошук інваріантів для циклу має лінійну асимптотику, то загалом традиційний метод має наступну обчислювальну складність:

$$O(|E||N|^2 + |N|^2 + |N|) = O(|E||N|^2)$$

Оскільки граф потоку даних містить інформацію про циклічні конструкції, перший і другий етапи пропускаються. Обчислювальна складність запропонованого методу складається із композиції складності побудови дерева домінації, пост-домінації, та пошуку вершин, інваріантних для циклів. Останній етап, аналогічно до класичного пошуку у графі потоку команд, лінійний за обчислювальною складністю. Побудова дерева домінації і пост-домінації має складність  $O(|N|^2)$ , якщо використовувати алгоритм, запропонований в [11].

Це означає, що при використанні графу залежності станів та значень асимптотика оптимізації пошуку інваріантів циклічних конструкцій зменшується у  $|E|$  кількість разів.

Також варто зазначити, що процес винесення обчислення за межі циклу складається лише з переміщення ребер залежності для однієї з вершин у графі. Для традиційного методу з графом потоку команд можливо буде необхідним повторне обчислення допоміжних структур даних – графу потоку даних, множин *in*, *out*, тощо.

## 2.5 Згортка умовних конструкцій

### 2.5.1 Опис запропонованого методу

Структура графу залежності та значень передбачає існування двох різних конструкцій, що мають семантику умовного переходу, тому має сенс застосування двох різних методів для згортки кожного типу вершин.

#### 2.5.1.1 Згортка $\gamma$ -вершин

Згортка констант на графі залежності станів та значень може призвести до створення конструкцій, наведених на Рисунку 25.

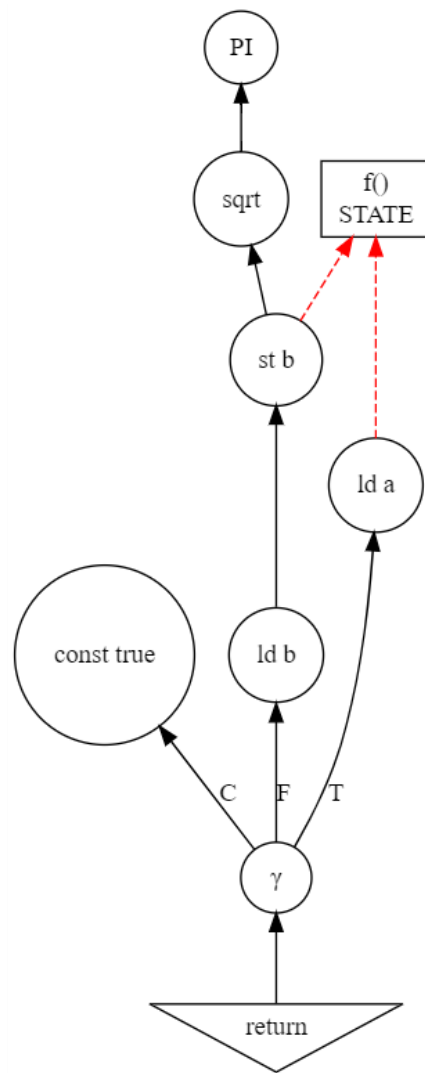


Рисунок 25 – Граф залежності для функції із збитковою вершиною  $\gamma$

Для програми, представленою на графі з Рисунку 24, значення на вхідному порті С вершини  $\gamma$  завжди істинне. Це означає, що не змінюючи роботи програми, можна видалити з графу саму вершину умовного переходу, та зв'язати усі вхідні ребра вершини  $\gamma$  із вершиною, куди був направлений її порт Т. В результаті для наведеного вище графу буде отримана програма, зображена на Рисунку 26.

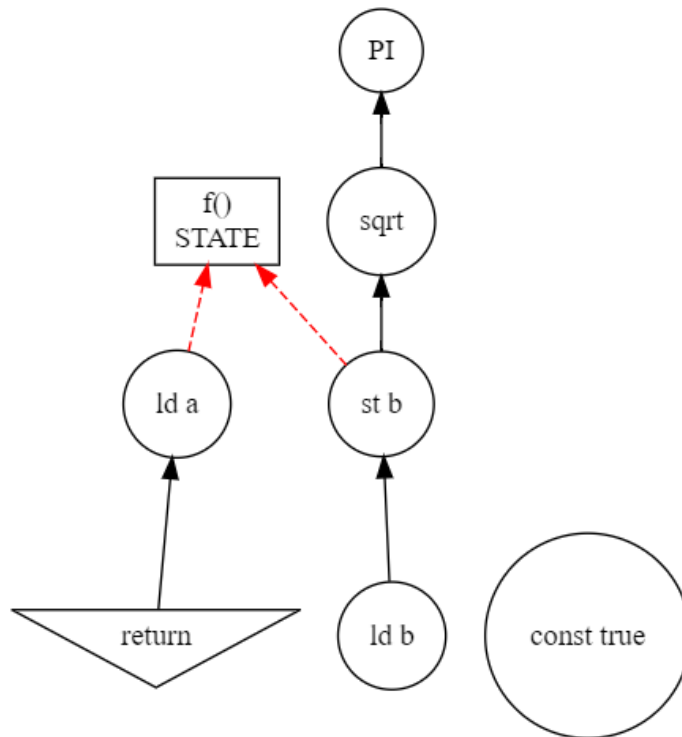


Рисунок 26 – Граф для програми після видалення умовної вершини

Після видалення циклічних вершин зазвичай залишаються побічні надлишкові вершини, які не мають залежності від вершини виходу з програми. В такому випадку достатньо один раз використати метод видалення мертвого коду. В результаті граф залежності станів та значень буде мати тривіальну форму, наведену на Рисунку 27.

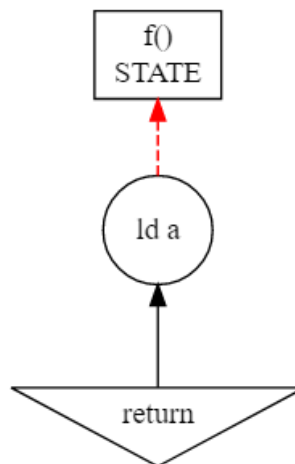


Рисунок 27 – Граф для програми після завершення методу згортки  $\gamma$ -вершин

Найбільший вплив згортка  $\gamma$ -вершин має на розмір згенерованого транслятором коду програми. У деяких випадках (наприклад, у

розглянутому вище) подібні зміни до програми можуть призвести до каскадного видалення мертвого коду. В такому випадку необхідно звертати увагу на те, що ребра залежності стану необхідно зберігати. Для розглянутого прикладу відомо, що виклик зовнішньої функції *sqrt* не впливає на загальний стан програми, а тому його можна вважати мертвим.

Варто зазначити, що методи оптимізації згортки умовних вершин функціонально аналогічні оптимізації згортки гілок для графу потоку команд[5], оскільки в результаті виконання обох методів оптимізації умовні конструкції замінюються безумовним переходом до певної мітки в коді.

### 2.5.1.2 Згортка $\theta$ -вершин

Основна різниця між звичайним оператором умовного переходу та циклічною конструкцією в процесі згортки вершин є те, що можливо видалити лише цикли, для яких виконується наступна умова:

$$\theta_c = false \quad (10)$$

Тобто, на графі залежності станів та значень можливо видалити лише цикли, які не будуть виконуватися – приклад такої програми наведений на Рисунку 26. Нескінчені цикли, які не є мертвим кодом, залишаться в програмі з таким підходом. Таким чином забезпечується збереження результату роботи програми.

Видалення циклічних конструкцій з графу залежності та значень складається з декількох етапів:

1. Усі ребра, направлені до портів  $X_i$  вершини  $\theta^{tail}$  перенаправляються до відповідних виходів  $I_i$  вершини  $\theta^{head}$ .
2. Видалення усіх вершин, для яких виконується умова (11):

$$v \text{ dom } \theta^{head} \wedge v \text{ pdom } \theta^{tail} \quad (11)$$

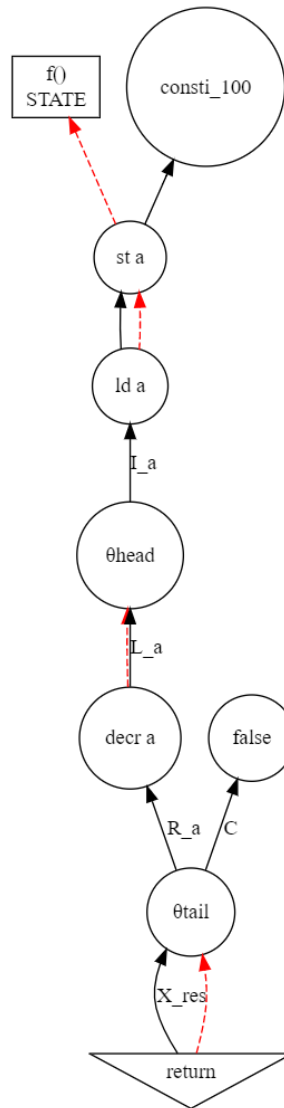


Рисунок 28 – Граф залежності станів і значень для програми, в якій можливо згорнути циклічну вершину

Граф на Рисунку 28 приблизно відповідає наступній програмі:

```

int f() {
    int a = 100;
    while (false) {
        a--;
    }
    return a;
}

```

Після застосування методу згортки  $\theta$ -вершин програма буде мати вигляд, наведений на Рисунку 29.

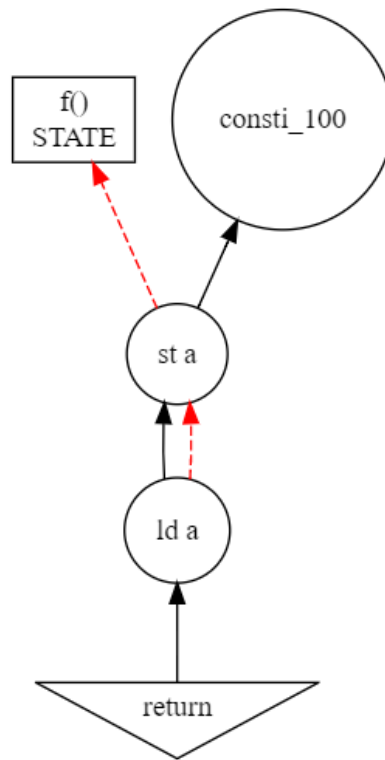


Рисунок 29 – Граф залежності станів і значень після застосування методу згортки  $\theta$ -вершин

У розглянутому випадку видалення циклічних вершин дозволить скоротити розмір генерованої програми, а також зменшити час виконання програми, оскільки не будуть виконані зайві операції порівняння та переходу.

### 2.5.2 Порівняння запропонованих методів із стандартним

Методи згортки умовних конструкцій на графі потоку команд та на графі залежності станів та значень мають суттєві відмінності, якщо порівнювати програми, отримані на виході трансляторів.

Джерелом різниці є відмінність у представленні умовних конструкцій в цих графах: граф потоку команд має лише один тип вершини з умовним переходом, в той час як граф залежності станів та значень визначає два типи вершин, що можуть подавати до вихідних портів різні значення в залежності від вхідних умов – вершини  $\gamma$  та  $\theta$ . Широкі можливості, які відкривають ці вершини для інших методів оптимізації (таких, як згортка констант)



накладають обмеження на можливість видалення самих умовних конструкцій.

Основною задачею в даному випадку є відсутність можливості відобразити оператор безумовного переходу на графі залежності станів і значень, що означає, що для деяких конструкцій неможливо провести оптимізацію, як у графі потоку команд. Приклад програми у формі графу потоку команд, який неможливо оптимізувати за допомогою графу залежності станів та значень, наведений на Рисунку 30. Граф для програми, отриманої в результаті оптимізації за допомогою графу потоку команд, також наведений на Рисунку 30.

