Довідка
про відсутність плагіату у випускній кваліфікаційній роботі

Міністерство освіти і науки України
Український державний університет науки та технологій

Кафедра «Комп'ютерні інформаційні технології»

ДОВІДКА

За результатами перевірки випускної кваліфікаційної роботи здобувача вищої освіти
Яковенка Богдана Миколайовича
(прізвище, ім'я, по батькові)
на тему: «Розробка методу визначення відповідності тексту програми графічному представленню алгоритму»
в роботі не виявлено порушень академічної доброчесності.

Керівник ВКР _____ Олена Кураєтнеєк

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Український державний університет науки і технологій

Кафедра Комп'ютерні інформаційні технології

«ДО ЗАХИСТУ»

Завідувач кафедри

_____ /Вадим ГОРЯЧКІН/

«_17_» _____12_____ 20_21_ р.

ДИПЛОМНА РОБОТА

на здобуття освітнього ступеня «магістр»

Галузь знань **12 Інформаційні технології**

Спеціальність **121 Інженерія програмного забезпечення**

Тема **Розробка методу визначення відповідності тексту програми графічному представленню алгоритму**

Theme **Development of a method for determining the correspondence of the program text to the graphical representation of the algorithm**

Керівник дипломної роботи доц. _____ Олена КУРОП'ЯТНИК

Нормоконтролер доц. _____ Олена КУРОП'ЯТНИК

Студент групи П32021 _____ Богдан ЯКОВЕНКО

Student Bohdan YAKOVENKO

Дніпро – 2021

Дніпровський національний університет залізничного транспорту
імені академіка В. Лазаряна

Факультет Комп'ютерних технологій і систем кафедра Комп'ютерні інформаційні технології
Спеціальність Інженерія програмного забезпечення

«ЗАТВЕРДЖУЮ»
Завідувач кафедри
_____ В.І. Шинкаренко
(підпис)
«17» 12 2021 р.

ЗАВДАННЯ

до дипломної роботи на здобуття ОС магістр
(освітній ступінь)
студента групи ПЗ2021 Яковенко Богдана Миколайовича
(номер групи) (ПІБ)

1 Тема дипломної роботи: Розробка методу визначення відповідності тексту програми графічному представленню алгоритму
затверджена наказом по університету від «18» листопада 2020 р. № 690 ст.
2 Термін подання студентом закінченої роботи «3» грудня 2021р.
3 Вихідні дані до дипломної роботи _____
_____
_____
4 Зміст пояснювальної записки (перелік питань до розробки): аналіз сучасного стану методів визначення відповідності тексту програми графічному представленню алгоритму, обгрунтування напрямку дослідження, проектування та розробка ПЗ, дослідження відповідності тексту програми графічному представленню алгоритму, огляд питань охорони та безпеки праці в надзвичайних ситуаціях.
5 Перелік демонстраційного матеріалу: презентація на тему визначення відповідності тексту програми графічному представленню алгоритму, результати експериментів, висновки; відео демонстрації роботи розробленої програмної системи для проведення досліджень.

6 Консультанти (з назвами розділів):

Розділ	Консультант	Підпис, дата	
		Завдання видав	Завдання прийняв
Техніко-економічні розрахунки	доц. Гненний М.В.	18.10.21	24.10.21
Охорона праці	доц. Саблін О.І.	26.10.21	24.11.21

КАЛЕНДАРНИЙ ПЛАН

№ пор.	Назва розділів дипломної роботи	Термін виконання розділів роботи	Примітка
1	Вступ		
2	Аналіз сучасного стану дослідження проблеми за науковими літературними джерелами		
3	Аналіз сучасного стану програмно-апаратного забезпечення, яке потребує вдосконалення для вирішення проблем дослідження		
4	Постановка задачі, технічне завдання	11.10.21 – 17.10.21	30%
5	Техніко-економічні показники		
6	Розробка інструментальних засобів дослідження		
7	Виконання досліджень	08.11.21 – 14.11.21	60%
8	Оформлення тез доповідей		
9	Оформлення статті у фаховий журнал		
10	Оформлення пояснювальної записки		
11	Розробка демонстраційних матеріалів	29.11.21 – 05.12.21	100%

Дата видачі завдання «18» листопада 2020 р.

Керівник дипломної роботи _____ О.С. Куроп`ятник
(підпис) (ПІБ)

Завдання прийняв до виконання _____ Б.М. Яковенко
(підпис) (ПІБ)

РЕФЕРАТ

Об’єктом дослідження є процеси визначення відповідності тексту програми графічному представленню алгоритму. Предметом дослідження є метод зіставлення та оцінки схожості графічного представлення алгоритму та тексту програми.

Метою дослідження в контексті даної роботи є розробка методу визначення відповідності тексту програми графічному представленню алгоритму.

Методи дослідження: конструктивно-продукційне моделювання, що базується на використанні апарату формальних граматик, теорія графів (способи представлення графів, методи обходу графів). При розробці програмної реалізації побудованого методу була застосована технологія об’єктно-орієнтованого проектування, та моделювання на основі UML.

Результати та їх новизна: розроблено метод зіставлення фрагментів документів представлених різними штучними мовами, а саме програмного коду та блок-схеми алгоритму. Метод формалізовано засобами конструктивно-продукційного моделювання.

Пояснювальна записка складається зі вступу, п’яти розділів, висновків, бібліографічного списку та п’яти додатків.

Вступ описує суть, мету та актуальність роботи (3 сторінки).

Перший розділ: аналіз сучасного стану методів визначення відповідності тексту програми графічному представленню алгоритму (14 сторінок).

Другий розділ: обґрунтування напрямку дослідження (15 сторінок).

Третій розділ: опис процесу проектування і розробки ПЗ (10 сторінок).

Четвертий розділ: дослідження відповідності тексту програми графічному представленню алгоритму (22 сторінки).

П’ятий розділ: огляд питань охорони та безпеки праці в надзвичайних ситуаціях (9 сторінок).

Додатки: технічне завдання, робочий проект, тези двох доповідей, наукова стаття.

Таблиць – 4 , рисунків – 50, бібліографія – 60 джерел.

Ключові слова: конструктивно-продукційне моделювання; конструктор; графове представлення тексту; граф керування програми; алгоритм; відповідність алгоритму

ЗМІСТ

ВСТУП

Актуальність роботи. В академічному середовищі останнім часом дуже гостро постає проблема неправомірних запозичень в навчальних та наукових роботах.

Вже існують розроблені моделі та методи для визначення запозичень у документах, в тому числі структурованих. Вони орієнтовані переважно на природомовні тексти, проте наукові роботи можуть також містити частини, написані штучними мовами, а саме фрагменти програмного коду, формули тощо, а також зображення (графічні представлення алгоритмів, UML-діаграми та ін.).

Фрагменти документів, написаних природними та штучними мовами не можна обробляти однаковими методами. Окремою задачею є порівняння між собою фрагментів різними штучним мовами. Тому доцільним є розробка методу для порівняння програмного коду та графічних елементів між собою, оскільки існуючі методи виявлення запозичень у наукових роботах працюють лише з фрагментами написаними однією мовою.

Метод, що розроблюється може бути додатковим елементом обробки академічних робіт для виявлення плагіату.

Тема роботи: «Розробка методу визначення відповідності тексту програми графічному представленню алгоритму».

Об’єктом дослідження є процеси визначення відповідності тексту програми графічному представленню алгоритму.

Предметом дослідження є метод зіставлення та оцінки схожості графічного представлення алгоритму та тексту програми.

Мета та задачі дослідження. Метою магістерської роботи є розробка методу та засобів визначення відповідності тексту програму графічному представленню алгоритму.

Для досягнення мети буде поставлено та вирішено такі задачі:

- дослідження відомих методів розпізнавання геометричних фігур;
- дослідження відомих методів розпізнавання тексту;
- розробка та формалізація методу визначення відповідності тексту алгоритму, що включає: попередню обробку тексту програми, побудову його проміжного

представлення у вигляді списку керуючих елементів та елементів процесу, побудова графів керування програми для тексту програми і її алгоритму та зіставлення графів керування;

- розробка програмного забезпечення для визначення відповідності тексту програми графічному представленню алгоритму за розробленим методом.

Методи дослідження. Для вирішення поставлених задач було використано: конструктивно-продукційне моделювання, що базується на використанні апарату формальних граматик та знайшло відображення у застосуванні конструкторів та методів їх перетворення; теорію графів (способи представлення графів, методи обходу графів).

При розробці програмної реалізації побудованого методу була застосована технологія об’єктно-орієнтованого проектуванн та моделювання на основі UML.

Наукова новизна. Розроблено метод зіставлення фрагментів документів представлених різними штучними мовами, а саме програмного коду та блок-схеми алгоритму. Метод формалізовано засобами конструктивно-продукційного моделювання.

Практичне значення. Впровадження запропонованого методу дозволило розробити засіб автоматизованого зіставлення графічного представлення алгоритму та його програмної реалізації, який може бути використано для:

- аналізу студентами власних програм, з метою отримання навичок розробки програм за відомим алгоритмом;
- виявлення запозичень у академічних роботах, які містять фрагменти різними штучними мовами.

Апробація результатів дослідження та публікації. Результати магістерської роботи доповідались на семінарі кафедри КІТ 22.02.2021 р., представлено всеукраїнських науково-практичних конференціях: Науково-технічний прогрес на транспорті (29 березня 2021 р.) та Наука і сталий розвиток транспорту (28 жовтня 2021 року).

Основні результати представлені у науковій статті «Визначення відповідності тексту алгоритму програми на основі конструктивно-продукційної моделі графа керування» (Наука та прогрес транспорту. Вісник Дніпропетровського національного університету залізничного транспорту, 2021, № 4 (94)).

1 АНАЛІЗ СУЧАСНОГО СТАНУ МЕТОДІВ ВИЗНАЧЕННЯ ВІДПОВІДНОСТІ ТЕКСТУ ПРОГРАМИ ГРАФІЧНОМУ ПРЕДСТАВЛЕННЮ АЛГОРИТМУ

1.1 Призначення та сфера застосування

Результатом даного дослідження має бути метод, який на основі конструктивно-продукційної моделі графу керування зіставляє між собою програмну реалізацію та блок-схему алгоритму задля визначення їх відповідності один одному. Даний метод буде покладено в основу програмного додатку, який автоматизує зіставлення тексту та алгоритму програми.

Програмний додаток може використовуватися учасниками навчального процесу спеціальності «Інженерія програмного забезпечення»: викладачами для автоматизації перевірки академічних робіт студентів, студентами для самоконтролю. Ще однією сферою застосування програмного додатку є перевірка документації, яка містить представлення алгоритмів та програм, на академічний плагіат.

1.2 Постановка задачі

Результатом даної роботи повинен бути метод та програмний додаток, котрий зіставляє між собою програмну реалізацію та блок-схему алгоритму задля визначення відповідності між ними.

Додаток повинен відповідати наступним вимогам:

- мінімізована потреба в діях за сторони користувача під час визначення відповідності між програмною реалізацією та блок-схемою алгоритму;
- розпізнавання програмного коду на мові C++ та побудова графу керування у вигляді списку суміжності;
- розпізнавання блок-схеми та побудова графу керування у вигляду списку суміжності;
- зіставлення між собою графів керування побудованих за програмним кодом та блок-схемою методом обходу в ширину та надання висновку щодо їх схожості у вигляді процентного співвідношення;

– простота в інтеграції нової функціональності, необхідної для подальших досліджень.

1.3 Аналіз предметної сфери

1.3.1 Опис проблеми. Актуальність дослідження

На сьогоднішній день не було знайдено жодного методу для порівняння між собою на предмет схожості програмної реалізації та блок-схеми алгоритму. Існуючі методи визначення схожості не розглядають блок-схему як графічне представлення алгоритму, а порівнюють лише програмні реалізації. Також слід зазначити, що розроблені методи в своїй більшості не використовують конструктивно-продукційне моделювання.

1.3.2 Огляд останніх досліджень та публікацій

Вже існуючі наукові роботи присвячені розв’язанню задачі виявлення плагіату у програмному коді [1 – 4] або визначення алгоритмічної схожості програмних реалізацій [5]. Наукові роботи, що розглядають та використовують блок-схему, як представлення алгоритму для вищезгаданих задач, не знайдено.

Існує велика кількість різних алгоритмів пошуку плагіату в природомовних текстах [6, 7], у тому числі в структурованих документах [8], та систем, що призначені для автоматичної перевірки на запозичення. Незважаючи на досить успішну практику в суміжних областях, точних підходів до ідентифікації плагіату в текстах, написаних штучними мовами, таких як програмний код, досить мала кількість. Існуючі в даній області алгоритми можна класифікувати наступним чином:

– текстові алгоритми;
– структурні алгоритми;
– семантичні алгоритми.

Особливістю текстових алгоритмів є те, що вони представляють коди програм у вигляді тексту, а точніше рядками над алфавітом, символ якого відповідає певному оператору або групі операторів мови програмування. Причому аргументи операторів ігноруються, що робить марними багато елементарних дій щодо приховання плагіату, наприклад перейменування змінних. Текстові алгоритму включають в себе як

найбільш ефективні сучасні алгоритми пошуку плагіату в програмних кодах, так і самі старі алгоритми. До текстових алгоритмів можна віднести метод відбитків [9 – 11] та метод просіювання [9, 11].

Структурні алгоритми, виходячи з їх назви, використовують структуру програного коду, зазвичай це граф потоку керування [12], або абстрактне синтаксичне дерево [13]. Внаслідок такого представлення програмного коду, більшість дій направлених на плагіат зводиться нанівець. Але всі алгоритми цього класу є край трудомісткими, тому зазвичай не використовуються на практиці.

Семантичні алгоритми багато в чому схожі на структурні та текстові. Наприклад, один з таких алгоритмів використовує представлення вихідного коду програми у вигляді графа з вершинами двох типів. Одні будуються із послідовності операторів, яким призначається певна семантика (наприклад, математичний вираз або цикл), інші задають відношення, у якому є сусідні з нею вершини (наприклад, входження). Між двома вершинами першого типу зазвичай стоїть вершина другого. Такі алгоритми, як і структурні, дуже трудомісткі, тому досить рідко зустрічаються на практиці.

Блок-схема [14] зазвичай представлена у вигляді цифрового зображення. Вона складається з функціональних блоків різної форми, пов’язаних між собою стрілками. У кожному блоці описується одна або кілька дій. Елементи блок-схеми можна розділити на два типи: графічні елементи та друковані символи.

До графічних елементів блок-схеми можна віднести: блок початку, блок воду/виводу, блок умови, блок дії, лінія, лінія з стрілкою. Друкованими символами блок-схеми є текст який розташований в графічних елементах.

Задача розпізнавання графічних елементів блок-схеми та наведеного в них тексту відноситься до задач цифрової обробки зображення, а саме класифікації об’єктів [15]. Безпосередньо класифікація є однією с задач машинного навчання.

Роботу з цифровим зображення можна умовно розділити на три етапи: попередня фільтрація та підготовка зображення, логічна обробка результатів фільтрації та прийняття рішень на основі логічної обробки.

Для фільтрації зображення використовують методи, які дозволяють виділити на зображенні необхідні області, без їх аналізу. Більшість цих методів застосовує єдине перетворення до всіх точок зображення. До таких методів відносяться: бінаризація зображення по порогу [16], перетворення Фур'є [17], Вейвлет-перетворення [18], фільтрація контурів та границь [19], кореляція [20] та ін.

Після фільтрації зображення ми отримуємо набір даних, які придатні для подальшої обробки. До методів, які дозволяють перейти від зображення до об'єктів на зображення відносяться: морфологія [21, 15], контурний аналіз [22], сегментація [23], Фур'є-дескриптори [15] та ін.

Після логічної обробки зображення необхідно застосувати методи, які не працюють безпосередньо з зображення, а дозволяють приймати рішення на основі попередньої обробки зображення. В багатьох випадах це задачі машинного навчання та прийняття рішень.

Задача розпізнавання тексту на зображенні є достатньо популярною, існує багато наукових робіт на цю тему [24 – 26]. Для побудови системи розпізнавання тексту використовують два підходи: метрики [27 – 31] та нейронні мережі [32, 33].

На сьогодні виділяють три основних підходу для вирішення задачі розпізнавання друкованих символів за допомогою метрик: шаблонний [27], структурний [28 – 30] і ознаковий [31].

Шаблонні методи розпізнавання друкованих символів. Першим етапом шаблонного методу є перетворення зображення тексту в растрове. Далі проводиться його порівняння з всіма наявними в базі даних шаблонами. Найбільш відповідним шаблоном є той, в якого буде найменша кількість точок, відмінних від розпізнаваного символу. В таких методах достатньо висока точність розпізнавання дефектних символів. Недоліком таких методів є неможливість працювати з шрифтом, який не закладений до бази даних.

Структурні методи розпізнавання друкованих символів. Такі методи зберігають інформацію не про крапкове написання символу, а про його топологію. Тобто еталон зберігає інформацію про взаємне розташування окремих зіставних частин символу. Таким методам неважливо розмір або шрифт розпізнаваного

символу, проте вони погано розпізнають символи з дефектами. Необхідно відмітити, що в теперішній час програма розпізнавання тексту заснована на скелетизації не використовується сама по собі, а комбінується с іншими методами розпізнавання, в першу чергу в комбінації с нейронною мережею.

Ознакові методи розпізнавання друкованих символів. Такі методи базуються на тому, що зображенню ставиться у відповідність N-мірний вектор ознак. Розпізнавання полягає в порівнянні його з набором еталонних векторів тої же розмірності. Основні переваги таких методів – простота реалізації, стійкість до зміни форм символів, висока швидкодія та низька кількість відказів. До недоліків відносять нестійкість до дефектів зображення та втрата інформації про взаємне розплодження елементів символу.

Існуючі підходи до розпізнавання фігур можна розділити наступним чином: це методи, засновані на контурі [34], засновані на області, просторовому домені і домені перетворення; інформаційно збережені (IP) і інформаційно незбережені методи (NIP). Однак підходи до виділення і поданням фігур зазвичай поділяються, в залежності від способів обробки, на одномірні функції уявлення фігури (Onedimensional function), апроксимацію полігонів (Polygonal approximation), взаємозв'язок просторових ознак (Spatial interrelation feature), моменти (Moments), методи розподілу шкали (Scalespace methods), домени перетворення фігури (Shape transform domains). Для уявлення нескладних фігур на основі контурів часто використовують: комплексні координати, функцію відстані, дотичний кут, кривизну контуру і Фур‘є дескриптори. Всі ці методи (крім Фур‘є дескрипторів) входять до клас «одновимірні функції уявлення фігури».

1.4 Огляд програмних аналогів

На поточний момент не було знайдено жодного публічного програмного додатку, який би визначав відповідність тексту та алгоритму програми на основі конструктивно-продукційної моделі графу керування.

Але слід зазначити, що існує програмне забезпечення в сферах близьких до досліджуваних, наприклад у сфері виявлення плагіату в програмному коді.

1.4.1 Огляд функціоналу онлайн-сервісу перевірки коду на плагіат Copyleaks

Copyleaks – це хмарна платформа для виявлення плагіату та автоматизованого оцінювання на основі штучного інтелекту та машинного навчання.

Передова технологія Copyleaks допомагає виявляти плагіат, порушення авторських прав і витік вмісту з веб-сторінок. Copyleaks допомагає академічним установам, LMS та іншим платформам електронного навчання, студентам, видавцям, юристам з інтелектуальної власності, підприємствам та багатьом іншим відстежувати вміст і гарантувати його оригінальність.

До функціоналу даного програмного додатку, згідно з інформацією з офіційного сайту [35], входить:

- запобігання викрадення контенту з сайту;
- порівняння текстових документів для виявлення схожості мж ними;
- пошук дублікатів файлів та документів в внутрішній мережі;
- порівняння між собою веб-сайтів та веб-сторінок на предмет виявлення дублювання контенту;
- перевірка програмного коду на предмет виявлення плагіату.

Слід зазначити, що програмний додаток має декілька версій, які відрізняються ціною та кількістю можливих перевірок. Версії поділяються на два типи: для навчання та для бізнесу.

В табл. 1.1 представлено вартість версій програмного додатку відповідно до наданої на офіційному сайті [36, 37].

Таблиця 1.1 – Вартість версій програмного додатку Copyleaks

Для бізнесу		Для навчання	
Вартість (USD)	Кількість перевірок	Вартість (USD)	Кількість перевірок
1	2	3	4
Безкоштовно	20	Безкоштовно	20
9,99	100 разів на місяць	10,99	100 разів на місяць
21,99	250 разів на місяць	24,99	250 разів на місяць

Продовження таблиці 1.1

1	2	3	4
34,99	500 разів на місяць	40,99	500 разів на місяць
65,99	1000 разів на місяць	75,99	1000 разів на місяць
159,99	2500 разів на місяць	184,99	2500 разів на місяць
299,99	5000 разів на місяць	349,99	5000 разів на місяць
579,99	10000 разів на місяць	679,99	10000 разів на місяць

На рис. 1.1 – 1.8 представлено інтерфейс та функціонал онлайн сервісу Copyleaks.

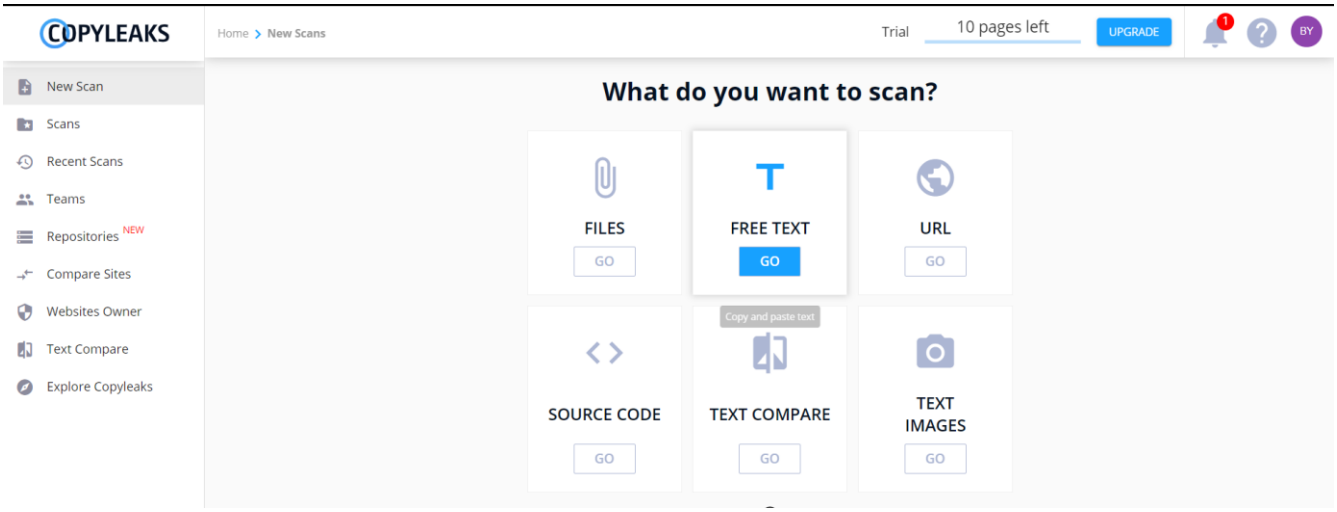


Рисунок 1.1 – Інтерфейс онлайн сервісу Copyleaks

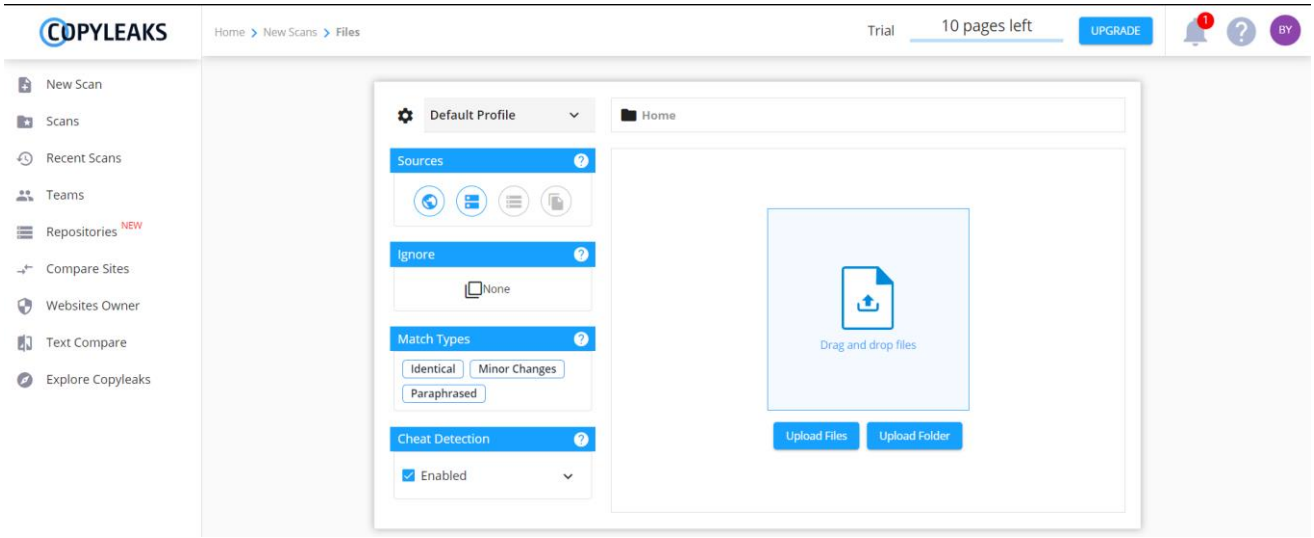


Рисунок 1.2 – Вікно для порівняння файлів

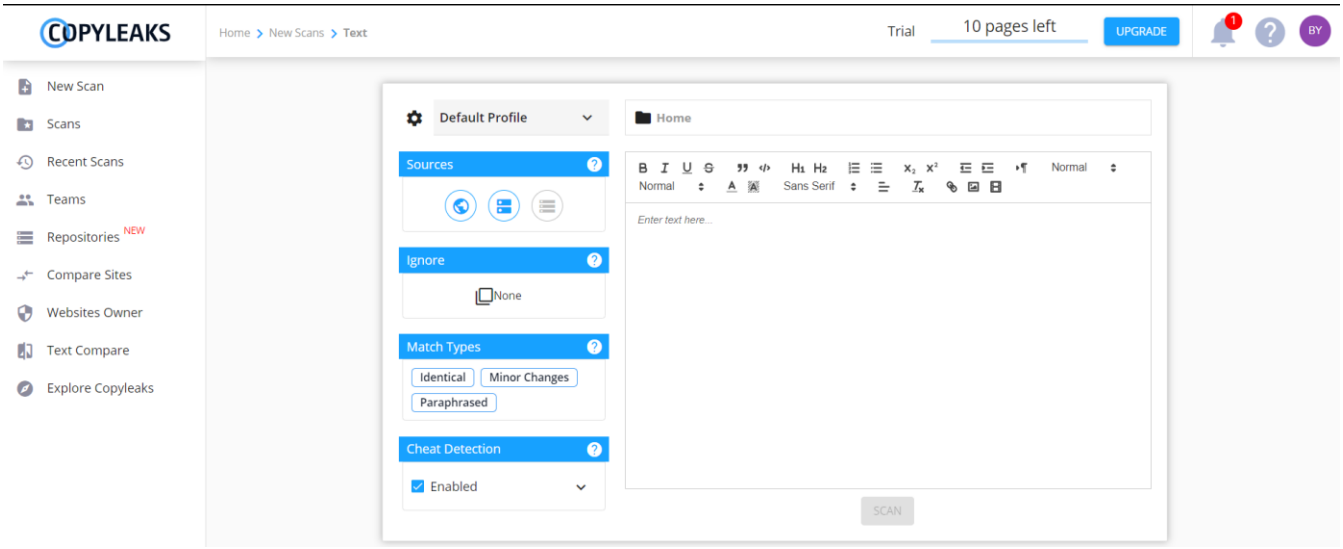


Рисунок 1.3 – Вікно для виявлення плагіату в тексті

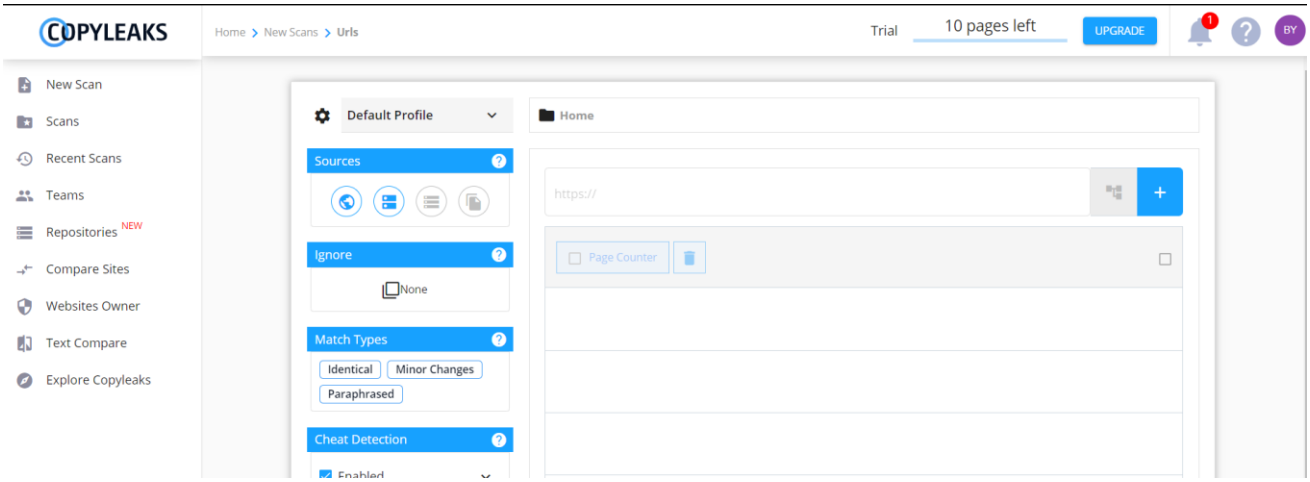


Рисунок 1.4 – Вікно для порівняння веб-сторінок

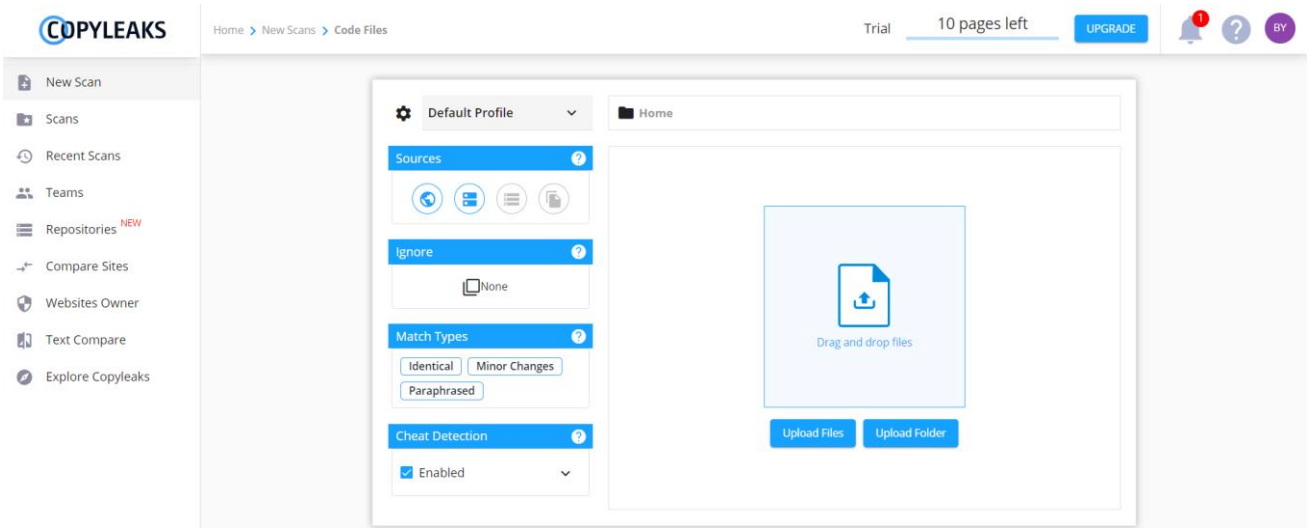


Рисунок 1.5 – Вікно для виявлення плагіату програмного коду

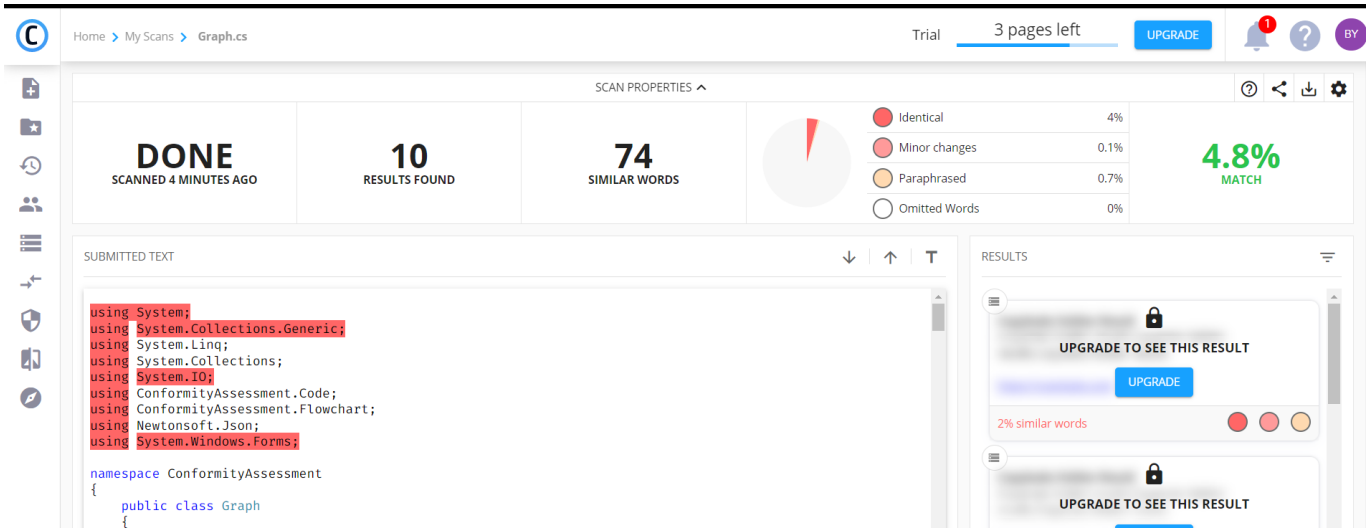


Рисунок 1.6 – Виявлення плагіату у програмному коді на мові C#

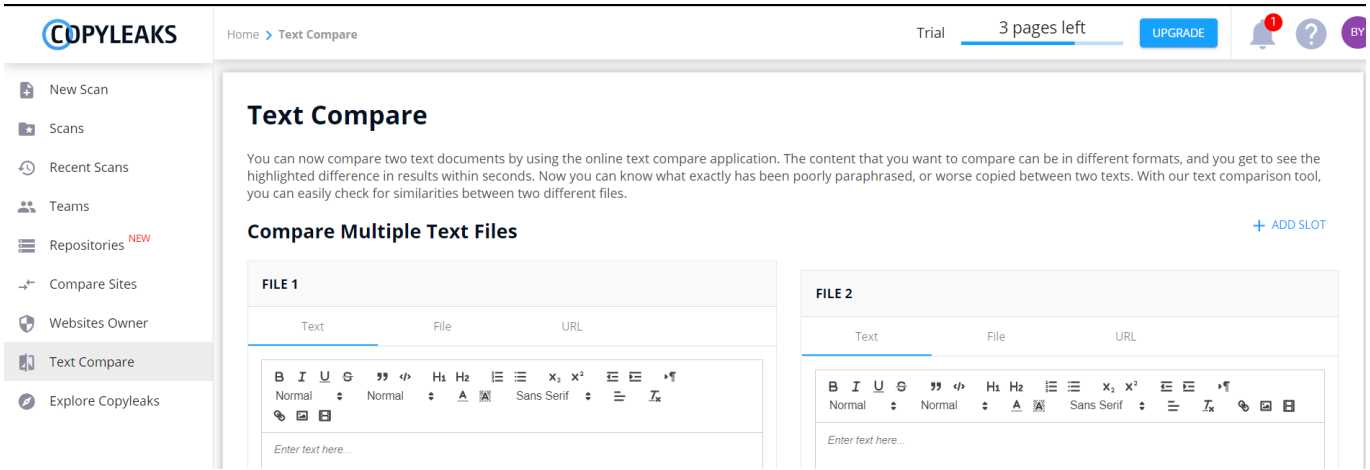


Рисунок 1.7 – Вікно для порівняння текстів між собою

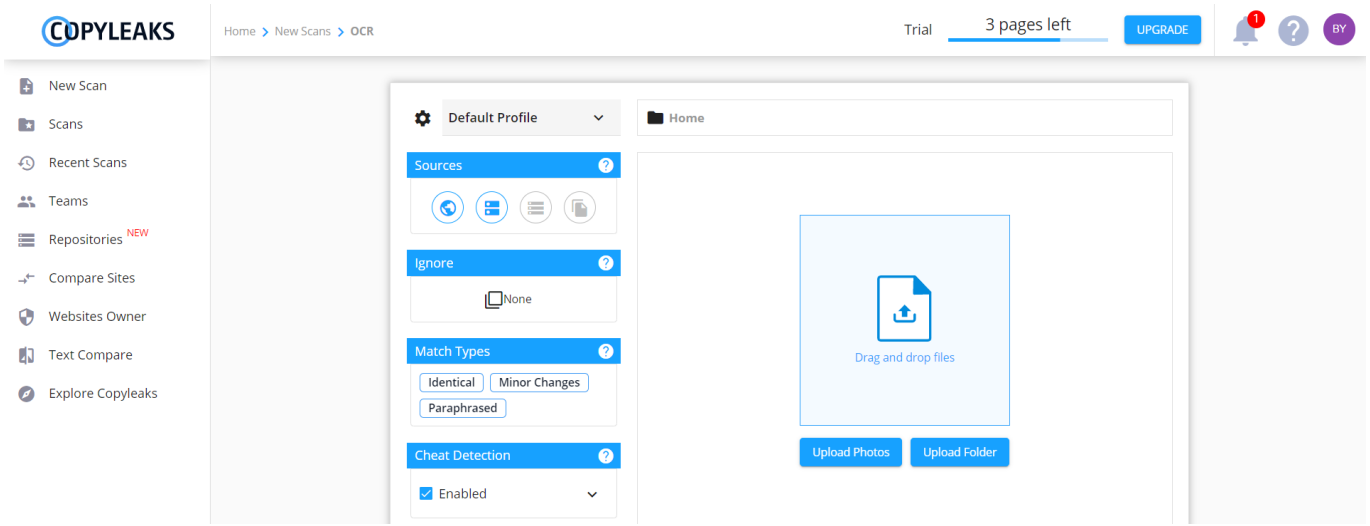


Рисунок 1.8 – Вікно для порівняння текстів з зображенням між собою

До переваг даного програмного продукту можна віднести:

- великий функціонал з виявлення плагіату;
- зручний та інтуїтивно зрозумілий інтерфейс;
- наявність перекладу інтерфейсу на різні мови.