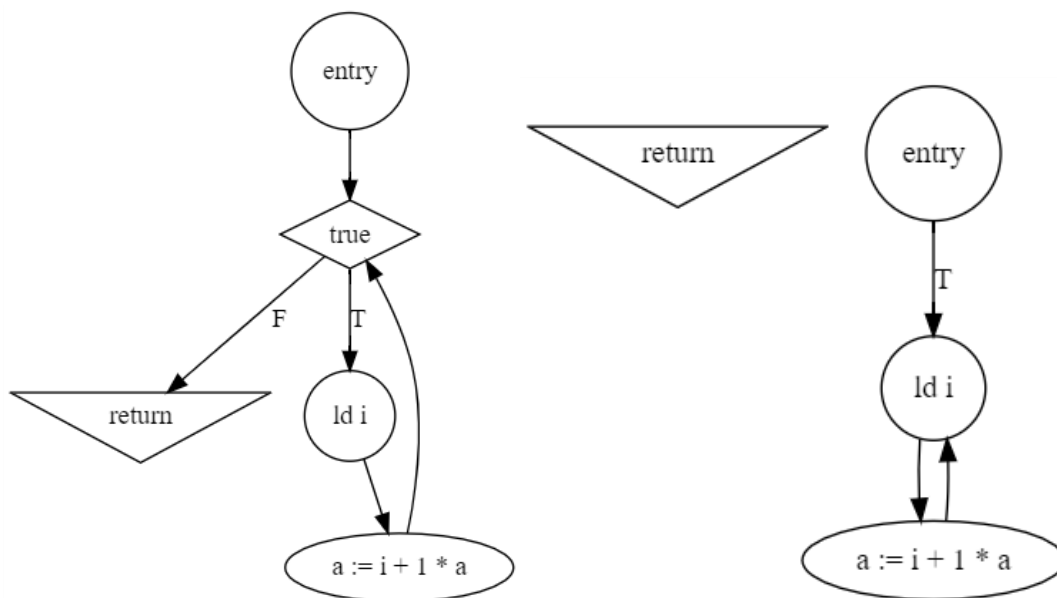


Рисунок 30 – Граф потоку команд до і після оптимізації згортки умовних конструкцій

Загалом, описаний випадок оптимізації дозволяє зменшити час лише на одне порівняння, що є однією конструкцією в асемблері, а це означає, що такі випадки можна ігнорувати, оскільки загальний вплив на швидкодію згенерованої програми малий. Одним із методів вирішення цієї проблеми може бути створення додаткового правила під час генерації вихідного програмного коду. Також необхідно зазначити, що цикли без умови виходу не представляють інтерес для користувачів мови програмування, що не має

інструкцій безумовного завершення циклу, тобто можна із впевненістю дійти висновку, що така програма була написана помилково.

За винятком зазначених вище відмінностей, характеристики двох методів однакові, оскільки вони потребують лише видалення вершини і перенаправлення декількох ребер у вхідному графі, тобто обчислювальна складність в середньому випадку становить  $O(|N|)$ . При цьому, метод, що використовує граф потоку команд, буде потребувати додаткові витрати на підтримку допоміжних структур даних (граф потоку даних, множини *in*, *out*, тощо) в актуальному стані.

### 3. РЕАЛІЗАЦІЯ МЕТОДІВ ТА АНАЛІЗ ОТРИМАНИХ РЕЗУЛЬТАТІВ

#### 3.1 Розробка оптимізуючих трансляторів для порівняння

Розробка будь-якого транслятора невід’ємно пов’язана із мовами програмування. Для методів оптимізації, які були згадані в дисертації, мова програмування повинна відповідати наступним вимогам:

1. В мові програмування повинна бути можливість використовувати типові високорівневі конструкції керування потоку команд: умова, цикли.
2. В мові програмування повинні бути присутні засоби описання процедур та функцій.

Для наочності було обрано підмножину мови C, яка містить лише вказані вище можливості. Граматика мови знаходиться у Додатку 2, але варто привести короткий перелік наявних синтаксичних конструкцій:

1. Оголошення функцій:

```
int func () {}
```

2. Оголошення локальних змінних примітивних типів для функції:

```
int x = 0;
```

3. Арифметичні та булеві операції: **&&, ||, +, -, \*, /**.

4. Конструкції керування потоком команд: **if...else, while() {}, for(;;) {}**.

5. Оператор повернення значення з функції:

```
return x.
```

6. Примітивні типи даних: **int, bool**.

7. Виклик сторонніх функцій, які можуть мати чи не мати побічні ефекти:

```
random(x);  
sin(x);
```

В ході роботи було розроблено програмний комплекс, який надає наступні можливості:

1. Лексичний аналіз вхідної програми та побудова синтаксичного дерева.

2. Побудова графу потоку команд на основі синтаксичного дерева.
3. Побудова графу залежності станів та значень для вхідної програми.
4. Оптимізація вхідної програми за допомогою графу залежності станів та значень.
5. Оптимізація вхідної програми за допомогою графу потоку команд та інших допоміжних структур даних.
6. Вимір основних характеристик програмного комплексу під час його роботи: витрачений час, максимальний об'єм використовуваної оперативної пам'яті.

Для реалізації програмного комплексу, що надає можливості для порівняльного аналізу класичних методів оптимізації у трансляторах з методами на графі залежності станів та значень, було обрано мову Common Lisp версії SBCL.

LISP – сімейство мов програмування загального призначення з довгою історією. Перша специфікація мови була розроблена у 1958 році в Массачусетському Технологічному Інституті для дослідження проблем, пов'язаних із штучним інтелектом. Завдяки відмінностям, які виділяли цю мову від її конкурентів, мова набула досить широкого використання не тільки у галузі дослідження штучного інтелекту, а й інших наукових напрямках, серед яких виділяються галузі обробки природніх мов (Natural language processing – NLP) та мов програмування.

Основна ідея мов програмування сімейства LISP – це уніфікація процесу роботи з даними та програмним кодом. Будь-яка програма з синтаксичної точки зору є списком із значень або виразів. Виконання програми – лише обчислення виразів для отримання значень, які передаються у інші вирази як елементи списків.

Наслідком такого підходу є те, що програма в процесі виконання здатна створювати нові команди, які їй необхідно виконати далі. Подібні маніпуляції над вхідною програмою зазвичай виконуються за допомогою достатньо простої, але потужної системи макросів. У традиційних мовах

програмування підтримка системи макросів зазвичай передбачає лише текстову маніпуляцію над текстом вхідної програми. Завдяки тому, що в LISP програмний код – це дані, і навпаки, створення зручних і безпечних макросів стає досить простою задачею.

Велике значення в LISP також має лямбда-числення. Лямбда-числення – альтернативна модель для визначення комп'ютерних систем, функціонально аналогічна до машини Тюринга [9], що в контексті опису мови програмування означає, що в LISP відсутнє поняття «операція», яке застосовують для процедурних мов програмування. Натомість використовується аплікація функцій, що дозволяє використовувати єдиний синтаксис для арифметичних операцій, декларації функцій, декларації змінних, тощо.

При цьому варто звернути увагу на те, що сучасний стандарт Common Lisp також містить у собі повноцінну об'єктно-орієнтовну модель під назвою CLOS (Common Lisp Object System), завдяки якій можна в залежності від поставленої практичної задачі використовувати типові методи вирішення у стилі функціонального програмування, або об'єктно-орієнтованого.

### 3.2 Структура розробленого програмного забезпечення

Розроблене програмне забезпечення структурно з модулів, структура взаємодії яких зображена на Рисунку 31.

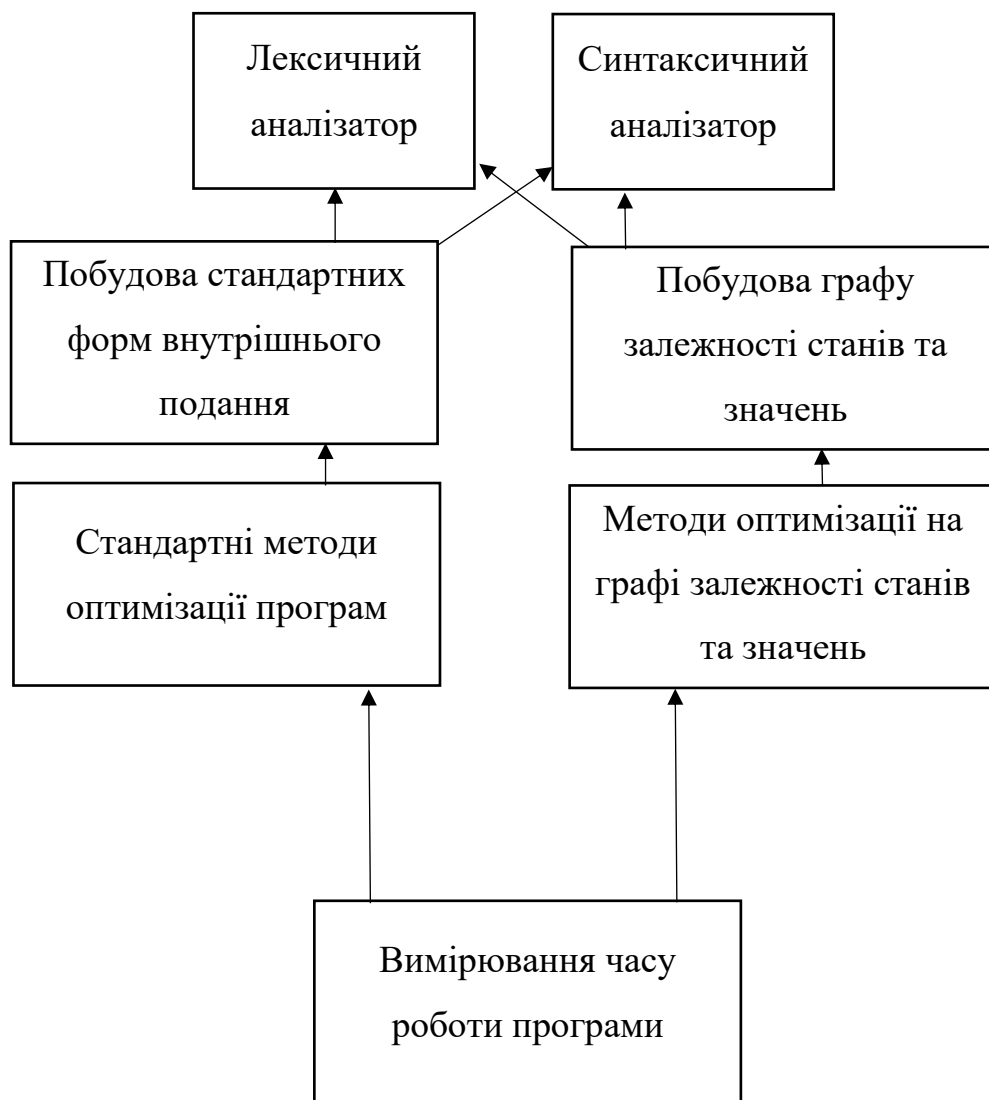


Рисунок 31 – Структура взаємодії модулів розробленого програмного комплексу

Така архітектура програмного комплексу дозволяє відокремити незалежні частини – модулі створення та реалізації методів оптимізації, для яких існує можливість для повторного використання спільних модулів – лексичного та синтаксичного аналізу.

### 3.3 Порівняння результатів роботи окремих методів оптимізації

Порівняльний аналіз методів оптимізації повинен складатися із декількох етапів, для того щоб показати усі основні характеристики двох підсистем. По-перше, необхідно дослідити характеристики часу виконання і розмір використаної пам'яті для кожного методу оптимізації окремо – це

продемонструє відмінності у окремих випадках роботи транслятора, але не буде відповідати загальній швидкодії транслятору.

### 3.3.1 Порівняння запропонованого методу видалення мертвого коду із стандартним

Для перевірки методу оптимізації видалення мертвого коду було застосовано програми, в яких половина коду може вважатися мертвою. Шаблон вхідної програми наступний:

```
int f()
{
    int x1 = 1;
    int x2 = 1;
    int x3 = x1 + 1;
    int x4 = x2 + 1;
    int x5 = x3 + 1;
    ...
    int x999 = x997 + 1;
    int x1000 = x998 + 1;
    return x999;
}
```

Кожна змінна із парним індексом не впливає на результат виконання вказаної функції. Тобто, половина тіла функції є мертвим кодом, а половина використовується для знаходження результату. Такий метод дозволяє в певній мірі показати реальну швидкодію даного методу оптимізації в порівнянні із стандартним. Порівняльні результати для таких програм наведені на Рисунку 32 і Рисунку 33.

Для випробувань змінним значенням виступає кількість змінних  $x$ , які присутні у вхідній програмі.

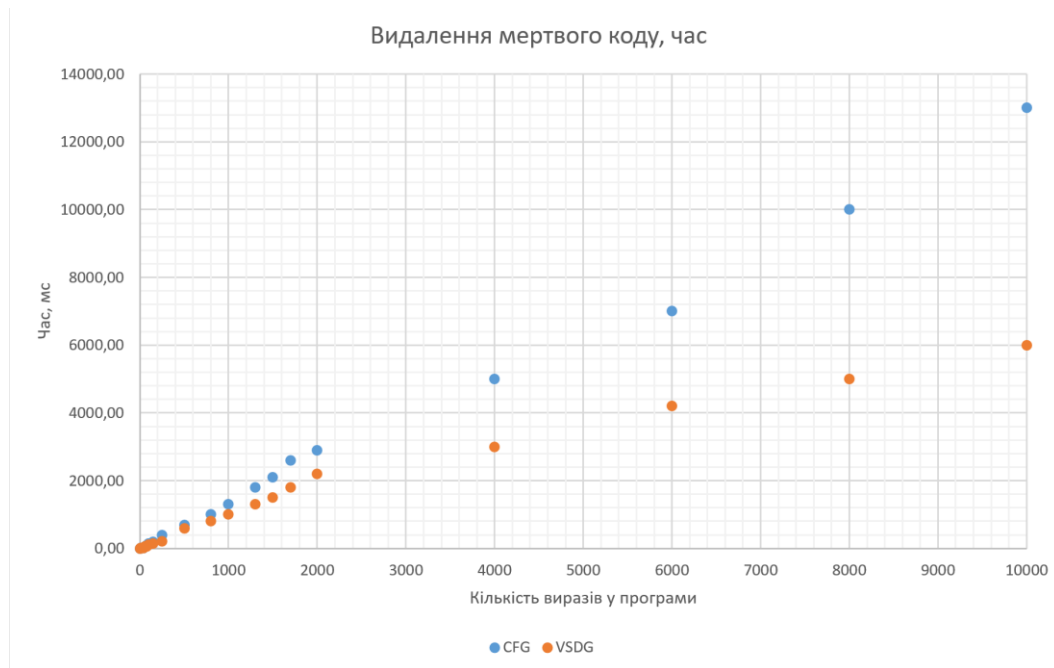


Рисунок 32 – Порівняння методів видалення мертвого коду за часовою характеристикою

З Рисунок 31 видно, що при достатньо великому розмірі програми застосування запропонованого методу дозволяє суттєво зменшити час виконання методу оптимізації. Це досягається завдяки тому, що немає потреби побудови допоміжних структур даних – графу залежності даних, та застосування аналізу потоку даних (тобто, побудова множин *in, out*).

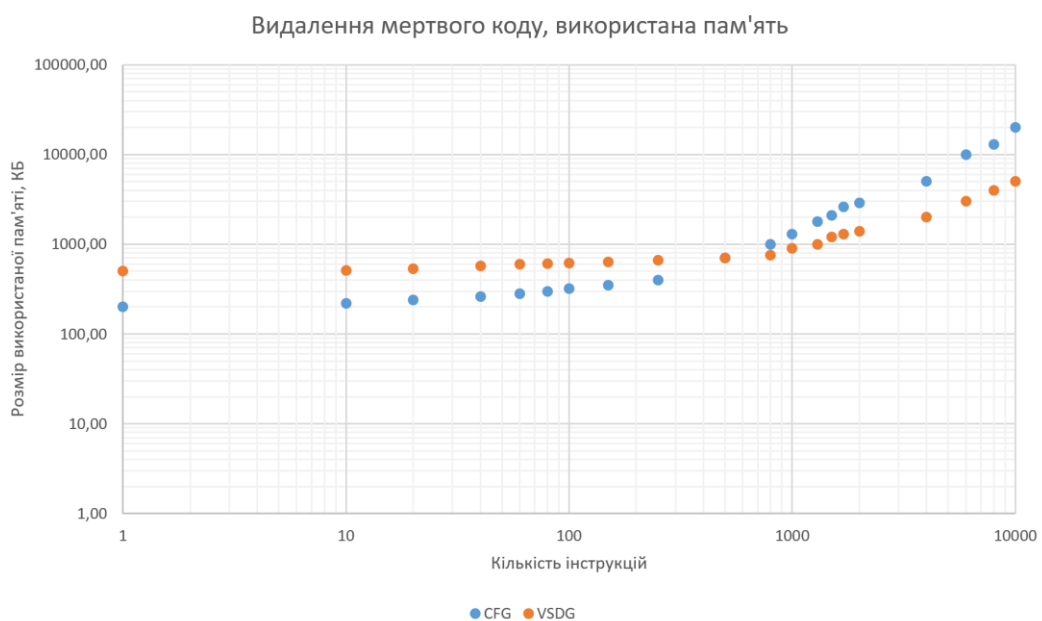


Рисунок 33 – Порівняння методів видалення мертвого коду за просторовою характеристикою



Як видно з Рисунку 32, граф залежності станів та значень на малих розмірах програми використовує більше пам'яті, ніж традиційний метод. Але вже на достатньо великих розмірах програми (близько 800 виразів) запропонований метод дозволяє суттєво знизити розмір використаної пам'яті під час оптимізації.

### 3.3.2 Порівняння запропонованого методу визначення інваріантів циклів із стандартним

Для визначення інваріантів у циклах було створено множину програм, які містять один великий за розміром цикл, із різною кількістю інструкцій, інваріантних для тіла циклу. Шаблон створеної програми наступний:

```
int f()
{
  for(int i=0;i<100;i++)
  {
    int x1 = 1;
    int x2 = x1;
    int x3 = x3;
    ...
    int x100 = x99;
    int a = i * x100;
  }
  return a;
}
```

Обчислення усіх змінних  $x$  є інваріантним відносно циклу програми, в той час як обчислення  $a$  – ні. Кількість інваріантних змінних задається параметром експерименту, результати зображені на Рисунку 34.



Рисунок 34 – Порівняння методів визначення інваріантів циклів при змінній кількості інваріантних інструкцій

З Рисунку 34 видно, що запропонований метод займає час, не менший ніж стандартний метод. Різниця між методами невелика, оскільки найскладніший етап для стандартного пошуку інваріантів циклів – пошук меж циклічних конструкцій, тут виконується лише один раз. Змінивши вхідні програми таким чином, щоб кількість циклічних конструкцій була параметром, було отримано результати, зображені на Рисунку 35.

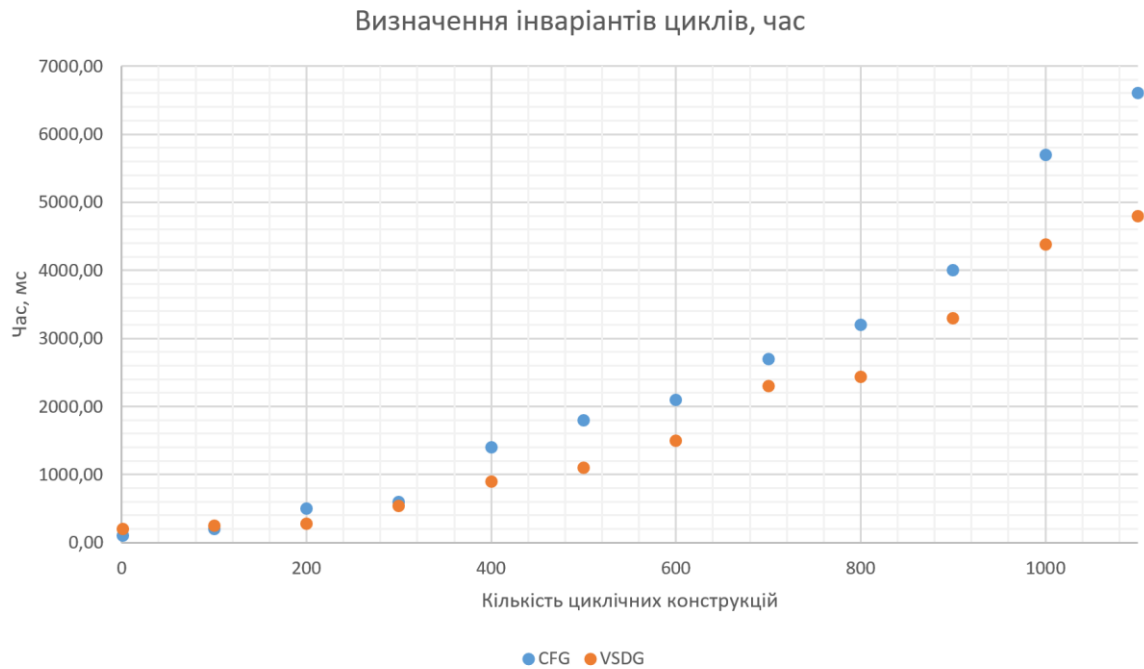


Рисунок 35 – Порівняння методів визначення інваріантів циклів при змінній кількості циклічних конструкцій

З рисунку видно, що при збільшенні кількості циклічних конструкцій складність пошуку для стандартного методу стає достатньо значною, щоб суттєво збільшити різницю між методами, що розглядаються, що підтверджується розрахунками складності двох методів.

### 3.3.3 Порівняння запропонованого методу видалення спільних виразів із стандартним

Для порівняння методів видалення спільних виразів було створено множину програм, які відображують найгірший випадок форми вхідних даних для обох методів:

```
int f() {
    int x1 = f1(f2(f3(...(f100(1))...));
    if (true) {
        x2 = f1(f2(f3(...(f100(1))...));
        return x1 + x2;
    } else {
        if (true) {
            x3 = f1(f2(f3(...(f100(1))...));
            return x3;
        } else {
            ...
            return xn;
        }
    }
}
```

Результати експериментів зображені на Рисунках 36-37.

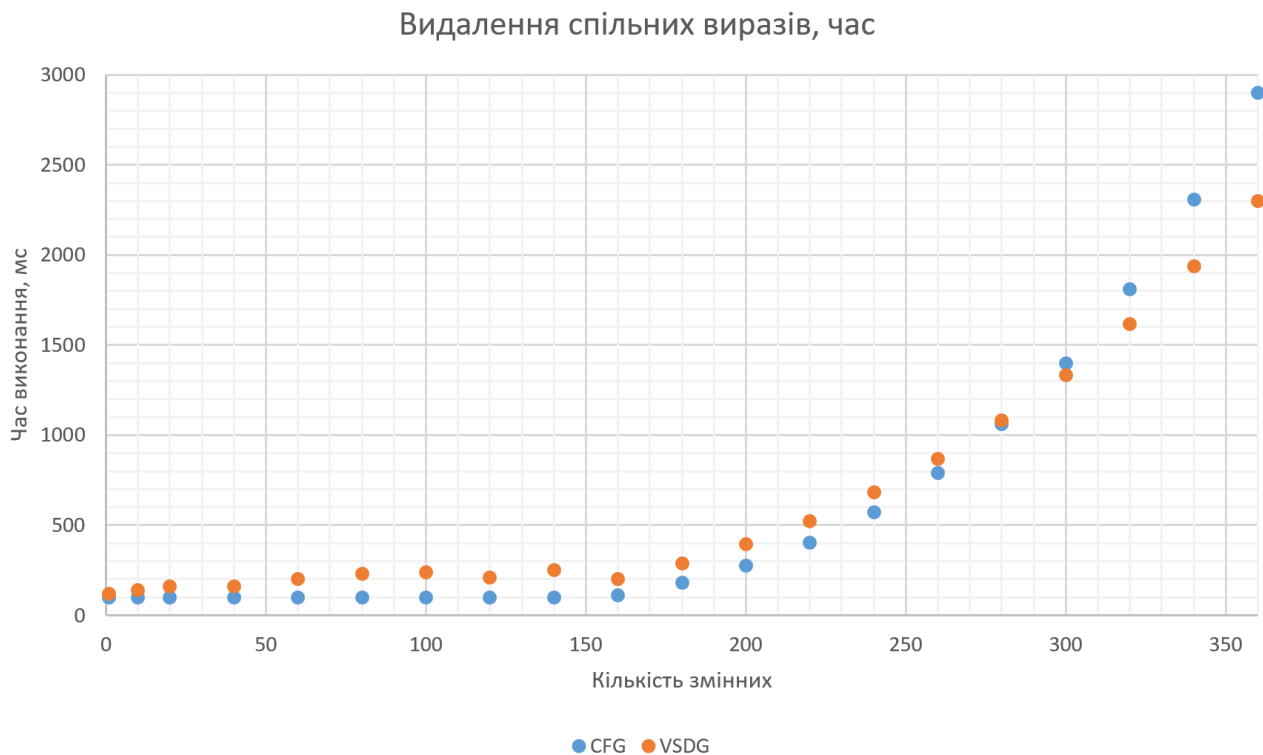


Рисунок 36 – Порівняння методів видалення спільних виразів за часовою характеристикою

Як видно з Рисунку 36, при збільшенні кількості виразів, які присутні в програмі, запропонований метод дозволяє суттєво пришвидшити час роботи методу. Це досягається завдяки тому, що час роботи не залежить від кількості вершин із багатьма ребрами, а також завдяки відсутності етапу глобальної нумерації змінних, а також завдяки спрощенню процедури видалення самих виразів. На малих розмірах програми кількість умовних виразів недостатньо велика, і тому додаткові обходи по графу залежності станів та значень займають більший час, ніж загальний час виконання оптимізації.