До недоліків даного програмного продукту можна віднести:

- велика вартість програмного додатку;
- досить обмежена безкоштовна ознайомча версія програмного додатку.

1.5 Огляд літератури

1.5.1 Конструктори і композитні конструктори

Поняття узагальненої конструктивно-продукційної структури [38] або узагальненого конструктора (УК) є основою для конструктивно-продукційного моделювання. Даний механізм був створений в результаті дослідження та аналізу різних граматичних структур та систем: графічних [39], індексних [39, 40], програмних [39, 40], стохастичних [41], атрибутивних [42] та ін., а також R- та L-систем [43].

Особливістю даного підходу є використання атрибутивних множин:

- елементів носія (може бути розширена);
- операцій, пов’язаних з алгоритмами виконання та виконавцем;
- конструкцій.

Узагальненим конструктором є трійка [38]:

$$C_G = \langle M, \Sigma, \Lambda \rangle, \tag{1.1}$$

де M – неоднорідний носій, який включає конструктивні елементи з атрибутами та може розширюватися, Σ – сигнатура операцій (і відповідних відношень) зв’язування, підстановки і виводу, операцій над атрибутами, Λ – множина тверджень інформаційного забезпечення конструювання (ІЗК). ІЗК (конструктивна аксіоматика) включає онтологію, мету, правила, обмеження, початкові умови та умови завершення конструювання.

Наявність набору атрибутів w_i у елемента носія m_i позначається як ${}_{w_i}m_i$, а те, що w_{i_j} є атрибутом елемента m_i – $w_{i_j} \leftlbrace m_i$. Набір атрибутів є кортежем довжини k $w_i = \langle w_{i_1}, w_{i_2}, \ldots, w_{i_k} \rangle \in W_1 \times W_2 \times \ldots \times W_k = W$. Носій – $M = \{ {}_{w_i}m_i \}$.

Сигнатура складається з імен операцій, які мають набір атрибутів.

Довільна операція сигнатури представляється як ${}_{v}\circ \in \Sigma$, де $v = \langle v_1, v_2, \ldots, v_k \rangle \in V_1 \times V_2 \times \ldots \times V_k = V$. Сигнатура Σ складається з множин операцій: Ξ – зв'язування, Θ – виведення, Φ – операцій над атрибутами і правил продукцій – Ψ. $\Sigma = \langle \Xi, \Psi, \Theta, \Phi \rangle$, де правило – набір операцій, які виконуються в певній послідовності.

Згідно онтології УК, формою ${}_{w}l$ з атрибутом w називається набір терміналів і нетерміналів, що об'єднуються операціями зв'язування:

${}_{w_l}l = {}_{w_0}\otimes\left({}_{w_1}m_1, {}_{w_2}m_2, \ldots, {}_{w_k}m_k \right)$ для $\forall {}_{w_i}m_i \in M$, де $\otimes$ – будь-яка операція зв'язування з Σ;

${}_{w_l}l = {}_{w_j}m_j$, якщо $l = {}_{w_0}\otimes\left(\varepsilon, \ldots, \varepsilon, {}_{w_j}m_j, \varepsilon, \ldots, \varepsilon \right)$;

${}_{w_l}l = {}_{w_0}\otimes\left({}_{w_1}l_1, {}_{w_2}l_2, \ldots, {}_{w_k}l_k \right)$, якщо ${}_{w_1}l_1, {}_{w_2}l_2, \ldots, {}_{w_k}l_k$ – форми.

Конструкцією називається форма, яка містить тільки термінали [38].

Правила підстановки мають вигляд $\psi_r : \langle s_r, g_r \rangle \in \Psi$, де s_r – відношення підстановки, g_r – набір операцій над атрибутами. Відношення підстановки – двомісне відношення з атрибутами ${}_{w_i}l_i \quad {}_{w_p} \rightarrow {}_{w_j}l_j$ [38]. Для форми ${}_{w_l}l = {}_{w_0}\otimes\left({}_{w_1}l_1, {}_{w_2}l_2, \ldots, {}_{w_h}l_h, \ldots, {}_{w_k}l_k \right)$ і доступного відношення підстановки ${}_{w_p}\left({}_{w_h}l_h, {}_{w_q}l_q \right)$ (може бути записане як ${}_{w_h}l_h \rightarrow {}_{w_q}l_q$) такого, що ${}_{w_h}l_h \prec {}_{w_l}l$ (${}_{w_h}l_h$ є частиною ${}_{w_q}l_q$), результатом тримісної операції підстановки ${}_{w_l^*}l^* = {}_{w_p}\rightarrow\left({}_{w_h}l_h, {}_{w_q}l_q, {}_{w_l}l \right)$ буде форма ${}_{w_l^*}l^* = {}_{w_0}\otimes\left({}_{w_1}l_1, {}_{w_2}l_2, \ldots, {}_{w_q}l_q, \ldots, {}_{w_k}l_k \right)$ [38]. Послідовність відношень підстановки s_n будемо записувати як

$s_n = \langle l_i \to l_j, \ldots, l_m \to l_k \rangle$, де l_i, l_j, l_m, l_k – сентенціальні форми. Відношення підстановки може бути записано в скороченій формі $s_m = \langle l_i \to l_j \mid l_k \rangle$, де l_i, l_j, l_k – форми; що еквівалентно $s_m = \langle l_i \to l_j \rangle$, $s_n = \langle l_i \to l_k \rangle$.

Для заданої форми ${}_{w_l} l = {}_{w_0} \otimes ({}_{w_1} l_1, {}_{w_2} l_2, \ldots, {}_{w_h} l_h, \ldots, {}_{w_k} l_k)$ і доступного відношення підстановки ${}_{w_p} \to \left({}_{w_h} l_h, {}_{w_q} l_q \right)$ такого, що ${}_{w_h} l_h$ – підформа ${}_{w_l} l$ (${}_{w_h} l_h \prec {}_{w_l} l$), результатом тримісної операції підстановки ${}_{w_l^*} l^* = {}_{w_p} \Rightarrow \left({}_{w_h} l_h, {}_{w_q} l_q, {}_{w_l} l \right)$ буде форма ${}_{w_l^*} l^* = {}_{w_0} \otimes \left({}_{w_1} l_1, {}_{w_2} l_2, \ldots, {}_{w_q} l_q, \ldots, {}_{w_k} l_k \right)$, де $\Rightarrow \in \Theta$.

Операція часткового виведення ($|\Rightarrow (\Psi, {}_{w_i} l_i)$) полягає у:

- виборі одного з доступних правил підстановки ${}_{\vec{d}_r} \psi_r : \langle s_r, g_r \rangle \in \Psi$ з відношеннями підстановки s_r та виконанні на його основі операцій підстановки, де $\vec{d}$ – вектор доступності правил: якщо $d_r = 1$ правило доступне, якщо $d_r = 0$ – правило не доступне;
- виконанні операцій над атрибутами g_r.

Операція виведення ($\|\Rightarrow (\Psi, {}_{w_l} l)$, $\|\Rightarrow \in \Theta$) полягає у покроковому перетворенні форм, починаючи з початкового нетермінала і закінчуючи конструкцією, що задовольняє умові закінчення виведення. Таке перетворення передбачає циклічне виконання операцій часткового виведення. Умовою закінчення виведення є відсутність нетерміналів в формі. Перетворення конструктору дозволяють застосовувати його для різних предметних областей, що робить його досить універсальним.

Призначення конструктору полягає у формуванні множин конструкцій за допомогою операцій сигнатури, що задаються правилами ІЗК.

Для формування конструкцій необхідно виконувати ряд уточнюючих перетворень конструктору [38]:

- спеціалізацію ($_S\mapsto$) – визначення предметної області застосування конструктора;
- інтерпретацію ($_I\mapsto$) – зв'язування операцій сигнатури з алгоритмами виконання деякої алгоритмічної структури;
- конкретизацію ($_K\mapsto$) – розширення ІЗК множиною правил продукцій, завдання конкретних множин нетермінальних і термінальних символів з їх атрибутами і, при необхідності, значень атрибутів;
- реалізацію ($_R\mapsto$) – послідовне виконання операції виводу для побудови конструкцій.

Виділяють різні типи конструкторів залежно від їх функцій, а саме: перетворювач , породжувач та розпізнавач.

1.5.2 Граф потоку керування

Граф потоку керування (Control Flow Graf, CFG) [12] – є загальноприйнятим засобом візуального представлення множини всіх можливих шляхів виконання програми у вигляді графу.

У графі потоку керування кожний вузол (вершина) відповідає базовому блоку – прямолінійній ділянці коду, що не містить в собі ні операцій передачі керування, ні точок, на яке керування передається з інших частин програми. Є лише два винятки: точка, на яку виконується перехід, є першою інструкцією в базовому блоці, і базовий блок, що завершується інструкцією переходу. Спрямовані дуги використовуються в графі для представлення інструкцій переходу. Також в більшості реалізацій додано два спеціалізованих блоки: вхідний блок, через який керування входить в граф і вихідний блок, який завершує всі шляхи в даному графі.

Структура CFG вкрай важлива для багатьох оптимізацій компіляторів і для утиліт статичного аналізу коду. За допомогою графу керування можна визначати недосяжні фрагменти коду, деякі види зациклення програм, можливість перегрупування операторів для використання можливостей процесора з оптимізації

кешування пам’яті. Також граф керування може використовуватися під час контролю коректності оптимізуючих перетворень і при процедурному аналізі.

На рисунку 1.9 зображено макет графа потоку керування

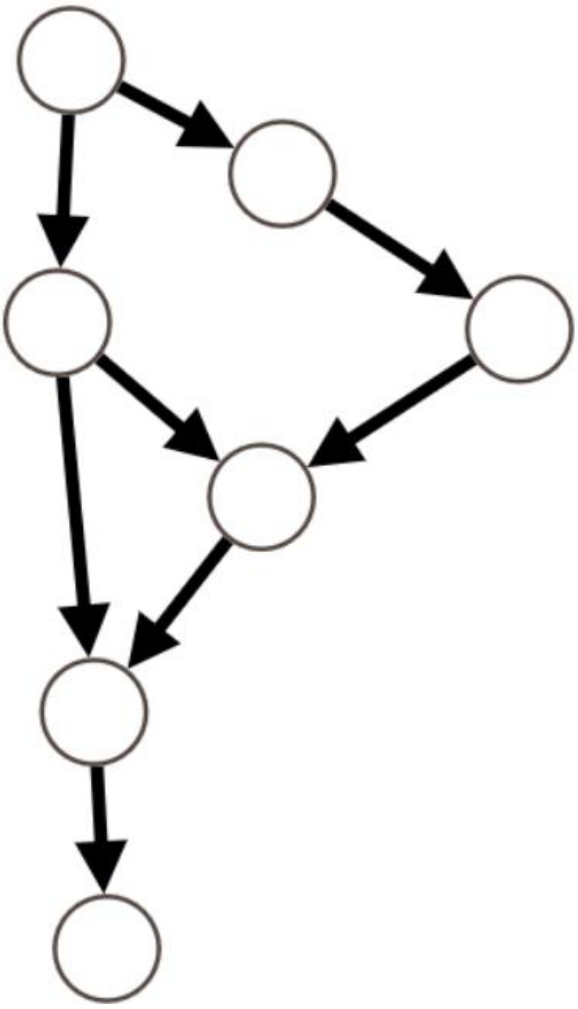


Рисунок 1.9 – Макет графу потоку керування

Висновки до першого розділу

В даному розділі були підняті питання стосовно поставленої мети, її доцільності та наукової новизни. Також були визначені основні інструменти та початкові вимоги до майбутнього програмного додатку. Було знайдено та проаналізовано наукові роботи та дослідження, які є близькими до досліджуваної області. Було проведено пошук та огляд програмних аналогів. На основі досліджень в області розпізнавання образів на цифровому зображенні, було прийнято рішення розглядати блок-схему у форматі JSON, оскільки алгоритми для роботи з зображення дуже трудомісткі.

2 ОБГРУНТУВАННЯ МЕТОДУ ВИЗНАЧЕННЯ ВІДПОВІДНОСТІ ТЕКСТУ ПРОГРАМИ ГРАФІЧНОМУ ПРЕДСТАВЛЕННЮ АЛГОРИТМУ

Вивчення, побудова та реалізація алгоритмів є невід'ємною частиною навчального процесу студентів спеціальності «Інженері програмного забезпечення». Однак розроблений алгоритм не завжди відповідає кінцевій реалізації, тому студентам для самоконтролю, а викладачам для оцінювання якості роботи студента необхідна перевірка відповідності написаного програмного коду розробленому алгоритму.

Для розробки методу визначення відповідності тексту програми графічному представленню алгоритму доцільним є використання апарату конструктивно-продукційного моделювання, що ґрунтується на застосуванні конструктивних властивостях формальних граматик. Апарат надає такі переваги:

- динамічний, неоднорідний носій, що дозволяє описати будь-які об'єкти та їх властивості;
- наявність не лише операцій, а й алгоритмів їх виконання, що дозволяє написати модель виконавця, яка в подальшому може бути реалізована програмно.

Використання конструктивно-продукційного підходу до розробки даного методу дасть подальший розвиток методам конструктивно-продукційного моделювання в задачах обробки текстів, написаних штучними мовами.

2.1 Методика визначення відповідності тексту програми графічному представленню алгоритму

Для визначення відповідності тексту та алгоритму програми пропонується метод на основі конструктивно-продукційного моделювання. Метод має такі складові-кроки:

1. Попередня обробка тексту програми, що передбачає видалення не значимих частин коду (коментарі, пробіли тощо) та його розбиття на лексеми;
2. Проміжне представлення тексту програми, отриманого на попередньому кроці, у вигляді списку керуючих елементів (реалізують алгоритмічні

структури вибору та циклу, а також операції передачі керування) та елементів процесу (відповідають алгоритмічній конструкції слідування);

3. Побудова графу керування програми за списком керуючих елементів;
4. Побудова графу керування за алгоритмом програми, що представлений блок-схемою;
5. Зіставлення графів керування програми та алгоритму шляхом обходу в ширину [44].

2.1.1 Конструктивно-продукційна модель списку керуючих елементів

Спеціалізований конструктор для перетворення програмного коду програми у проміжне представлення у вигляді списку має вигляд:

$$C = \langle M, \Sigma, \Lambda \rangle_S \mapsto C_T = \langle M_T, \Sigma_T, \Lambda_T \rangle, \quad (2.1)$$

де M_T – неоднорідний розширюваний носій, що складається з термінальних та нетермінальних елементів, Σ_T – множина операцій і відношень на елементах M_T, Λ_T – ІЗК, що включає онтологію, мету, обмеження, правила, умови початку і завершення конструювання.

Онтологія конструктора C_T. Розширюваний носій складається з множини термінальних та нетермінальних елементів $M_T = \{T_T \cup N_T\}$. Терміналами є $T_T = \{txt, list, \{P_i\}, \{O_i\}, \{Ob_i\}, \{L_i\}, \varepsilon\}$, де $txt \supset \{Ob_i\} \cup \{O_i\} \cup \{P_i\}$ – послідовність операторів програмного коду, $list$ – конструкція, що містить послідовність операторів програмного коду та допоміжних лексем, P_i – позначення процесу, що відповідає будь-якій послідовності операторів, що реалізує алгоритмічну структуру «слідування», $\{O_i\} = \{return, \ break, continue\}$ – множина керуючих операторів без тіла, $\{Ob_i\} = \{if, else, do, while, for, switch, case, default\}$ – множина керуючих операторів, що можуть мати власне тіло, $\{L_i\} = \{"++", "--"\}$ – множина допоміжних лексем для заповнення списку, ε – порожній елемент (символ).

Розглянемо сигнатуру:

$$\Sigma_T = \langle \Theta_T, \{\rightarrow\}, \{+\} \rangle \cup \Psi_G, \tag{2.2}$$

де $\Theta_T = \{\Rightarrow, |\Rightarrow, ||\tilde{\Rightarrow}\}$ – множина операцій виводу, $\rightarrow$ – відношення підстановки, Ψ_G – множина правил продукції виводу $\psi_i = \langle \hat{s}_i, \breve{s}_i \rangle$, де i номер правила, $\hat{s}_i$ – правило підстановки для розпізнавання тексту програмного коду, $\breve{s}_i$ – правило підстановки для побудови стеку, $+(el, list)$ – операція додавання нової вершини до списку, де el – значення нової вершини, $list$ – список.

Метою конструювання є побудова проміжного представлення програмного коду у вигляді списку.

Обмеження конструктору C_T накладаються конструкцією програмного коду.

Початкова умова конструювання: η – нетермінал, з якого починається побудова конструкції списку, α – нетермінал, з якого починається розпізнавання тексту програми.

Умова завершення конструювання: форма не містить нетерміналів, побудована конструкція списку відповідає програмного коду.

В результаті конкретизації конструктора $C_{T\ K \mapsto K} C_T$ маємо такі правила підстановки:

$$\hat{s}_0 = \langle \alpha \rightarrow \gamma\alpha \mid \beta\alpha \rangle. \tag{2.3}$$

$$\hat{s}_1 = \langle \beta \rightarrow P_1 \mid P_2 \ldots P_n \rangle, \tag{2.4}$$

$$\breve{s}_1 = \langle \eta \rightarrow +(p, list)\eta \rangle.$$

$$\hat{s}_2 = \hat{s}_3 = \langle \gamma \rightarrow C_1 \mid C_2 \ldots C_n \rangle, \tag{2.5}$$

$$\breve{s}_2 = \langle \eta \rightarrow +(Ob_i, list) + ("++", list)\eta + ("--", list)\eta \rangle.$$

$$\breve{s}_3 = \langle \eta \rightarrow +(O_i, list)\eta \rangle. \tag{2.6}$$

$$\breve{s}_4 = \langle \eta \to \varepsilon \rangle. \tag{2.7}$$

В ході інтерпретації виконаємо зв'язування операцій сигнатури Σ_T з алгоритмами їх виконання:

$$\langle C_T = \langle M_T, \Sigma_T, \Lambda_T \rangle, C_A = \langle M_A, \Sigma_A, \Lambda_A \rangle \rangle \; {}_I\mapsto {}_{I,C_A} C_T = \langle M_T, \Sigma_T, \Lambda_1 \rangle, \tag{2.8}$$

де $M_A \supset V_A, V_A = \{A_i^0 \mid_{X_i}^{Y_i}\}$ – множина базових алгоритмів, X_i, Y_i – множини визначення та значень алгоритму $A_i^0 \mid_{X_i}^{Y_i}$, $\Lambda_A = \left\{ M_A = \bigcup\limits_{A_i^0 \in V_A} \left(X\left(A_i^0\right) \subset Y\left(A_i^0\right) \right) \cup \Omega\left(C_T\right) \right\}$ – неоднорідний носій, $\Omega\left(C_T\right)$ – множина конструкцій списків, які задовольняють C_T. $\Lambda_1 = \Lambda_T \cup \Lambda_A \cup \Lambda_2$.

Конструктор ${}_{I,C_A} C_T$ включає алгоритми виконання операцій: $\Lambda_2 = \left\{ \left(A_1^0 \mid_{A_i, A_j}^{A_i \cdot A_j} \hookleftarrow \cdot \right), \quad \left(A_2^0 \mid_{S}^{A_1} \hookleftarrow : \right), \quad \left(A_3 \mid_{el, list}^{list} \hookleftarrow + \right), \quad \left(A_4 \mid_{l_h, l_q, f_i}^{f_i} \hookleftarrow \Rightarrow \right), \quad \left(A_5 \mid_{f_i, \Psi}^{f_j} \hookleftarrow \mid \Rightarrow \right), \left(A_6 \mid_{\sigma, \Psi}^{\bar{\Omega}} \hookleftarrow \| \Rightarrow \right) \right\}$.

Результатом реалізації конструктора (2.8) є множина конструкцій списків, які є проміжним представленням тексту програми та відповідному йому графу керування.

2.1.2 Конструктивно-продукційна модель списку керуючих елементів

Спеціалізований конструктор для побудови графу керування програми за її проміжним представлення у вигляді списку керуючих операторів має вигляд:

$$C = \langle M, \Sigma, \Lambda \rangle \; {}_S\mapsto C_G = \langle M_G, \Sigma_G, \Lambda_G \rangle, \tag{2.9}$$

де M_G – неоднорідний розширюваний носій, Σ_G – множина операцій і відношень на елементах M_G, Λ_G – ІЗК.

Онтологія конструктора C_G. Розширюваний носій складається з множини термінальних та нетермінальних елементів $M_G = \{T_G \cup N_G\}$. Терміналами є конструкції графів та їх складові, проміжного представлення програмних кодів у вигляді списків та допоміжних конструкцій:

$$T_G = \{G \cup V \cup E \cup list \cup stack\_temp \cup stack\_root \cup stack\_break \cup \cup stack\_next \cup stack\_continue \cup stack\_switch \cup temp \cup l\_vertex\}, \quad (2.10)$$

де $G = \langle V, E \rangle$ – конструкція графу, $V = \{v_i\}$, $E = \{e_i\}$ – множина вершин та дуг відповідно, $list$ – конструкція побудована C_T, допоміжні конструкції-стеки вершин графу з навантаженням: у вигляді назв керуючих операторів – $stack\_temp$, які є коренем під графу, що утворюється при роботі з операторами $\{if, else, while, for, case\}$ – $stack\_root$; "$break$" – $stack\_break$; "$continue$" – $stack\_continue$; які мають спільні дуги з вершиною, що буде додана – $stack\_next$; "$case$" – $stack\_switch$; $temp$ зберігає вершину допоміжних стеків, l_vertex зберігає вершину $list$. Допоміжні конструкції стеків є впорядкованим набором елементів, що формуються за принципом LIFO.

Вершина графу має атрибути $w_v \hookleftarrow v = \langle index, name \rangle$, де $index$ – індекс вершини, приймає цілочислені значення, $name$ – навантаження вершини. Атрибути дуги $w_e \hookleftarrow e = \langle start, finish \rangle$, де $start, finish$ – цілочислені індекси інцендентних вершин.

Граф має атрибути $w_G = \langle begin, end, current_v, current_e \rangle$, де $begin$ – початкова вершина графу, end – кінцева вершина графу, $current_v = \langle name, index, prev \rangle$ – поточна вершина при формуванні графу, де $prev$ – попередня вершина графу, $current_e = \langle start, finish \rangle$ – поточна дуга при формуванні графу.

Атрибути вершини проміжного представлення програмного коду l_vertex – $w_l = \langle name, prev \rangle$, де $name$ – навантаження вершини списку, $prev$ – навантаження попередньої вершини списку.

Атрибути вершини допоміжного стеку $temp$ – $w_{ST} = w_v$.

Розглянемо сигнатуру Σ_G:

$$\Sigma_G = \langle \Xi_G, \Theta_G, \Phi_G, \{\rightarrow\} \rangle \cup \Psi_G, \tag{2.11}$$

де $\Xi_G = \left\{ \tilde{\cup}, \cup \right\}$ – множина операцій зв'язування, $\Theta_G = \{\Rightarrow, |\Rightarrow, ||\tilde{\Rightarrow}\}$ – множина операцій виводу, $\Phi_G = \{\div, :=, \%, \#, +, -, --, \times\}$ – множина операцій над атрибутами, $\rightarrow$ – відношення підстановки.

Розглянемо операції над атрибутами:

- $\div(c, n, L)$ – виконання n операцій із списку L, якщо $c = true$;
- $a := b$ – присвоєння, копіює значення операнду b в a;
- $\%(index, V)$ – знаходження навантаження вершини, індекс якої дорівняє $index$, у множині вершин V;
- $\#Q$ – обчислення потужності множини, визначає число, яке дорівнює кількості елементів в Q;
- $+(el, stack)$ – додавання елементу el в допоміжний стек $stack$;
- $-(el, stack)$ – вилучення вершини допоміжного стеку $stack$ в el при умові, що $stack$ не порожній;
- $--(el, list)$ – вилучення вершини списку $list$ та збереженні її в el при умові, що $list$ не порожній;
- $\times(a, b)$ – множення двох чисел, передбачає знаходження третього числа, що є їхнім добутком.

Операція об'єднання графів $w_G \hookleftarrow G = = \tilde{\cup}(w_1 \hookleftarrow G_1, w2 \hookleftarrow G_2)$ передбачає формування нового графа $w_G \hookleftarrow G$, що включає об'єднані множини вершин і дуг вихідних графів $w_G \hookleftarrow G = \langle V, E \rangle$, де $V = V_1 \cup V_2, E = E_1 \cup E_2$, $w_1 \hookleftarrow G_1 = \langle V_1, E_1 \rangle$, $w_2 \hookleftarrow G_2 = \langle V_2, E_2 \rangle$, при цьому $\cup$ – традиційна операція об'єднання.

Відношення підстановки має вигляд:

$$\psi_i = \langle s_i, g_i \rangle,\ s_i = \langle \overline{s}_i, \tilde{s}_i \rangle,\ g_i = \langle \overline{g}_i, \tilde{g}_i \rangle, \tag{2.12}$$

де $\overline{s}_i, \tilde{s}_i$ – відношення підстановки для роботи зі списком і побудови конструкції графу відповідно, $\overline{g}_i, \tilde{g}_i$ – операції над атрибутами списку та графа, його вершин і дуг відповідно. У разі якщо операція над атрибутами не виконується, відношення підстановки має вигляд $\psi_i = \langle s_i, \varepsilon \rangle$.

Операція повного виводу $\|\tilde{\Rightarrow}(\Psi, w_l \lrcorner l)$ та більш детальна інформація щодо інших операцій наведена у роботі [12, 13]. Результатом операції виводу є конструкції-граф.

Метою конструювання є побудова графу керування програми за проміжним представленням у вигляді $list \in \Omega(C_T)$.

Обмеження конструктору C_G накладаються виходячи з навантаження вершин у списку.

Початкова умова конструювання: σ – нетермінал, з якого починається побудова конструкції графу керування, ρ – нетермінал, з якого починається вилучення вершин з $list \lrcorner C_T$.

Умова завершення конструювання: форма не містить нетерміналів, конструкція списку порожня.

В результаті конкретизації конструктора $C_G\ {}_K\mapsto {}_K C_G$ маємо нижченаведені правила підстановки.

Правило ініціалізації графу має вигляд:

$$\overline{s}_1 = \varepsilon,\ \overline{g}_1 = \varepsilon,\ \tilde{s}_1 = \langle \sigma \to G\alpha \rangle, \tag{2.13}$$

$$\tilde{g}_1 = \langle index \lrcorner v_1 := 1, name \lrcorner v_1 := "begin", index \lrcorner v_2 := 0, name \lrcorner v_2 := "end", begin \lrcorner G := v_1,$$

$$end \lrcorner G := v_2, current \lrcorner G := v_1, V = \{v_1, v_2\} \rangle.$$

Побудова графу передбачає вилучення вершини конструкції *list* ($\overline{s}_2$), тому далі, для правил $G\alpha$ та $G\beta$ якщо $\overline{s}_i$ або $\overline{g}_i$ не вказано, то $\overline{s}_i = \overline{s}_2$, $\overline{g}_i = \overline{g}_2$ відповідно. А для правил $G\phi$, $G\beta$ та $G\sigma$, якщо $\overline{s}_i$ або $\overline{g}_i$ не вказано, то $\overline{s}_i = \varepsilon$, $\overline{g}_i = \varepsilon$.

$$\overline{s}_2 = \left\langle \rho \to --\left(l\_vertex, list\right)\right\rangle,\ \overline{g}_2 = \varepsilon. \tag{2.14}$$

Правило для додавання нових вершин до графу має вигляд:

$$\tilde{s}_2 = \left\langle G\alpha\ {}_{d_1} \to \tilde{\cup}\left(G, G^*\right)\alpha \right\rangle, \tag{2.15}$$

$$\tilde{g}_2 = \Big\langle \div\Big(name\hookleftarrow l\_vertex = P \mid el \in \left\{Cb_i\right\} \setminus \left\{"do","else"\right\}, 9, d_1 := true, G^* = V^* \cup E^*,$$

$$V^* = \left\{v\right\}, E^* = \left\{e\right\}, name\hookleftarrow v := name\hookleftarrow l\_vertex, index\hookleftarrow v := \#G, start\hookleftarrow e := index\hookleftarrow current\hookleftarrow G,$$

$$finish\hookleftarrow e := index\hookleftarrow v, current\hookleftarrow G := v\Big)\Big\rangle.$$

Далі наведемо правила для додавання вершин до конструкції *stack_temp*:

вершина "*do*":

$$\tilde{s}_3 = \left\langle G\alpha\ {}_{d_2} \to G\alpha \right\rangle, \tag{2.16}$$

$$\tilde{g}_3 = \Big\langle \div\Big(name\hookleftarrow l\_vertex = "++"\&\ name\hookleftarrow prev\hookleftarrow$$

$$\hookleftarrow l\_vertex = "do", 2, d_2 := true, +\left(-1\times\left(\#G+1\right), stack\_temp\right)\Big\rangle.$$

вершина "*else*":

$$\tilde{s}_4 = \left\langle G\alpha\ {}_{d_3} \to G\alpha \right\rangle, \tag{2.17}$$

$$\tilde{g}_4 = \Big\langle \div\Big(name\hookleftarrow l\_vertex = "++"\&\ name\hookleftarrow prev\hookleftarrow$$

$$\hookleftarrow l\_vertex = "else", 2, d_3 := true, +\left(-1, stack\_temp\right)\Big\rangle.$$

інші вершини:

$$\tilde{s}_5 = \left\langle G\alpha \ {}_{d_4} \to G\alpha \right\rangle, \tag{2.18}$$

$$\tilde{g}_5 = \left\langle \div\left(name\text{↵}l\_vertex = "++"\& name\text{↵}prev\text{↵}\right.\right.$$

$$\left.\left.\text{↵}l\_vertex != el \in \{"do","else"\}, 2, d_4 := true, +\left(index\text{↵}current\text{↵}G, stack\_temp\right)\right)\right\rangle.$$

Далі наведемо правила для обробки різних вершин конструкцій:

"*return*" конструкції *list*:

$$\tilde{s}_6 = \left\langle G\alpha \ {}_{d_5} \to \tilde{\cup}\left(G, G^*\right)\alpha \right\rangle, \tag{2.19}$$

$$\tilde{g}_6 = \left\langle \div\left(name\text{↵}l\_vertex = "return", 5, d_5 := true,\right.\right.$$

$$\left.\left. G^* = E^*, E^* = \{e\}, start\text{↵}e_i := index\text{↵}current\text{↵}G,\ finish\text{↵}e_i := index\text{↵}end\text{↵}G\right)\right\rangle.$$

"*break*" конструкції *list*:

$$\tilde{s}_7 = \left\langle G\alpha \ {}_{d_6} \to G\alpha \right\rangle, \tag{2.20}$$

$$\tilde{g}_7 = \left\langle \div\left(name\text{↵}l\_vertex = "break", 2, d_6 := true, +\left(index\text{↵}current\text{↵}G, stack\_break\right)\right)\right\rangle.$$

"*continue*" конструкції *list*:

$$\tilde{s}_8 = \left\langle G\alpha \ {}_{d_7} \to G\alpha \right\rangle, \tag{2.21}$$

$$\tilde{g}_8 = \left\langle \div\left(name\text{↵}l\_vertex = "continue", 2, d_7 := true, +\left(index\text{↵}current\text{↵}G, stack\_continue\right)\right)\right\rangle.$$

"−−" конструкції *list*:

$$\tilde{s}_9 = \left\langle G\alpha \ {}_{d_8} \to G\phi\alpha \right\rangle, \tag{2.22}$$

$$\tilde{g}_9 = \left\langle \div\left(name\hookleftarrow l\_vertex = "--", 1, d_8 := true\right)\right\rangle.$$

"*else*" конструкції *stack _ temp*:

$$\tilde{s}_{10} = \left\langle G\phi \ {}_{d_9} \to G\phi \right\rangle, \tag{2.23}$$

$$\tilde{g}_{10} = \left\langle -(temp, stack\_temp), \div\left(temp = -1, 2, d_8 := true, +\left(index\hookleftarrow current\hookleftarrow G, stack\_root\right)\right)\right\rangle.$$

"*do*" конструкції *stack _ temp*:

$$\tilde{s}_{11} = \left\langle G\phi \ {}_{d_{10}} \to \tilde{\cup}\left(G\beta, G^*\right)\chi\phi \right\rangle, \tag{2.24}$$

$$\tilde{g}_{11} = \left\langle -(temp, stack\_temp), \div\left(temp < -1, 5, d_{10} := true, G^* = E^*, E^* = \{e\},\right.\right.$$

$$\left.\left. start\hookleftarrow e_i := index\hookleftarrow current\hookleftarrow G,\ finish\hookleftarrow e_i := temp \times -1\right)\right\rangle.$$

Правило для додавання нової вершини до графу має вигляд:

$$\tilde{s}_{12} = \left\langle G\beta \to \tilde{\cup}\left(G, G^*\right)\right\rangle, \tag{2.25}$$

$$\tilde{g}_{12} = \left\langle G^* = V^* \cup E^*, V^* = \{v\}, E^* = \{e\}, name\hookleftarrow v := name\hookleftarrow l\_vertex, index\hookleftarrow v := \#G,\right.$$

$$\left. start\hookleftarrow e := index\hookleftarrow current\hookleftarrow G,\ finish\hookleftarrow e := index\hookleftarrow v, current\hookleftarrow G := v\right\rangle.$$