З Рисунку 37 можна зробити висновок, що індекс глобальної нумерації виразів на великих програмах починає займати помітно більше пам'яті, ніж запропонований метод, який має лінійну просторову складність.



Рисунок 37 – Порівняння методів видалення спільних виразів за просторовою характеристикою

В цілому для достатньо великих програм отримані практичні результати підтверджують очікувані теоретичні дані.

### 3.3.4 Порівняння запропонованого методу згортки констант із стандартним

Для дослідження методів згортки констант був застосований наступний набір правил для згортки:

1.  $c1 + c2 = (c1 + c2)$  – заміна додавання констант результатом;
2.  $c1/c2 = (c1/c2)$  – заміна множення констант результатом;
3.  $a + 0 = a$ ;
4.  $a * 0 = 0$ ;
5.  $a * c1 + b * c1 = (a + b) * c1$ ;
6.  $a * c1 - b * c1 = (a - b) * c1$ ;
7.  $a/c1 + b/c1 = (a + b)/c1$ ;
8.  $a/c1 - b/c1 = (a - b)/c1$ ;

На вхід трансляторів подавалися програми, для яких можна було застосувати вказані вище правила, наприклад:

```
int f() {  
    int x = 2 + 3 * 0 - 15;  
    return x;  
}
```

Вирази, над якими виконується оптимізація згортки констант, були створені автоматично, кількість операцій, присутня у виразах, виступає параметром для випробування. Результати випробувань зображені на Рисунку 38.

### Згортка констант

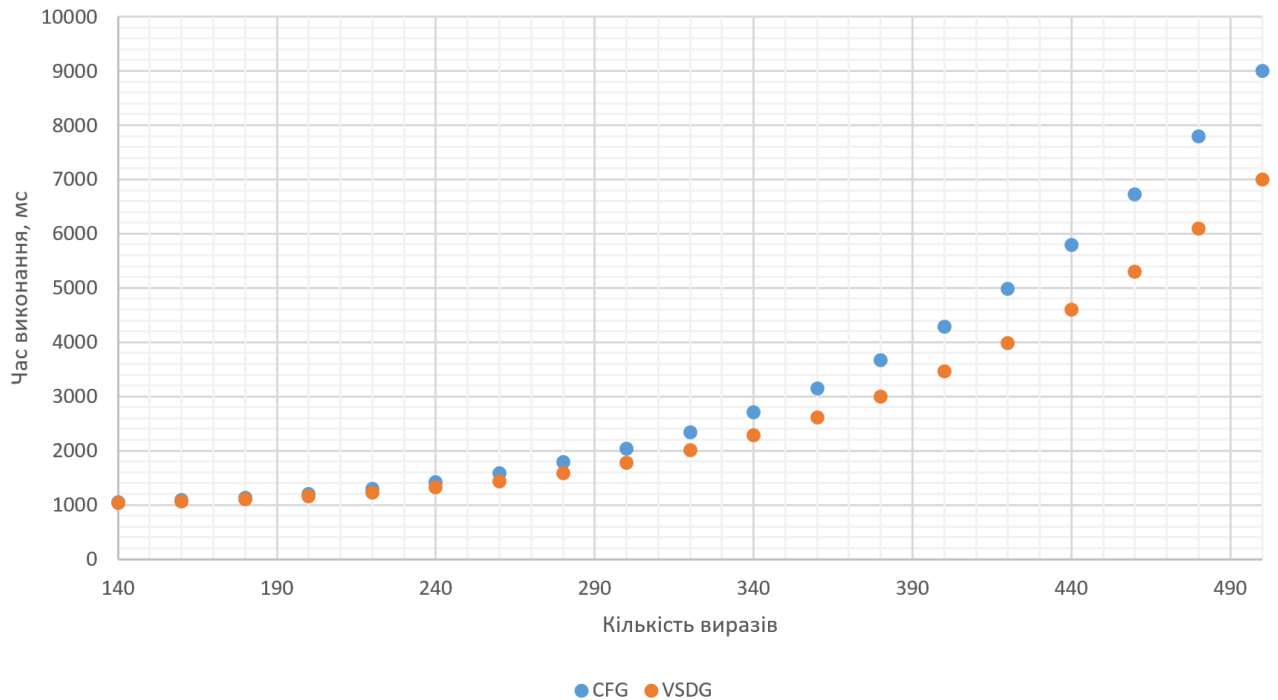


Рисунок 38 – Порівняння методів згортки констант за часовою характеристикою

Практичні результати доводять, що при достатньо великій кількості виразів, які піддаються згортанню, запропонований метод дозволяє зменшити кількість обчислень на величину, яка за теоретичними обчисленнями залежить від розміру програми за кубічним законом. Хоча загальна характеристика алгоритмічної складності запропонованого методу не змінилася, на практиці різниця між запропонованим методом і стандартним достатньо велика.

#### 3.3.5 Порівняння запропонованого методу згортки умовних конструкцій із стандартним

Оскільки граф залежності станів та значень має декілька видів умовних конструкцій, порівнювати швидкодію методів згортки умовних констант необхідно, використовуючи вхідні програми, які мають усі доступні типи умовних конструкцій – тобто, конструкції умовного вибору та циклу з

передумовою. Таким чином, було створено множину програм наступної форми:

```
int f() {
    int x = 0;
    if (true) { x++; } else { x--; }
    if (false) { x++; } else { x--; }
    while(false) { x++; }
    ...
    return x;
}
```

Змінною величиною для таких програм є кількість умовних конструкцій в програмі. Результати випробувань зображені на Рисунку 39.

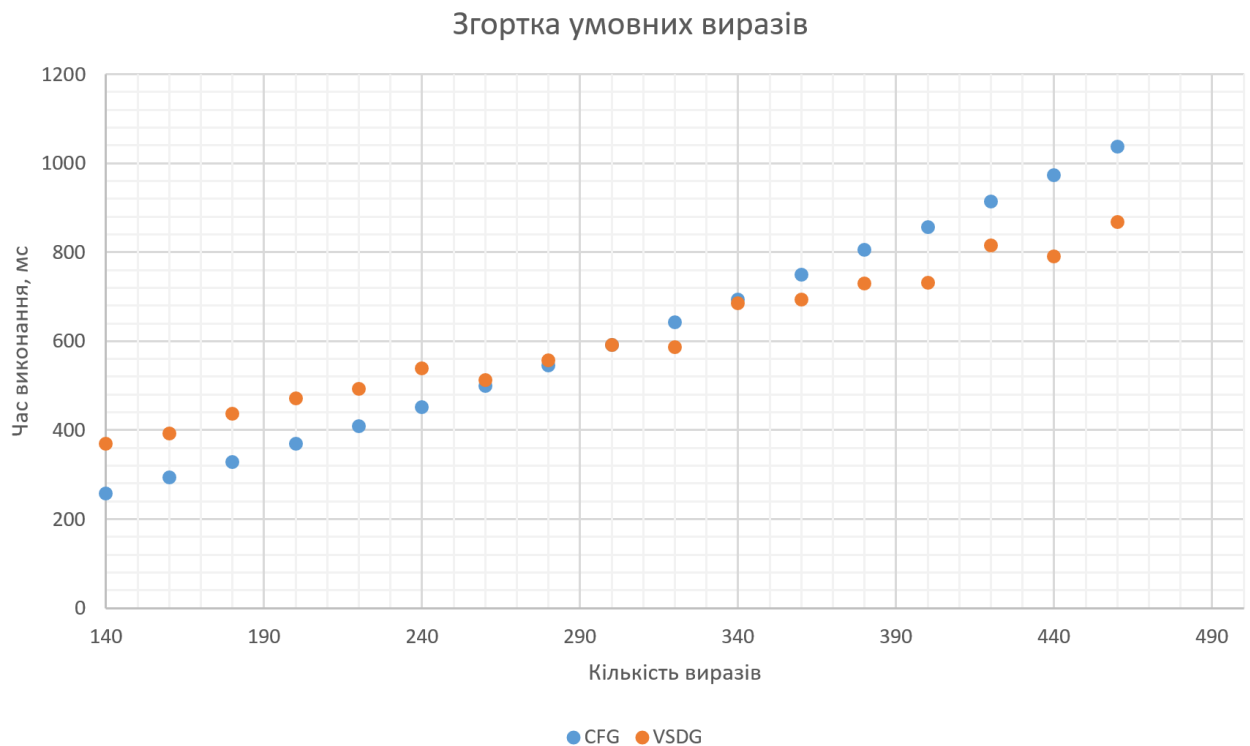


Рисунок 39 – Порівняння методів згортки умовних виразів за часовою характеристикою

На Рисунку 39 видно, що при малому розмірі програми запропонований метод суттєво повільніший, ніж стандартний. Це пояснюється тим, що для запропонованого методу необхідний додатковий повний перелік проміжного подання програми – для вершин  $\theta$  та  $\gamma$ . Але при великій кількості конструкцій вартість додаткового переліку стає низькою в



порівнянні із вартістю видалення умовних виразів для стандартного методу, який потребує оновлення внутрішніх структур даних після завершення роботи. Для запропонованого методу немає потреби проводити повторні обчислення таких додаткових даних, як граф потоку даних, множин досяжних значень, тощо. Це надає запропонованому методу помітну перевагу в порівнянні із стандартним методом.

### 3.4 Порівняння роботи транслятору, що використовує запропоновані методи із транслятором, який використовує стандартні методи

На практиці транслятори зазвичай не використовують лише один метод оптимізації під час генерації програмного коду – наприклад, популярний компілятор мови C під назвою GCC (Gnu C Compiler) має декілька налаштувань, що дозволяє регулювати «рівнем оптимізації». Ці налаштування дозволяють вмикати чи вимикати цілу низку методів оптимізації генерованого коду. Для експериментів було створено можливість запуску трансляторів в режимі з ввімкненою окремою оптимізацією, або із циклічним застосуванням усіх методів оптимізації, доки вони будуть мати вплив на внутрішнє подання програми. Псевдокод такого підходу наведено нижче:

```
(defun optimize_loop (program)
  (let* ((w1? (dead-code-elimination program))
        (w2? (invariant-code-motion program))
        (w3? (constant-folding program))
        (w4? (branch-folding program)))
    (if (or w1? w2? w3? w4?)
        program
        (optimize_loop program))))
```

Тобто робота транслятора вважається завершеною тільки якщо жоден метод оптимізації не мав впливу на програму.

Також необхідно порівняти час роботи трансляторів враховуючи і не враховуючи витрати на побудову внутрішнього подання. Граф залежності станів та значень – більш складна структура, ніж граф потоку команд, і його побудова містить декілька додаткових етапів у порівнянні із стандартною

формою подання, тому варто дослідити, у яких випадках можливо використання графу залежності станів та значень буде менш раціональним рішенням у порівнянні із стандартним методом.

Порівняння часу роботи двох трансляторів – достатньо складна задача, оскільки множина можливих вхідних даних надзвичайно велика, і її неможливо описати лише одною характеристикою – наприклад, кількість інструкцій в програмі. Завдяки накладеним обмеженням на граматику вхідної мови транслятора множина варіацій вхідних програм була суттєво зменшена, але все ще необхідно провести декілька різних досліджень.

Загальну роботу трансляторів можна перевірити, даючи на вхід комбінацію із програм, наведених у минулих розділах, поєднаних в одну функцію. Це дозволить створити модель програми, близької до тої, яка може зустрітися в справжніх програмах – в якій усі запропоновані методи оптимізації будуть впливати на кінцевий результат. Фрагмент такої функції наведено далі.

```
int f() {
  while(a < 3) {
    if(true || false) {
      int xdead1 = sin(cos(sin(...0.5)));
      if (false || true) {
        int xdead2 = xdead1 + sin(cos(sin(...0.5)));
        ...
      }
      if (false || true) {
        int xalive1 = sin(cos(sin(...0.5)));
        if (false || true) {
          int xalive2 = xalive1 + sin(cos(sin(...0.5)));
          ...
        }
        return xAliveN;
      }
    }
    return 0;
  }
}
```

Подібна структура функції дозволяє перевірити, яким чином взаємодіють окремі методи оптимізації в трансляторах: усі арифметичні

вирази мають спільну частину, значення якої може бути знайдене під час компіляції, винесене за межі окремих виразів запису до змінних, а потім за межі створеного циклу, оскільки являється інваріантним. Недоліком вказаної методики дослідження швидкодії трансляторів є те, що після застосування кожного окремого методу оптимізації програми один раз, транслятор дійде фіксованої точки. Розмір вхідних даних в даному випадку рівний кількості рядків у програмі. Результати проведених експериментів наведені на Рисунках 40 – 42.

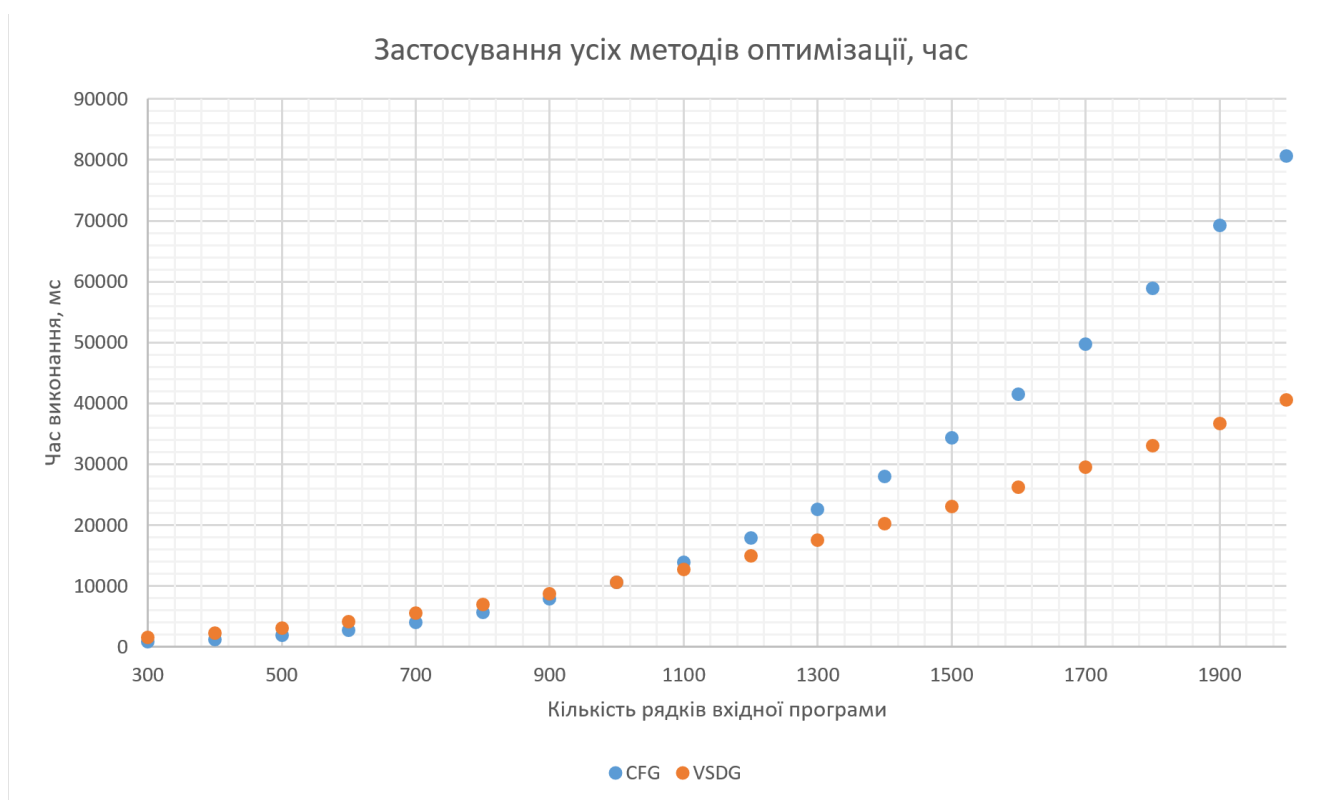


Рисунок 40 – Порівняння розроблених трансляторів за часовою характеристикою – не включаючи час на побудову форми проміжного подання

З Рисунку 40 видно, що на невеликих розмірах програми транслятор, що використовує запропоновані методи оптимізації має приблизно однакову швидкодію в порівнянні із стандартними методами. При збільшенні розміру вхідної програми стандартні методи починають сильно поступатися у швидкодії запропонованим методам. Варто зазначити, що дані, наведені на

Рисунку 40 не показують загальний час роботи системи – лише час, витрачений на застосування усіх методів оптимізації.

На Рисунку 41 зображені результати експериментів, де час роботи містить лише етапи початкової обробки програм: лексичний і синтаксичний аналіз, побудова форми внутрішнього подання.

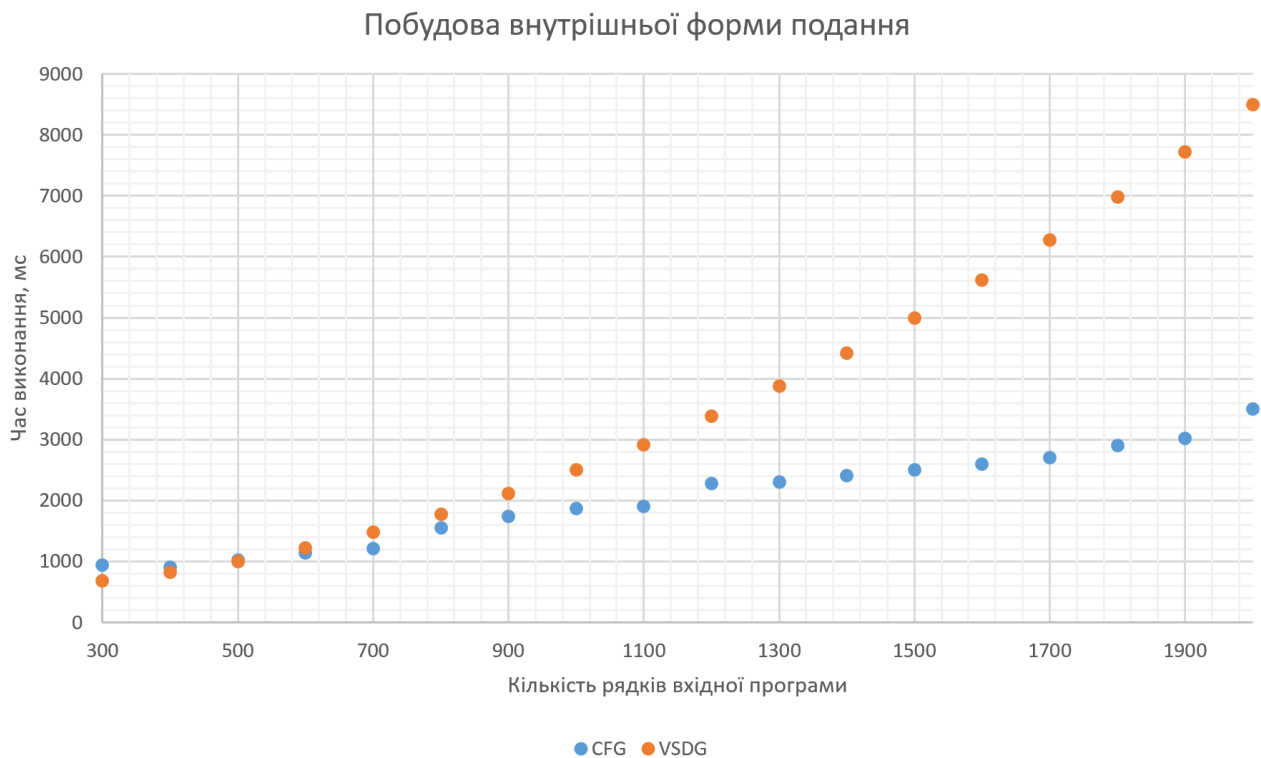


Рисунок 41 – Порівняння часу побудови внутрішньої форми подання для трансляторів

Побудова графу залежності станів та значень займає набагато більший час в порівнянні із побудовою такої простої форми внутрішнього подання, як граф потоку команд. Незважаючи на це, оскільки для вхідної програми усі розроблені методи оптимізації можуть мати вплив, то транслятор, що використовує стандартні методи оптимізації, витрачає зайвий час на побудову і підтримку додаткових структур даних, що відповідають за аналіз потоку даних в програмі, з тою різницею, що побудова цих структур виконується при першій потребі – при застосуванні певної оптимізації. Наприклад, якщо над програмою виконується оптимізація згортки умовних

виразів, то не буде побудований граф потоку даних, та не потрібно шукати множини досяжних значень.

З іншого боку, важно уявити сучасний транслятор, який буде виконувати лише одну чи дві оптимізації над вхідною програмою. Як видно з Рисунку 40, потреба виконувати додаткові обчислення для кожної оптимізації на великих обсягах вхідної програми стає недоліком в порівнянні із графом залежності станів та значень, де подібна інформація зберігається одразу після побудови.

Важливою характеристикою двох трансляторів також є розмір використаної пам'яті. Під розміром використаної пам'яті мається на увазі максимальний обсяг програми в оперативній пам'яті під час роботи транслятора. Результати цих досліджень наведені на Рисунку 42.