Правило для обробки вершин "*while*" та "*for*" конструкції *stack _ temp* має вигляд:

$$\tilde{s}_{13} = \left\langle G\phi_{\ d_{11}} \to \tilde{\cup}\left(G, G^*\right)\chi\phi \right\rangle, \tag{2.26}$$

$$\tilde{g}_{13} = \Big\langle -(temp, stack\_temp), \div\left(\%\left(temp, V\right) = "while" \,|\, " for", 6, d_{11} := true, G^* = E^*, E^* = \{e\},\right.$$

$$\left. start \hookleftarrow e_i := index \hookleftarrow current \hookleftarrow G,\ finish \hookleftarrow e_i := temp, +(temp, stack\_root)\right)\Big\rangle.$$

Далі представлено правила для обробки вершин конструкцій $stack\_root$ та $stack\_continue$ в правилі $\tilde{s}_{13}$.

$$\tilde{s}_{14} = \left\langle G\chi \to \tilde{\cup}\left(G, G^*\right)\chi \right\rangle, \tag{2.27}$$

$$\tilde{g}_{14} = \Big\langle G^* = E^*, E^* = \{e\}, -(temp, stack\_root),$$

$$start \hookleftarrow e := temp,\ finish \hookleftarrow e := finish \hookleftarrow current_e \hookleftarrow G\Big\rangle.$$

$$\tilde{s}_{15} = \left\langle G\chi \to \tilde{\cup}\left(G, G^*\right)\chi \right\rangle, \tag{2.28}$$

$$\tilde{g}_{15} = \Big\langle G^* = E^*, E^* = \{e\}, -(temp, stack\_continue), start \hookleftarrow e := temp,$$

$$finish \hookleftarrow e := finish \hookleftarrow current_e \hookleftarrow G \,|\, start \hookleftarrow current_e \hookleftarrow G\Big\rangle.$$

$$\tilde{s}_{16} = \left\langle \chi \to \varepsilon \right\rangle. \tag{2.29}$$

Правила для обробки вершини "if" конструкції $stack\_temp$ має вигляд:

$$\tilde{s}_{17} = \left\langle G\phi_{\ d_{12}} \to G\phi \right\rangle, \tag{2.30}$$

$$\tilde{g}_{17} = \Big\langle -(temp, stack\_temp), \div\left(\%\left(temp, V\right) = "if ",\right.$$

$$\left. 3, d_{12} := true, +(temp, stack\_next), +(index \hookleftarrow current \hookleftarrow G, stack\_root)\right)\Big\rangle.$$

$$\tilde{s}_{18} = \left\langle G\phi_{\ d_{13}} \to G\phi \right\rangle, \tag{2.31}$$

$$\tilde{g}_{18} = \Big\langle -(temp, stack\_temp), \div\left(\%\left(temp, V\right) = "if ",\right.$$

$$3, d_{13} := true, +(temp, stack\_root), +(index \hookleftarrow current \hookleftarrow G, stack\_root))\rangle.$$

Далі представлено правило для обробки вершини "*case*" та "*default*" конструкції $stack\_temp$:

$$\tilde{s}_{19} = \langle G\phi\ _{d_{14}} \to G\phi \rangle, \tag{2.32}$$

$$\tilde{g}_{19} = \langle -(temp, stack\_temp), \div(\%(temp, V) =$$

$$= "case" | "default", 2, d_{14} := true, +(temp, stack\_switch))\rangle.$$

Правила для обробки вершини "*switch*" конструкції $stack\_temp$ має вигляд:

$$\tilde{s}_{19} = \langle G\phi\ _{d_{15}} \to G\delta\phi \rangle, \tag{2.33}$$

$$\tilde{g}_{19} = \langle -(temp, stack\_temp), \div(\%(temp, V) = "switch", 1, d_{15} := true)\rangle.$$

Правила для обробки вершин конструкцій $stack\_switch$ має наступний вигляд:

$$\tilde{s}_{20} = \left\langle G\delta \to \tilde{\cup}(G, G^*)\delta \right\rangle, \tag{2.34}$$

$$\tilde{g}_{20} = \langle G^* = E^*, E^* = \{e\}, -(temp, stack\_switch),$$

$$start \hookleftarrow e := start \hookleftarrow current_e \hookleftarrow G,\ finish \hookleftarrow e := temp)\rangle.$$

$$\tilde{s}_{21} = \langle \delta \to \varepsilon \rangle. \tag{2.35}$$

Правила для обробки вершини конструкції $stack\_break$ має вигляд:

$$\tilde{s}_{22} = \langle G\phi \to G\phi \rangle, \tag{2.36}$$

$$\tilde{g}_{22} = \langle -(temp, stack\_break), +(temp, stack\_next) \rangle .$$

$$\tilde{s}_{23} = \langle \phi \to \varepsilon \rangle . \tag{2.37}$$

Правило для обробки вершини конструкції $stack\_next$ має вигляд:

$$\tilde{s}_{24} = \left\langle G\alpha \to \tilde{\bigcup}\left(G, G^*\right)\alpha \right\rangle, \tag{2.38}$$

$$\tilde{g}_{24} = \langle G^* = E^*, E^* = \{e\}, -(temp, stack\_next),$$

$$start \hookleftarrow e := temp,\ finish \hookleftarrow e := index \hookleftarrow current_v \hookleftarrow G \rangle .$$

Правило для обробки вершини конструкції $stack\_root$ має вигляд:

$$\tilde{s}_{25} = \left\langle G\alpha \to \tilde{\bigcup}\left(G, G^*\right)\alpha \right\rangle, \tag{2.39}$$

$$\tilde{g}_{25} = \langle G^* = E^*, E^* = \{e\}, -(temp, stack\_root),$$

$$start \hookleftarrow e := temp,\ finish \hookleftarrow e := index \hookleftarrow current_v \hookleftarrow G \mid index \hookleftarrow end \hookleftarrow G \rangle .$$

Правила для додавання дуги до останньої вершини графу має вигляд:

$$\tilde{s}_{26} = \left\langle G\alpha \to \tilde{\bigcup}\left(G, G^*\right)\alpha \right\rangle, \tag{2.40}$$

$$\tilde{g}_{26} = \langle G^* = E^*, E^* = \{e\}, start \hookleftarrow e := index \hookleftarrow current_v \hookleftarrow G,\ finish \hookleftarrow e := index \hookleftarrow end \hookleftarrow G \rangle .$$

Наступне правило дозволяє завершити обробку вершин конструкції $stack$:

$$\tilde{s}_{27} = \langle \alpha \to \varepsilon \rangle . \tag{2.41}$$

В ході інтерпретації виконаємо зв'язування операцій сигнатури Σ_G з алгоритмами їх виконання:

$$\left\langle C_G = \left\langle M_G, \Sigma_G, \Lambda_G \right\rangle, C_{A'} = \left\langle M_{A'}, \Sigma_{A'}, \Lambda_{A'} \right\rangle \right\rangle_I \mapsto_I \mapsto_{I, C_{A'}} C_G = \left\langle M_G, \Sigma_G, \Lambda_3 \right\rangle \quad (2.42),$$

де $M_{A'} \supset V_{A'}, V_{A'} = \left\{ A_i^0 \mid_{X_i}^{Y_i} \right\}$ – множина базових алгоритмів, X_i, Y_i – множини визначення та значень алгоритму $A_i^0 \mid_{X_i}^{Y_i}$,

$\Lambda_{A'} = \left\{ M_{A'} = \bigcup\limits_{A_i^0 \in V_A} \left(X\left(A_i^0\right) \subset Y\left(A_i^0\right) \right) \bigcup \Omega\left(C_G\right) \right\}$ – неоднорідний носій, $\Omega\left(C_G\right)$ – множина конструкцій графів, які задовольняють C_G. $\Lambda_3 = \Lambda_G \bigcup \Lambda_{A'} \bigcup \Lambda_4$.

Конструктор ${}_{I, C_A} C_G$ включає алгоритми виконання операцій:

$\Lambda_4 = \left\{ \left(A_1^0 \mid_{A_i, A_j}^{A_i \cdot A_j} \hookleftarrow \cdot \right), \quad \left(A_2^0 \mid_S^{A_1} \hookleftarrow : \right), \quad \left(A_3 \mid_{l_h, l_q, f_i}^{f_i} \hookleftarrow \Rightarrow \right), \quad \left(A_4 \mid_{f_i, \Psi}^{f_j} \hookleftarrow \mid\Rightarrow \right), \quad \left(A_5 \mid_{\sigma, \Psi}^{\bar{\Omega}} \hookleftarrow \mid\mid\Rightarrow \right),$

$\left(A_6 \mid_{c,n,L}^{L} \hookleftarrow \div \right), \quad \left(A_7 \mid_{a,b}^{a} \hookleftarrow := \right), \quad \left(A_8 \mid_{index, V}^{name} \hookleftarrow \% \right), \quad \left(A_9 \mid_{Q}^{x} \hookleftarrow \# \right), \quad \left(A_{10} \mid_{el, stack}^{stack} \hookleftarrow + \right),$

$\left(A_{11} \mid_{el, stack}^{el} \hookleftarrow - \right), \left(A_{12} \mid_{el, list}^{el} \hookleftarrow -- \right), \left(A_{13} \mid_{a,b}^{c} \hookleftarrow \times \right), \left(A_{14} \mid_{Q_1, Q_2}^{Q} \hookleftarrow \bigcup \right), \left(A_{15} \mid_{G_1, G_2}^{G} \hookleftarrow \tilde{\bigcup} \right) \right\}$.

Результатом реалізації конструктора (2.42) є множина конструкцій графів керування, побудованими за конструкціями з $\Omega\left(C_T\right)$.

Таким чином, правила (2.13) – (2.41) дозволяють перетворити текст програми, попередньо побудований за правилами (2.3) – (2.7), у граф керування. При цьому враховані оператори керування $\{C_i\} \bigcup \{Cb_i\}$. Інші оператори мають уніфіковане представлення у вигляді вершини-процесу.

Висновки до другого розділу

В даному розділі було обґрунтовано напрям обраного дослідження та викладено загальна методика виконання наукового дослідження.

Були визначено два основних композитних конструктора, котрі використовуються для визначення відповідності тексту програми графічному представленню алгоритму, а саме:

- C_T – конструктор побудови проміжного представлення у вигляді списку керуючих елементів за програмним кодом на мові C++;
- C_G – конструктор побудови графу потоку керування за проміжним представленням у вигляді списку керуючих елементів.

Також було проведено зв’язування операцій сигнатури конструкторів з алгоритмами їх виконання.

3 ПРОЕКТУВАННЯ ТА РОЗРОБКА ІНСТРУМЕНТАЛЬНОГО ЗАБЕЗПЕЧЕННЯ ДЛЯ ВИЗНАЧЕННЯ ВІДПОВІДНОСТІ ТЕКСТУ ПРОГРАМИ ГРАФІЧНОМУ ПРЕДСТАВЛЕННЮ АЛГОРИТМУ

3.1 Зовнішнє проектування

3.1.1 Вхідні дані

Вхідними даними програмного додатку є:

- програмний код на мові програмування C++;
- файл .cpp з програмним кодом на мові програмування C++;
- файл .json з зображенням блок-схеми, що має наступний формат: blocks [text, type], arrows [startIndex, endIndex], де blocks і arrows – множина блоків та ліній з стрілкою блок-схеми відповідно, text – текст блоку, type – тип блоку, startIndex – індекс блоку який є початком лінії з стрілкою та endIndex – індекс блоку який є кінцем лінії з стрілкою.

3.1.2 Вихідні дані

Вихідними даними програмного додатку є:

- граф керування побудований за програмним кодом та представлений у вигляді списку суміжності;
- граф керування побудований за блок-схемою та представлений у вигляді списку суміжності;
- текстове повідомлення з висновком, щодо відповідності тексту програми до зображення блок-схеми у вигляді процентного співвідношення;
- текстові повідомлення, щодо успішності або неуспішності виконаних операцій.

3.1.3 Функціональні характеристики

Програмний додаток має наступні функціональні характеристики:

- введення або завантаження з файлу тексту програми;
- завантаження зображення блок-схеми з json-файлу;
- побудова графів керування за блок-схемою і програмним кодом та зіставлення їх між собою методом пошуку в ширину;

– надання висновку користувачеві програмного продукту, щодо відповідності тексту програми до зображення блок-схеми у вигляді процентного співвідношення.

Функціональні аспекти системи зображено на рис. 3.1 – 3.4 за допомогою діаграм прецедентів [45]. Для підвищення ступеню деталізації діаграми виконаємо окремо моделювання кожного прецеденту.

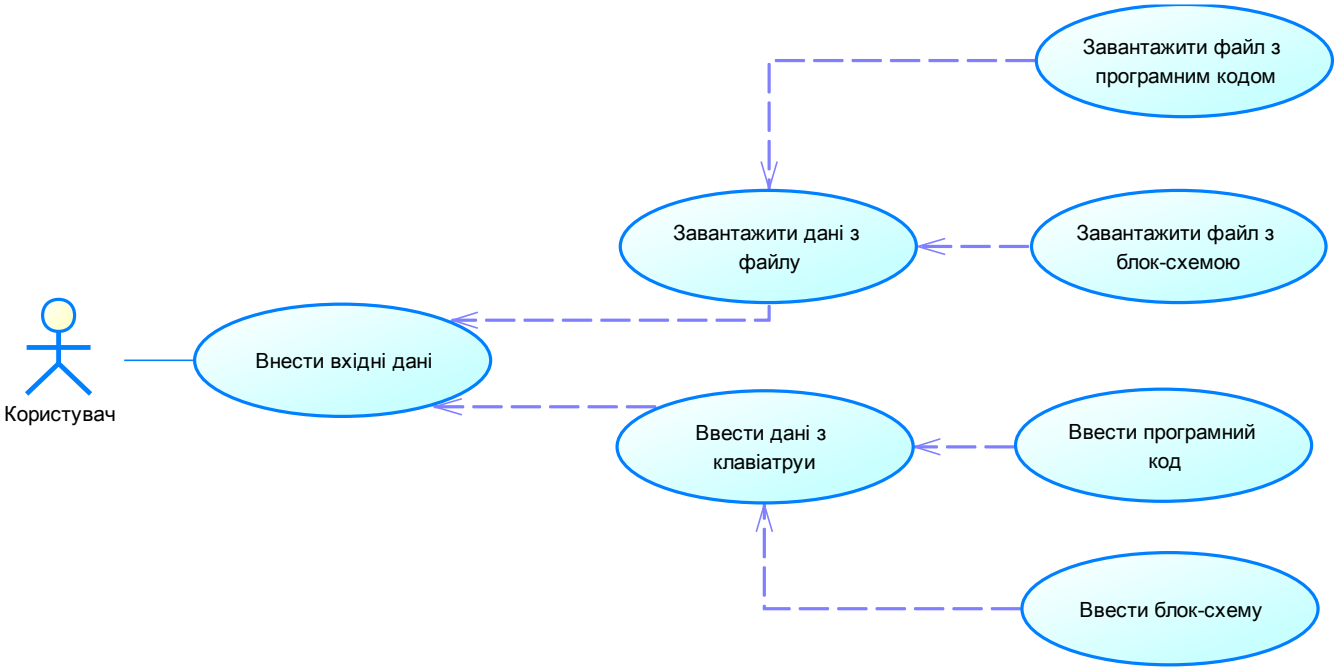


Рисунок 3.1 – Діаграма прецедентів внесення даних користувачем до системи

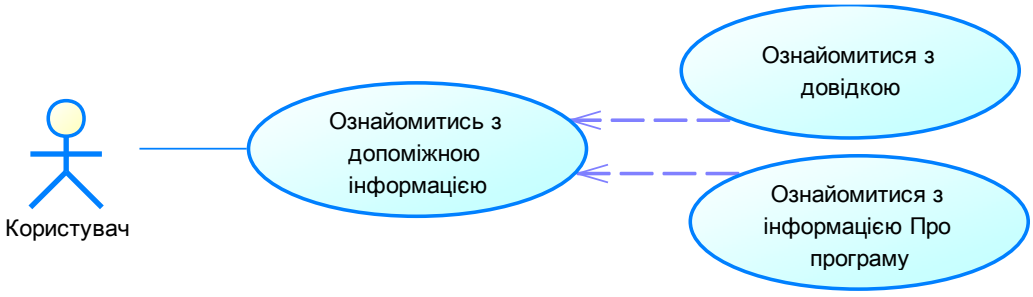


Рисунок 3.2 – Діаграма прецедентів перегляду користувачем довідкової інформації

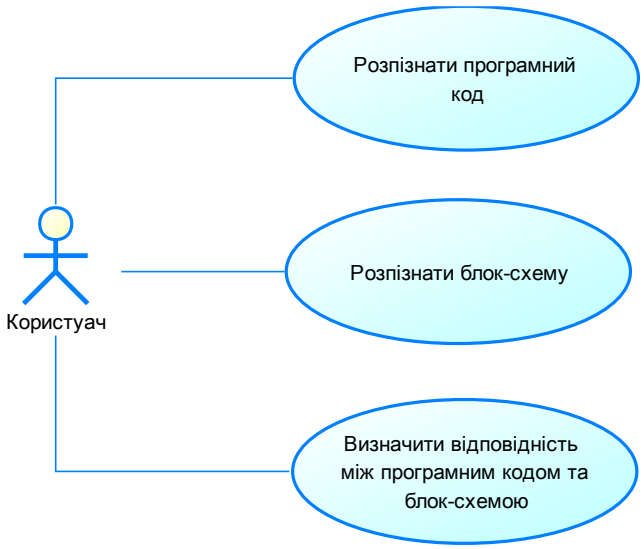


Рисунок 3.3 – Діаграма прецедентів визначення користувачем відповідності блок-схеми та програмного коду

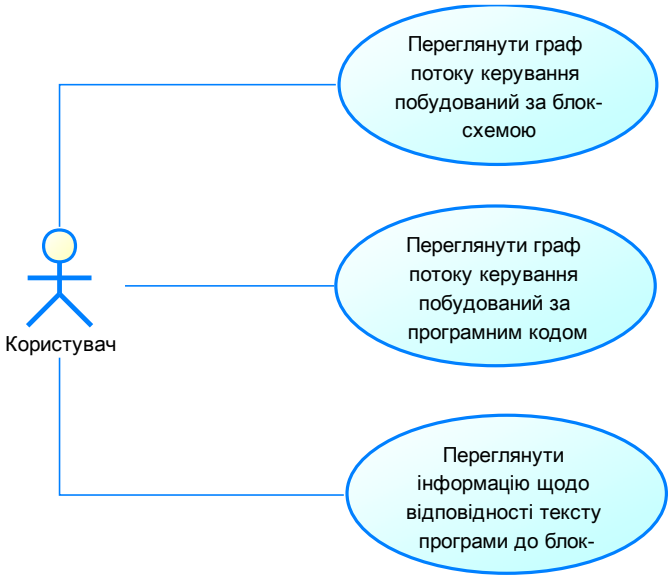


Рисунок 3.4 – Діаграма прецедентів перегляду користувачем вихідних даних додатку

3.2 Внутрішнє проектування

Для реалізації вищезазначених функціональних характеристик програмного додатку, було спроектовано та розроблено класи і форми, які відповідають за користувацький інтерфейс користувача та визначення відповідності тексту програми графічному представленню алгоритму.

3.2.1 Проектування класів системи

Розроблені класи відобразимо на діаграмах класів [46] рис. 3.5 – 3.7.

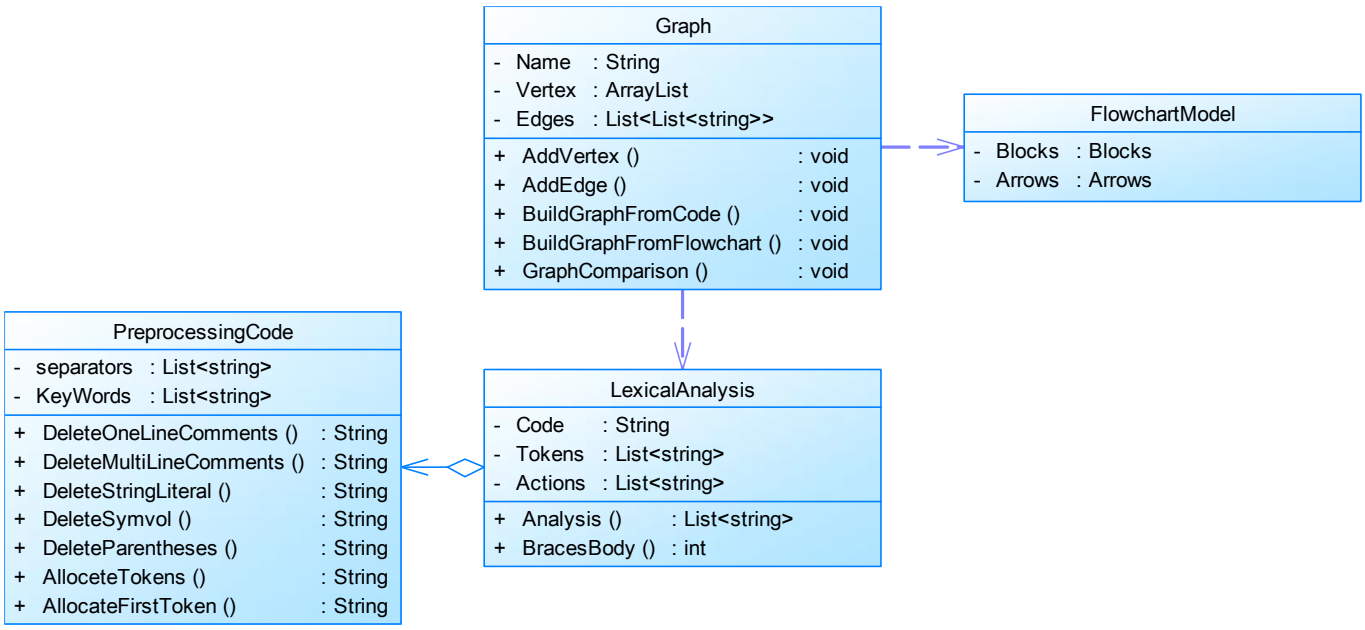


Рисунок 3.5 – Діаграма класів рівня логіки

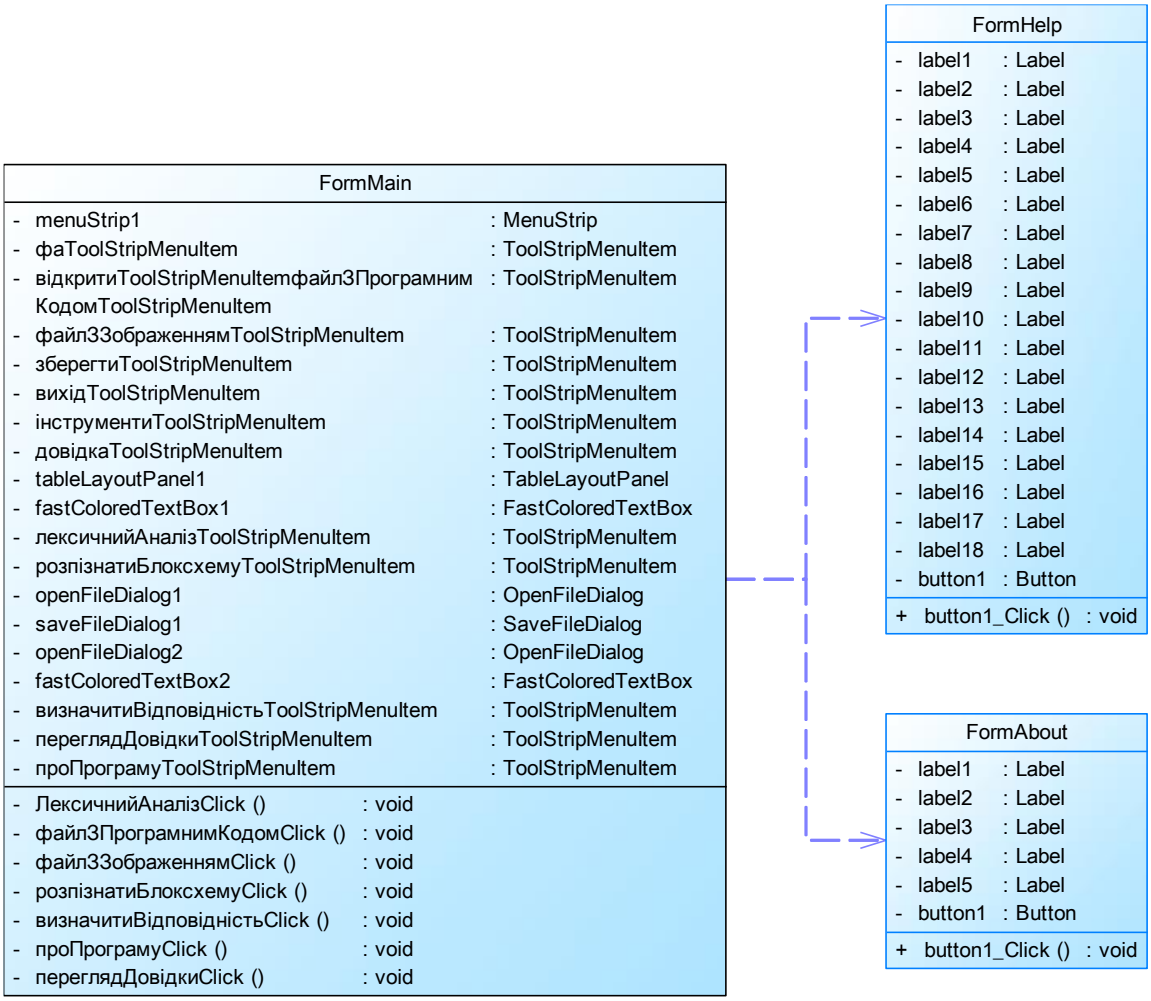


Рисунок 3.6 – Діаграма класів рівня представлення

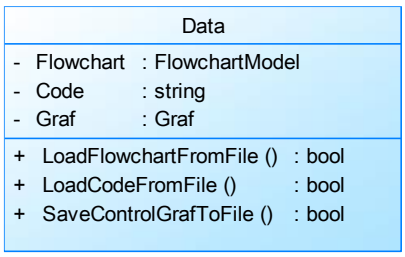


Рисунок 3.7 – Діаграма класів рівня даних

Для розроблених діаграм класів було виділено наступні рівні:

- логіки – включає класи, що відповідають за логіку програми, а саме за визначення відповідності тексту програми графічному представленню алгоритмів;
- представлення – включає форми, які відповідають за користувацький інтерфейс користувача;
- дані – включає класи, які відповідають за роботу з вхідними та вихідними даними додатку.

Розглянемо кожний розроблений клас більш детально:

- PrepocessingCode – клас рівня логіки, який відповідає за попередню обробку програмного коду, а саме: видалення одно строкових та багато строкових коментарів, видалення строкових та символьних літералів, видалення виразів в круглих дужках, видалення незначимих частин програмного коду таких як пробіли та виділення лексем програмного коду;
- LexocalAnalysis – клас рівня логіки, який відповідає за лексичний аналіз програмного коду та побудову його проміжного представлення у вигляді списку керуючих операторів;
- FlowcharModel – клас рівня логіки, який зберігає модель представлення блок-схеми;
- Graf – клас рівня логіки, який відповідає за побудову та зберігання графу потоку керування за проміжним представлення програмного коду у вигляді списку керуючих операторі та представлення блок-схеми у json-форматі;
- About – клас рівня представлення, що описує екранну форму, яка містить дані щодо програмного продукту;

- Help – клас рівня представлення, що описує екранну форму, яка містить керівництво користувача з використання програмного додатку;
- Main – клас рівня представлення, що описує екранну форму інтерфейсу користувача;
- Data – клас рівня даних, який відповідає за роботи з вхідними та вихідними даними програмного додатку, а саме завантаження програмного коду та блок-схеми з файлу та збереження побудованого графу потоку керування в файл.

3.2.2 Проектування інтерфейсу користувача

Інтерфейс користувача повинен базуватись на наступних принципах [47]:

- простий – інтерфейс користувача зрозумілий на інтуїтивному рівні;
- естетичний – програмні об’єкти інтерфейсу користувач повинні мати естетичну та ергономічну привабливість;
- продуктивний – інтерфейс користувача реалізується таким чином, щоб уникнути складної ієрархічності вікон та/або екранів, а також зайвих дій з клавіатурою та мишею;
- налагоджуваний – програмні об’єкти інтерфейсу користувача повинні мати можливість налагодження.

Для забезпечення вищеозначених вимог було обрано графічний інтерфейс користувача (GUI – Graphical User Interface) [46], який включає в себе діалог на основі меню та діалог на основі екранних форм.

GUI має наступні характеристики:

- забезпечує візуальне відображення інформації та об’єктів;
- забезпечує візуальний зворотній зв’язок в процесі виконання користувачами дій та задач;
- надає візуальне відображення дій користувача або системи;
- надає користувачам можливість налагодити та персоналізувати інтерфейс та інтерактивні дії;
- надає можливість безпосереднього маніпулювання об'єктами та інформацією на екрані.

Спроектований інтерфейс користувача відображено на рис. 3.8 – 3.10.

Інтерфейс користувача містить наступні елементи:

- екранна форма ConformityAssessment, яка є головною формою та містить користувацький інтерфейс;
- екранна форма Про програму, яка містить інформацію щодо програмного продукту;
- екранна форма Довідка, яка містить керівництво користувача;
- головне меню, що містить наступні пункти: файл, інструменти та довідка.

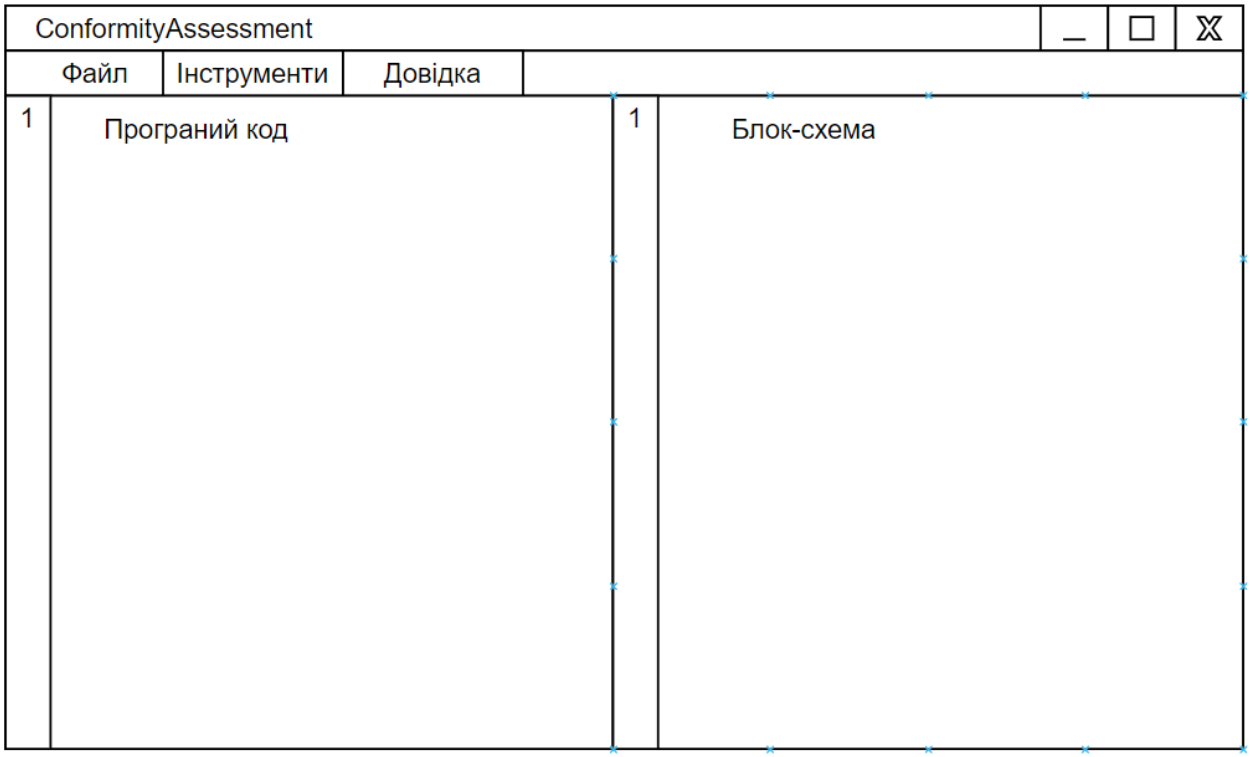


Рисунок 3.8 – Ескіз екранної форми ConformityAssessment

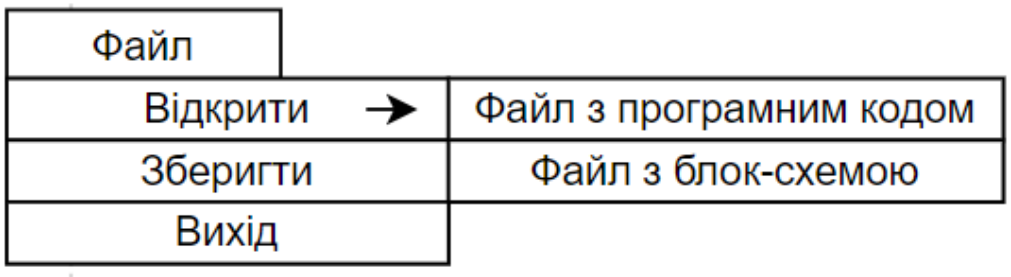


Рисунок 3.9 – Ескіз пункту головного меню Файл

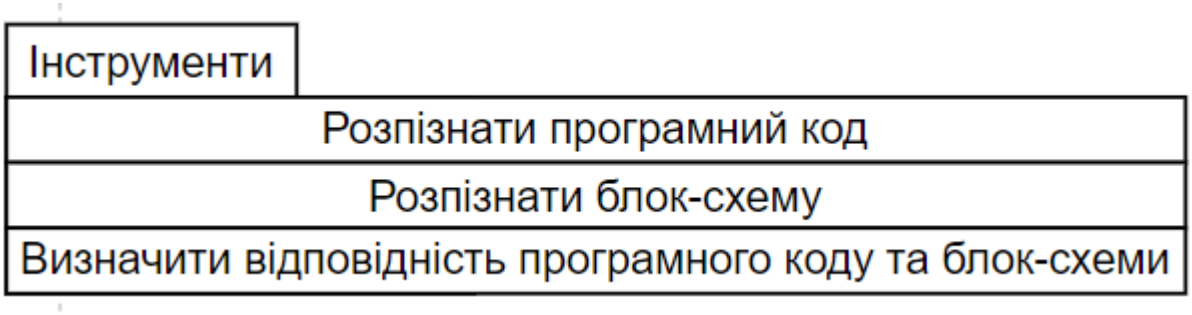


Рисунок 3.10 – Ескіз пункту головного меню Інструменти

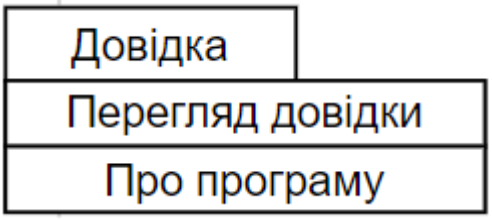


Рисунок 3.11 – Ескіз пункту головного меню Довідка

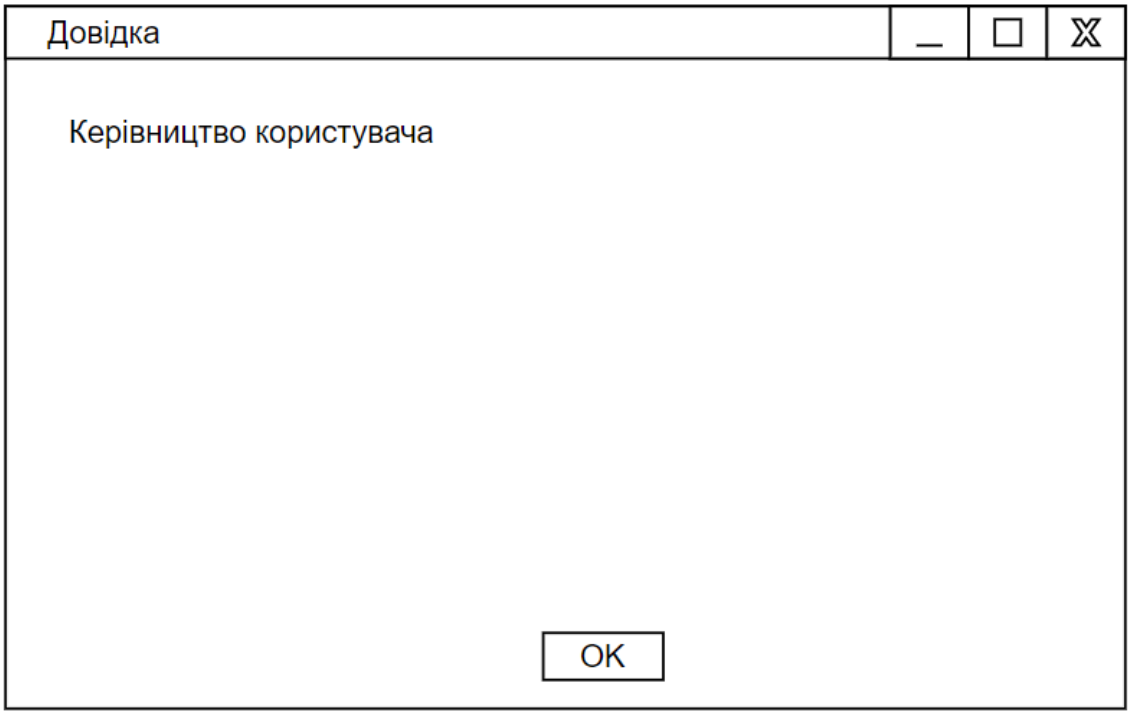


Рисунок 3.12 – Ескіз екранної форми Довідка

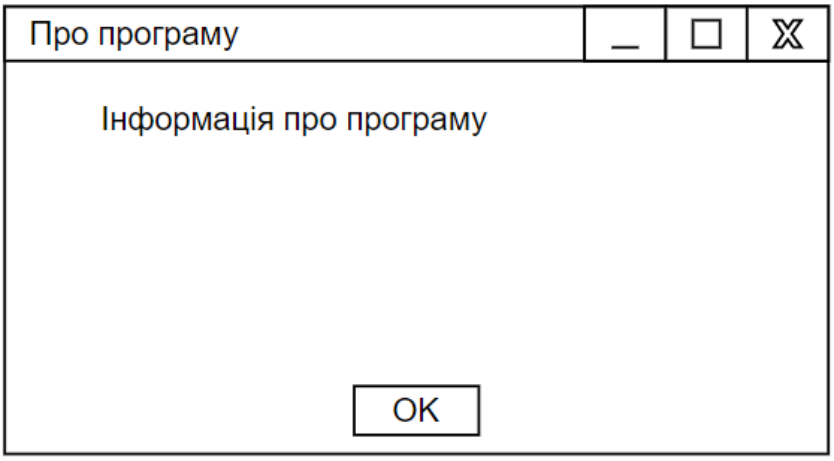


Рисунок 3.13 – Ескіз екранної форми програму

Взаємодія користувача з графічним інтерфейсом відображена на діаграмі станів [48] (рис. 3.14)

Висновки до третього розділу

В даному розділі було виконано проектування та розробку інструментального забезпечення для визначення відповідності алгоритму та його графічного зображення.