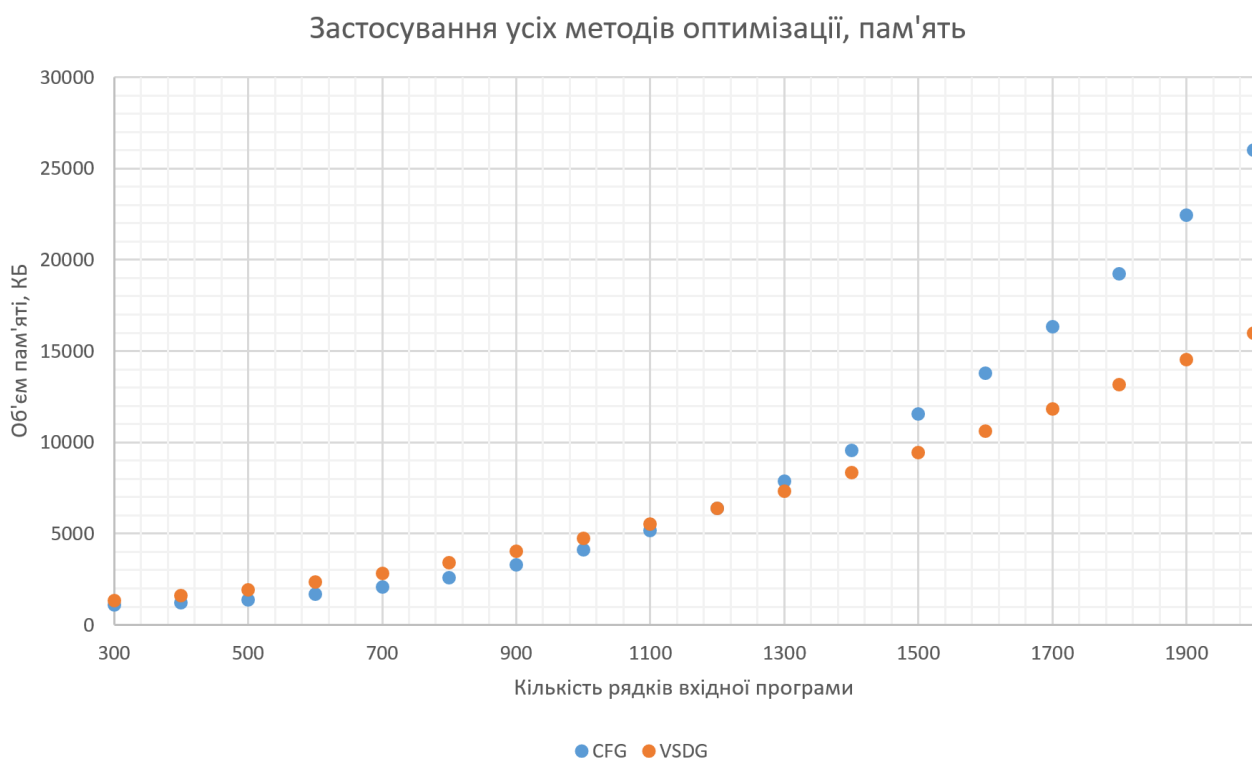


Рисунок 42 – Порівняння розміру використаної пам'яті для трансляторів

При малих розмірах програми граф залежності станів та значень програє стандартним методам по об'єму використаної пам'яті – оскільки структура графу передбачає більшу кількість ребер і вершин самого графу. Різниця не дуже велика, оскільки в стандартних методах замість більшої кількості

ребер необхідно зберігати додатковий граф, і також необхідно тримати в коректному стані множини досяжних значень, множин *in, out* для кожної вершини графу. При збільшенні розміру програми стає помітною різниця між двома трансляторами – внаслідок того, що деякі з запропонованих методів мають суттєво меншу просторову складність.

## ВИСНОВКИ

Граф залежності станів та значень – зручна і ефективна форма внутрішнього подання програми для використання у сучасних трансляторах. В багатьох випадках використання такої форми дозволяє зменшити час виконання оптимізації вхідної програми та об'єм використаної при цьому пам'яті. Це стає можливим завдяки модифікованій структурі графу, яка дозволяє зберігати додаткову важливу інформацію про програму прямо в тому ж внутрішньому поданні, замість того, щоб використовувати додаткові структури даних, які також необхідно підтримувати в актуальному стані.

Найбільш ефективним використання графу залежності станів та значень буде у випадку, коли необхідно вхідну програму провести через велику кількість різних методів оптимізації. Якщо компілятор передбачає застосування лише застосування одного методу оптимізації вхідної програми, витрати на створення графу залежності станів та значень можуть бути занадто великими у порівнянні із стандартними методами.

Запропонований модифікований метод оптимізації видалення коду показує кращі результати, ніж стандартний метод, по параметрам швидкодії та використання оперативної пам'яті. Це досягається завдяки тому, що в запропонованому методі не проводяться додаткові обчислення, що стосуються побудови графу потоку даних, та проведення аналізу потоку даних в програмі.

Запропонований модифікований метод оптимізації видалення спільних виразів показує кращі результати по швидкодії, ніж стандартний метод, завдяки тому, що в запропонованому методі не використовується додатковий етап глобальної нумерації виразів. При цьому, на невеликих розмірах програми, стандартний метод працює швидше.

Запропонований модифікований метод оптимізації згортки констант показує кращі результати по швидкодії та по використаній пам'яті, ніж

стандартний метод. Такі показники досягаються тому, що запропонований метод не вимагає перетворення внутрішнього подання у форму з єдиними записами до змінних.

Запропонований модифікований метод оптимізації визначення інваріантів для циклів показує кращі результати по швидкодії, ніж стандартний метод. Це можливо завдяки особливостям структури внутрішнього подання, що усуває потребу в визначенні меж природніх циклів в програмах, що було одним з найважчих етапів оптимізації у стандартному методі.

Запропонований модифікований метод оптимізації згортки умовних конструкцій показує кращі результати, ніж стандартний метод на великих розмірах програм. Це відбувається завдяки спрощеним механізмам перетворення внутрішнього подання, ніж при використанні стандартних методів. На невеликих програмах використання стандартного методу показує кращі результати, ніж запропонований метод, оскільки використовується менше обходів графу внутрішнього подання.

Варто звернути увагу, що визначені теоретичні показники алгоритмічної і просторової складності не завжди відображаються у результатах досліджень в очікуваній мірі. Для описаних методів це пояснюється тим, що розміри графу залежності станів та значень та розмір відповідного йому графу потоку команд не завжди прямо пропорційні – граф залежності станів та значень за визначенням буде мати більше вершин і ребер, ніж звичайний граф потоку команд.

Також використання графу залежності станів та значень не рекомендоване у випадку, якщо вихідною мовою є мова високого рівня, а не асемблер, чи інша мова, близька до машинної (наприклад, байт-код віртуальної машини)[2]. В процесі побудови графу залежності станів та значень втрачається багато інформації, яка необхідна для того, щоб була можливою ефективна генерація коду на мовах програмування високого рівня. Це означає, що у випадку, коли вхідна і вихідна мова у транслятора –



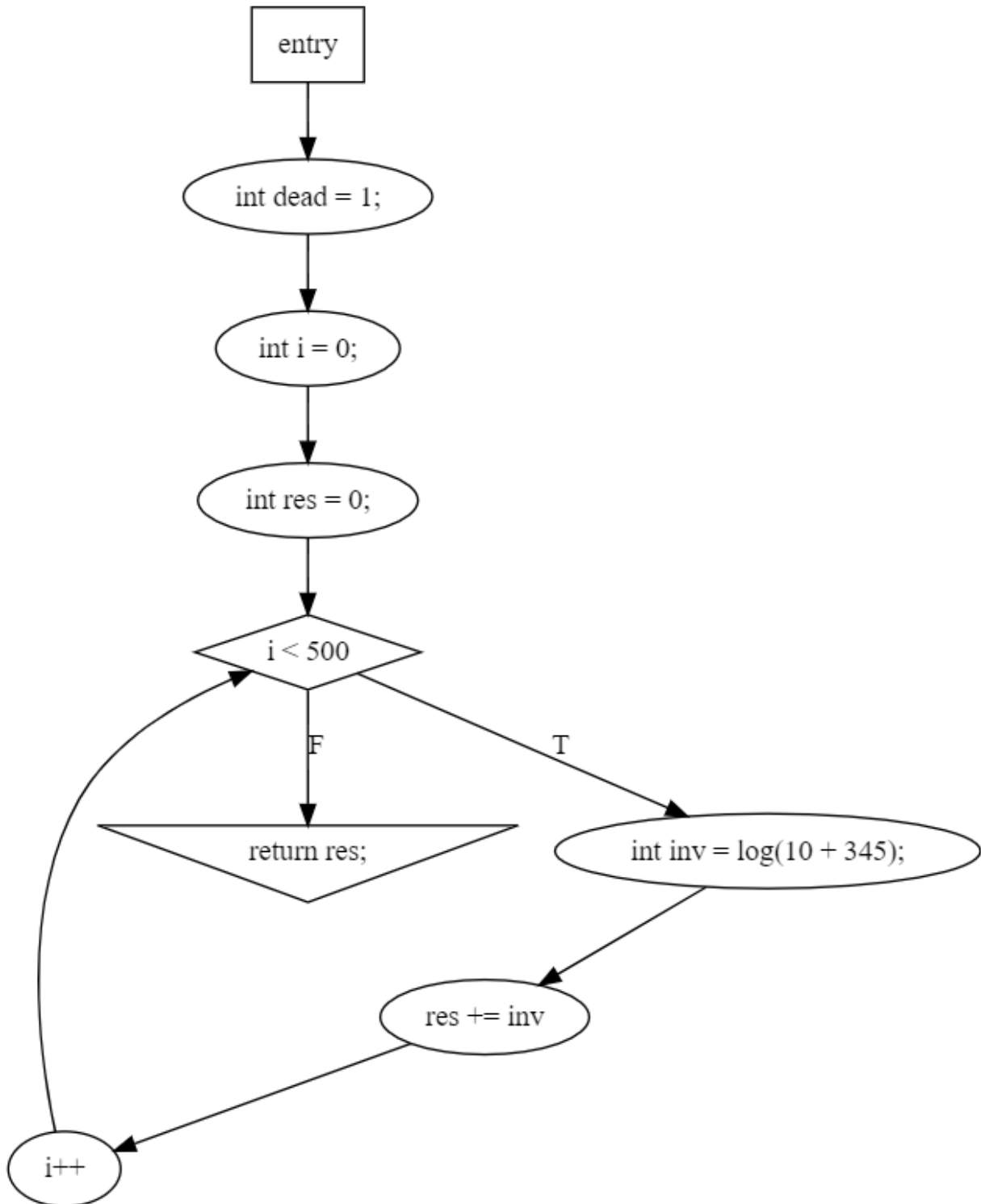
однакова (наприклад транслятор – статичний аналізатор коду, який повинен виконувати автоматичні оптимізації вхідних програм), то повернення внутрішньої форми подання у вигляд, близький до початкового – нетривіальна задача, яка досі не вирішена в повній мірі. Для таких трансляторів стандартні методи оптимізації все ще мають перевагу.

## СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. *Aho A., Сети P., Ульман Д.* Компиляторы: принципы, технологии и инструментарий: Пер. с англ./ Альфред Ахо, Рави Сети, Джеффри Ульман// издательский дом Вильямс, 2008. С. 554-590
2. *Johnson N., Mycroft A.* Combined Code Motion and Register Allocation using the Value State Dependence Graph./ Johnson N., Mycroft A.// 12th Intl. Conf. on Compiler Construction(CC'03) (April 2003), vol. 2622 of LNCS, pp. 1–16
3. Daniel Weise, Roger F. Crew, Michael D. Ernst, and Bjarne Steensgaard, Value dependence graphs: Representation without taxation/ *Weise D., Crew R., Ernst M., Steensgaard B.* //Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Portland, OR), January 1994, pp. 297–310
4. *Skiena S.*, The Algorithm Design Manual / Steven S Skiena// Springer-Verlag London 2008 . p. 480.
5. *L.H. Lee, J. Scott, B. Moyer, and J. Arends.* Low-cost branch folding for embedded applications with small tight loops./ L.H. Lee, J. Scott, B. Moyer, J. Arends // Proc. 32nd Int'l Symp. Microarchitecture, pp.103–111, 1999
6. *David A. McAllester.* On the complexity analysis of static analyses. / David A. McAllester // Journal of the ACM, 49(4):512–537, 2002
7. *N. Saleena, V. Paleri,* Global value numbering for redundancy detection: A simple and efficient algorithm/ N. Saleena, V. Paleri // Proceedings of the 29th Annual ACM Symposium on Applied Computing, SAC '14, ACM, New York, NY, USA, 2014, pp. 1609–1611
8. *Muchnick, Steven S.*, Advanced Compiler Design and Implementation./ Muchnick, S. // Morgan Kaufmann, 1997
9. *Randy Allen, Ken Kennedy,* Optimizing Compilers for Modern Architectures: A Dependence-based Approach/ Allen R., Kennedy K. // Morgan Kaufmann, 2001
10. *J. R. Ullmann.* An algorithm for subgraph isomorphism. / Ullmann J. R. // J. ACM, 23(1):31–42, 1976.
11. *R. E. Tarjan.* Finding dominators in directed graphs/ Tarjan. R. E. //. SIAM J. Comput., 3(1):62–89, Mar. 1974

# МЕТОДИ ОПТИМІЗАЦІЇ ПРИ ТРАНСЛЯЦІЇ НА ГРАФІ ЗАЛЕЖНОСТІ СТАНІВ ТА ЗНАЧЕНЬ

## Граф потоку команд

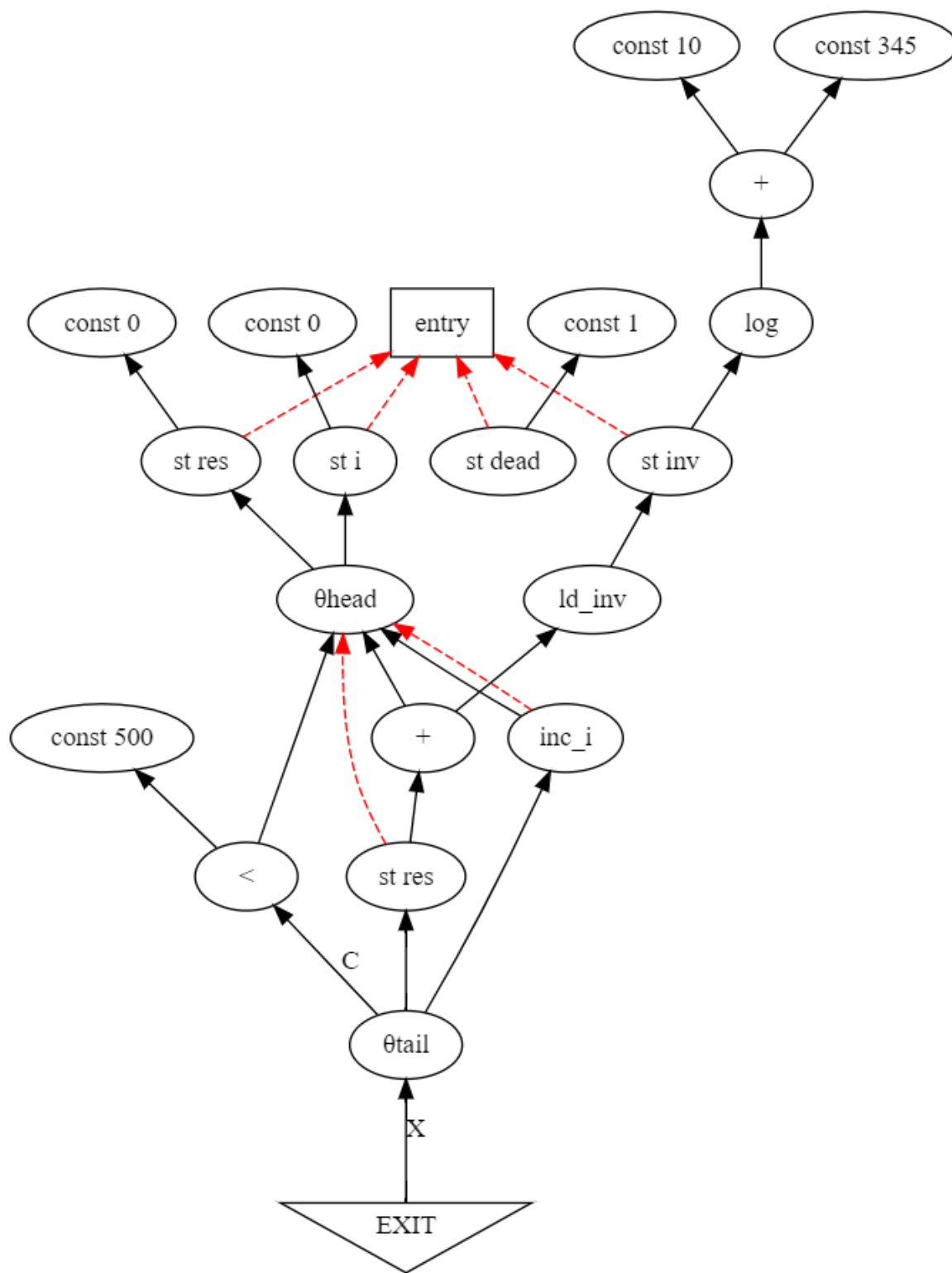


КПІ ім. Ігоря Сікорського, ФПМ, група КВ-62м

Подзе Олександр Сергійович

# МЕТОДИ ОПТИМІЗАЦІЇ ПРИ ТРАНСЛЯЦІЇ НА ГРАФІ ЗАЛЕЖНОСТІ СТАНІВ ТА ЗНАЧЕНЬ

Граф залежності станів та значень



КПІ ім. Ігоря Сікорського, ФПМ, група КВ-62м

Подзе Олександр Сергійович

# МЕТОДИ ОПТИМІЗАЦІЇ ПРИ ТРАНСЛЯЦІЇ НА ГРАФІ ЗАЛЕЖНОСТІ СТАНІВ ТА ЗНАЧЕНЬ

Порівняння алгоритмічної складності розроблених методів із стандартними

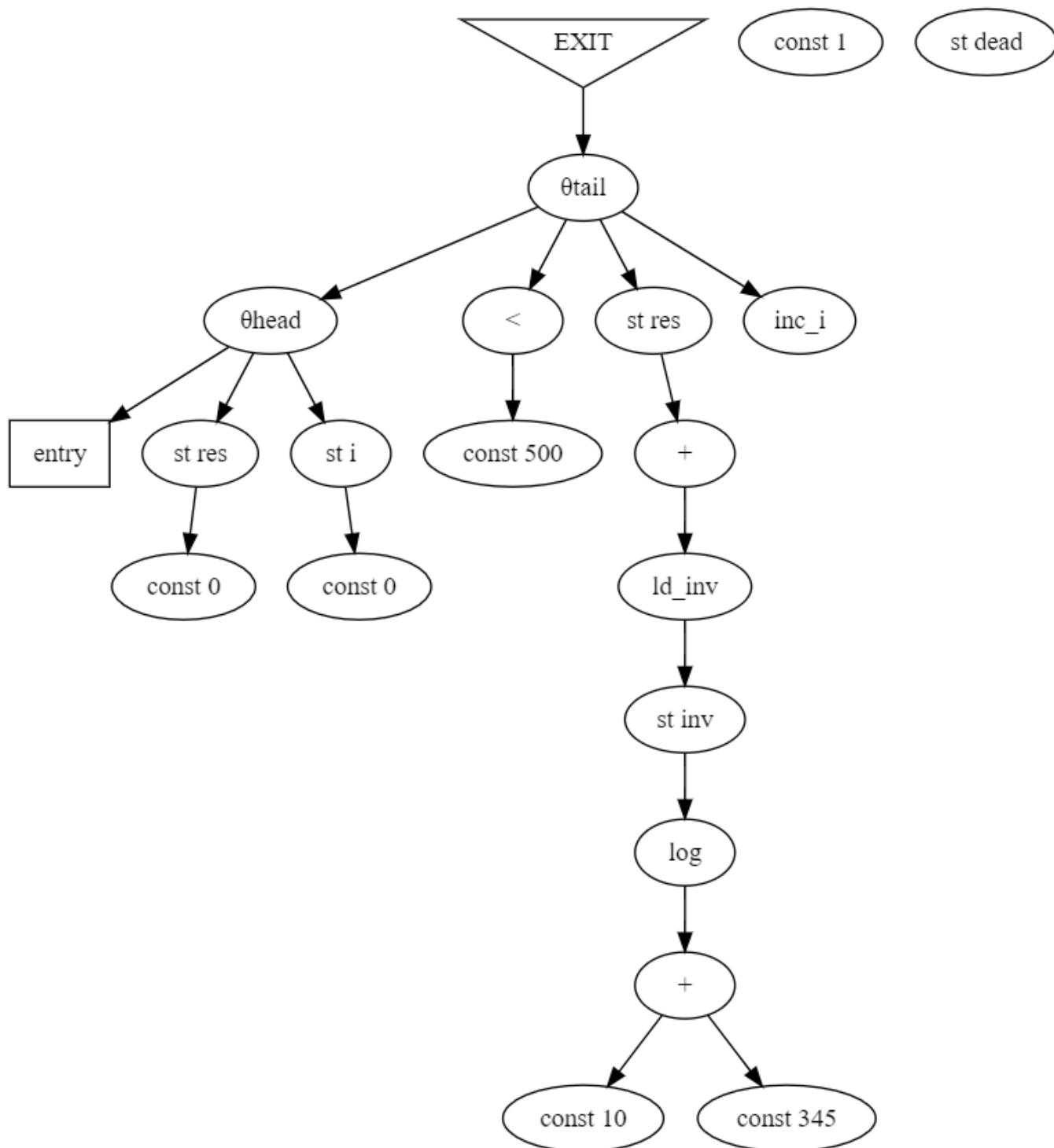
Назва методу	Стандартний метод		Запропонований метод	
	Час	Простір	Час	Простір
Видалення мертвого коду	$O( N ^2)$ в середньому $O( N ^4)$ в найгіршому	$O( N ^2)$	$O( N )$	$O( N )$
Видалення спільних виразів	$O(expr^3 +  N ^2)$ в середньому, $O( N ^4)$ в найгіршому	$O(expr^2)$	$O( N ^2)$ в середньому $O( N ^3)$ в найгіршому	$O( N )$
Згортка констант	$O( N ^4 + \max(N, E)^3)$	$O( N )$	$O( N ^4)$	$O( N )$
Видалення інваріантів циклів	$O( E  N ^2)$	$O( N )$	$O( N ^2)$	$O( N )$
Згортка умовних конструкцій	$O( N  + refresh\_cost)$	$O(1 + refresh\_cost)$	$O( N )$	$O(1)$