Було визначено вхідні та вихідні дані програмного додатку, а також його функціональні характеристики. Також було визначено архітектуру додатку, яка складається з трьох рівнів: представлення, логіки та даних Було спроектовано основні класи системи та їх взаємодію між собою.

Було обрано модель інтерфейсу користувача, а саме GUI, яка складається з діалогу на основі меню та діалогу на основі екранних форм. Також було спроектовано та реалізовано екранні форми програмного додатку.

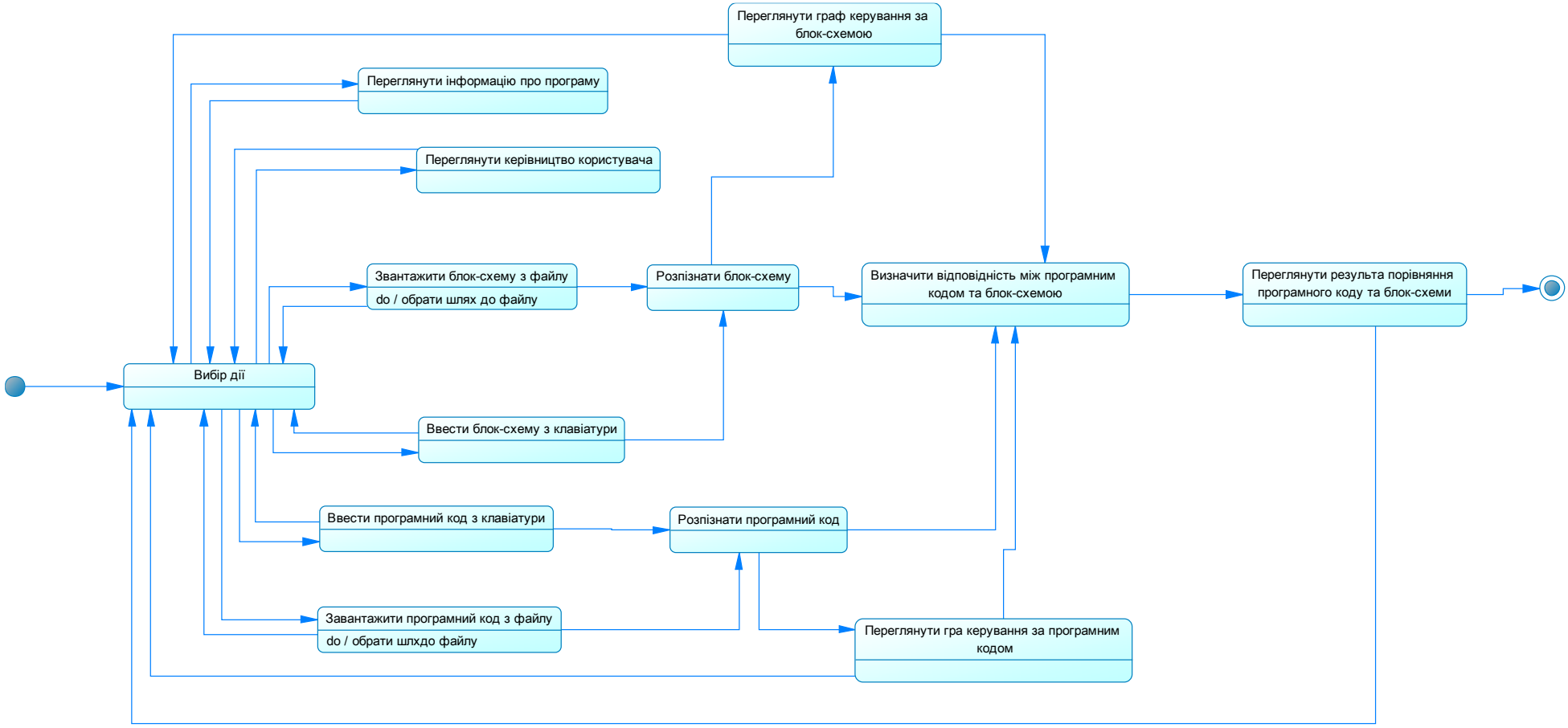


Рисунок 3.14 – Діаграма станів взаємодії користувача з вікном Про програму

4 ДОСЛІДЖЕННЯ ВІДПОВІДНОСТІ ТЕКСТУ ПРОГРАМИ ГРАФІЧНОМУ ПРЕДСТАВЛЕННЮ АЛГОРИТМУ

4.1 Підготовка експерименту

4.1.1 Класифікація алгоритмів

Визначимо основні види алгоритмів [49{]:

- лінійний алгоритм;
- алгоритм розгалуження;
- циклічний алгоритм.

Якщо розглядати алгоритм в цілому, то він може містити в собі всі види одночасно, адже окремі його ділянки можна розглядати як під алгоритми.

Визначмо види алгоритмів та їх реалізацій для проведення експерименту:

- лінійний алгоритм;
- алгоритм розгалуження;
- циклічний алгоритм з пост умовою;
- циклічний алгоритм з передумовою;
- вкладеність циклічних алгоритмів;
- вкладеність алгоритмів розгалуження;
- вкладеність розгалуження в циклічний алгоритм.

4.1.2 Набір даних для експерименту

Для кожного з вищезазначених видів алгоритмів визначимо набір даних, а саме пару: код на мові програмування C++ (лістинги 1 – 9) та його блок-схему (рис. 4.1 – 4.9).

Блок-схема містить програмний код, оскільки розроблений метод для розпізнавання блок-схеми потребує уточнення типів вузлів, що позначають керуючі конструкції.

Лістинг 1. Лінійний алгоритм:

```
#include <iostream>
#include <cstdlib>
// Строки C++.
#include <string>

int main()
```

```
{
    using namespace std; // Искать имена в std.

    string user_name = "user"; // Определить переменную.
    cout << "Hello, " << user_name << "!" << endl;

    user_name = "The Great Whale"; // Изменить значение
переменной.
    cout << "I am " << user_name;
    return EXIT_SUCCESS; // Возвратим ОС "код успеха".
}
```

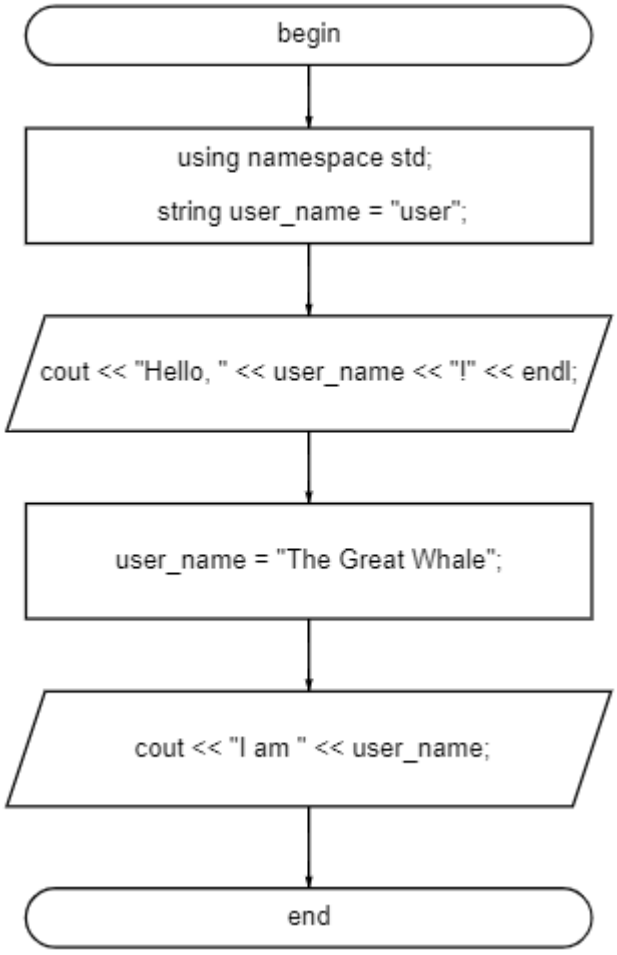


Рисунок 4.1 – Блок-схема лінійного алгоритму

Лістинг 2. Алгоритм розгалуження:

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    double x = 0;
    cout << "x = ";
    cin >> x;
```

```
        cout << "x*x ";
        if (x * x < 2) // Условие.
            cout << " < ";
        else // Альтернатива.
            cout << " > ";
        cout << "2" << endl;
        return EXIT_SUCCESS;
    }
```

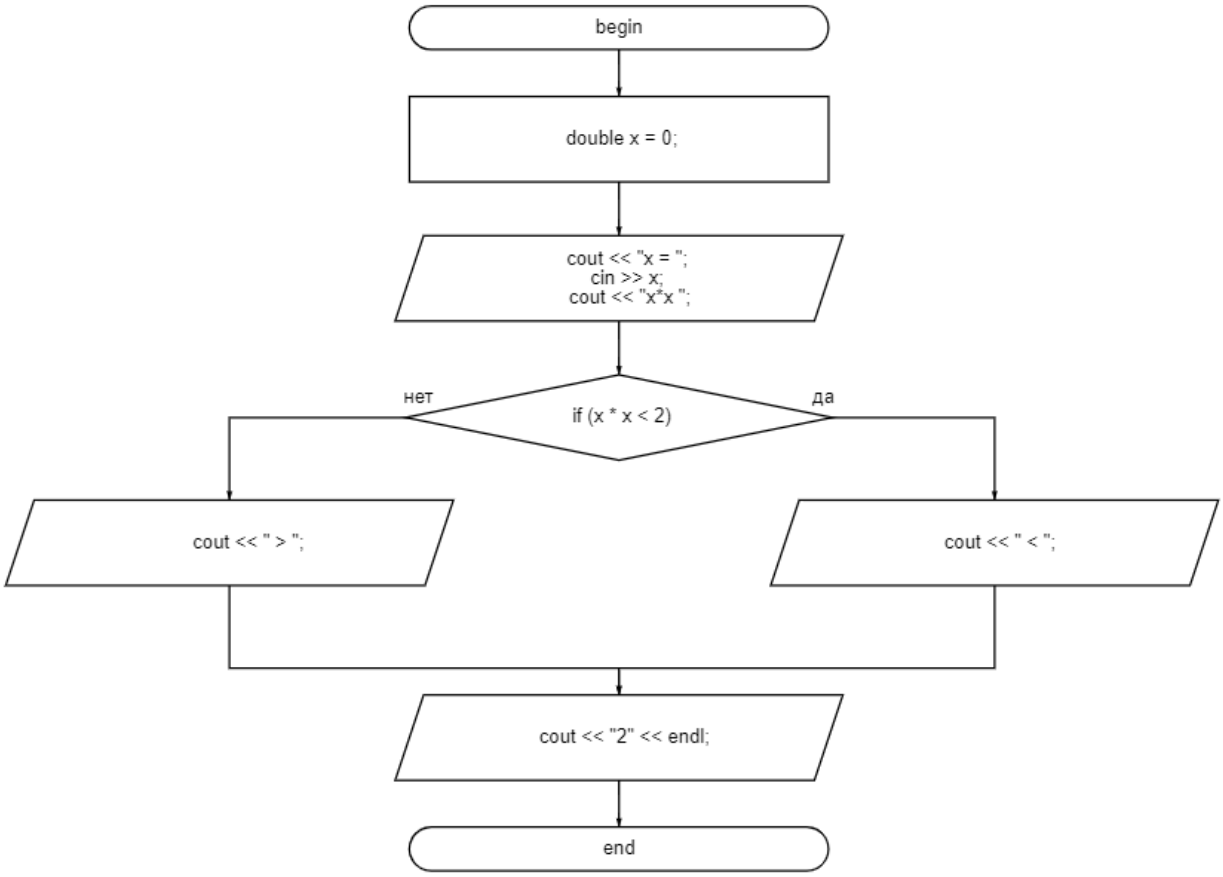


Рисунок 4.2 – Блок-схема алгоритму розгалуження

Лістинг 3. Циклічний алгоритм з пост умовою:

```
    #include <iostream>

    int main()
    {
        // Переменная choice должна быть объявлена вне цикла do
while
        int choice;

        do
        {
            std::cout << "Please make a selection: \n";
            std::cout << "1) Addition\n";
            std::cout << "2) Subtraction\n";
            std::cout << "3) Multiplication\n";
```

```
            std::cout << "4) Division\n";
            std::cin >> choice;
        } while (choice != 1 && choice != 2 &&
            choice != 3 && choice != 4);

        // Что-то делаем с переменной choice, например, используем
оператор switch

        std::cout << "You selected option #" << choice << "\n";

        return 0;
    }
```

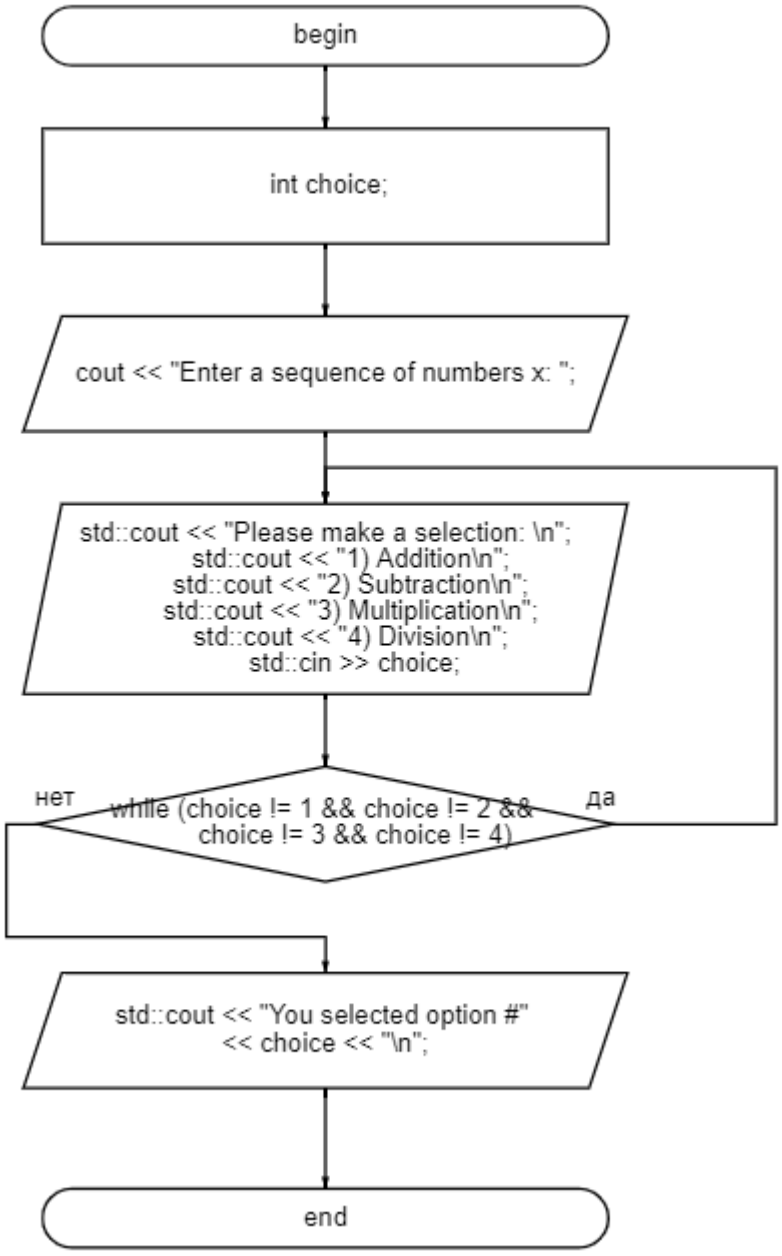


Рисунок 4.3 – Блок-схема циклічного алгоритму з пост умовою

Лістинг 4. Циклічний алгоритм з передумовою:

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    double x = 0;
    cout << "Enter a sequence of numbers x: ";
    while (cin >> x) // Условие продолжения выполнения цикла.
    {
        cout << "x*x < 2  == " << sqr_lt_2(x) << endl;
    }
    return EXIT_SUCCESS;
}
```

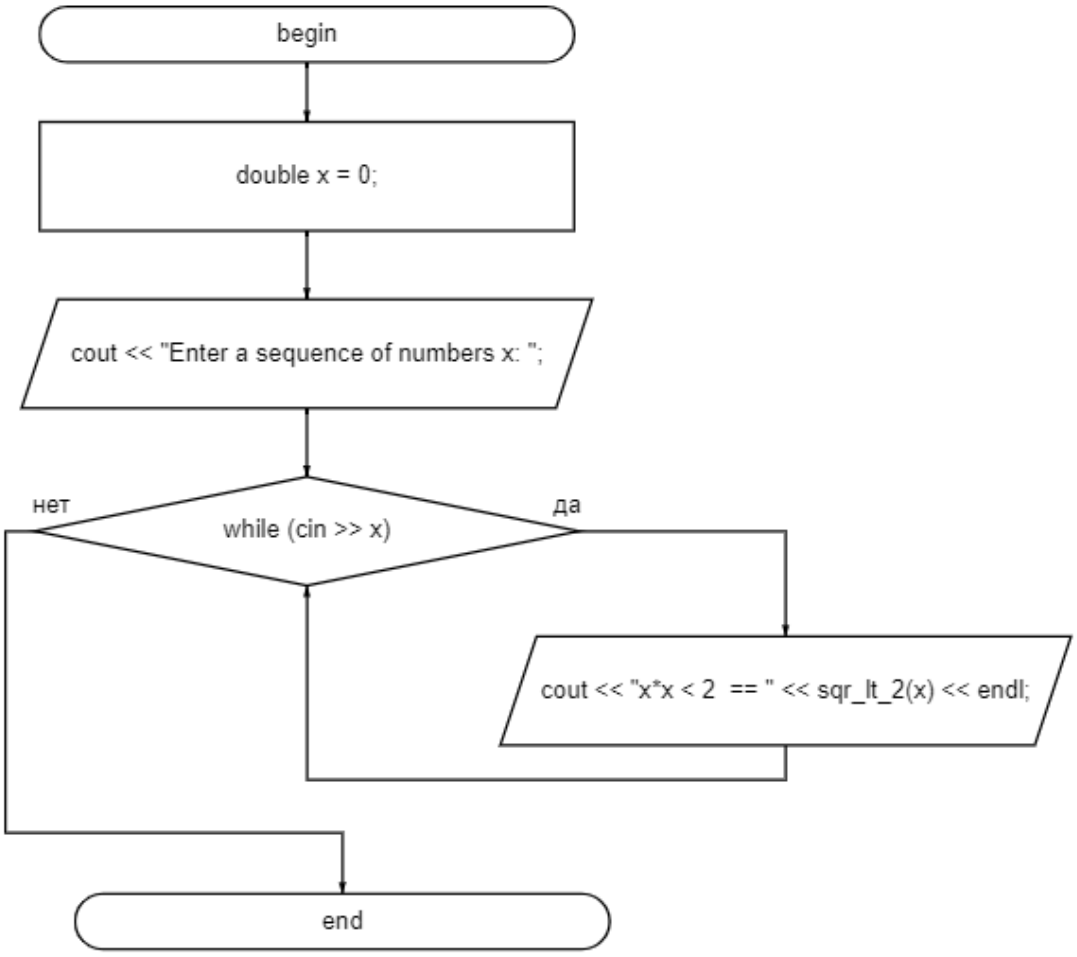


Рисунок 4.4 – Блок схема циклічного алгоритму з передумовою

Лістинг 5. Вкладені конструкції циклу:

```
#include <iostream>
using namespace std;
int main()
{
    for (int f = 2; f <= 9; f++)
```

```
    {
        for (int s = 1; s <= 9; s++)
        {
            cout << f << " * " << s << " = " << f * s;
            cout << endl;
        }
        cout << endl;
    }
    cout << endl;
    return 0;
}
```

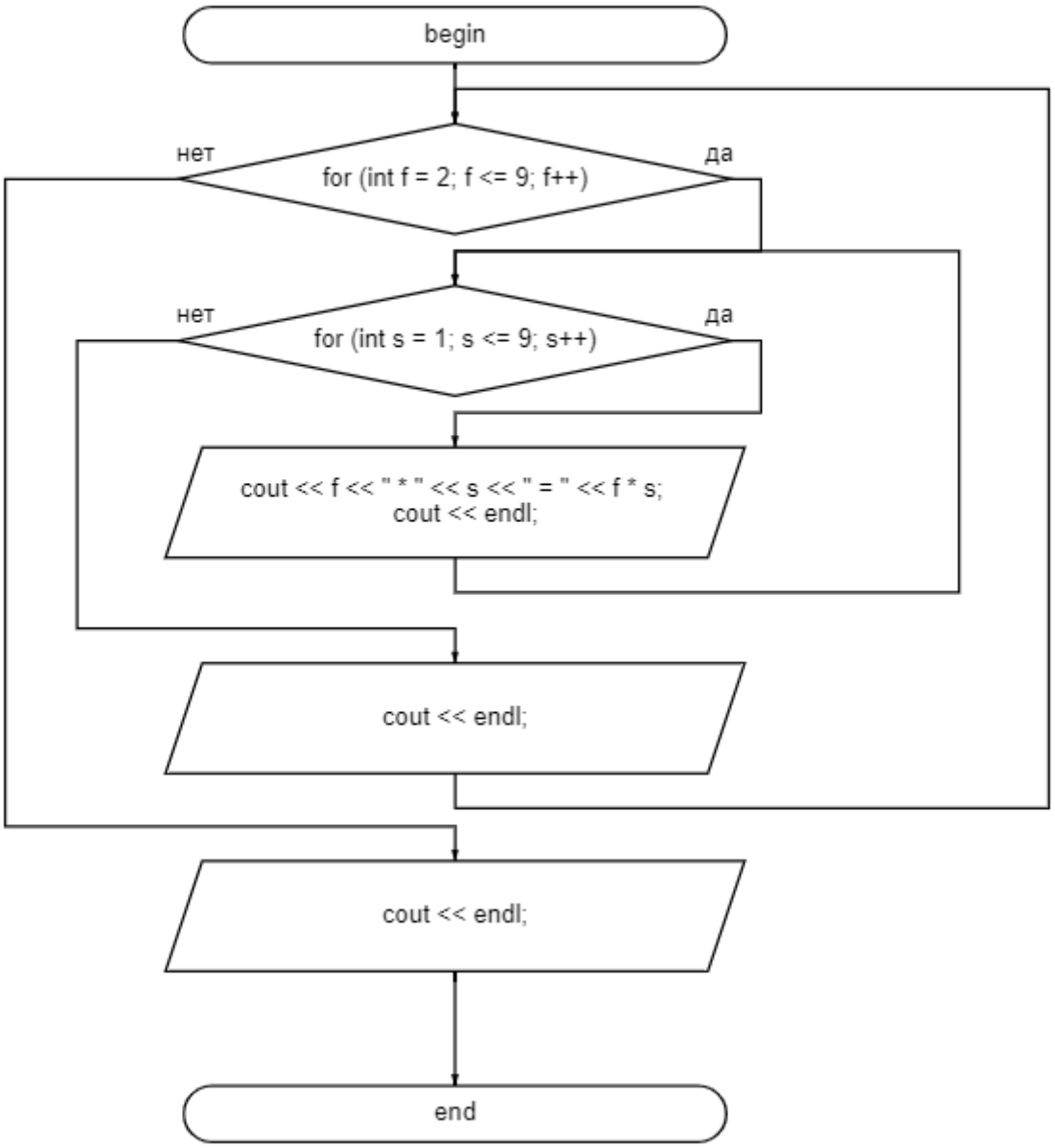


Рисунок 4.5 – Блок-схема вкладених конструкцій циклу

Лістинг 6. Вкладені умовні конструкції:

```
#include <stdio.h>
```

```
int main(void)
{
     int magic = 123; /* искомое число */
     int guess;
     printf("Enter your guess: ");
     scanf("%d", &guess);
     if (guess == magic)
     {
          printf("** Right ** ");
          printf("%d is the magic number", magic);
     }
     else
     {
          printf(".. Wrong .. ");
          if (guess > magic) printf("Too high");
          else printf("Too low");
     }
     return 0;
}
```

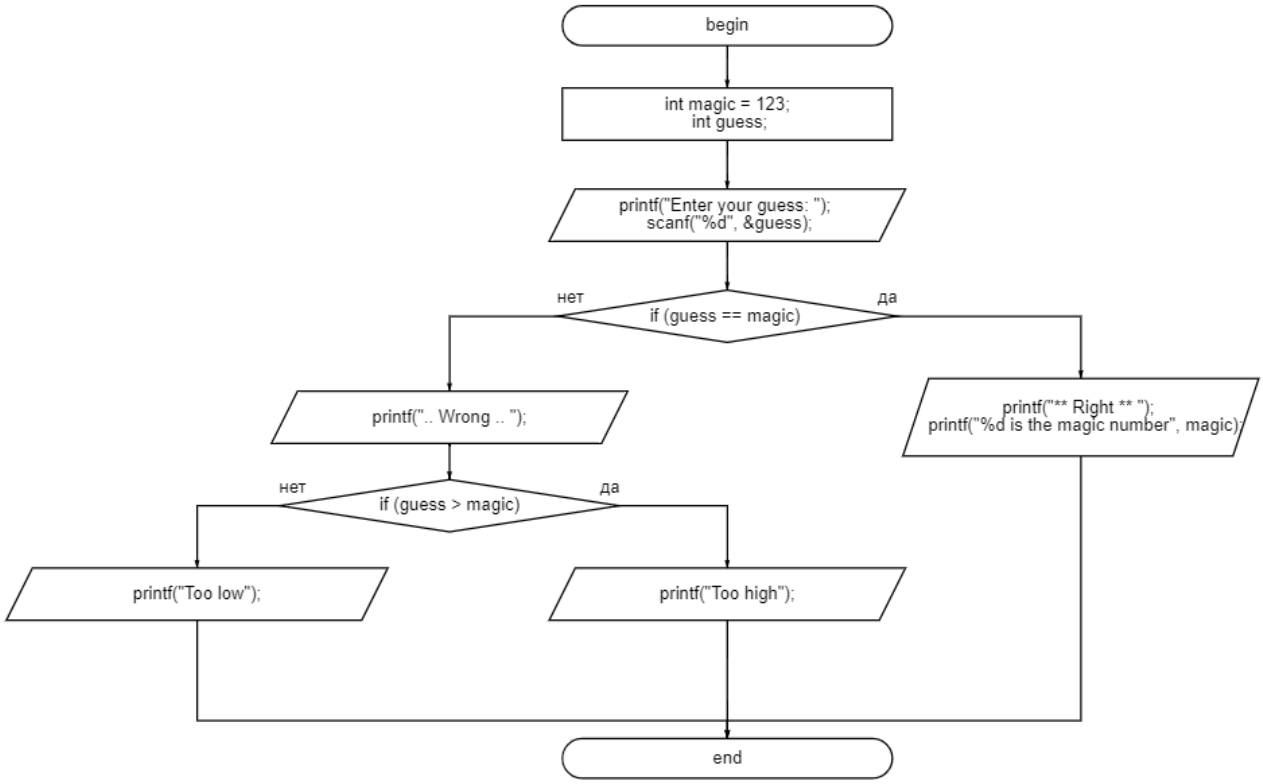


Рисунок 4.6 – Блок схема вкладених умовних конструкцій

Лістинг 7. Вкладеність розгалуження в циклічний алгоритм:

```
#include <stdio.h>
int main(void)
{
     int magic = 123; /* искомое число */
     int guess;
     printf("Enter your guess: ");
     scanf("%d", &guess);
```

```
    while (guess != magic)
    {
         printf(".. Wrong .. ");
         if (guess > magic) printf("Too high");
         else printf("Too low");

    }
    printf("** Right ** ");
    printf("%d is the magic number", magic);
    return 0;
}
```

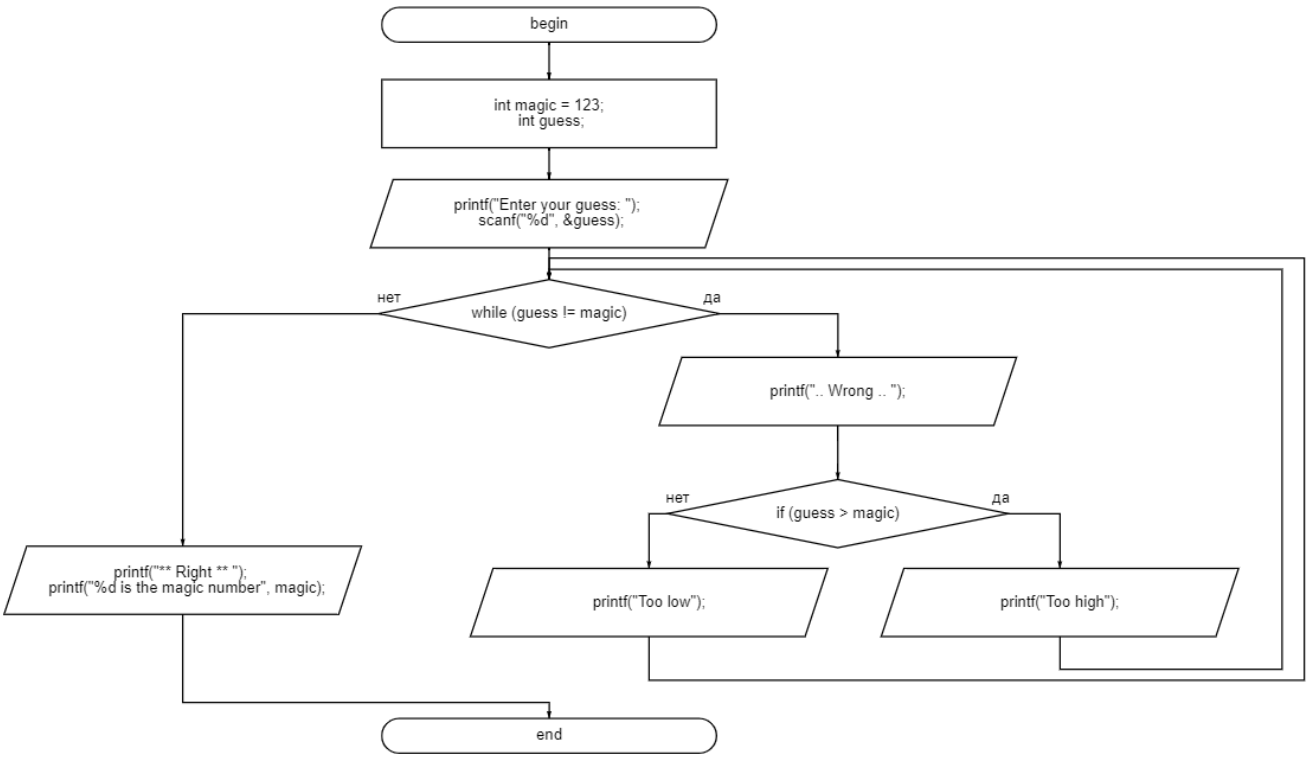


Рисунок 4.7 – Блок схема вкладеного розгалуження в циклічний алгоритм

Лістинг 8. Алгоритм розгалуження:

```
#include <iostream>
#include <cstdlib>
using namespace std;

int main()
{
    double x = 0;
    cout << "Enter a sequence of numbers x: ";
    while (cin >> x) // Условие продолжения выполнения цикла.
    {
        cout << "x*x < 2  == " << sqr_lt_2(x) << endl;
    }
    return EXIT_SUCCESS;
}
```

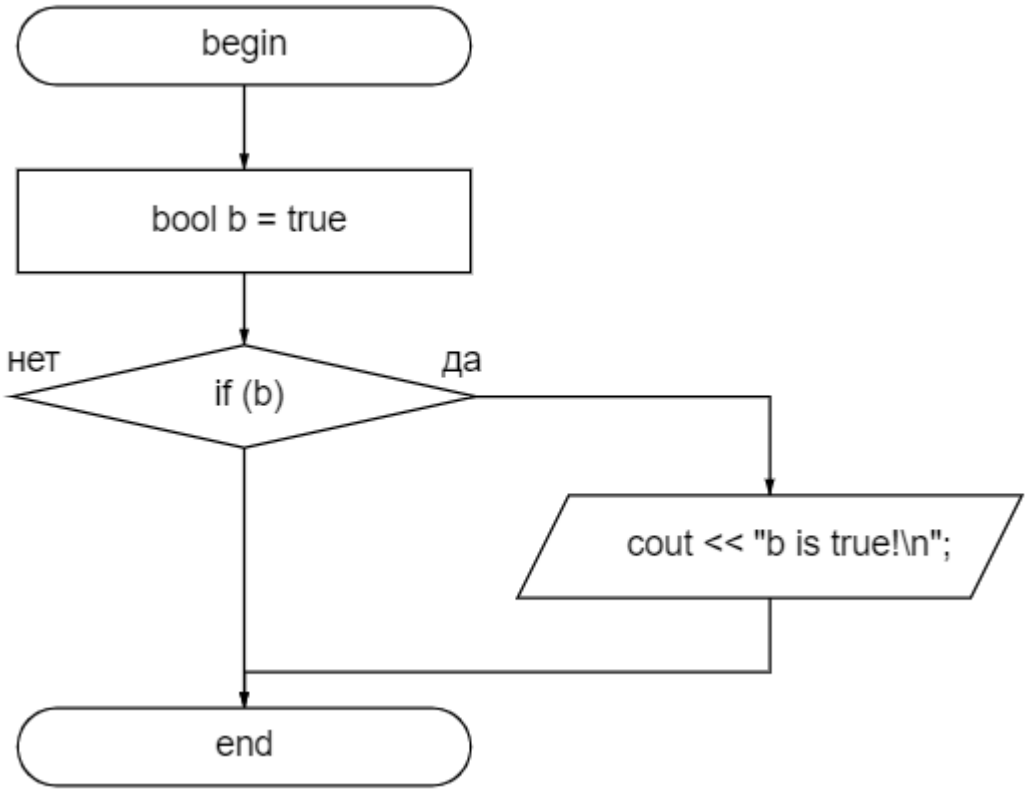


Рисунок 4.8 – Блок-схема алгоритму розгалуження

Лістинг 9. Циклічний алгоритм

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{

    double x = 0;

    cout << "Enter a sequence of numbers x: ";

    while (cin >> x) // Условие продолжения выполнения цикла.
    {

        cout << "x*x < 2  == " << sqr_lt_2(x) << endl;

    }

    return EXIT_SUCCESS;

}
```

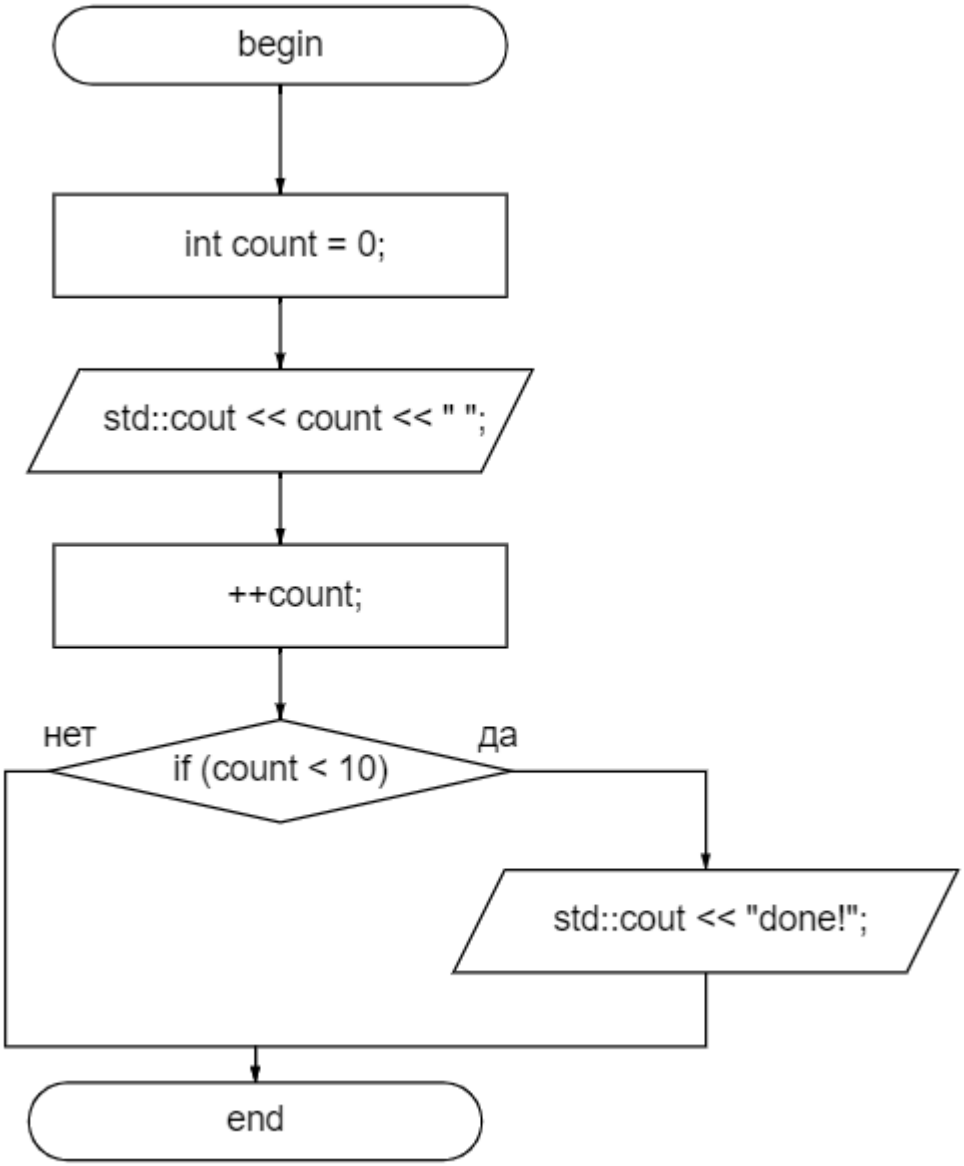


Рисунок 4.9 – Блок-схема алгоритму розгалуження

4.2 Проведення експерименту

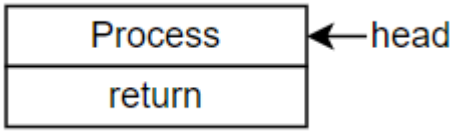
Для кожного лістингу з набору даних визначимо: список керуючих операторів побудований за програмним кодом, послідовність виконання правил для побудови графу керування за списком керуючих операторів, очікуваний граф керування, побудований за списком керуючих операторів та отриманий граф керування, побудований за списком керуючих операторів у вигляді списку суміжності. Оскільки блок-схема є орієнтованим графом, то для неї достатньо визначити отриманий граф керування побудований за моделлю блок-схеми.

Лістинг 1 було перетворено у *list* – конструкцію побудованою за допомогою C_T (рис. 4.10, а), head – голова списку, принцип роботи списку – FIFO. Граф потоку керування (рис. 4.10, б) будується за допомогою послідовного виконання правил:

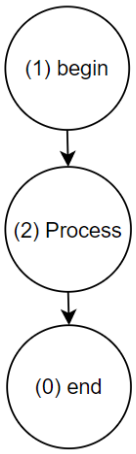
$$\sigma_{\,1} \to G\alpha_{\,2} \to \tilde{\cup}\left(G, G^*\right)\alpha_{\,6} \to \tilde{\cup}\left(G, G^*\right)_{\,27} \to \varepsilon\,. \tag{4.1}$$

Графи керування, побудовані за допомогою програмного забезпечення, представлено у вигляді списку суміжності для:

– програмного коду на рис. 4.10, в;
– блок-схеми на рис 4.11.



а)



б)

в)

```
0 end -
1 begin - 2
2 Process - 0
```

Рисунок 4.10 – Схема конструкцій, побудованих C_T, C_G та програмним додатком: а) список керуючих операторів б) очікуваний граф керування в) отриманий граф керування

```
0 end -
1 begin - 2
2 Process - 0
```

Рисунок 4.11 – Граф керування у вигляді списку суміжності побудований за блок-схемою

Лістинг 2 було перетворено у *list* – конструкцію побудованою за допомогою C_T (рис. 4.12, а), head – голова списку, принцип роботи списку – FIFO. Граф потоку керування (рис. 4.12, б) будується за допомогою послідовного виконання правил:

$$\sigma_{1} \to G\alpha_{2} \to \tilde{\cup}\left(G,G^{*}\right)_{2} \to \tilde{\cup}\left(G,G^{*}\right)_{5} \to G\alpha_{2} \to \tilde{\cup}\left(G,G^{*}\right)_{9} \to G\phi\alpha_{17} \to \quad (4.2)$$

$$_{17} \to G\phi_{23} \to G\alpha_{4} \to G\alpha_{2} \to \tilde{\cup}\left(G,G^{*}\right)_{9} \to G\phi\alpha_{10} \to G\phi_{25} \to \tilde{\cup}\left(G,G^{*}\right)_{2} \to$$

$$_{2} \to \tilde{\cup}\left(G,G^{*}\right)_{6} \to \tilde{\cup}\left(G,G^{*}\right)_{27} \to \varepsilon .$$

Графи керування, побудовані за допомогою програмного забезпечення, представлено у вигляді списку суміжності для:

– програмного коду на рис. 4.12, в;
– блок-схеми на рис 4.13.

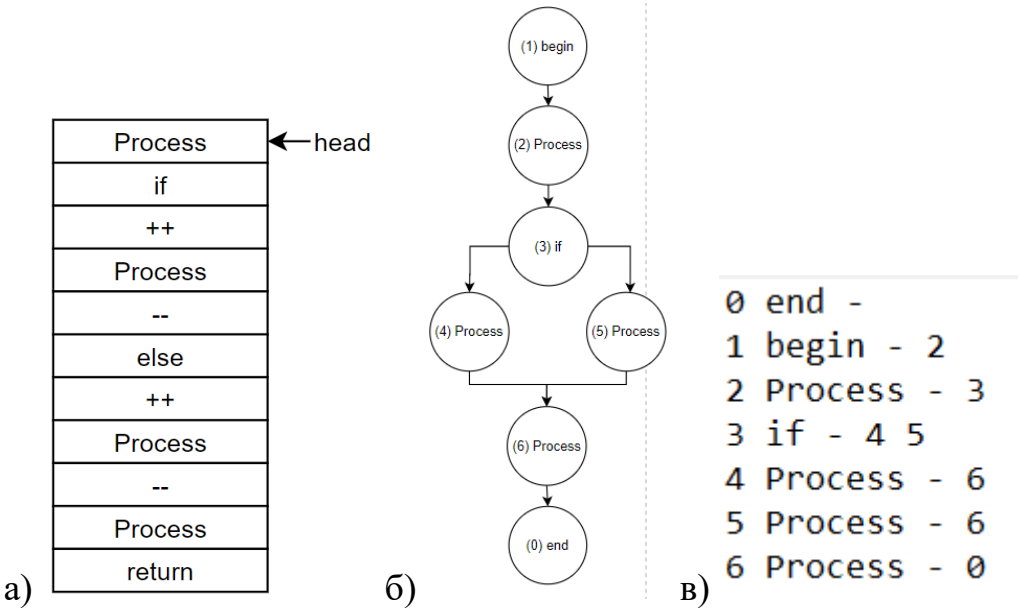


Рисунок 4.12 – Схема конструкцій, побудованих C_T, C_G та програмним додатком: а) список керуючих операторів б) очікуваний граф керування в) отриманий граф керування

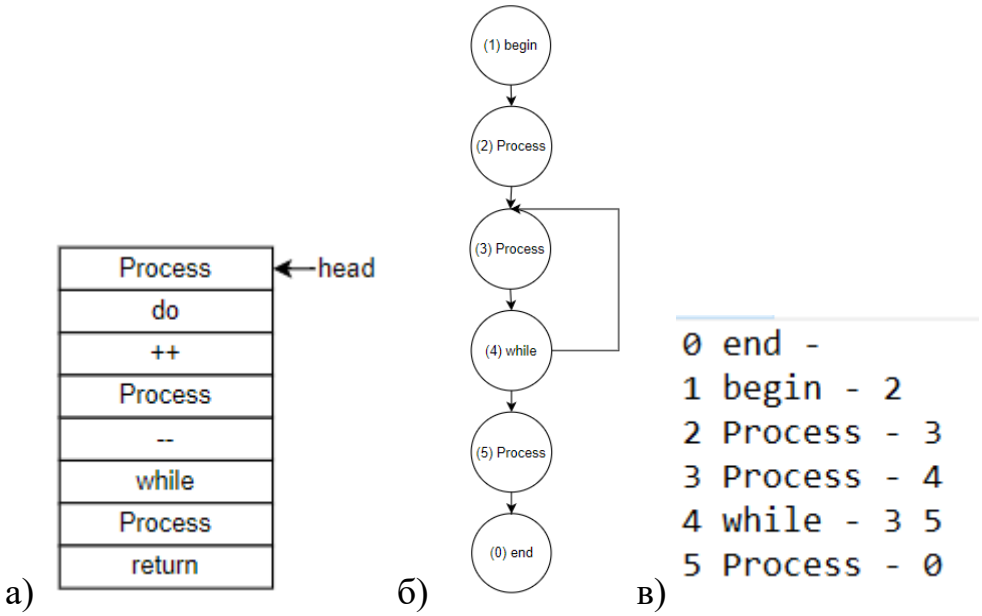
```
0 end -
1 begin - 2
2 Process - 3
3 if - 4 5
4 Process - 6
5 Process - 6
6 Process - 0
```

Рисунок 4.13 – Граф керування у вигляді списку суміжності побудований за блок-схемою

Лістинг 3 було перетворено у *list* – конструкцію побудованою за допомогою C_T (рис. 4.14, а), head – голова списку, принцип роботи списку – FIFO. Граф потоку керування (рис. 4.14, б) будується відповідно з вищезазначеними прикладами за допомогою формул 2.13 – 2.41.

Графи керування, побудовані за допомогою програмного забезпечення, представлено у вигляді списку суміжності для:

– програмного коду на рис. 4.14, в;
– блок-схеми на рис 4.15.



а) б) в)

Рисунок 4.14 – Схема конструкцій, побудованих C_T, C_G та програмним додатком: а) список керуючих операторів б) очікуваний граф керування в) отриманий граф керування

```
0 end -
1 begin - 4
2 Process - 0
3 Process - 5
4 Process - 3
5 while - 2 3
```

Рисунок 4.15 – Граф керування у вигляді списку суміжності побудований за блок-схемою

Лістинг 4 було перетворено у *list* – конструкцію побудованою за допомогою C_T (рис. 4.16, а), head – голова списку, принцип роботи списку – FIFO. Граф потоку

керування (рис. 4.16, б) будується відповідно з вищезазначеними прикладами за допомогою формул 2.13 – 2.41.

Графи керування, побудовані за допомогою програмного забезпечення, представлено у вигляді списку суміжності для:

– програмного коду на рис. 4.16, в;
– блок-схеми на рис 4.17.

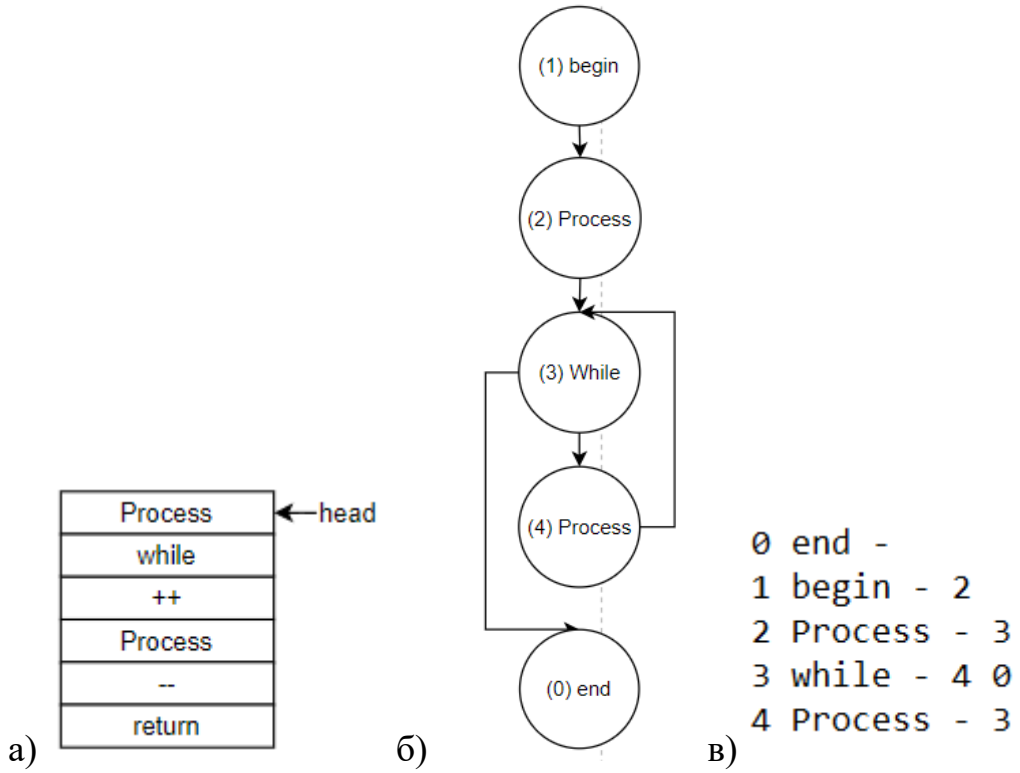


Рисунок 4.16 – Схема конструкцій, побудованих C_T, C_G та програмним додатком: а) список керуючих операторів б) очікуваний граф керування в) отриманий граф керування

```
0 end -
1 begin - 3
2 Process - 4
3 Process - 4
4 while - 2 0
```

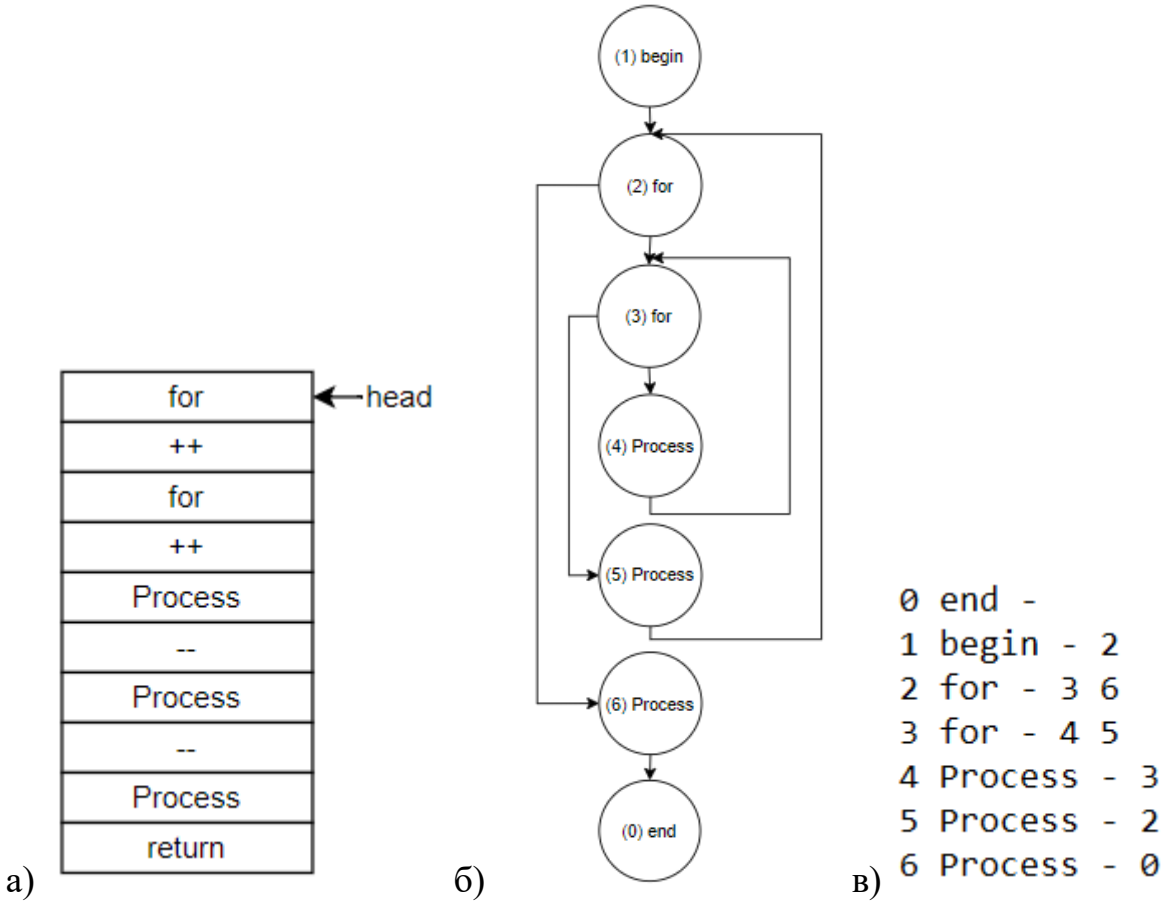
Рисунок 4.17 – Граф керування у вигляді списку суміжності побудований за блок-схемою

Лістинг 5 було перетворено у *list* – конструкцію побудованою за допомогою C_T (рис. 4.18, а), head – голова списку, принцип роботи списку – FIFO. Граф потоку

керування (рис. 4.18, б) будується відповідно з вищезазначеними прикладами за допомогою формул 2.13 – 2.41.

Графи керування, побудовані за допомогою програмного забезпечення, представлено у вигляді списку суміжності для:

– програмного коду на рис. 4.18, в;
– блок-схеми на рис 4.19.



а) б) в)

Рисунок 4.18 – Схема конструкцій, побудованих C_T , C_G та програмним додатком: а) список керуючих операторів б) очікуваний граф керування в) отриманий граф керування

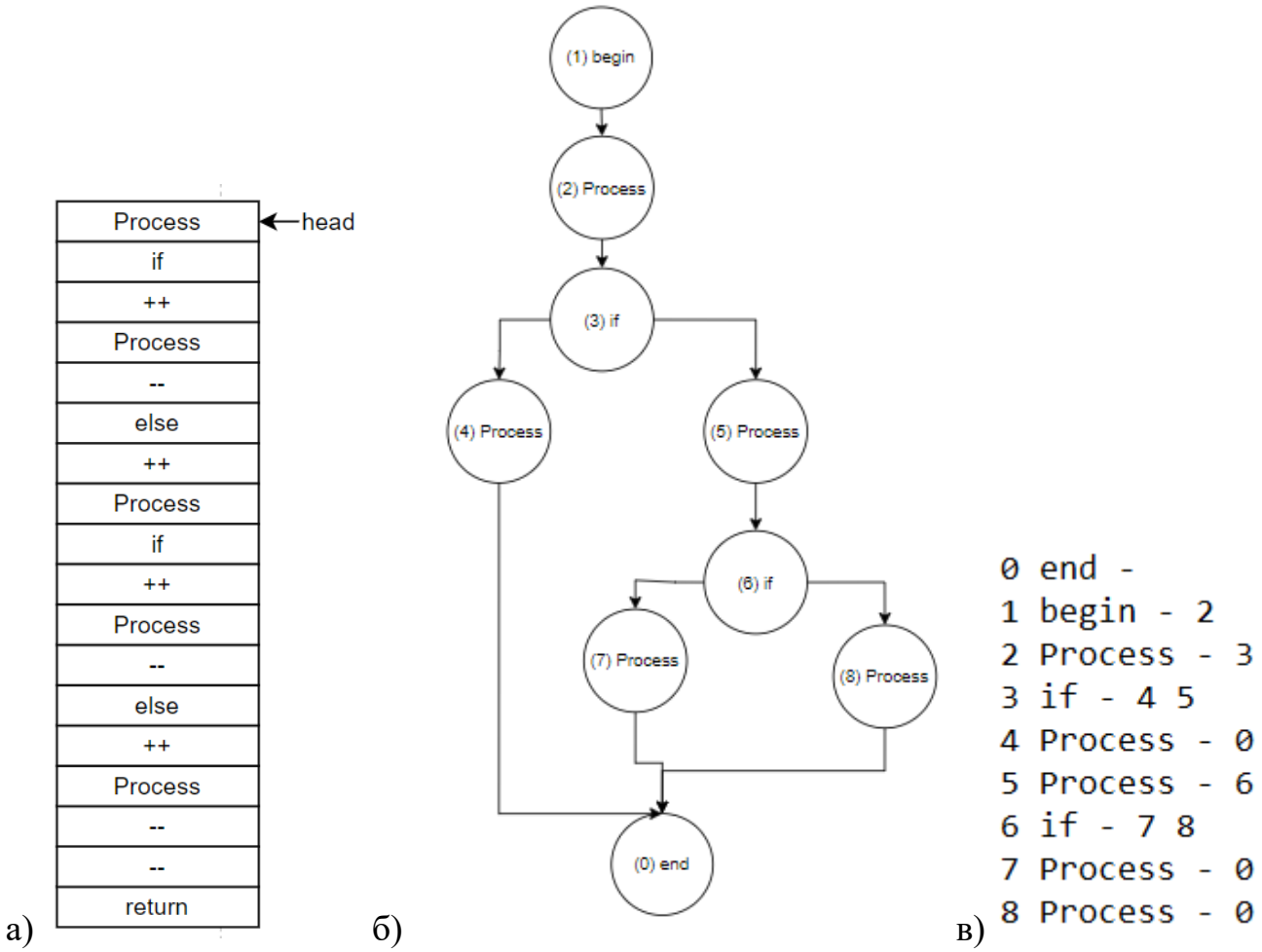
```
0 end -
1 begin - 2
2 for - 3 6
3 for - 4 5
4 Process - 3
5 Process - 2
6 Process - 0
```

Рисунок 4.19 – Граф керування у вигляді списку суміжності побудований за блок-схемою

Лістинг 6 було перетворено у *list* – конструкцію побудованою за допомогою C_T (рис. 4.20, а), head – голова списку, принцип роботи списку – FIFO. Граф потоку керування (рис. 4.20, б) будується відповідно з вищезазначеними прикладами за допомогою формул 2.13 – 2.41.

Графи керування, побудовані за допомогою програмного забезпечення, представлено у вигляді списку суміжності для:

– програмного коду на рис. 4.20, в;
– блок-схеми на рис 4.21.



а) б) в)

Рисунок 4.20 – Схема конструкцій, побудованих C_T, C_G та програмним додатком: а) список керуючих операторів б) очікуваний граф керування в) отриманий граф керування

```
0 end -
1 begin - 2
2 Process - 3
3 if - 8 5
4 Process - 0
5 Process - 6
6 if - 7 4
7 Process - 0
8 Process - 0
```

Рисунок 4.21 – Граф керування у вигляді списку суміжності побудований за блок-схемою

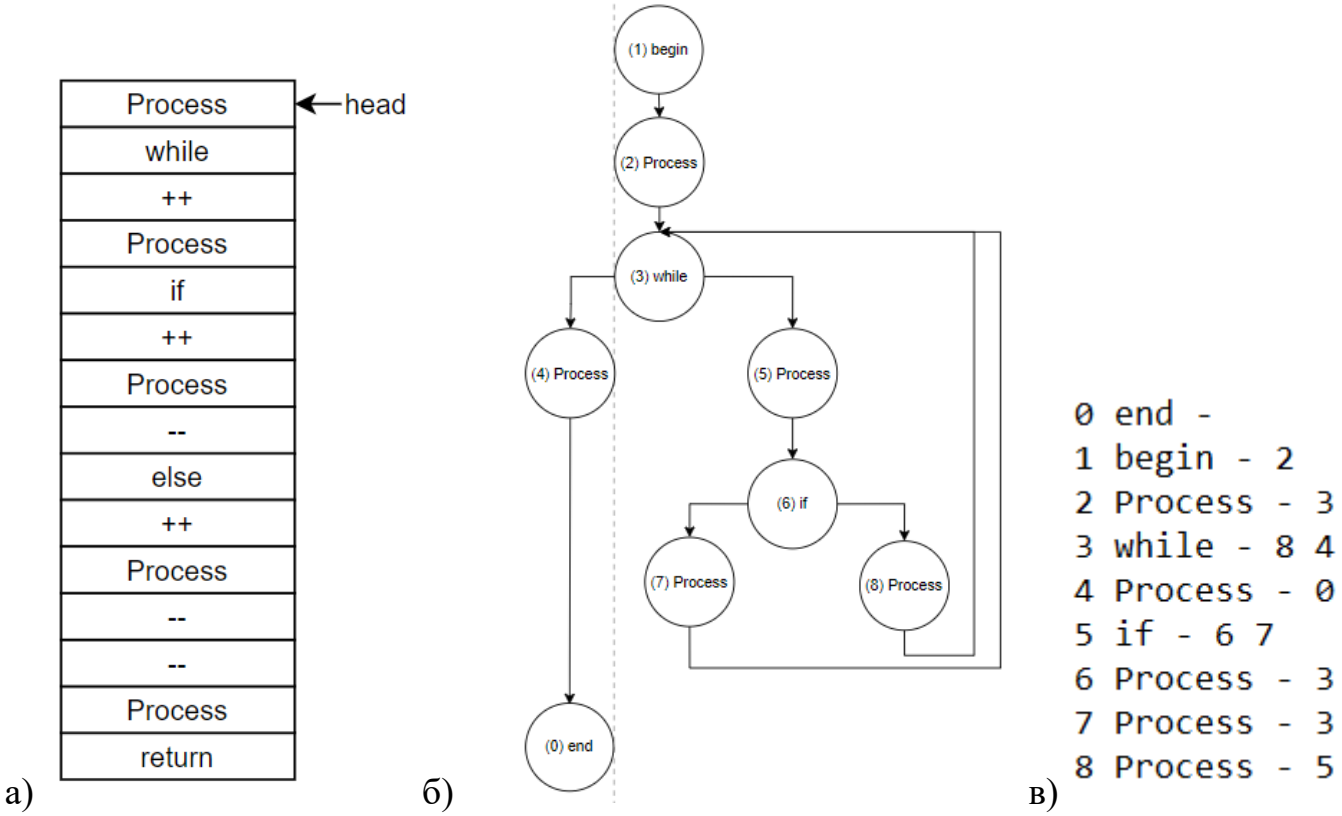
Лістинг 7 було перетворено у *list* – конструкцію побудованою за допомогою C_T (рис. 4.23, а), head – голова списку, принцип роботи списку – FIFO. Граф потоку керування (рис. 4.23, б) будується відповідно з вищезазначеними прикладами за допомогою формул 2.13 – 2.41.

Графи керування, побудовані за допомогою програмного забезпечення, представлено у вигляді списку суміжності для:

– програмного коду на рис. 4.23, в;
– блок-схеми на рис 4.22.

```
0 end -
1 begin - 2
2 Process - 3
3 while - 8 4
4 Process - 0
5 if - 6 7
6 Process - 3
7 Process - 3
8 Process - 5
```

Рисунок 4.22 – Граф керування у вигляді списку суміжності побудований за блок-схемою



а) б) в)

Рисунок 4.23 – Схема конструкцій, побудованих C_T, C_G та програмним додатком: а) список керуючих операторів б) очікуваний граф керування в) отриманий граф керування

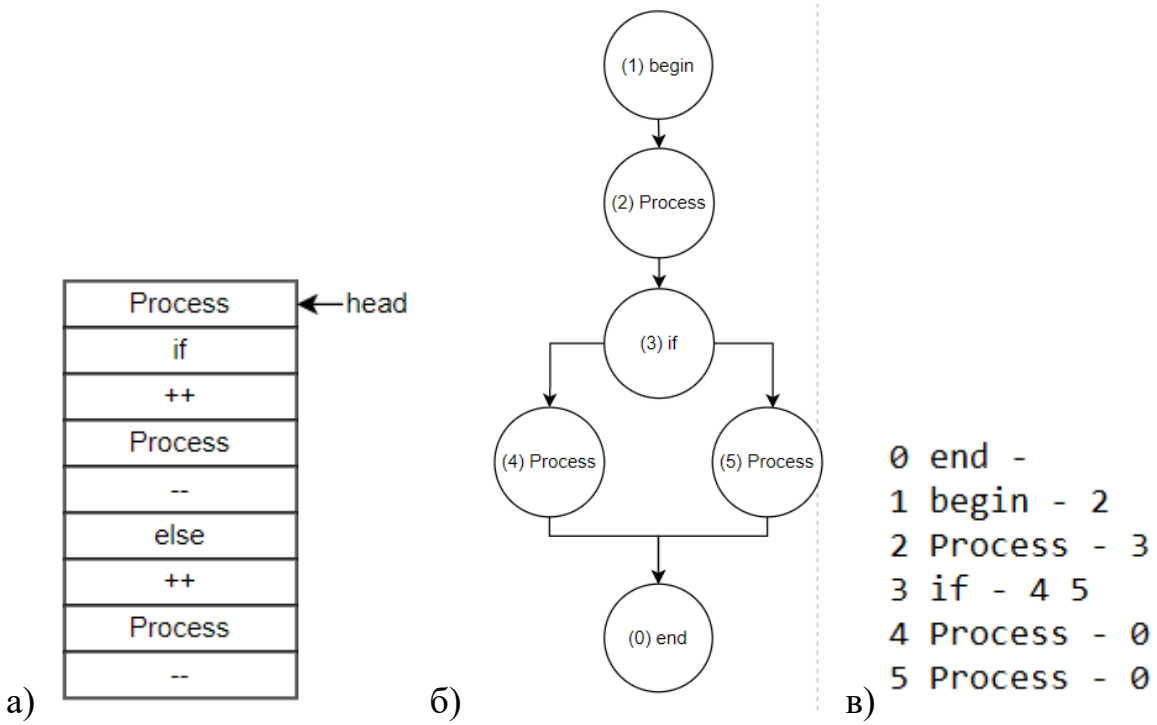
Лістинг 8 було перетворено у *list* – конструкцію побудованою за допомогою C_T (рис. 4.25, а), head – голова списку, принцип роботи списку – FIFO. Граф потоку керування (рис. 4.25, б) будується відповідно з вищезазначеними прикладами за допомогою формул 2.13 – 2.41.

Графи керування, побудовані за допомогою програмного забезпечення, представлено у вигляді списку суміжності для:

– програмного коду на рис. 4.25, в;
– блок-схеми на рис 4.24.

```
0 end -
1 begin - 2
2 Process - 3
3 if - 4 0
4 Process - 0
```

Рисунок 4.24 – Граф керування у вигляді списку суміжності побудований за блок-схемою



а) б) в)

Рисунок 4.25 – Схема конструкцій, побудованих C_T, C_G та програмним додатком: а) список керуючих операторів б) очікуваний граф керування в) отриманий граф керування

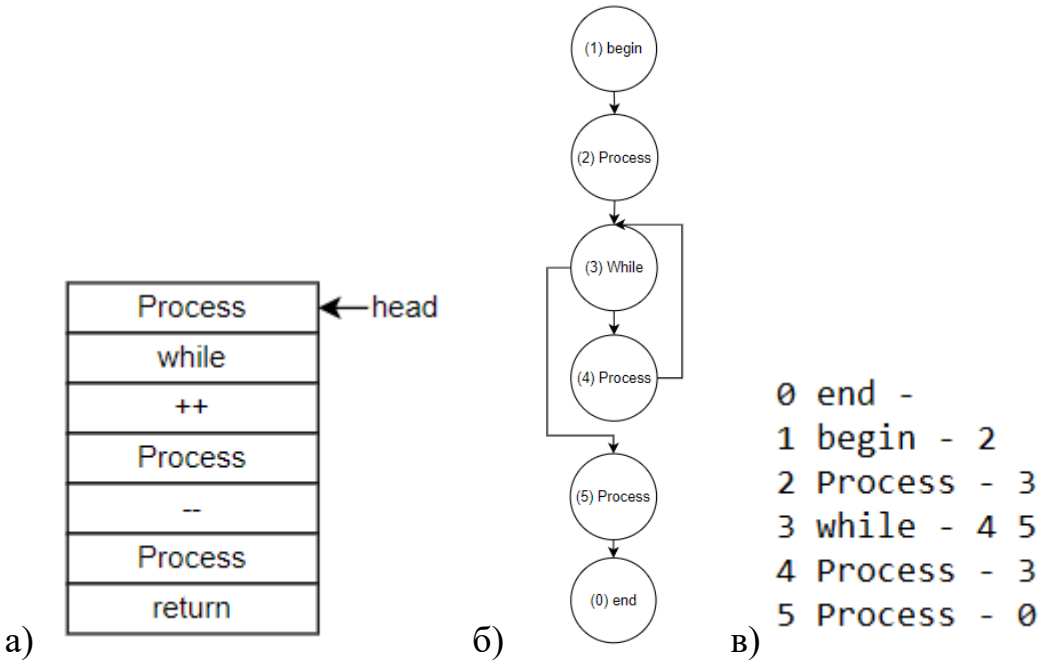
Лістинг 9 було перетворено у *list* – конструкцію побудованою за допомогою C_T (рис. 4.27, а), head – голова списку, принцип роботи списку – FIFO. Граф потоку керування (рис. 4.27, б) будується відповідно з вищезазначеними прикладами за допомогою формул 2.13 – 2.41.

Графи керування, побудовані за допомогою програмного забезпечення, представлено у вигляді списку суміжності для:

– програмного коду на рис. 4.27, в;
– блок-схеми на рис 4.26.