КПІ ім. Ігоря Сікорського, ФПМ, група КВ-62м

Подзе Олександр Сергійович

# МЕТОДИ ОПТИМІЗАЦІЇ ПРИ ТРАНСЛЯЦІЇ НА ГРАФІ ЗАЛЕЖНОСТІ СТАНІВ ТА ЗНАЧЕНЬ

## Дерево пост-домінаторів

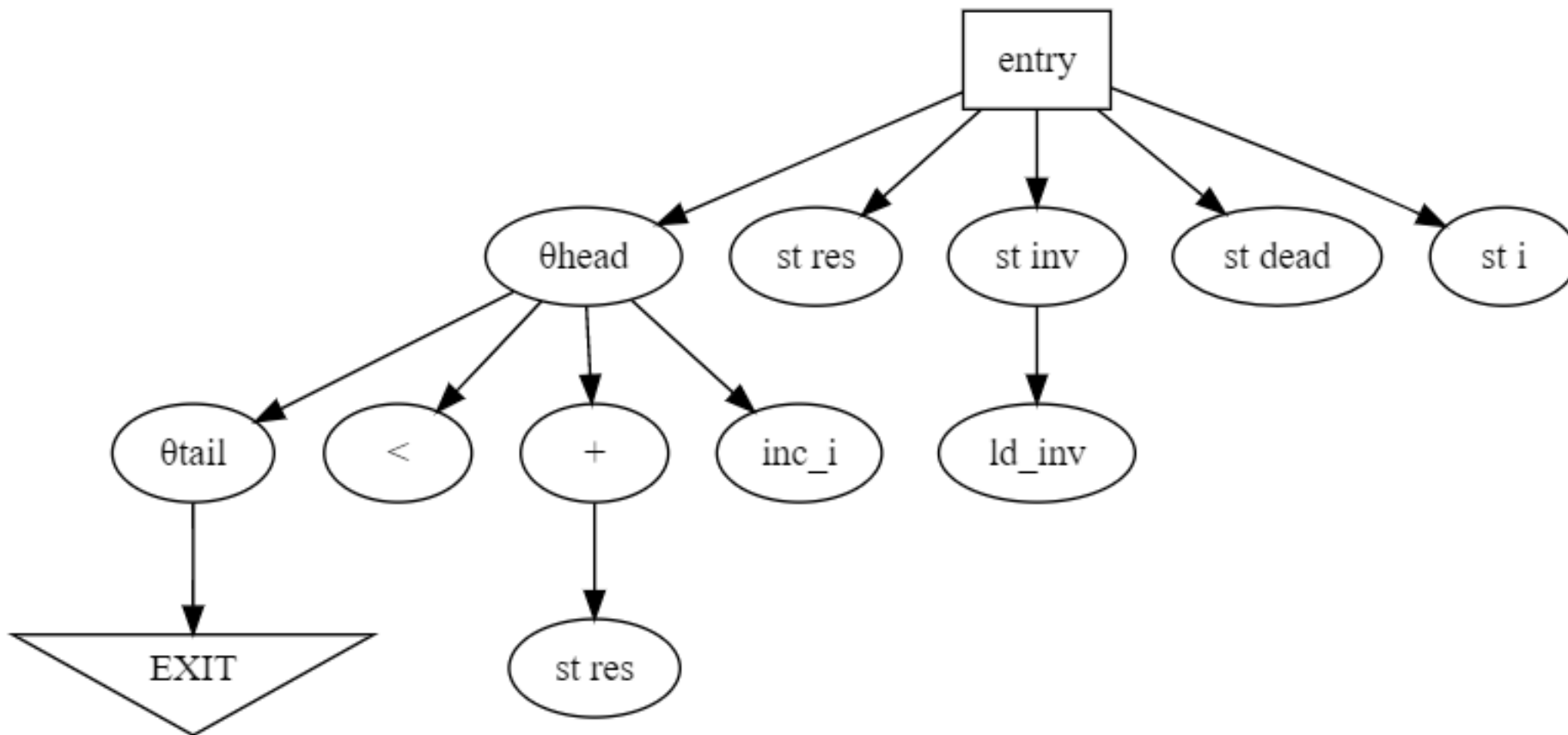


КПІ ім. Ігоря Сікорського, ФПМ, група КВ-62м

Подзе Олександр Сергійович

# МЕТОДИ ОПТИМІЗАЦІЇ ПРИ ТРАНСЛЯЦІЇ НА ГРАФІ ЗАЛЕЖНОСТІ СТАНІВ ТА ЗНАЧЕНЬ

Дерево домінаторів

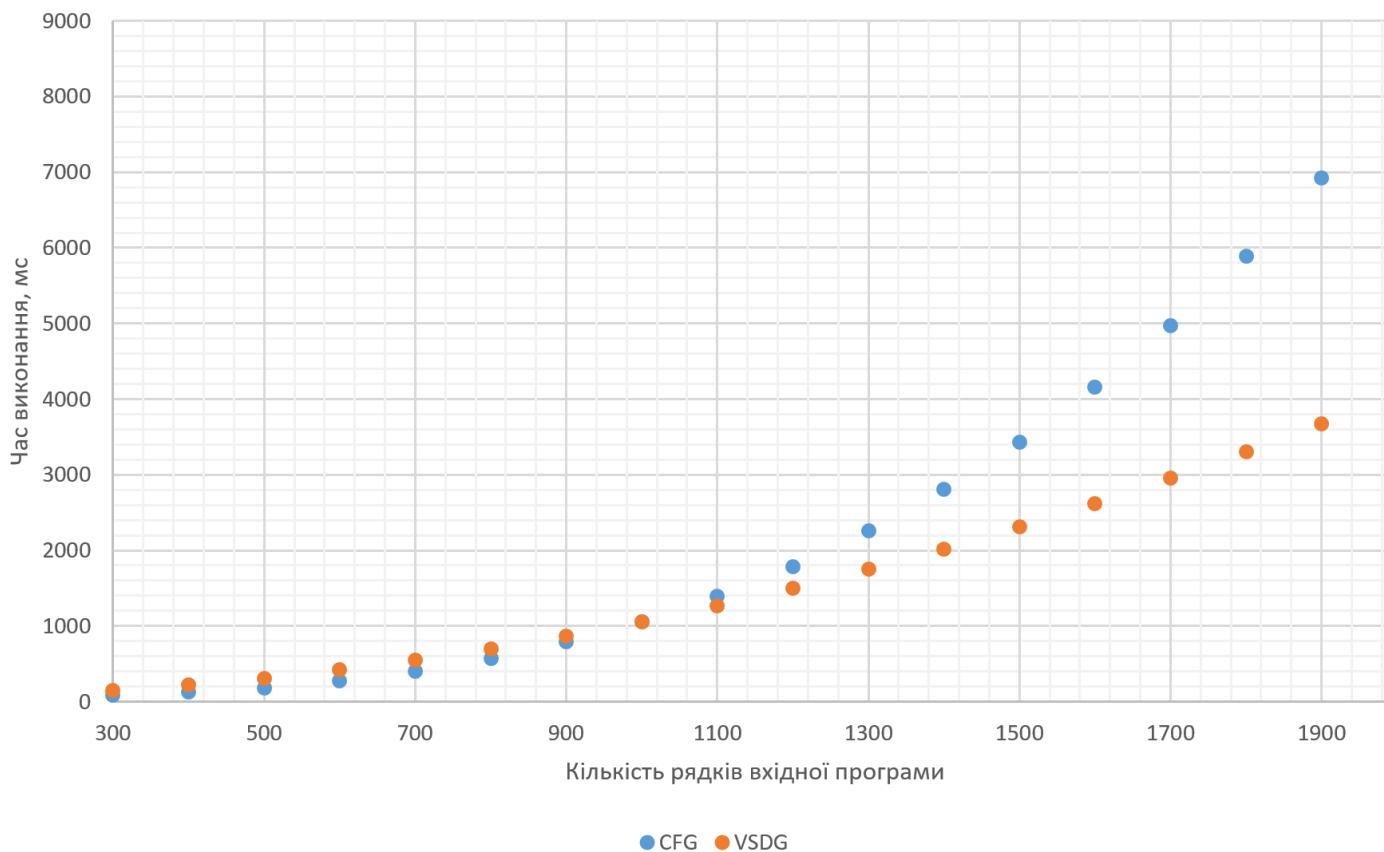


КПІ ім. Ігоря Сікорського, ФПМ, група КВ-62м

Подзе Олександр Сергійович

# МЕТОДИ ОПТИМІЗАЦІЇ ПРИ ТРАНСЛЯЦІЇ НА ГРАФІ ЗАЛЕЖНОСТІ СТАНІВ ТА ЗНАЧЕНЬ

Порівняльний графік швидкодії роботи запропонованих методів із стандартними



КПІ ім. Ігоря Сікорського, ФПМ, група КВ-62м

Подзе Олександр Сергійович



## ДОДАТОК 8. Граматика використаної мови програмування

```
primaryExpression: Identifier | Constant | '(' expression ')' ;

postfixExpression
    : primaryExpression
    | postfixExpression '(' argumentExpressionList? ')'
    | postfixExpression '++'
    | postfixExpression '--' ;

argumentExpressionList : assignmentExpression ;

unaryExpression : postfixExpression | '++' unaryExpression | '--' unaryExpression ;

unaryOperator : '&' | '*' | '+' | '-' | '~' | '!' ;

castExpression : '(' typeName ')' castExpression | unaryExpression ;

multiplicativeExpression
    : castExpression
    | multiplicativeExpression '*' castExpression
    | multiplicativeExpression '/' castExpression
    ;

additiveExpression
    : multiplicativeExpression
    | additiveExpression '+' multiplicativeExpression
    | additiveExpression '-' multiplicativeExpression
    ;

shiftExpression
    : additiveExpression
    | shiftExpression '<<' additiveExpression
    | shiftExpression '>>' additiveExpression
    ;

relationalExpression
    : shiftExpression
    | relationalExpression '<' shiftExpression
    | relationalExpression '>' shiftExpression
    ;

equalityExpression
    : relationalExpression
```

```

    | equalityExpression '==' relationalExpression
    | equalityExpression '!=' relationalExpression
;
andExpression:  equalityExpression;
exclusiveOrExpression  :  andExpression;
inclusiveOrExpression  :  exclusiveOrExpression;
logicalAndExpression  :  inclusiveOrExpression  |  logicalAndExpression '&&' inclusiveOrExpression  ;
logicalOrExpression   :  logicalAndExpression  |  logicalOrExpression '||' logicalAndExpression   ;
assignmentExpression
    :  conditionalExpression
    |  unaryExpression assignmentOperator assignmentExpression
    |  DigitSequence
;
assignmentOperator  :  '=' | '+='  ;
expression  :  assignmentExpression  |  expression ',' assignmentExpression  ;
constantExpression  :  conditionalExpression  ;
declaration  :  declarationSpecifiers initDeclaratorList ';' |  declarationSpecifiers ';'  ;
declarationSpecifiers  :  declarationSpecifier+  ;
declarationSpecifiers2  :  declarationSpecifier+  ;
declarationSpecifier
    :  storageClassSpecifier
    |  typeSpecifier
    |  typeQualifier
    |  functionSpecifier
    |  alignmentSpecifier
;
initDeclaratorList  :  initDeclarator  |  initDeclaratorList ',' initDeclarator  ;
initDeclarator  :  declarator  |  declarator '=' initializer  ;
typeSpecifier:  'bool' | 'int';
nestedParenthesesBlock  :  ( ~( '(' | ')' )  |  '(' nestedParenthesesBlock ')'  )*  ;
typeQualifierList  :  typeQualifier  |  typeQualifierList typeQualifier  ;
parameterTypeList  :  parameterList  |  parameterList ',' '...'  ;
parameterList  :  parameterDeclaration  |  parameterList ',' parameterDeclaration  ;
parameterDeclaration

```

```

: declarationSpecifiers declarator
| declarationSpecifiers2 abstractDeclarator?
;

identifierList : Identifier | identifierList ',' Identifier ;

typeName : specifierQualifierList ;

abstractDeclarator : pointer | pointer? directAbstractDeclarator gccDeclaratorExtension* ;

directAbstractDeclarator
: '(' abstractDeclarator ')' gccDeclaratorExtension*
| '[' typeQualifierList? assignmentExpression? ']'
| '[' 'static' typeQualifierList? assignmentExpression ']'
| '[' typeQualifierList 'static' assignmentExpression ']'
| '[' '*' ']'
| '(' parameterTypeList? ')' gccDeclaratorExtension*
;

typedefName: Identifier;

initializer: assignmentExpression ;

initializerList : designation? initializer | initializerList ',' designation? initializer ;

designation : designatorList '=' ;

designatorList : designator | designatorList designator ;

statement
:
| compoundStatement
| expressionStatement
| selectionStatement
| iterationStatement
| jumpStatement
;

compoundStatement : '{' blockItemList? '}' ;

blockItemList : blockItem | blockItemList blockItem ;

blockItem : statement | declaration ;

expressionStatement : expression? ';' ;

selectionStatement : 'if' '(' expression ')' statement 'else' statement ;

iterationStatement
: While '(' expression ')' statement

```

```

    | Do statement While '(' expression ')' ';'
    | For '(' forCondition ')' statement
;

forCondition
    : forDeclaration ';' forExpression? ';' forExpression?
    | expression? ';' forExpression? ';' forExpression?
;

forDeclaration    : declarationSpecifiers initDeclaratorList | declarationSpecifiers ;
forExpression     : assignmentExpression | forExpression ',' assignmentExpression ;
externalDeclaration : functionDefinition | declaration ;
functionDefinition : declarationSpecifiers? declarator declarationList? compoundStatement ;
declarationList   : declaration | declarationList declaration ;

LeftParen  : '(';
RightParen : ')';

LeftBracket : '[';
RightBracket : ']';

LeftBrace : '{';
RightBrace : '}';

Less : '<';
LessEqual : '<=';

Greater : '>';
GreaterEqual : '>=';

LeftShift : '<<';
RightShift : '>>';

Identifier
    : IdentifierNondigit
      ( IdentifierNondigit
        | Digit
      )*
;

fragment
IdentifierNondigit: Nondigit | UniversalCharacterName ;

fragment

```

Digit : [0-9];

Constant : IntegerConstant | BooleanConstant ;

fragment

IntegerConstant: DecimalConstant;

fragment

DecimalConstant : NonzeroDigit Digit\* ;

fragment

Sign : '+' | '-';

DigitSequence : Digit+ ;