```
0 end -
1 begin - 2
2 Process - 3
3 if - 4 0
4 Process - 0
```

Рисунок 4.26 – Граф керування у вигляді списку суміжності побудований за блок-схемою



а) б) в)

Рисунок 4.27 – Схема конструкцій, побудованих C_T, C_G та програмним додатком: а) список керуючих операторів б) очікуваний граф керування в) отриманий граф керування

4.3 Результат експерименту

Схожість побудованих графів керування виражається в числовому еквіваленті за формулою:

$$Match(A,B) = \frac{|A'| + |B'| - 2}{|A| + |B| - 2} \times 100, \tag{4.3}$$

де A, B – досліджувані графи, $|A|, |B|$ – кількість вершин у відповідних графах, $|A'|, |B'|$ – кількість помічених верши.

Обчислимо схожість для очікуваних графів керування побудованих за лістингом 1 та рис 4.1:

$$Match(A,B) = \frac{|A'| + |B'| - 2}{|A| + |B| - 2} \times 100 = \frac{|3| + |3| - 2}{|3| + |3| - 2} = 100\%, \tag{4.4}$$

За аналогією обчислимо схожість графів для кожної пари з набору даних. Обчислена схожість для очікуваних графів та для отриманих графів, яка отримана за допомогою програмного забезпечення відображена у табл. 4.1.

Таблиця 4.1 – Схожість графів керування

Номер пари	Очікуваний результат	Отриманий результат
1	100	100
2	100	100
3	100	100
4	100	100
5	100	100
6	100	100
7	100	100
8	88	88,89
9	65	66,7

Висновок до четвертого розділу

В даному розділі описані випробування та їх результати. Було обчислено очікувану схожість між програмним кодом та блок-схемою. Також було отримано схожість, що обчислювалась за допомогою програмного додатку.

Були виконані випробування для наступних типів алгоритмів:

- лінійний алгоритм;
- алгоритм розгалуження;
- циклічний алгоритм з пост умовою;
- циклічний алгоритм з передумовою;
- вкладеність циклічних алгоритмів;
- вкладеність алгоритмів розгалуження;
- вкладеність розгалуження в циклічний алгоритм.

Для кожної пари з набору даних було побудовано: список керуючих операторів за програмним кодом, очікуваний граф потоку керування за списком керуючих операторів та очікуваний граф потоку керування за блок-схемою. Також аналогічні графи було побудовано за допомогою програмного додатку, та представлено у вигляді списку суміжності.

Всі отримані результати співпадають з очікуваними. Отже, можна стверджувати, що розроблений метод визначення відповідності тексту програми та його графічного зображення працює коректно.

5 ОХОРОНА ПРАЦІ ТА БЕЗПЕКА В НАДЗВИЧАЙНИХ СИТУАЦІЯХ

Розроблений програмний додаток складається з однієї клієнтської частини. Даний компонент програмного продукту призначений для розгортання на персональних ЕОМ, таких як персональний комп'ютер та ноутбук. Для взаємодії з програмним додатком необхідний повний комплект аксесуарів – комп'ютера миша, клавіатура та монітор. До основних дій при взаємодії з ПЗ можна віднести:

- завантаження вхідних даних до програмного додатку;
- перегляд та аналіз побудованих графів керування за вхідними даними;
- перегляд результатів порівняння між собою вхідних даних.

Для дотримання належного рівня працездатності працівників робочі місце та середовище повинні відповідати певним нормам безпеки. Також від самих працівників потребується дотримання визначених норм та правил роботи.

За регулювання вищезазначених елементів використовуються група нормативно-правових актів з охорони праці (скорочено НПАОП) та державні санітарні правила і норми роботи (скорочено ДСанПіН).

Виходячи з того, що робота з програмним додатком перш за пов'язана з взаємодією із комп'ютерними пристроями, слід використовувати НПАОП та ДСанПіН, які відносяться до роботи з комп'ютерною технікою та відповідним приміщенням.

5.1 Вимоги безпеки при виконанні робіт на робочому місці

Вимоги стосовно умов праці в основному трактуються законодавством України, а саме Законом України «Про охорону праці» від 2002 р. [50].

У свою чергу, основні положення, які трактують основні правила та вимоги, які стосуються професії та робочого процесу інженера-програміста можна поділити на дві умовні групи:

- загальні;
- відносно роботи з комп'ютерною технікою.

До загальних можна віднести наступні положення про санітарні норми, правила та вимоги:

- Державні санітарні норми виробничого шуму, ультразвуку та інфразвуку ДСН 2.3.6.037-99, затверджені постановою Головного державного санітарного лікаря України від 01.12.99 р. № 37 [51];
- Державні санітарні норми виробничої загальної та локальної вібрації ДСН 3.3.6.039-99, затверджені постановою Головного державного санітарного лікаря України від 01.12.99 р. № 39 [52];
- Державні санітарні норми мікроклімату виробничих приміщень ДСН 3.3.6.042-99, затверджені постановою Головного державного санітарного лікаря України від 01.12.99 р. № 42 [53];
- Загальні вимоги стосовно забезпечення роботодавцями охорони праці працівників, затверджені наказом МНС від 25.01.2012 р. № 67.

До нормативно-правових актів, пов’язаних з організацією експлуатації комп’ютерної техніки, належать:

- НПАОП 0.700-7.15-18 Вимоги щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями, затверджені від 14.02.2018 р. № 207 [54];
- Правила безпеки під час навчання в кабінетах інформатики навчальних закладів системи загальної середньої освіти (НПАОП 80.0-1.12-04) [55];
- Інструкція щодо надання послуг з ремонту побутової радіоелектронної апаратури [56];
- Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин (ДСанПіН 3.3.2-007-98) [57];

При розробці програмного додатку основою робочого процесу є взаємодія з ЕОМ. Відповідно до НПАОП 0.700-7.15-18 [58] можна визначити наступні групи вимог:

- вимоги безпеки до робочих місць працівників з екранними пристроями;
- мінімальні вимоги безпеки до екранних пристроїв;

– мінімальні вимоги безпеки під час роботи з екранними пристроями.

Вимоги безпеки до робочих місць працівників з екранними пристроями:

– робоче місце повинне буде спроектоване таким чином, щоб надавати працівнику необхідний простір для зміни робочого положення та рухатися;
– випромінення від екранних пристроїв (до даного терміну відносять такі фактори як шум, вібрації, забруднювачі, температура) повинне бути зведене до граничного допустимого рівня відповідно до потреб безпеки та охорони здоров'я;
– організація та розташування робочого місця та його елементів повинні відповідати ергономічним, антропологічним, психофізіологічним вимогам та характеру робіт;
– освітлення робочого місця повинне надавати необхідний контраст між екраном та навколишнім середовищам та відповідати вимогам ДСанПІН 3.3.2.007-98 [57];
– мікроклімат повинен підтримуватися на постійному рівні та відповідно до вимог Санітарних норм мікроклімату виробничих приміщень ДСН 3.3.6.042-99 [53];
– робочій стіл або поверхня повинна мати достатній розмір та поверхню з низькою відбивною здатністю та також допускати гнучкість при розміщенні документів, пристроїв (екрану або клавіатури) та відповідного устаткування;
– робоче крісло повинне дозволяти працівнику приймати зручне положення та легко рухатися.

Мінімальні вимоги безпеки під час роботи з екранними пристроями:

– на початку кожного дня необхідно провидити очистку екранного пристрою від пилу та інших забруднень;
– в кінці робочого дня та при виникненні аварійної ситуації необхідно відключити екранний пристрій від мережі;

– при виконанні робіт операторського типу у приміщеннях необхідно підтримувати оптимальні умови мікроклімату відповідно до ДСН 3.3.6.042-99 [53].

Не допускається:

– виконувати технічне обслуговування, ремонт або налагодження екранних пристроїв на робочому місці працівника та під час роботи;
– відключати захисні пристрої, самому змінювати конструкцію та складові пристроїв або їх технічне налагодження;
– працювати з екранними пристроями які мають прояви несправності (виникають нехарактерні сигнали, нестабільне зображення та інше).

5.2 Шкідливі виробничі фактори на підприємстві

Шкідливий виробничий фактор – це елемент робочого середовища, який може впливати на робітника підчас робочого процесу, а також при певних умовах може викликати захворювання або зменшення працездатності. Відповідно до тривалості та інтенсивності впливу шкідливого виробничого фактору, він може становити небезпеку для здоров‘я персоналу.

Виходячи з робочого процесу інженера-програміста можна визначити наступні шкідливі виробничі фактори робочого середовища:

– поганий мікроклімат приміщення (підвищена або знижена температура повітря, запиленість, загазованість повітря, підвищена або знижена вологість повітря);
– недостатня освітленість робочого місця;
– занадто високий рівень шуму;
– занадто високий рівень іонізуючого випромінювання;
– занадто високий рівень електромагнітних полів;
– занадто високий рівень статичної електрики;
– небезпека поразки електричним струмом;
– недостатня контрастність або яскравість екрана монітору;
– порушення ергономічних норм.

5.2.1 Мікроклімат приміщення

Мікроклімат робочого приміщення – це клімат внутрішнього середовища визначеного приміщення, який визначається фізичними показниками, які у свою чергу мають вплив на організм людини.

До показників мікроклімату можна віднести:

– температуру повітря;
– відносну вологість;
– швидкість руху повітря.

Відповідно від пори року, характеру трудового процесу і характеру виробничого приміщення встановлені рекомендовані значення для кожного з показників. Відповідно до класифікації тяжкості робочого процесу та ДСН 3.3.6.042-99 [53] у таблиці 5.1 представлені значення параметрів мікроклімату та показники робочого приміщення, де проводилася робота над розробкою програмного продукту.

Таблиця 5.1 – Нормативні та визначені параметри мікроклімату приміщення

Період року	Параметри мікроклімату	Нормативна величина	Величина виміряна у приміщенні
Холодний	Температура повітря в приміщенні	22-24°С	23,5°С
	Відносна вологість	40-60%	54,7%
	Швидкість руху повітря	До 0.1 м/с	0,084 м/с
Теплий	Температура повітря в приміщенні	23 - 25° С	24,32°С
	Відносна вологість	40-60%	50,2%
	Швидкість руху повітря	До 0.1 – 0.2 м/с	0,094 м/с

Для підтримання нормативних параметрів мікроклімату слід використовувати зволожувачі повітря. Також приміщення повинно бути обладнане кондиціонерами повітря, або мати систему вентиляції.

Також у контексті мікроклімату приміщення слід розглянути показники подачі свіжого повітря до робочого приміщення відповідно до пори року, об‘єму

приміщення та кількості людей. У таблиці 5.2 представлені рекомендовані норми подачі свіжого повітря в приміщення з ПК.

Таблиця 5.2 – Норми подачі свіжого повітря в приміщення з ПК

Характеристика приміщення	Об’ємна витрата свіжого повітря, що подається в приміщення, м3 на одну людину в годину
Об’єм до 20м^3 на людину	не менше 30
20 - 40 м^3 на людину	не менше 20
Більше 40 м^3 на людину	може бути використана природна вентиляція

Виходячи з отриманих даних можна стверджувати, що робоче приміщення в якому проходив процес розробки програмного забезпечення повністю відповідає визначеним нормам та вимогам.

5.2.2 Освітлення робочого місця

Виходячи з того, що робочий процес розробки програмного додатку проходить у приміщенні, освітлення повинне наближатися до оптимальних умов зорового сонячного рівня. Загалом вимоги до освітлення приміщень із встановленими ПК наступні:

– загальна освітленість 300 лк та 750 лк комбінована – при виконанні зорових робіт високої точності;
– загальна освітленість 200 лк та 300 лк комбінована – при виконанні зорових робіт середньої точності;

Слід зазначити, що випромінювачі світла повинні бути встановлені таким чином, щоб не створювати полисків на поверхні екрана. Саме поле зору повинне бути освітлено рівномірно.

Процес розробки програмного додатку слід віднести до типу робіт середньої точності. У свою чергу, значення загальної освітленості робочого місця дорівнювало 215,62 лк, а загальна освітленість дорівнювало 284,75 лк. Для освітлення робочого приміщення було використано природній та штучний джерела освітлення, які були встановлені відповідно до вимог.

Також слід зазначити, що екран монітора також є джерелом світла. При довгій роботі з даним пристроєм з‘являється стомленість очей. Це може бути обумовлене високим рівнем яскравості, встановленим користувачем. Також миготіння та нечітка картинка можуть ускладнювати робочий процес. Для уникнення подібних ситуацій слід персонально встановлювати рівень яскравості зображення, а частота кадрів у свою чергу повинне бути не меншим за 75 Гц для ЕЛТ-моніторів та 60 Гц для ЖКИ-моніторів.

5.2.3 Шум та вібрації

Вібрації – це коливання твердих тіл, частин апаратів, машин, устаткування, споруд, що сприймаються організмом людини як струс. У свою чергу шум – це звуковий процес, який викликає дискомфорт та негативно впливає на організм людини.

Після тривалої дії шуму інтенсивності вище 80 дБ можуть виникнути патології слухового органу та негативно впливає на нервову систему. Під дією шуму працівник швидко втомлюється, що може привести до виробничих помилок під час робочого процесу.

Зниження рівня шуму та вібрації може бути досягнуто наступними засобами:

– стіни та стеля приміщення можуть бути облицьовані звуковбирний матеріалами;
– використанням м‘яких килимків, прокладок із гуми або повсті або спеціальних кожухів;
– використанням працівниками навушників, вкладишів та ін.

Джерелом шуму та вібрацій у робочому середовищі працівників сфери програмування загалом є різного роду апаратура. Основними представниками є:

– персональний комп‘ютер або ноутбук та його складові;
– принтери;
– інша комп’ютерна апаратура (телевізійні екрани, та ін.).

Головним та єдиним джерелом шуму та вібрацій у робочому середовищі, у якому розроблявся програмний продукт, був саме персональний комп‘ютер. Загальний рівень шуму становив 35 дБ, а рівень вібрацій ПК — 65,9 дБ. Слід

відзначити, що рівень вібрації на самому робочому місці дорівнює нулю (не фіксується пристроями). Дані значення відповідно до ДСН 3.3.6.039-99 [52] відповідають вимогам.

5.3 Дії працівників в надзвичайних ситуаціях

5.3.1 Порядок надання домедичної допомоги

Відповідно до Наказу МОЗ №398 від 16.06.2014 «Про затвердження порядків надання домедичної допомоги особам при невідкладних станах» [58] встановлений перелік документів, кожний з яких являє собою порядок дій при наданні домедичної допомоги. Загалом даний документ включає в себе перелік із 29 протоколів дій на окремі випадки, які потребують невідкладної допомоги.

В контексті роботи над програмним додатком найбільший ризик для працівника походив від великої кількості електронних приладів, які у свою чергу є джерелом ризику ураження електричним струмом. Відповідно до Порядку надання домедичної допомоги постраждалим при ураженні електричним струмом та блискавкою [59] нижче представлений порядок дій при наданні домедичної допомоги при ураженні електричним струмом:

- переконатися у відсутності небезпеки;
- якщо джерело струму продовжує діяти на постраждалого, необхідно вимкнути його від електромережі за допомогою електронепровідного засобу;
- провести огляд життєвих показників постраждалого;
- викликати бригаду медичної допомоги;
- при відсутності у постраждалого дихання необхідно провести серцево-легеневу реанімацію
- при відсутності свідомості надати постраждалому стабільного положення;
- накласти стерильну пов‘язку на місце опіку;
- забезпечити нагляд за потерпілим до прибуття бригади невідкладної допомоги;
- повторно зателефонувати диспетчеру при погіршенні стану потерпілого.

5.3.2 Пожежна безпека

Задля запобігання пожежі у робочому приміщенні необхідно дотримуватися загальних правил пожежної безпеки [60].

У виникненні ознак пожежі необхідно дотримуватися наступних правил:

- повідомити службу пожежної безпеки про факт пожежі за вашою адресою;
- за можливістю вжити заходів протидії пожежі за допомогою вогнегасника;
- розпочати процес евакуації приміщення;
- повідомити керівництво про факт пожежі;
- перед тим як схопитися за ручку дверей, доторкніться до неї тільною стороною долоні задля перевірки температури;
- обов‘язково зачиняйте за собою усі двері;
- при великій задимленості приміщення пересувайтесь поповзом.

Висновки до п’ятого розділу

У даному розділі були розглянути основні питання, які стосуються безпеки та охорони праці під час процесу розробки програмного додатку.

Були розглянуті основоположні документі, які визначають необхідні норми та вимоги до робочого процесу, а також в цілому до середовища. Були визначені основні параметри мікроклімату, вібрації, шуму та освітлення для конкретного робочого. Після їх визначення результати були співставленні з нормативними значеннями, описаними у відповідних документах.

Також були розглянуті порядки дій при наданні домедичної допомоги та дії при пожежній небезпеці. Були розглянуті відповідні нормативні документи з протоколами дій для запобігання та під час самої надзвичайної ситуації.

ВИСНОВКИ

В результаті роботи над дипломною роботою було розроблено метод визначення відповідності тексту програми графічному представленню алгоритму. Метод було розроблено з використанням конструктивно-продукційного підходу.

Було проведено аналіз сучасного стану методів визначення відповідності між програмним кодом та блок-схемою. Дослідження показали, що на даний момент не існує жодного методу для порівняння між собою на предмет схожості програмної реалізації та блок-схеми алгоритму, яка використовують конструктивно-продукційне моделювання.

Було визначено два конструктора, котрі використовуються для визначення відповідності тексту програми графічному представленню алгоритму, а саме:

- C_T – конструктор побудови проміжного представлення у вигляді списку керуючих елементів за програмним кодом на мові C++;
- C_G – конструктор побудови графу потоку керування за проміжним представленням у вигляді списку керуючих елементів.

Також було визначено основні алгоритми для кожного конструктору:

- для конструктору ${}_{I,C_A}C_T$: $\left(A_1^0 \,|_{A_i,A_j}^{A_i \cdot A_j} \hookleftarrow \cdot\right)$, $\left(A_2^0 \,|_{S}^{A_1} \hookleftarrow :\right)$, $\left(A_3 \,|_{el,list}^{list} \hookleftarrow +\right)$, $\left(A_4 \,|_{l_h,l_q,f_i}^{f_i} \hookleftarrow \Rightarrow\right)$, $\left(A_5 \,|_{f_i,\Psi}^{f_j} \hookleftarrow |\Rightarrow\right)$, $\left(A_6 \,|_{\sigma,\Psi}^{\bar{\Omega}} \hookleftarrow ||\Rightarrow\right)$;
- для конструктору ${}_{I,C_A}C_G$: $\left(A_1^0 \,|_{A_i,A_j}^{A_i \cdot A_j} \hookleftarrow \cdot\right)$, $\left(A_2^0 \,|_{S}^{A_1} \hookleftarrow :\right)$, $\left(A_3 \,|_{l_h,l_q,f_i}^{f_i} \hookleftarrow \Rightarrow\right)$, $\left(A_4 \,|_{f_i,\Psi}^{f_j} \hookleftarrow |\Rightarrow\right)$, $\left(A_5 \,|_{\sigma,\Psi}^{\bar{\Omega}} \hookleftarrow ||\Rightarrow\right)$, $\left(A_6 \,|_{c,n,L}^{L} \hookleftarrow \div\right)$, $\left(A_7 \,|_{a,b}^{a} \hookleftarrow :=\right)$, $\left(A_8 \,|_{index,V}^{name} \hookleftarrow \%\right)$, $\left(A_9 \,|_{Q}^{x} \hookleftarrow \#\right)$, $\left(A_{10} \,|_{el,stack}^{stack} \hookleftarrow +\right)$, $\left(A_{11} \,|_{el,stack}^{el} \hookleftarrow -\right)$, $\left(A_{12} \,|_{el,list}^{el} \hookleftarrow --\right)$, $\left(A_{13} \,|_{a,b}^{c} \hookleftarrow \times\right)$, $\left(A_{14} \,|_{Q_1,Q_2}^{Q} \hookleftarrow \cup\right)$, $\left(A_{15} \,|_{G_1,G_2}^{G} \hookleftarrow \tilde{\cup}\right)$.

Алгоритми було реалізовано у програмному додатку, що визначає відповідність між програмним кодом на мові C++ та блок-схемою у json-форматі.

Було проведено випробування, які показали коректність розробленого методу. Отже, мета дипломної роботи була досягнута, оскільки розломлений метод визначення відповідності тексту програми графічному представленню алгоритму, довів свою працездатність.

Напрям подальших досліджень:

- вдосконалити розроблений метод, для розпізнавання образів на цифровому зображення, що сприймати блок-схему як зображення;
- розглядати елементи блок-схеми, як фігури, які не містять тексту.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. Малышев Е.В., Смелов В.В. Алгоритм распознавания плагиатов кодов программ // Труды БГТУ. Серия 3: Физико-математические науки и информатика. . 2018. № 1 (206). С.135 – 138.
2. Вихтенко Э.М., Карманов Д.А., Син Д.З. Информационная система «плагиат в программах студентов» //Вестник Тихоокеанского государственного университета. – 2019. – №. 3. – С. 25 – 34.
3. Pandit A.A., Toksha G. Review of plagiarism detection technique in source code // International Conference on Intelligent Computing and Smart Communication 2019. Singapore: Springer, 2020. С.393 – 405. DOI: 10.1007/978-981-15-0633-8_38
4. Сарвар С. Выявление характерных особенностей программ для борьбы с компьютерным пиратством на основе интеллектуального анализа графов // Труды Института системного программирования РАН. 2019. № 31 (2). С.171 – 186. DOI: 10.15514/ISPRAS-2019-31(2)-12
5. Никитин В.Д., Иванов А.П. Разработка методов оценки сходства алгоритмов на основе графовых моделей // Magyar Tudományos Journal. 2018. № 23. С.36-41.
6. Khaled F., H. Al-Tamimi M.S. Plagiarism Detection Methods and Tools: An Overview // Iraqi Journal of Science. 2021. № 62 (8). С.2771-2783. DOI: 10.24996/ijs.2021.62.8.30
7. Kulkarni S., Govilkar S., Amin D. Analysis of Plagiarism Detection Tools and Methods // SSRN. 2021. № 1. С.1-7 URL: https://ssrn.com/abstract=3869091. DOI: 10.2139/ssrn.3869091
8. Шинкаренко В.И., Куропятник Е.С. Проблемы выявления плагиата и анализ инструментального программного обеспечения для их решения // Наука та прогрес транспорту. 2017. № 1 (67). С.131 – 142.
9. Евтифеева О. А. и др. Анализ алгоритмов поиска плагиата в исходных кодах программ //Научно-технический вестник информационных технологий, механики и оптики. – 2007. – №. 39.

10. Осипов И. и др. Методы обнаружения плагиата в текстах студенческих работ //Альманах научных работ молодых ученых университета итмо. – 2016. – С. 95 – 98.
11. Половикова О. Н., Иванова В. Е. Разработка детектора автоматической проверки на плагиат блоков программного кода для образовательной среды //Высокопроизводительные вычислительные системы и технологии. – 2020. – Т. 4. – №. 1. – С. 173 – 178.
12. Бланк Я. А., Савкин М. К. Граф потока управления //им. НЭ Баумана© Издательство МГТУ им. НЭ Баумана, 2016. – 2016. – С. 75.
13. «Введение в абстрактное синтаксическое дерево» [Ел. ресурс]. Available: https://russianblogs.com/article/62501151692/
[Дата звернення: 20.10.2021].
14. Блок-схемы алгоритмов. ГОСТ. Примеры [Ел. ресурс]. Режим доступу: https://pro-prof.com/archives/1462
[Дата звернення: 20.10.2021].
15. Яне, Б. Цифровая обработка изображений / Б. Яне ; Б. Яне ; пер. с англ. А. М. Измайловой. – Москва : Техносфера, 2007. – С. 528 – 575. – (Мир цифровой обработки). – ISBN 978-5-94836-122-2.
16. Журавель І. М. Метод бінаризації металографічних зображень з оптимальним порогом //Штучний інтелект. – 2012.
17. Гусев В. Ю., Крапивенко А. В. Методика фильтрации периодических помех цифровых изображений //Труды МАИ. – 2012. – №. 50. – С. 34 – 34.
18. Седов М. О. Адаптивное дискретное вейвлет-преобразование //T-Comm-Телекоммуникации и Транспорт. – 2012. – №. 9.
19. Huo Y. K. et al. An adaptive threshold for the Canny Operator of edge detection //2010 international conference on image analysis and signal processing. – IEEE, 2010. – С. 371 – 374.
20. Корреляция и отслеживание цифрового изображения [Ел. ресурс]. Режим доступу: https://star-wiki.ru/wiki/Digital_image_correlation_and_tracking [Дата звернення: 20.10.2021].

21. Огнев И. В., Сидорова Н. А. Обработка изображений методами математической морфологии в ассоциативной осцилляторной среде //Известия высших учебных заведений. Поволжский регион. Технические науки. – 2007. – №. 4.
22. Сакович И. О., Белов Ю. С. Обзор основных методов контурного анализа для выделения контуров движущихся объектов //Инженерный журнал: наука и инновации. – 2014. – №. 12 (36).
23. Самойленко Д. Е. Структурная сегментация изображений //Штучний інтелект. – 2004. – №. 4. – С. 521 – 528.
24. Мальченко Є. Є. Розробка системи розпізнавання тексту на зображенні : дис. – КПІ ім. Ігоря Сікорського, 2020.
25. Шерстюк В. Г. Розробка програмних інструментів розпізнавання тексту по зображеннях документів. – 2020.
26. Помаз С. А. Оптичне розпізнавання тексту за допомогою нейромережі : дис. – Чернігів, 2021.
27. Багдонас А. Читающее устройство «РУТА 701» //Автоматизация ввода письменных знаков в электронно-вычислительные машины: Докл. научно-техн. совещ. – 1968. – С. 96 – 121.
28. Котович Н. В., Славин О. А. Распознавание скелетных образов //Информационные технологии и вычислительные системы. – 2000. – №. 4. – С. 204-215.
29. Метляев С. В. Распознавание скелетных образов //Решетневские чтения. – 2009. – Т. 2. – №. 13.
30. Афонасенко А. В., Елизаров А. И. Обзор методов распознавания структурированных символов //Доклады Томского государственного университета систем управления и радиоэлектроники. – 2008. – №. 2 – 1 (18).
31. Абрамов Е. С. Моделирование систем распознавания изображений (на примере печатных текстов). – 2006.
32. Марчук Д. К., Скачков В. О. Використання нейронної мережі для розпізнавання друкованих символів. – 2017.

33. Гришанов К. М., Рыбкин С. В. Тестирование сверточной нейронной сети в задачах машинного зрения //Электронный журнал: наука, техника и образование. – 2017. – №. 2. – С. 186 – 193.
34. Местецкий Л. М., Рейер И. А. Распознавание формы растровых бинарных изображений плоских фигур с использованием морфинга контуров границы. – 2000.
35. Copyleaks Plagiarism Software [Ел. ресурс]. Режим доступу: https://copyleaks.com/
[Дата звернення: 20.10.2021].
36. Copyleaks Plagiarism Software Pricing [Ел. ресурс]. Режим доступу: https://copyleaks.com/ru/pricing/product/businesses/step/1
[Дата звернення: 20.10.2021].
37. Copyleaks Plagiarism Software Pricing [Ел. ресурс]. Режим доступу: https://copyleaks.com/ru/pricing/product/education/step/1
[Дата звернення: 20.10.2021].
38. Шинкаренко В. И. Конструктивно-продукционные структуры и их грамматические интерпретации. I. Обобщенная формальная конструктивнопродукционная структура / В. И. Шинкаренко, В. М. Ильман // Кибернетика и системный анализ. – Киев, 2014. – том 50, №5 – с. 8 – 16
39. Фу К. Структурные методы в распознавании образов. Пер. с англ. Под ред. Айзермана М. А. [Текст] / К. Фу – М.: «Мир», 1977 – 317 с.
40. Алферова З. В. Теория алгоритмов [Текст] / З. В. Алферова – М.: Статистика, 1973 – 137 с.
41. Ту Дж., Гонсалес Р. Принципы распознавания образов. Пер. на рус. яз. [Текст] / Дж. Ту, Р. Гонсалес – М.: «Мир», 1978 – 414 с.
42. Андон Ф. И. Алгеброалгоритмические модели и методы параллельного программирования / Ф. И. Андон, А. Е. Дорошенко, Г. Е. Цейтлин, Е. А. Яценко – К.: Академпериодика, 2007. – 634 с.
43. Кроновер Р. М. Фракталы и хаос в динамических системах. Основы теории. М., 2000. — 352 с.

44. Алгоритмы. Построение и анализ:[пер. с англ.] / Т. Кормен та ін. Москва: Вильямс, 2005. 1296 с.
45. QAInfo Use-case-diagrams [Ел. ресурс]. Режим доступу: https://www.quality-assurance-group.com/use-case-diagrams/
[Дата звернення: 25.10.2021].
46. Uk.education – Діаграма класів [Ел. ресурс]. Режим доступу: https://uk.education-wiki.com/7917237-class-diagram
[Дата звернення: 25.10.2021].
47. Основні поняття інтерфейсів користувача та засоби їх проектування [Ел. ресурс]. Режим доступу: http://elar.khnu.km.ua/jspui/bitstream/123456789/1415/2/Rozdil1.pdf
[Дата звернення: 25.10.2021].
48. Діаграма станів [Ел. ресурс]. Режим доступу: http://mmsa.kpi.ua/sites/default/files/disciplines/%D0%A0%D0%BE%D0%B7%D1%80%D0%BE%D0%B1%D0%BA%D0%B0%20%D1%96%20%D1%82%D0%B5%D1%81%D1%82%D1%83%D0%B2%D0%B0%D0%BD%D0%BD%D1%8F%20%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC/didkovska_m_v_testing_lecture_4.pdf
[Дата звернення: 25.10.2021].
49. Понятие алгоритма. Виды алгоритмов [Ел. ресурс]. Режим доступу: https://intuit.ru/studies/courses/16740/1301/lecture/25624
[Дата звернення: 10.11.2021].
50. Закони України «Про охорону праці» [із змінами], – К., 2002; «Про загальнообов'язкове державне соціальне страхування». К., 2012 р.; [із змінами], - К., 2014.
51. ДСН 2.3.6.037-99, Державні санітарні норми виробничого шуму, ультразвуку та інфразвуку затверджені від 01.12.99 р. № 37;
52. ДСН 3.3.6.039-99 Державні санітарні норми виробничої загальної та локальної вібрації, затверджені від 01.12.99 р. № 39;

53. ДСН 3.3.6.042-99, Державні санітарні норми мікроклімату виробничих приміщень затверджені від 01.12.99 р. № 42;
54. НПАОП 0.700-7.15-18 Вимоги щодо безпеки та захисту здоров’я працівників під час роботи з екранними пристроями, затверджені від 14.02.2018 р. № 207;
55. НПАОП 80.0-1.12-04 Правила безпеки під час навчання в кабінетах інформатики навчальних закладів системи загальної середньої освіти, затверджений 16.03.2004 р. № 81;
56. Інструкція щодо надання послуг з ремонту побутової радіоелектронної апаратури, затверджені від 27.08.2000 р. № 20;
57. ДСанПіН 3.3.2.007-98 Гігієнічні вимоги до організації роботи з візуальними дисплейними терміналами електронно-обчислювальних машин, затверджені від 10.12.1998 р. № 7;
58. Наказ №398 Про затвердження порядків надання домедичної допомоги особам при невідкладних станах від 16.06.2014;
59. Порядку надання домедичної допомоги постраждалим при ураженні електричним струмом та блискавкою, затверджений від 16.06.2014 р. № 398;
60. НАПБ А.01.001-2014 Правила пожежної безпеки в Україні, затверджений від 30.12.2014 Наказом № 1417

ДОДАТКИ

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖУЮ

Перший проректор Дніпровського національного університету залізничного транспорту імені академіка В. Лазаряна

Борис БОДНАР

СИСТЕМА ВИЗНАЧЕННЯ ВІДПОВІДНОСТІ ТЕКСТУ ПРОГРАМИ ГРАФІЧНОМУ ПРЕДСТАВЛЕННЮ АЛГОРИТМУ

Технічне завдання

ЛИСТ ЗАТВЕРДЖЕННЯ

1116130.01222-01-ЛЗ

Завідувач кафедри КІТ

Вадим ГОРЯЧКІН

Керівник розробки

Олена КУРОП`ЯТНИК

Виконавець

Богдан ЯКОВЕНКО

Нормоконтролер

Олена КУРОП`ЯТНИК

2021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖЕНО

1116130.01222-01

СИСТЕМА ВИЗНАЧЕННЯ ВІДПОВІДНОСТІ ТЕКСТУ ПРОГРАМИ ГРАФІЧНОМУ ПРЕДСТАВЛЕННЮ АЛГОРИТМУ

Технічне завдання

Аркушів 21

2021

АНОТАЦІЯ

Документ 1116130.01222-01 ««Система визначення відповідності тексту програми графічному представленню алгоритму». Технічне завдання» входить до складу програмної документації додатку, який надає можливість порівнювати між собою алгоритм (програмна реалізація) та його графічне представлення (блок-схема).

У даному документі представлено призначення програми, область застосування програмного продукту, основні вимоги, стадії та строки розробки проекту, технічні та техніко-економічні показники, що пред’являються до програмного продукту.

ЗМІСТ

ВСТУП

Невід’ємною частиною навчального процесу студентів, що навчаються за спеціальністю Інженерія програмного забезпечення, є вивчення, побудова та реалізація алгоритмів. Найбільш популярним методом зображення алгоритму, є блок-схема, тому вона найчастіше зустрічається в звітності студентів з лабораторних робіт. Однак блок-схема, що розроблена під час проектування програми, може дещо відрізнятися від кінцевої реалізації. Тому необхідно постійно контролювати відповідність між алгоритмом та його реалізацію, що займає чимало часу як в студентів так і у викладачів.

«Система визначення відповідності тексту програми графічному представленню алгоритму» дозволяє автоматизувати процес порівняння блок-схеми та її реалізації.

1 ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки є наказ № 690 ст. від 18.11.2020 ректора Дніпровського національного університету залізничного транспорту імені академіка В. Лазаряна «Про затвердження керівників та затвердження тем магістерських робіт».

Тема дипломної роботи: Розробка методу визначення відповідності тексту програми графічному представленню алгоритму.

Керівник дипломної роботи – Куроп’ятник О. С.

2 ПРИЗНАЧЕННЯ РОЗРОБКИ

2.1 Функціональне призначення

Основною метою даного програмного додатку є зіставлення графічного представлення алгоритму та його програмної реалізації з метою оцінки схожості.

2.2 Експлуатаційне призначення

Основне призначення програмного додатку полягає в автоматизації процесу перевірки відповідності блок-схеми її реалізації.

3 ВИМОГИ ДО ПРОГРАМНОГО ПРОДУКТУ

3.1 Вимоги до функціональних характеристик

Вимоги до функціональних характеристик наступні:

- введення або завантаження з файлу тексту програми;
- завантаження зображення блок-схеми з json-файлу;
- побудова графів керування за блок-схемою і програмним кодом та зіставлення їх між собою методом пошуку в ширину;
- надання висновку користувачеві програмного продукту, щодо відповідності тексту програми до зображення блок-схеми у вигляді процентного співвідношення.

Вхідні дані:

- текст програми на мові С++;
- файл .cpp з програмним кодом;
- файл .json з зображенням блок-схеми, що формується за допомогою онлайн конструктора [7].
- Вихідні дані:
- граф керування побудований за програмним кодом представлений у вигляді списку суміжності;
- граф керування побудований за блок-схемою представлений у вигляді списку суміжності;
- текстове повідомлення з висновком щодо відповідності тексту програми до зображення блок-схеми у вигляді процентного співвідношення;
- текстові повідомлення, щодо успішності або неуспішності виконаних операцій.

3.2 Вимоги до надійності

Вимоги до надійності програмного додатку наступні:

- програма повинна не допускати пошкодження даних під час своєї роботи;
- програма повинна не допускати невимушеної втрати пам‘яті;

– програма повинна стабільно працювати та кількість відмов системи не повинна перевищувати однієї відмови на 2000 запусків системи.

3.3 Умови експлуатації

Дане програмне рішення може використовуватись в умовах, відповідних до умов, які описані в документу [1].

Для забезпечення сталого функціонування програмного продукту необхідно дотримуватись таких умов:

– мінімальні навички роботи з ПК;
– програмним продуктом може користуватися людина, яка ознайомилася з керівництвом користувача;
– ЕОМ, які використовуються для роботи програмного продукту, повинні відповідати чинним вимогам та стандартам в Україні, нормативних актами з охорони праці [2];
– програмний комплекс повинен використовуватись в приміщеннях, призначених для роботи ЕОМ з наступними кліматичними умовами: температура – 21-25° С, відносна вологість повітря 40-60%.

3.4 Вимоги до складу і параметрів технічних засобів

Для роботи з додатком необхідний ПК, що має наступні мінімальні характеристики:

– процесор: двох’ядерний 32-, 64- або 86-бітний процесор з тактовою частотою 1.9 ГГц;
– оперативна пам’ять: 1 ГБ (для 32-бітних систем) або 2 ГБ (для 64-бітних систем);
– пам’ять на жорсткому диску: 500 МБ;
– периферійні пристрої: VGA-монітор з мінімальним розширенням екрану 1024x768, клавіатура, миша.

3.5 Вимоги до інформаційної і програмної сумісності

Програмний продукт має бути розроблений з використанням мови програмування C# за допомогою IDE MS Visual Studio.

Для роботи з додатком необхідно 100 Мб пам’яті для розміщення програмного додатку.

3.6 Вимоги до маркування і упаковки

У разі необхідності випуску даного програмного продукту у фізичному форматі, програмний продукт, а також електронна документація користувача повинні зберігатись на USB-носі.

Приклад маркування приведений на рис. 3.1:

Система визначення відповідності тексту програми
графічному представленню алгоритму
Яковенко Б.М.
ДНУЗТ Каф. КІТ
2021

Рисунок 3.1 – Приклад маркування

3.7 Вимоги до транспортування і зберігання

Транспортування повинно забезпечувати збереження програмного продукту, його цілісність та запобігання несанкціонованого доступу до нього. Транспортування фізичної упаковки програмного продукту повинно проводитись довіреною особою та здійснюватися комплектами в упаковці з термоплівки та піни, яка захищає носій від різного виду пошкоджень. Місце зберігання програмного продукту повинне бути сухим з відсутністю пилу при температурі 21-25°С і вологості 40-60%, з уникненням впливу вологи та шкідників.

4 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ

Програмна документація представляє собою перелік наступних документів:

- специфікація;
- текст програми;
- опис програми;
- керівництво користувача. Керівництво з визначення відповідності тексту програми графічному представленню алгоритму.

Вся документація до програми повинна задовольняти вимогам державного стандарту до оформлення програмних документів ГОСТ 19.101-77 «Єдина система програмної документації. Види програм та програмних документів» [3].

5 ТЕХНІКО-ЕКОНОМІЧНІ ПОКАЗНИКИ

Основна мета розробки техніко-економічного обґрунтування (ТЕО) – дати фінансову оцінку передбачуваних витрат та одержуваного корисного результату, а також оцінити прибутковість проекту і, в кінцевому підсумку, економічну доцільність його розробки та впровадження.

Початковим етапом розрахунку величини трудових витрат розробників є оцінка розміру програмного забезпечення. Основні відмінності методик, що застосовуються в оцінці трудовитрат, полягають у використовуваному типі критерію оцінки якості (кількісний або якісний) [4].

Згідно моделі COCOMO, розмір проекту S вимірюється в рядках коду LOC (KLOC), а трудовитрати в людино-місяцях [5, 6].

$$E = a \cdot S^b \cdot EAF \tag{5.1}$$

де E – витрати праці на проект (в людино-місяцях);

S^b – розмір коду (в KLOC);

EAF – фактор уточнення витрат (effort adjustment factor).

Для простих систем, a = 2,4; b = 1,05.

Розмір програмного коду програмного засобу – 2500 рядків:

$$E = 2{,}4 \cdot 2{,}5^{1{,}05} \cdot 1 = 6{,}3 \tag{5.2}$$

Отже, згідно моделі COCOMO, орієнтовні трудовитрати на проект складуть приблизно 6,3 людино-місяці.

Нижче наведені розрахунки вартості розробки «Система визначення відповідності тексту програми графічному представленню алгоритму». Основними статтями витрат прийняті:

- основна заробітна плата;
- відрахування на соціальні потреби;
- накладні витрати;
- витрати на персональний комп’ютер і ліцензійні базові програмні засоби.

Основна заробітна плата (ОЗП) оцінює працю інженера-програміста зі створення програмного продукту і визначається виходячи з кількості розробників,

часу виконання розробки (годин), а також заробітної плати в розрахунку на одну годину. Розрахунок заробітної платні проводиться по формі табл. 5.1. [9]

Таблиця 5.1 – Фонд місячної заробітної плати

№ п/п	Посада виконавця	Оклад, грн/міс	Кількість		Сума зарплати грн
			чол	місяців	
1	інженер-програміст	21374	1	6,3	134656

Описаний в проекті програмний продукт буде розроблений одним програмістом в період з 8.03.21 до 24.09.21, що складає 200 дні або 29 робочих тижнів. Витрати робочого часу прийняті за 40 годин у тиждень. Погодинна ставка кваліфікованого інженера–програміста складає 133,58 грн/год. Таким чином, витрачено робочого часу:

$$t_{\text{розробки}} = N_{\text{чол}} \cdot N_{\text{тиж}} \cdot N_{\text{год}}, \quad (5.3)$$

де $N_{\text{чол}}$ – кількість виконавців, *чол*;

$N_{\text{тиж}}$ – тривалість розробки;

$N_{\text{год}}$ – витрати робочого часу, год;

$$t_{\text{розробки}} = 1 \cdot 29 \cdot 40 = 1160 \text{ чол/год.} \quad (5.4)$$

ОЗП визначається за формулою:

$$\text{ОЗП} = t_{\text{розробки}} \cdot N \cdot K_{KB}, \quad (5.5)$$

Де $t_{\text{розробки}}$ – витрати праці у чол/год*;*

N – погодинна ставка;

K_{KB} – коефіцієнт кваліфікації програміста, приймається 0,75.

ОЗП складає:

$$\text{ОЗП} = 1160 \cdot 133{,}58 \cdot 0{,}75 = 116215 \text{ грн.} \quad (5.6)$$

Відрахування на соціальні потреби встановлюються у відсотках від суми заробітної плати:

$$C_{\text{соц}} = \frac{\text{ОЗП} \cdot 22\%}{100\%}$$

$$C_{\text{соц}} = \frac{116215 \cdot 22\%}{100\%} = 25567 \text{ грн.} \quad (5.7)$$

Отримані результати за (6) та (7) підсумовуються. Вони складають 141782 грн. та визначають основні прямі витрати.

Накладні витрати враховують загальногосподарчі витрати по забезпеченню проведення роботи: витрати на опалення, електроенергію, амортизація будівель, зарплату адміністративного персоналу та інше. Вони визначаються в процентах (30 – 40%) від суми прямих витрат:

$$\text{С}_{\text{накл}} = \frac{(\text{ОЗП} + \text{С}_{\text{соц}}) \cdot 35\%}{100\%}; \quad (5.8)$$

$$\text{С}_{\text{накл}} = \frac{(116215 + 25567) \cdot 35\%}{100\%} = 49624 \text{ грн.} \quad (5.9)$$

На протязі усього терміну використання нової техніки підприємство щорічно витрачає певні кошти, пов’язані з її експлуатацією.

Експлуатаційні витрати на персональний комп’ютер визначаються протягом терміну розробки програмного засобу в залежності від вартості комп’ютеру. В експлуатаційні витрати входять:

- вартість витратних матеріалів;
- витрати на ремонт;
- заробітна плата ремонтника;
- оренда приміщення;
- додаткові витрати – прибирання приміщення, охорона, оренда, комунальні послуги;
- амортизаційні витрати на персональний комп'ютер і програмне забезпечення;
- витрати на електроенергію ($\text{С}_{\text{ел}}$), які визначаються за формулою:

$$\text{С}_{\text{ел}} = P \cdot B \cdot T_{\text{розр}}, \quad (5.10)$$

де Р – потужність комп’ютера та допоміжних споживачів електричної енергії, приймається 0,45 кВт/год;

B – вартість 1 кВт/година, складає 3,61 грн; [10]

$T_{\text{розр}}$ – час роботи з ЕВМ, приймається рівним робочому часу.

Витрати на електроенергію визначаються так:

$$C_{\text{ел}} = 0{,}45 \cdot 3{,}61 \cdot 1160 = 1884 \text{ грн.} \quad (5.11)$$

Витрати на витратні матеріали ($\text{С}_{\text{вм}}$) протягом всього терміну експлуатації приблизно 10% від вартості комп'ютеру. Вартість робочої станції для комфортної роботи програміста в середньому 25000 грн. [11], термін експлуатації – 5 років. Отже, можна визначити ці витрати за період створення програмного засобу:

$$\text{С}_{\text{вм}} = \text{В}_{\text{ком}} \cdot \frac{N_{\text{Д}}}{N_{\text{експ}} \cdot 365} \cdot \frac{10\%}{100\%}, \quad (5.12)$$

де B_{KOM} – вартість персонального комп'ютеру;

$N_{\text{Д}}$ – кількість днів розробки програмного продукту;

$N_{\text{експ}}$ – термін експлуатації персонального комп'ютеру.

Витрати на витратні матеріали визначаються так:

$$C_{\text{вм}} = 25000 \cdot \frac{200}{5 \cdot 365} \cdot \frac{10}{100} = 274 \text{ грн.} \quad (5.13)$$

Заробітна плата ремонтника ($C_{\text{рем}}$) визначена наступним чином: на ремонт 50 комп'ютерів потрібен один інженер-системотехнік. Його середньомісячна заробітна плата приймається 12000 грн [9]. Тоді в перерахунку на один комп'ютер його заробітна плата за період розробки програмного продукту складає:

$$C_{\text{рем}} = \frac{C_{\text{рем}}'}{N_{KOM}} \cdot T_{\text{міс}}, \quad (5.14)$$

де $C_{\text{рем}}'$ – середньомісячна заробітна плата;

N_{KOM} – кількість комп'ютерів на одного ремонтника.

$T_{мес}$ – час розробки програмного продукта, міс.

Заробітна плата ремонтника ($C_{\text{рем}}$) буде складати:

$$C_{\text{рем}} = \frac{12000}{50} \cdot 6 = 1440 \text{ грн.}$$

За статистикою витрати на комплектуючи вироби (С_{KOM}) для ремонту персонального комп'ютера складає 10% від його вартості за термін його експлуатації, тобто рівні витратам на витратні матеріали:

$$C_{KOM} = C_{BM} = 274 \text{ грн.} \quad (5.16)$$

Амортизаційні відрахування на персональний комп'ютер (АПК) визначені з положення, що амортизаційний період в даний час дорівнює терміну морального старіння обчислювальної техніки і складає 3 роки. Отже, за 3 роки амортизаційні

відрахування на персональний комп'ютер дорівнюють вартості комп'ютера. За період проектування амортизаційні відрахування складуть:

$$\text{АКП} = \text{В}_{КОМ} \cdot \frac{N_{\text{Д}}}{N_{\text{експ}} \cdot 365}; \quad (5.17)$$

$$\text{АКП} = 25000 \cdot \frac{200}{3 \cdot 365} = 4566$$

Амортизаційні відрахування на програмне забезпечення (АПЗ) залежать від його циклу заміни. Якщо прийняти термін морального старіння для Windows 5 років та MS Visual Studio 2019 за 3 роки то амортизаційні відрахування на програмне забезпечення дорівнюють його вартості.

Для функціонування персонального комп'ютера використовувалася операційна система Windows 10, для написання програмного забезпечення - програмне середовище MS Visual Studio 2019.

$$\text{АКП}_w = 6582 \cdot \frac{200}{5 \cdot 365} = 715,$$

$$\text{АКП}_w = 14000 \cdot \frac{200}{3 \cdot 365} = 2557.$$

Розрахунок амортизаційних відрахувань на програмне забезпечення зведений в табл. 5.2.

Додаткові витрати ($C_{\text{дод}}$): прибирання приміщень, охорона, комунальні послуги важко оцінити точно і прийняти рівними 50% заробітної плати інженера-програміст, тобто 10687 гривень на місяць.

Оренду приміщень приймемо рівною 3500 гривень на місяць відповідно до реальної пропозиції [12].

Сумарні експлуатаційні витрати на один персональний комп'ютер складають:

$$C_{\text{експ}} = C_{\text{ел}} + C_{\text{ВМ}} + C_{\text{рем}} + C_{КОМ} + \text{АПК} + \text{АПО} + C_{\text{ор}} + C_{\text{дод}}; \quad (5.18)$$

$$C_{\text{експ}} = 1884 + 274 + 1440 + 274 + 4566 + 3272{,}7 + 3500 + 10687 = \\ = 25897.7 \text{ грн.} \quad (5.19)$$

Таблиця 5.2 – Використовуване програмне забезпечення

Найменування програмного забезпечення	Вартість програмного забезпечення, *грн*	Джерело придбання	Амортизаційні відрахування, *грн*
WINDOWS 10 PROFESSIONAL	6582	http://mtsoft.kiev.ua/product/windows-10-professional	715
MS VISUAL STUDIO 2019	14000	https://visualstudio.microsoft.com/ru/vs/pricing/	2557
Всього:	20582	-	3272

Результати розрахунків зведено у табл. 5.3.

Таблиця 5.3 – Експлуатаційні витрати на ПК і ПЗ.

Найменування витрат	Витрати, грн
Витрати на електроенергію	1884
Вартість витратних матеріалів	274
Витрати на ремонт	1440
Витрати на комплектуючи вироби	274
Амортизація персонального комп’ютера	4566
Амортизація програмного забезпечення	3272,7
Оренда приміщення	3500
Додаткові витрати	10687
Всього	25897.7

Таким чином, витрати на створення програмного продукту складають:

$$C_{\text{розробки}} = \text{ОЗП} + C_{\text{соц}} + C_{\text{накл}} + C_{\text{експ}}; \quad (5.20)$$

$$C_{\text{розробки}} = 116215 + 25567 + 49624 + 25897.7 = \\ = 217303.7 \text{ грн} \quad (5.21)$$

Розрахунок витрат зведено у табл. 5.4.

Таблиця 5.4 – Кошторис витрат на розробку програмного засобу

Найменування витрат	Витрати, грн
Основна заробітна плата	116215
Відрахування на соціальні потреби	25567
Накладні витрати	49624
Експлуатаційні витрати	25897.7
Всього	217303.7

За отриманими значеннями техніко-економічних показників проекту складено кошторис витрат на розробку «Система визначення відповідності тексту програми графічному представленню алгоритму». За результатами розрахунків, приблизна вартість розробки складає 217303.7 грн.

6 СТАДІЇ ТА ЕТАПИ РОЗРОБКИ

Стадії розробки наведені в табл. 6.1.

Таблиця 6.1 – Стадії та етапи розробки

Стадії	Зміст робіт	Терміни виконання
1	2	3
1. Технічне завдання	Постановка завдання Збір початкових матеріалів Вибір і обґрунтування критеріїв ефективності і якості програми, що розробляється Обґрунтування необхідності проведення науководослідних робіт Визначення структури вхідних і вихідних даних Попередній вибір методів рішення задач Обґрунтування доцільності застосування раніше розроблених програм Визначення вимог до технічних засобів Обґрунтування принципової можливості рішення поставленої задачі Визначення вимог до програми Техніко-економічне обґрунтування розробки програми Визначення стадій, етапів і термінів розробки програми і документації на неї Вибір мов програмування Визначення необхідності проведення науководослідних робіт на подальших стадіях Узгодження і затвердження технічного завдання	25.01.2021 – 10.03.2021

Продовження таблиці 6.1

1	2	3
2. Робочий проект	Програмування і налагодження програми Розробка програмних документів відповідно до вимог ГОСТ 19.101-77 Розробка, узгодження і затвердження програми і методики випробувань Проведення попередніх і інших видів випробувань Коригування програми і програмної документації за наслідками випробувань	11.03.2021 - 24.05.2021
3. Впровадження	Підготовка і передача програми і програмної документації для супроводу і (або) виготовлення	25.05.2021 - 11.06.2021

7 ПОРЯДОК КОНТРОЛЮ І ПРИЙМАННЯ

Контроль за виконанням роботи виконує керівник розробки.

Прийом програмного продукту здійснюється прийомною комісією і керівником розробки.

БІБЛІОГРАФІЧНИЙ СПИСОК

1. ДСанПіН 3.3.2-007-98. Державні санітарні правила і норми роботи з візуальними дисплейними терміналами електронно-обчислювальних машин [Текст] / Постанова Головного державного санітарного лікаря України від 10 грудня 1998 р. № 7 – К.,1998.
2. ДСН 3.3.6.042-99. Санітарні норми мікроклімату виробничих приміщень [Текст]/ Постанова Головного Державного санітарного лікаря України від 01.12.1999 № 42 – К., 1999.
3. ГОСТ 19.101-77. Виды програм и программных документов [Текст]/ Постановление Государственного комитета стандартов Совета Министров СССР от 20 мая 1977 г. – М., 1977.
4. Борисенков, Евгений. «Методики оценки трудозатрат по разработке программного обеспечения информационных систем.»
5. «Методики оценки трудозатрат по разработке программного обеспечения информационных систем» [Ел. ресурс]. Available: http://repository.enu.kz/
6. bitstream//12881/metodika-trudozatrat.pdf. [Дата звернення: 05.07.2021].
7. «Методики оценки трудозатрат» [Ел. ресурс]. Available: http://www.hups.mil.gov.ua/periodic-app/article/11953/soi_2014_8_33.pdf. [Дата звернення: 05.07.2021].
8. Programforyou: редактор блок-схем. URL: https://programforyou.ru/block-diagram-redactor (дата останнього звернення: 05.07.2021).
9. «Dou.ua Статистика зарплат програміств в Україні» [Ел. ресурс]. Available: https://jobs.dou.ua/salaries/#period=jun2021&city=Dnipro&title=Junior%20Software%20Engineer&language=C%23%2F.NET&spec=&exp1=0&exp2=10 [Дата звернення: 05.07.2021].
10. «Тарифы на распределение электроэнергии для предприятий» [Ел. ресурс]. Available: https://index.minfin.com.ua/tariff/electric/prom/2021-01-01/ [Дата звернення: 05.07.2021].

11. «Ноутбук Lenovo ThinkPad E14 Gen 2 (20TA0027RT) Black» [Ел. ресурс]. Available: https://rozetka.com.ua/lenovo-20ta0027rt/p308766743 [Дата звернення: 05.07.2021].
12. «Сайт оголошень OLX» [Ел. ресурс]. Available: https://www.olx.ua/d/obyavlenie/sdam-3-komn-kvartiru-na-pr-polyakirova-pod-tihiy-ofis-IDM9OXX.html#d542b01665;promoted [Дата звернення: 05.07.2021].

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖУЮ

Перший проректор Дніпровського

національного університету

залізничного транспорту

імені академіка В. Лазаряна

Борис БОДНАР

СИСТЕМА ВИЗНАЧЕННЯ ВІДПОВІДНОСТІ ТЕКСТУ ПРОГРАМИ ГРАФІЧНОМУ ПРЕДСТАВЛЕННЮ АЛГОРИТМУ

Робочий проект

1116130.01222-01-ЛЗ

ЛИСТ ЗАТВЕРДЖЕННЯ

Завідувач кафедри КІТ

Вадим ГОРЯЧКІН

Керівник розробки

Олена КУРОП`ЯТНИК

Виконавець

Богдан ЯКОВЕНКО

Нормоконтролер

Олена КУРОП`ЯТНИК

2021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖЕНО

1116130.01222-01

СИСТЕМА ВИЗНАЧЕННЯ ВІДПОВІДНОСТІ ТЕКСТУ ПРОГРАМИ ГРАФІЧНОМУ ПРЕДСТАВЛЕННЮ АЛГОРИТМУ

Специфікація

Аркушів 2

2021

Позначення	Найменування	Примітка
1116130.01222-01-ЛЗ	Лист затвердження	
1116130.01222-01 12 01-ЛЗ	Лист затвердження	
1116130.01222-01 12 01	Текст програми	
1116130.01222-01 13 01-ЛЗ	Лист затвердження	
1116130.01222-01 13 01	Опис програми	
1116130.01222-01 І3 01-ЛЗ	Лист затвердження	
1116130.01185-01 І3 01	Керівництво користувача. Керівництво з використання системи визначення відповідності тексту програми графічному представленню алгоритму	

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖЕНО

1116130.01222-01 13 01

СИСТЕМА ВИЗНАЧЕННЯ ВІДПОВІДНОСТІ ТЕКСТУ ПРОГРАМИ ГРАФІЧНОМУ ПРЕДСТАВЛЕННЮ АЛГОРИТМУ

Опис програми

Аркушів 15

2021

АНОТАЦІЯ

Документ 1116130.01222-01 13 01 ««Система визначення відповідності тексту програми графічному представленню алгоритму». Опис програми» входить до складу програмної документації додатку, який надає можливість порівнювати між собою алгоритм (програмна реалізація) та його графічне представлення (блок-схема).

У даному документі представлений опис програми клієнтської частини системи: функціональне призначення, опис логічної структури, використані технічні засоби, виклик і завантаження, вхідні і вихідні дані, опис інтерфейсу користувача, порядок роботи з програмою.

ЗМІСТ

1 ЗАГАЛЬНІ ВІДОМОСТІ

Програмний додаток «Система визначення відповідності тексту програми графічному представленню алгоритму» являє собою інструментарій для зіставлення між собою блок-схеми та її програмної реалізації для визначення їх відповідності.

Для коректного функціонування програми необхідно щоб на персональному комп’ютері було встановлено операційну систему Windows 7 або вище.

Програма реалізована на мові програмування C# у IDE MS Visual Studio 2019, а також мови опису даних JSON для опису блок-схеми.

2 ФУНКЦІОНАЛЬНЕ ПРИЗНАЧЕННЯ

Метою програмного додатку є визначення відповідності між блок-схемою та її програмною реалізацією на мові програмування C++.

Функціональні вимоги до додатку:

- розпізнавання програмного коду на мові C++ та побудова графу керування у вигляді списку суміжності;
- розпізнавання блок-схеми у форматі JSON-файлу та побудова графу керування у вигляду списку суміжності.
- зіставлення між собою графів керування побудованих за програмним кодом та блок-схемою методом обходу в ширину та надання висновку щодо їх схожості у вигляді процентного співвідношення.

Відповідно до функціональних вимог додатку було визначено наступні вимоги до даних програми:

- програмний код, який розпізнається має бути на мові програмування C++;
- блок-схема для розпізнавання має бути у вигляді JSON-файлу та мати наступний формат: blocks[text, type], arrows[startIndex, endIndex], де blocks і arrows – множина блоків та ліній з стрілкою блок-схеми відповідно, text – текст блоку, type – тип блоку, startIndex – індекс блоку який є початком лінії з стрілкою та endIndex – індекс блоку який є кінцем лінії з стрілкою.

3 ОПИС ЛОГІЧНОЇ СТРУКТУРИ

3.1 Алгоритм та методи програми

Програмний додаток складається з трьох логічних блоків:

- розпізнавання програмного коду та побудова проміжного представлення у вигляді списку керуючих операторів;
- зчитування блок-схеми з JSON-файлу та побудова її моделі;
- побудова графу керування за проміжним представлення у вигляді списку керуючих операторів і моделі блок-схеми та зіставлення їх між собою методом обходу в ширину на надання висновку щодо їх схожості у процентному співвідношенні.

Розпізнавання програмного коду складається з наступних етапів:

- попередня обробка програмного коду: видалення одно строкових та багато строкових коментарів, строкових літералів, конструкцій в круглих дужках та не значимих лексем, таких як пробіл, знаки табуляції та ін.;
- лексичний аналіз програмного коду, метою якого є виділення лексем та переведення їх в внутрішнє представлення;
- побудова списку керуючих операторів шляхом перебору виділених лексем.

Для зчитування блок-схеми з JSON-файлу в проміжну модель використовується бібліотека Newtonsoft.Json, яка дозволяє серіалізувати формат JSON.

Графи керування будуються за проміжними моделями програмного коду та блок-схеми у вигляді списку суміжності. Зіставлення графів між собою полягає у паралельному обході в ширину обох графів та відміченні однакових вершин. Визначення відповідності графів базується на процентному співвідношенні однакових вершин графів.

3.2 Структура програми з описом функцій складових частин і зв’язки між ними

На рис. 3.1 відображено основні класи рівня логіки та взаємозв‘язок між ними.

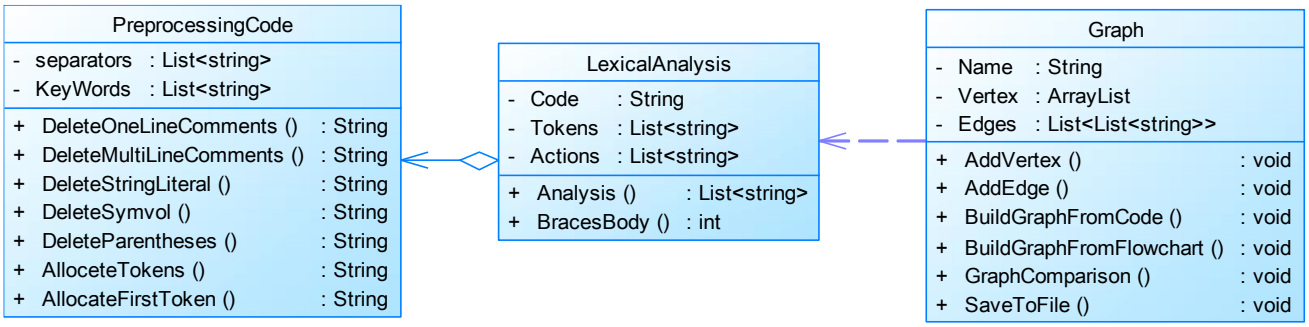


Рисунок 3.1 – Діаграма класів рівня логіки

Нижче представлений опис представлених програмних модулів.

PrepocessingCode.cs – відповідає за попередню обробку програмного коду.

LexocalAnalysis.cs – відповідає за лексичний аналіз програного коду та побудову проміжного представлення у вигляді списку керуючих операторів.

Graph.cs – відповідає за побудову графів керування за проміжним представленням програмного коду та блок-схеми та зіставлення їх для визначення їх відповіності.

3.3 Зв’язки програми з іншими програмами

Для коректної роботи програмного додатку необхідні такі програми:

– операційна система Windows 7 або більше;
– framework .NET 5.0;
– текстовий редактор для файлів txt-формату.

4 ВИКОРИСТАНІ ТЕХНІЧНІ ЗАСОБИ

Для коректного функціонування програмного забезпечення достатньо мати робочий персональний комп’ютер, що має наступні технічні характеристики:

- процесор з тактовою частотою 800 Мгц;
- не менше 256 Мб ОЗУ;
- 100 Мб вільного місця на жорсткому диску;
- архітектура x86 або x64;
- периферійні пристрої – VGA-монітор з мінімальним розширенням екрану 1024x768, клавіатура, миша.

5 ВИКЛИК ТА ЗАВАНТАЖЕННЯ

Програма викликається шляхом подвійного натискання лівою кнопкою миші значка виконавчого файлу програми «ConformityAssessment.exe», що знаходиться у папці де було встановлено програмний додаток.

Файли для завантаження у програмний додаток, а саме файли з розширенням .cpp та .json, що містять програмний код та блок-схему відповідно, можуть зберігатися у будь-якому місці зручному користувачеві.

Файли, що завантажує програмний додаток, зберігаються у папці «Graph», що знаходиться за шляхом, де було встановлено програмний додаток.

6 ВХІДНІ ДАНІ

Вхідні дані програмного продукту:

- програмний код на мові програмування C++;
- файл .cpp з програмним кодом на мові програмування C++;
- файл .json з зображенням блок-схеми, що має наступний формат: blocks[text, type], arrows[startIndex, endIndex], де blocks і arrows – множина блоків та ліній з стрілкою блок-схеми відповідно, text – текст блоку, type – тип блоку, startIndex – індекс блоку який є початком лінії з стрілкою та endIndex – індекс блоку який є кінцем лінії з стрілкою.

7 ВИХІДНІ ДАНІ

Вихідні дані програмного продукту:

- граф керування побудований за програмним кодом представлений у вигляді списку суміжності;
- граф керування побудований за блок-схемою представлений у вигляді списку суміжності;
- текстове повідомлення з висновком, щодо відповідності тексту програми до зображення блок-схеми у вигляді процентного співвідношення;
- текстові повідомлення, щодо успішності або неуспішності виконаних операцій.

8 ОПИС ІНТЕРФЕЙСУ КОРИСТУВАЧА

8.1 Опис станів програми

Стани програми наведено у табл. 8.1.

Таблиця 8.1 – Стани програми

№ стану	Назва стану	Опис стану	Рекомендовані дії
1	Запуск додатку	Додаток запускається.	Очікування клієнтського вікна додатку
2	Завантаження файлу з програмним кодом на мові C++	Вибір та завантаження файлу з формату .cpp	Вибрати необхідний файл з розширення .cpp
3	Введення програмного коду на мові C++	Введення програмного коду на мові C++ за допомогою вбудованого текстового редактору	Ввести програмний код на мові C++
4	Завантаження файлу з блок-схемою	Вибір та завантаження файлу з формату .json	Вибрати необхідний файл з розширення .json
5	Розпізнавання програмного коду	Розпізнавання програмного коду та побудова проміжного представлення у вигляд списку керуючих операторів	Обрати пункт головного меню Інструменти –> Розпізнати програмний код
6	Розпізнавання блок-схеми	Розпізнавання блок-схеми та побудова проміжної моделі	Обрати пункт головного меню Інструменти –> Розпізнати блок-схему
7	Визначення відповідності блок-схеми та програмного коду	Визначення відповідності блок-схеми та програмного коду шляхом їх зіставлення між собою	Обрати пункт головного меню Інструменти –> Визначення відповідності програмного коду та блок-схеми

8.2 Опис переходів між станами програми

Схема переходів програми наведена на рис. 8.1, де цифри є номером стану програми (табл. 8.1). Вийти з програми можливо під час усіх станів крім першого стану.

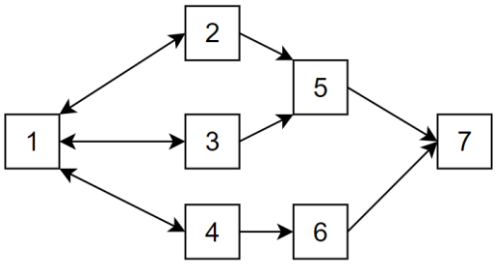


Рисунок 8.1 – Схема переходів між станами програми

8.3 Опис керування діалогом

Керування діалогом відбувається за допомогою екранних форм та головного меню програми.

8.4 Форматування екрана

Головне вікно програми можна форматувати, а саме змінювати його розмір. Інші засоби форматування не передбачені.

9 ПОРЯДОК РОБОТИ З ПРОГРАМОЮ

Порядок роботи з програмою передбачає наступну послідовність операцій:

- введення або завантаження програмного коду програми на мові C++;
- завантаження блок-схеми у форматі JSON;
- розпізнавання програмного коду;
- розпізнавання блок-схеми;
- визначення відповідності між блок-семою та програмним кодом.

10 ПОВІДОМЛЕННЯ

У табл. 10.1 наведені повідомлення користувачу, що можуть з’явитися під час роботи з програмою.

Таблиця 10.1 – Повідомлення користувачу

Текст повідомлення	Опис ситуації	Рекомендовані дії
Помилка відкриття файлу з програмним кодом	Файл з програмним кодом не було відкрито	Обрати коректний файл з програмним кодом у форматі .cpp
Помилка відкриття файлу з блок-схемою	Файл з блок-схемою не було відкрито	Обрати коректний файл з блок-схемою у форматі .json
Програмний код не введено!	Розпізнавання програмного коду не виконано оскільки вхідні дані не коректні	Ввести або відрити файл з програмним кодом на мові C++
Розпізнавання програмного коду виконано!	Розпізнавання програмного коду виконано успішно	Натиснути кнопку «ОК»
Невірний формат вхідних даних!	Розпізнавання блок-схеми не виконано оскільки вхідні дані не коректні	Відрити файл з блок-схемою у форматі JSON
Розпізнавання блок-схеми виконано!	Розпізнавання блок-схеми виконано успішно	Натиснути кнопку «ОК»
Невірний формат вхідних даних!	Зіставлення графів не виконано оскільки блок-схема або програмний код не було успішно розпізнано	Перевірити коректність вхідних даних та провести розпізнавання програмного коду та блок-схеми
Відповідність алгоритму до його реалізації складає:	Визначено відповідність блок-схеми та програмного коду	Натиснути кнопку «ОК»

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖЕНО

1116130.01222-01 І3 01

СИСТЕМА ВИЗНАЧЕННЯ ВІДПОВІДНОСТІ ТЕКСТУ ПРОГРАМИ ГРАФІЧНОМУ ПРЕДСТАВЛЕННЮ АЛГОРИТМУ

Керівництво користувача. Керівництво з використання системи визначення відповідності тексту програми графічному представленню алгоритму

Аркушів 12

2021

АНОТАЦІЯ

Документ 1116130.01222-01 І3 01 ««Система визначення відповідності тексту програми графічному представленню алгоритму». Керівництво користувача. Керівництво з використання системи визначення відповідності тексту програми графічному представленню алгоритму» входить до складу програмної документації додатку, який надає можливість порівнювати між собою алгоритм (програмна реалізація) та його графічне представлення (блок-схема).

У даному документі представлено призначення та умови застосування програмного продукту, інструкції з підготовки до роботи, а також опис основних операцій та аварійних ситуацій програмного додатку.

ЗМІСТ

ВСТУП

Програмний додаток «Система визначення відповідності тексту програми графічному представленню алгоритму» використовується студентами, що навчаються за спеціальності Інженерія програмного забезпечення для самоконтролю розроблених алгоритмів та їх реалізацій, а також викладачами для автоматизації процесу перевірки академічних робіт студентів. Ще однією сферою використання програмного додатку є перевірка академічних робіт на плагіат.

Програмний додаток надає можливість встановлення відповідності між блок-схемою та її програмною реалізацією на мові C++.

Для використання програми необхідні мінімальні навички роботи з ПК та операційною системою Windows 7 або вище.

Для коректної роботи з програмним додатком необхідно ознайомитися з описом програми та керівництвом викладача.

1 ПРИЗНАЧЕННЯ ТА УМОВИ ЗАСТОСУВАННЯ

1.1 Функціональне призначення програми

Основною метою даного програмного додатку є зіставлення графічного представлення алгоритму та його програмної реалізації з метою оцінки схожості.

До функціональних можливостей програмного додатку входить:

- розпізнавання програмного коду на мові C++ та побудова графу керування у вигляді списку суміжності;
- розпізнавання блок-схеми у форматі JSON та побудова графу керування у вигляді списку суміжності;
- надання висновку у вигляді текстового повідомлення, щодо відповідності блок-схеми та її програмної відповідності у процентному співвідношенні.

Основне призначення програмного додатку полягає в автоматизації процесу перевірки відповідності блок-схеми її реалізації.

1.2 Вимоги до складу і параметрів технічних засобів

Для коректного функціонування програмного забезпечення достатньо мати робочий персональний комп’ютер, що має наступні технічні характеристики:

- процесор з тактовою частотою 800 Мгц;
- не менше 256 Мб ОЗУ;
- 100 Мб вільного місця на жорсткому диску;
- архітектура x86 або x64;
- периферійні пристрої – VGA-монітор з мінімальним розширенням екрану 1024x768, клавіатура, миша.

1.3 Вимоги до інформаційної і програмної сумісності

Для коректного функціонування програми необхідно щоб на персональному комп’ютері було встановлено операційну систему Windows 7 або вище, Framework .NET 5 та текстовий редактор для файлів txt-формату.

2 ПІДГОТОВКА ДО РОБОТИ

Для виклику програми двічі натисніть лівою кнопкою миші по іконці виконавчого файлу програми «ConformityAssessment.exe», що знаходиться у папці де було розміщено програмний додаток.

Для роботи з програмним додатком завантажте файл с програмним кодом на мові С++ з розширенням .cpp та файл з блок-схемою з розширенням .json у будь-якому порядку.

3 ОПИС ОПЕРАЦІЙ

Після старту роботи з програмним додатком можливі наступні операції:

1. Завантаження програмного коду на мові програмування C++: завантажте файл з розширенням .cpp за допомогою пункту головного меню «Файл–>Відкрити–>Файл» з програмним кодом або за допомогою комбінації клавіш Ctrl+Shift+O;

2. Введення програмного коду на мові C++: введіть програмний код за допомогою вбудованого текстового редактору;

3. Завантаження блок-схеми: завантажте файл з розширенням .json за допомогою пункту головного меню «Файл–>Відкрити–>Файл з блок-схемою» або за допомогою комбінації клавіш Ctrl+Alt+Shift+O;

4. Розпізнавання програмного коду: оберіть пункт головного меню «Інструменти–>Розпізнати програмний код» або натисніть комбінацію клавіш Alt+F1. Виконати після 1 або 2 операції;

5. Розпізнавання блок-схеми: оберіть пункт головного меню «Інструменти–>Розпізнати блок-схему» або натисніть комбінацію клавіш Alt+F2. Виконати після 3 операції;

6. Визначення відповідності блок-схеми та її програмної реалізації: оберіть пункт головного меню «Інструменти–>Визначити відповідність програмного коду та блок-схеми» або натисніть комбінацію клавіш Alt+F3. Виконати після 4 та 5 операцій;

7. Перегляд довідки: оберіть пункт головного меню «Довідка–>Перегляд довідки» або натисніть клавішу F1.

4 АВАРІЙНІ СИТУАЦІЇ

У разі виявлення помилок у даних програми, необхідно закрити програмний додаток та перевірити коректність вхідних даних.

У разі виявлення інших аварійних ситуацій, необхідно закрити програмний додаток та зв’язатися з персоналом, що супроводжує даний додаток.

5 РЕКОМЕНДАЦІЇ ЩОДО ЗАСВОЄННЯ

Інтерфейс програмного додатку зображено на рис. 5.1.

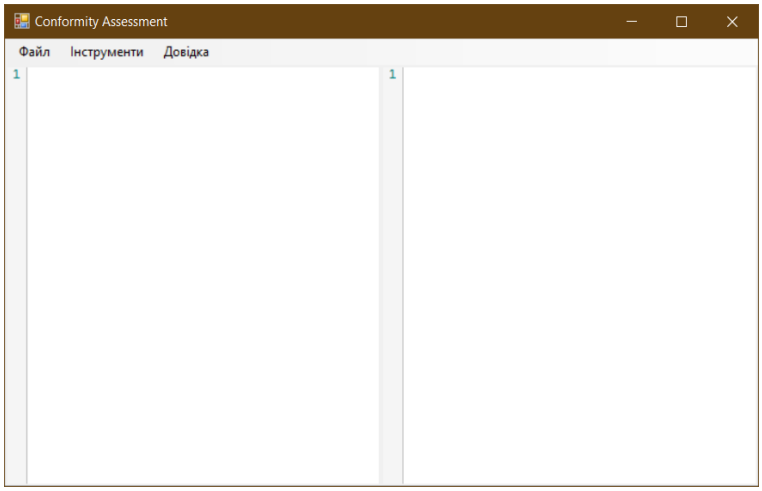


Рисунок 5.1 – Інтерфейс програмного додатку

Для завантаження файлів з програмним кодом на мові С++ та блок-схемою оберіть пункт головного меню Файл–>Відкрити (рис. 5.2).

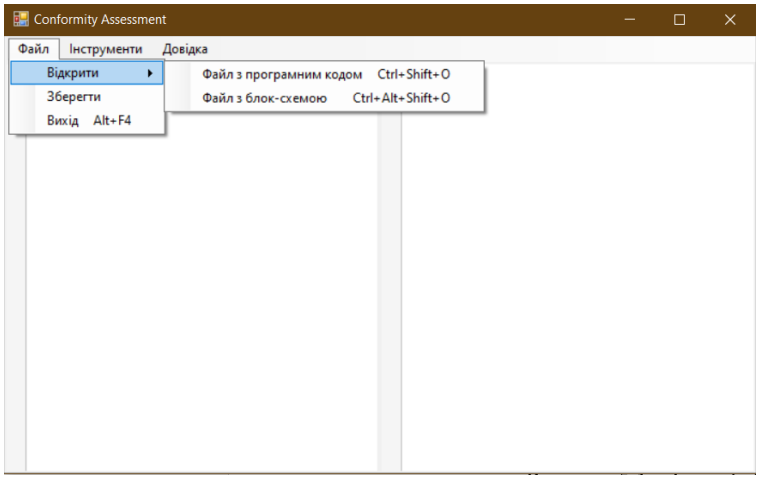


Рисунок 5.2 – Пункт головного меню Файл–>Відкрити

Щоб відредагувати вхідні дані скористайтеся вбудованим текстовим редактором додатку (рис. 5.3).

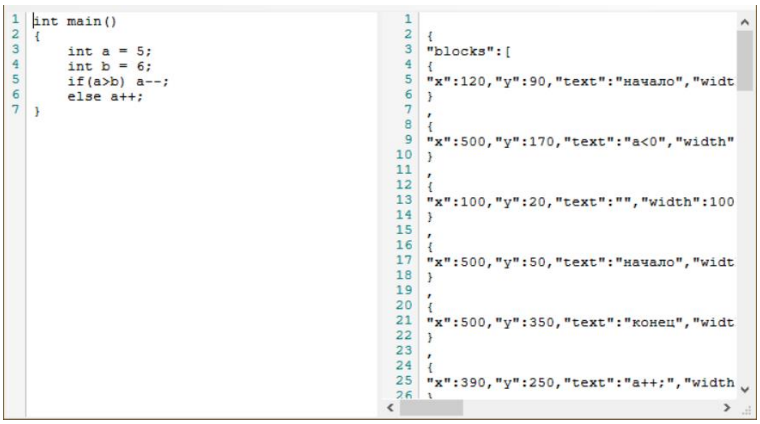


Рисунок 5.3 – Вбудований текстовий редактор додатку з вхідними даними

Для розпізнавання вхідних даних та порівняння їх між собою скористайтеся пунктом головного меню Інструменти (рис. 5.4).

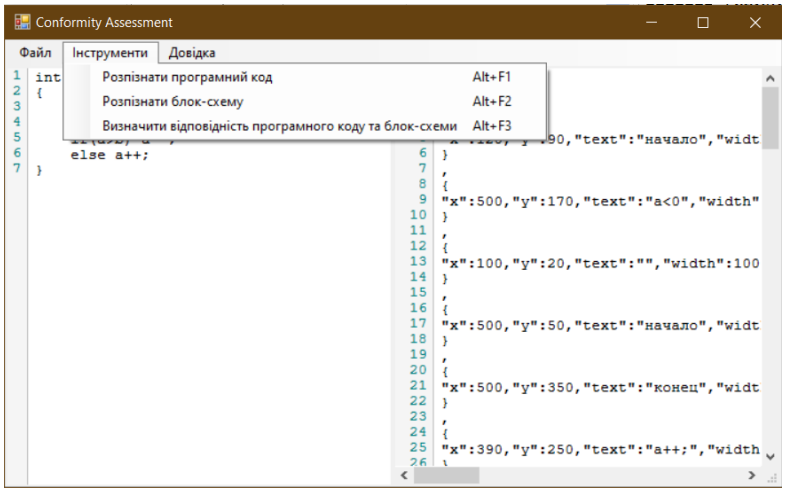


Рисунок 5.4 – Пункт головного меню Інструменти

Після виконання операцій з розпізнавання вхідних даних з’явиться інформаційне вікно про успішне або помилкове завершення операції (рис. 5.5-5.6).

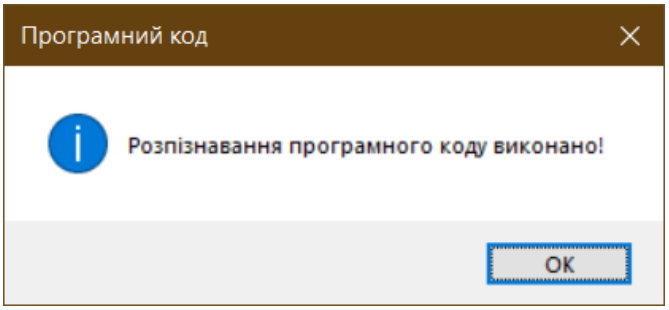


Рисунок 5.5 – Інформаційне вікно про успішне виконання операції

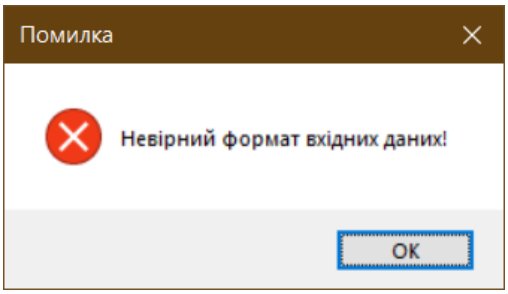


Рисунок 5.6 – Інформаційне вікно про неуспішне виконання операції

Після порівняння вхідних даних між собою з’явиться інформаційне з висновком щодо їх відповідності(рис. 5.7).

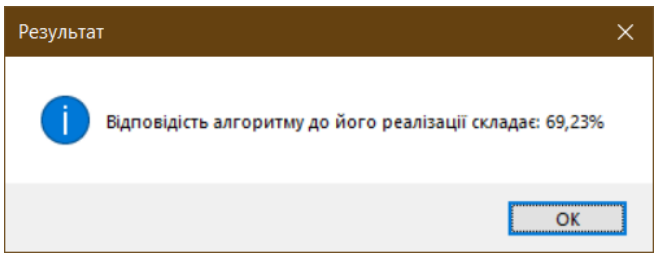


Рисунок 5.7 – Інформаційне вікно з висновком щодо відповідності вхідних даних

Для перегляду довідки скористайтесь пунктом головного меню Довідка (рис 5.8).

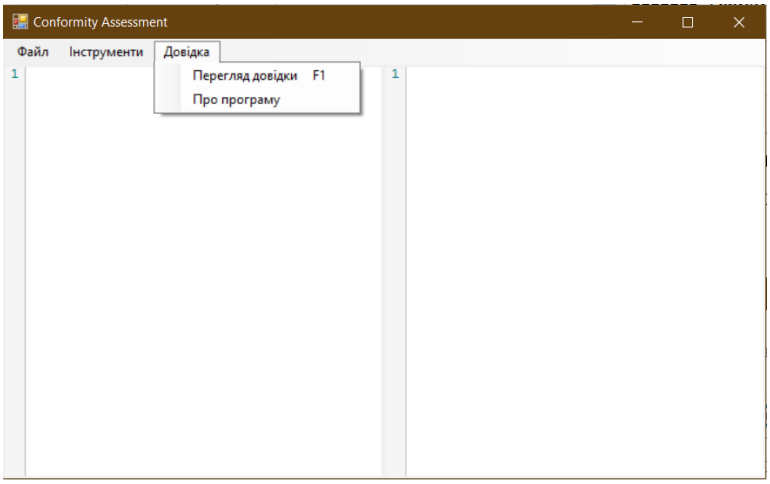


Рисунок 5.8 – Пункт головного меню Довідка

Інформаційні вікна програмного додатку зображено на рис. 5.9-5.10.

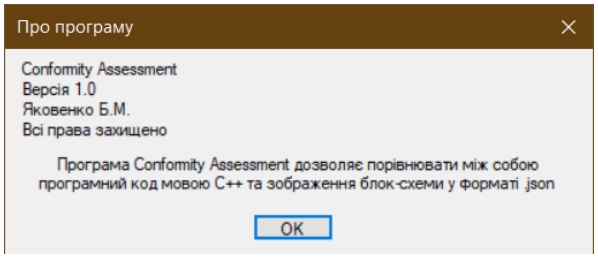


Рисунок 5.9 – Вікно «Про програму»

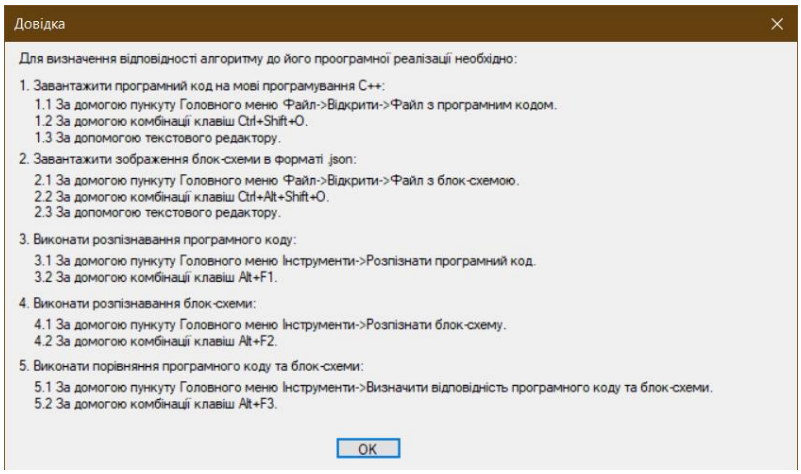


Рисунок 5.1 – Вікно «Довідка»

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖЕНО

1116130.01222-01 12 01

СИСТЕМА ВИЗНАЧЕННЯ ВІДПОВІДНОСТІ ТЕКСТУ ПРОГРАМИ ГРАФІЧНОМУ ПРЕДСТАВЛЕННЮ АЛГОРИТМУ

Текст програми

Аркушів 16

2021

АНОТАЦІЯ

Документ 1116130.01222-01 12 01 ««Система визначення відповідності тексту програми графічному представленню алгоритму». Текст програми» входить до складу програмної документації додатку, який надає можливість порівнювати між собою алгоритм (програмна реалізація) та його графічне представлення (блок-схема).

Документ містить текст програми. Програма реалізована на мові C# у програмному середовищі MS Visual Studio.

ЗМІСТ

1 СТРУКТУРА ПРОГРАМИ

Проект програмного додатку складається з двох рівнів: логіки та представлення. Рівень логіки (рис. 1.1) відповідає за розпізнавання програмного коду та блок-схеми, побудову графу керування за програмним кодом та блок-схемою і зіставлення графів керування між собою для надання висновку щодо їх відповідності. Рівень представлення (рис 1.2) відповідає за інтерфейс користувача.

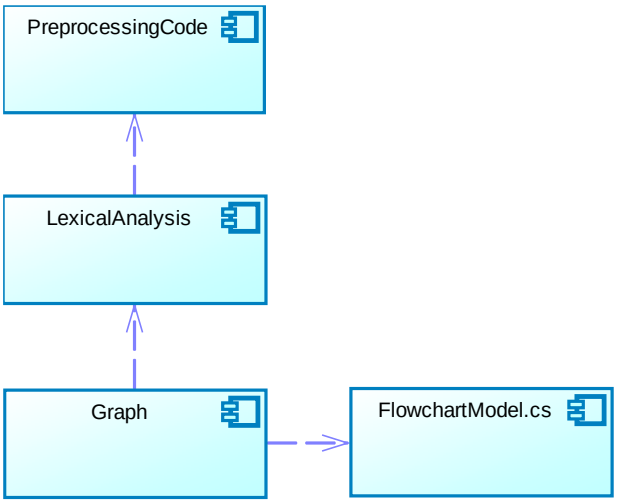


Рисунок 1.1 – Діаграма компонентів для рівня логіки

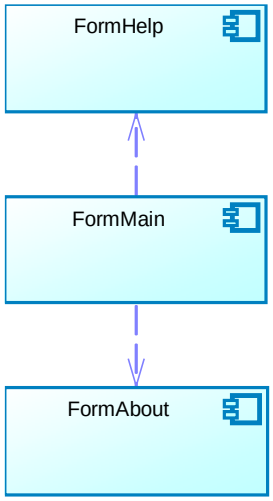


Рисунок 1.2 – Діаграма компонентів для рівня представлення

Файлова структура проекту відображена на рис. 1.3.

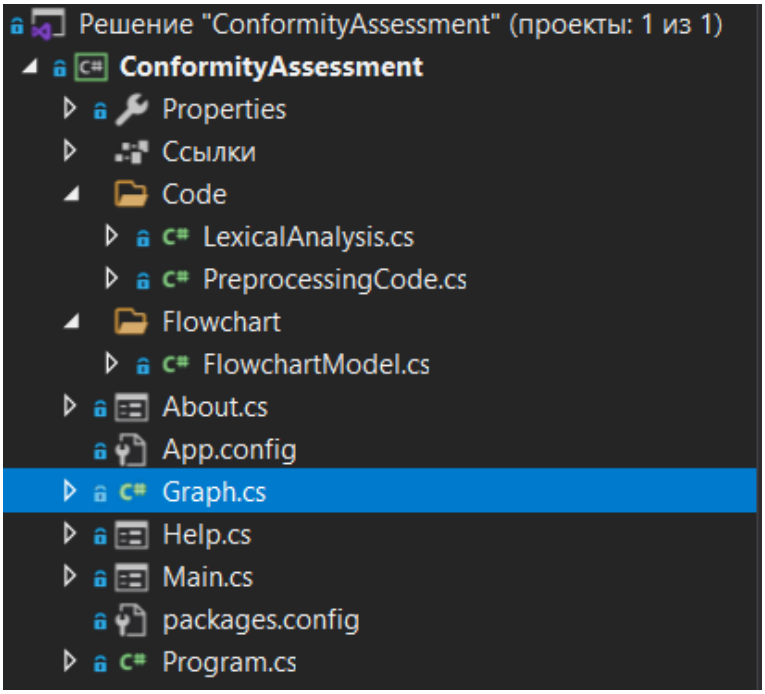


Рисунок 1.3 – Файлова структура проекту програмного продукту

Наведемо опис кожного файлу з структури:

Main.cs – файл рівня представлення, описує форму інтерфейсу користувача.

Help.cs – файл рівня представлення, описує форму, що містить керівництво користувача.

About.cs – файл рівня представлення, описує форму, що містить інформацію щодо програмного продукту.

PreprocessingCode.cs – файл рівня логіки, відповідає за попередню обробку тексту програми.

LexicalAnalysis.cs – файл рівня логіки, відповідає за лексичний аналіз програми.

FlowchartModel.cs – файл рівня логіки, зберігає модель, для десеріалізації даних блок-схеми.

Graph.cs – файл рівня логіки, відповідає за побудову та зіставлення графів керування.

Program.cs – є вхідною точкою програми.

2 ТЕКСТ ПРОГРАМИ

PreprocessingCode.cs

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace ConformityAssessment.Code
{
    class PreprocessingCode
    {
        private static readonly List<string>
separators = new List<string>()//колекция
разделителей
        { " ", "\n", "\r", ",", ".", "(",
")", "{", "}", "<", ">", "*",
              "/", "+", "-", "=", "&", "%",
"?", "!", ";", ":", "^" };

        private static readonly List<string>
KeyWorlds  = new List<string>()//колекция
ключевых строк
        {"if", "for", "switch", "do",
"break", "continue", "return", "while",
            "else", "case", "default"};

        public static string Delete-
OneLineComments(string code)//функция
удаление однострочных комметариев
        {
            try
            {
                int count;
                int index;
                for (int i = 0; i <
code.Length - 1; i++)
                {
                    if (code[i] == '/' &&
code[i + 1] == '/')
                    {
                        count = 0;
                        index = i;
                        while (index <
code.Length && code[index] != '\r')
                        {
                            count++;
                            index++;
                        }
                        code = code.Remove(i,
count);
                    }
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message,
"Помилка", MessageBoxButtons.OK, MessageBoxI-
con.Error);
            }
            return code;
        }

        public static string DeleteMul-
tiLineComments(string code)//функция удаление
многострочных комметариев
        {
            try
            {
                int count;
                int index;
                for (int i = 0; i <
code.Length - 1; i++)
                {
                    if (code[i] == '/' &&
code[i + 1] == '*')
                    {
                        count = 2;
                        index = i;
                        while (index <
code.Length - 2 && (code[index] != '*' ||
code[index + 1] != '/'))
                        {
                            count++;
                            index++;
                        }
                        code = code.Remove(i,
count);
                    }
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message,
"Помилка", MessageBoxButtons.OK, MessageBoxI-
con.Error);
            }

            return code;
        }

        public static string DeleteStringLit-
eral(string code)//аункция удаление строковых
""
        {
            try
            {
                for (int i = 0; i <
code.Length - 1; i++)
                {
                    int count;
                    int index;
                    if (code[i] == '\"')
                    {
                        count = 2;
                        index = i + 1;
                        while (index <
code.Length - 1 && code[index] != '\"')
                        {
                            if (code[index]
== '\\' && code[index + 1] == '\"')
                            {
                                count += 2;
```

```
                            index += 2;
                        }
                        else
                        {
                            count++;
                            index++;
                        }
                    }
                    code = code.Remove(i,
count);
                }
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message,
"Помилка", MessageBoxButtons.OK, MessageBoxI-
con.Error);
        }
        return code;
    }

    public static string DeleteSym-
vol(string code)//функция удаления символов
''
    {
        try
        {
            for (int i = 0; i <
code.Length - 1; i++)
            {
                if (code[i] == '\'')
                {
                    if (code[i + 1] ==
'\\')
                    {
                        code = code.Re-
move(i, 4);
                        i += 4;
                    }
                    else
                    {
                        code = code.Re-
move(i, 3);
                        i += 3;
                    }
                }
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message,
"Помилка", MessageBoxButtons.OK, MessageBoxI-
con.Error);
        }

        return code;
    }

    public static string DeleteParenthe-
sis(string code)//аункция удаления выражения
в круглых скобках ()
    {
        try
        {
            for (int i = 0; i <
code.Length - 1; i++)
            {
                if (code[i] == '(')
                {
                    int index = i + 1;
                    int count = 0;
                    int counter = 1;
                    while (counter > 0)
                    {
                        if (code[index]
== '('
                        {
                            counter++;
                        }
                        else if (code[in-
dex] == ')')
                        {
                            counter--;
                        }
                        count++;
                        index++;
                    }
                    code = code.Remove(i
+ 1, count - 1);
                }
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message,
"Помилка", MessageBoxButtons.OK, MessageBoxI-
con.Error);
        }
        return code;
    }

    public static List<string> Allocate-
Tokens(string code)//функция выделения лексем
    {
        var tokens = new List<string>();

        try
        {
            string lexem = default;
            for (int i = 0; i <
code.Length; i++)
            {
                if (separators.Con-
tains(code[i].ToString()))
                {
                    if (!String.IsNul-
lOrEmpty(lexem))
                    {
                        to-
kens.Add(lexem);
                        lexem =
String.Empty;
                    }
                    to-
kens.Add(code[i].ToString());
                }
                else
                {
                    lexem += code[i];
```

```
                    }
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message,
"Помилка", MessageBoxButtons.OK, MessageBoxI-
con.Error);
            }

            return tokens;
        }

        public static string AllocateFirstTo-
kens(string code)//функция выделения первой
лексемы
        {
            string lexem = default;
            try
            {

                for (int i = 0; i <
code.Length; i++)
                {
                    if (separators.Con-
tains(code[i].ToString()))
                    {
                        if (!String.IsNul-
lOrEmpty(lexem))
                        {
                            return Key-
Worlds.Contains(lexem.ToLower()) ? lexem :
"Process";
                        }
                    }
                    else
                    {
                        lexem += code[i];
                    }
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message,
"Помилка", MessageBoxButtons.OK, MessageBoxI-
con.Error);
            }
            return KeyWorlds.Con-
tains(lexem.ToLower()) ? lexem : "Process";
        }

    }
}
```

LexicalAnalysis.cs

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;

namespace ConformityAssessment.Code
{
    class LexicalAnalysis
    {
        private string Code { get;
set; }//програмный код функции на языке C++
        private List<string> Tokens { get;
set; } = new List<string>();//лексемы
програмного кода
```

```
        private List<string> Actions { get;
set; } = new List<string>();//список действий


        private readonly List<string> Separa-
tors = new List<string>()//колекция
разделителей
            { " ", "\n", "\r", ",", ".", "(",
")", "{", "}", "<", ">", "*",
              "/", "+", "-", "=", "&", "%",
"?", "!", ";", ":", "^" };
        //список ключевых слов
        private Dictionary<string, int> Key-
Worlds { get; set; } = new Dictionary<string,
int>()//колекция ключевых слов
        {
            {"if", 1}, {"for", 1}, {"switch",
1},  {"do", 1},
            {"break", 2}, {"continue", 2},
{"return", 2},
            {"while", 3},
            {"else", 4},
            {"case", 5},{"default", 5}
        };

        //конструктор
        public LexicalAnalysis (string code)
        {
            Code = code;
        }

        //лексический анализ кода
        public (List<string>, string) Analy-
sis()
        {
            string Name = default;
            try
            {
                //преддварительная обработка
кода
                Code = Preprocessing-
Code.DeleteOneLineComments(Code);
                Code = Preprocessing-
Code.DeleteMultiLineComments(Code);
                Code = PreprocessingCode.De-
leteSymvol(Code);
                Code = PreprocessingCode.De-
leteStringLiteral(Code);
                Code = Preprocessing-
Code.DeleteParenthesis(Code);
                //выделяем лексемы
                Tokens = Preprocessing-
Code.AllocateTokens(Code);
                //удаляем лексемы типа ' '
                for (int i = Tokens.Count -
1; i >= 0; --i)
                    if (Tokens[i] == " " ||
Tokens[i] == "\r" || Tokens[i] == "\n") To-
kens.RemoveAt(i);

                //находим имя функции
                Name = Tokens[Tokens.In-
dexOf("(") - 1];
                //обрабатываем тело функции
                BracesBody(Tokens.In-
dexOf("{"), false);
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message,
"Помилка", MessageBoxButtons.OK, MessageBoxI-
con.Error);
            }
            return (Actions,Name);

        }
        //обработка тела в {}
        public int BracesBody(int index, bool
flagcase)
        {
            try
            {
                //флаг первой лексемы
                bool f1 = true;

                //счетсчик для тела-ключевое
слово
                int counter = 0;
                //идем полексемно
                for (int i = index + 1; i <
Tokens.Count; i++)
                {
                    //если тело case
                    if (flagcase)
                    {
                        if (Tokens[i] == "}"
|| Tokens[i] == "case" || Tokens[i] == "de-
fault")
                        {
                            return i - 1;
                        }//если первая
лексема не ключевое слово
                    }
                    else
                    //если конец тела
                    if (Tokens[i] == "}")
                    {
                        if (i != Tokens.Count
- 1)
                        {
                            while (counter !=
0) { Actions.Add("--"); counter--; }
                            Actions.Add("--
");
                        }
                        return i;
                    }//если первая лексема не
ключевое слово
                    if (f1 && !KeyWorlds.Con-
tainsKey(Tokens[i]) && !Separators.Con-
tains(Tokens[i]))
                    {
                        Actions.Add("Pro-
cess");
                        f1 = false;
                    }
                    //если встретили клюевое
слово
```

```
                if (KeyWorlds.Contains-
Key(Tokens[i]))
                {
                    switch (KeyWorlds[To-
kens[i]])
                    {
                        //если первый тип
                        case 1:
                            {
                                Ac-
tions.Add(Tokens[i]);
                                Ac-
tions.Add("++");
                                i++;
                                while
(true)
                                {
                                    if
(Tokens[i] == ";")
                                    {

Actions.Add("Process");

Actions.Add("--");

f1 = true;

while (counter != 0) { Actions.Add("--");
counter--; }

break;
                                    }
                                    else
if (Tokens[i] == "{")
                                    {
                                        i
= BracesBody(i, false);


f1 = true;

break;
                                    }
                                    else
if (KeyWorlds.ContainsKey(Tokens[i]))
                                    {

i--;

counter++;

break;
                                    }
                                    i++;
                                }

                                break;
                            }
                        //если второй тип
                        case 2:
                            {
                                Ac-
tions.Add(Tokens[i]);
                                f1 =
true;
                                while
(true)
                                {
                                    if
(Tokens[i] == ";")
                                    {

break;
                                    }
                                    i++;
                                }
                                break;
                            }
                        //если третий тип
                        case 3:
                            {
                                if (To-
kens[i + 3] != ";")
                                {
                                    goto
case 1;
                                }
                                Ac-
tions.Add(Tokens[i]);
                                f1 =
true;
                                i += 3; ;
                                break;
                            }
                        //если четвертый
тип
                        case 4:
                            {
                                if (To-
kens[i + 1] != "if")
                                {
                                    goto
case 1;
                                }
                                Ac-
tions.Add(Tokens[i]);

                                break;
                            }
                        case 5:
                            {
                                int case-
begin = Tokens.IndexOf(":", i);
                                if (To-
kens[casebegin + 1] == "{")
                                { goto
case 1; }

                                Ac-
tions.Add(Tokens[i]);
                                Ac-
tions.Add("++");
                                i = Brac-
esBody(casebegin, true);
                                Ac-
tions.Add("--");
                                break;
                            }
                    }
                }
```

```
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show(ex.Message,
"Помилка", MessageBoxButtons.OK, MessageBoxI-
con.Error);
            }
            return 0;
        }


    }
}
```

FlowchartModel.cs

```
using System.Collections.Generic;
using Newtonsoft.Json;

namespace ConformityAssessment.Flowchart
{
    [JsonObject]
    public class FlowchartModel
    {
        [JsonProperty("blocks")]
        public IEnumerable<Blocks> Blocks
{ get; set; }

        [JsonProperty("arrows")]
        public IEnumerable<Arrows> Arrows
{ get; set; }

    }

    [JsonObject]
    public class Blocks
    {
        [JsonProperty("text")]
        public string Text { get; set; }

        [JsonProperty("type")]
        public  string Type { get; set; }
    }

    [JsonObject]
    public class Arrows
    {
        [JsonProperty("startIndex")]
        public int Begin { get; set; }

        [JsonProperty("endIndex")]
        public int End { get; set; }
    }
}
```

Graph.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Collections;
using System.IO;
using ConformityAssessment.Code;
using ConformityAssessment.Flowchart;
using Newtonsoft.Json;
using System.Windows.Forms;

namespace ConformityAssessment
{
    public class Graph
    {
        //Имя графа
       public string Name { get; set; }

        //список вершин графа
        public ArrayList Vertex { get; set; }

        //Список ребр графа
        public List<List<int>> Edge { get;
set; }

        //конструктор
        public Graph()
        {
            Vertex = new ArrayList();
            Edge = new List<List<int>>();

            //Добавление вершины конца
            Vertex.Add("end");
            Edge.Add(new List<int>());

            //Добавление вершины начала
            Vertex.Add("begin");
            Edge.Add(new List<int>());
        }

        public void AddVertex(string str)
        {
            Vertex.Add(str);
            Edge.Add(new List<int>());
        }

        public void AddEdge(int start, int
finish)
        {
            Edge[start].Add(finish);
        }

        public void BuildGrafFromCode(string
code)
        {
            try
            {
                if (code.Length < 10)
                {
                    throw new Excep-
tion("Програмний код не введено!");
                }

                LexicalAnalysis lexicalAnaly-
sis = new LexicalAnalysis(code);
```

```
            var resultLexicalAnalysis =
lexicalAnalysis.Analysis();
            List<string> actions = re-
sultLexicalAnalysis.Item1;
            Name = resultLexicalAnaly-
sis.Item2;

            var stackTemp = new
Stack<int>();//стек для хранения вершин,
используеться операциями ++ и --
            var addEdgeNextVertex = new
Stack<int>();//стек для хранения вершин, что
имеют ребро с вершиной, что будет добавлена
на следующем шаге
            var addEdgeRootVertex = new
Stack<int>();//стек для хранения вершин, что
являються корнем подграфа
            var stackBreak = new
Stack<int>();//стек для хранения вершин,
которые реализуют операцию break
            var stackContinue = new
Stack<int>();// стек для хранения вершин,
которые реализуют операцию continue
            var stackSwitch = new
Stack<int>();// стек для хранения вершин case

            bool f1 = false;//флаг, что
используеться для обозначении вершины как
корня подграфа
            bool f2 = false;//флаг, то
используеться для запрета дуги в вершину, что
будет добавлена

            int stackPop;//локальна
переменная для изьятия вершины из stackTemp



            for (int i = 0; i < ac-
tions.Count; i++)//пока action ну будет
пройден до конца
            {
                if (actions[i] ==
"++")//операция ++
                {
                    if (actions[i - 1] ==
"else") stackTemp.Push(-1);//если else,
заносим вместо идекса вершины -1
                    else if (actions[i -
1] == "do") stackTemp.Push(-1 * Ver-
tex.Count);//если do, заносим отреательный
индекс вершины, что что будет добавлена
                    else
stackTemp.Push(Vertex.Count - 1);//добавляем
идекс текущей вершины

                    if (actions[i - 1] ==
"switch") f2 = true;//если вершина, что
заноситься в стек switch, то запрещаем
выводить из нее дугу
                }
                else if (actions[i] == "-
-")//если операция --
                {
                    stackPop =
stackTemp.Pop();//извлекам индеск вершины из
stackTemp

                    if (stackPop == -
1)//если извлекаем else
                    {
                        //добавляем
последнюю вершину else к вершинам для
соединения с корнем
                        if (!f2)
                        {
                            if
(Edge[Edge.Count - 1].Count == 0) addEdge-
RootVertex.Push(Edge.Count - 1);
                        }
                        else f2 = !f2;


                        f1 = true;//ведем
дуги в корень

                        //если следущая
операция -- и извлекаем вершину while или for
то делаем ее корнем подграфа и выводим все
дуги из addEdgeRootVertex
                        if (i < ac-
tions.Count - 1 && actions[i + 1] == "--"
                            && (Ver-
tex[stackTemp.Peek()].ToString() == "while"
|| Vertex[stackTemp.Peek()].ToString() ==
"for"))
                        {
                            while (add-
EdgeRootVertex.Count != 0) this.AddEdge(add-
EdgeRootVertex.Pop(), stackTemp.Peek());
                            f1 = false;
                        }

                        f2 = true;
                    }
                    else if (stackPop <
0)//если извлекаем do
                    {
                        //добавляем новую
вершину while
                        i++;

this.AddVertex(actions[i]);
                        this.Add-
Edge(Edge.Count - 2, Edge.Count - 1);
                        //добавляем ребро
с while назад
                        this.Add-
Edge(Edge.Count - 1, stackPop * -1);
                        //добавляем ребро
из вершины continue в вершину while
                        while (stackCon-
tinue.Count != 0) this.AddEdge(stackCon-
tinue.Pop(), Edge.Count - 1);
                        //добавляем
вершину break к корню
                        while (stack-
Break.Count != 0) addEdgeNextVer-
tex.Push(stackBreak.Pop());
```

```
                        }
                        else switch (Ver-
tex[stackPop])
                            {
                                case "while":
                                    {

//если такое ребро уже есть то не дублируем
его

//добавляем ребро из последней вершины в
вершину while
                                        if
(!f2)
                                        {

if (Edge[Edge.Count - 1].IndexOf(stackPop) ==
-1) this.AddEdge(Edge.Count - 1, stackPop);
                                        }
                                        else
f2 = !f2;

//добавляем ребро из вершины continue в
вершину while
                                        while
(stackContinue.Count != 0) this.Add-
Edge(stackContinue.Pop(), stackPop);

//добавляем вершину while к вершинам для
соединения с корнем

addEdgeRootVertex.Push(stackPop);

//добавляем вершину break к корню
                                        while
(stackBreak.Count != 0) addEdgeRootVer-
tex.Push(stackBreak.Pop());


//рисуем ребра к корню
                                        f1 =
true;

//если следующаю вершину достаем while или
for до рисуем в нее все корни
                                        if (i
< actions.Count - 1 && actions[i + 1] == "--"
                                        &&
stackTemp.Count != 0 && (Ver-
tex[stackTemp.Peek()].ToString() == "while"
|| Vertex[stackTemp.Peek()].ToString() ==
"for"))
                                        {

while (addEdgeRootVertex.Count != 0)
this.AddEdge(addEdgeRootVertex.Pop(),
stackTemp.Peek());

f1 = false;
                                        }
                                        f2 =
true;

break;
                                    }
                                case "for":
                                    {
                                        goto
case "while";
                                    }
                                case "if":
                                    {

//если следующая вершина else
                                        if (i
< actions.Count - 1 && actions[i + 1] ==
"else")
                                        {

//добавляем ребро из if в первую вершину else

addEdgeNextVertex.Push(stackPop);

//добавляем последнюю вершину if к вершинам
для соединения с корнем

if (!f2) addEdgeRootVertex.Push(Edge.Count -
1);

else f2 = !f2;
                                        }
                                        else
                                        {

//если конструкция без else

//если с вершины не выходли ребра и нету
флага нереба

//добавляем последнюю вершину if к вершинам
для соединения с корнем

if (!f2)
                                            {

if (Edge[Edge.Count - 1].Count == 0) addEdge-
RootVertex.Push(Edge.Count - 1);
                                            }

else f2 = !f2;


//добавляем вершину if к вершинам для
соединения с корнем

addEdgeRootVertex.Push(stackPop);

//рисуем ребра к корню

f1 = true;
                                        }


//если следующаю вершину достаем while или
for до рисуем в нее все корни
                                        if (i
< actions.Count - 1 && actions[i + 1] == "--"
                                        &&
stackTemp.Count != 0 && (Ver-
tex[stackTemp.Peek()].ToString() == "while"
```

```
|| Vertex[stackTemp.Peek()].ToString() ==
"for"))
                                        {

while (addEdgeRootVertex.Count != 0)
this.AddEdge(addEdgeRootVertex.Pop(),
stackTemp.Peek());

f1 = false;
                                        }

                                        f2 =
true;

break;
                                    }

                                case "case":
                                    {

//Добавляем вершину case для соединения с
вершиной switch

stackSwitch.Push(stackPop);

//добавляем вершину break к корню
                                        while
(stackBreak.Count != 0) addEdgeRootVer-
tex.Push(stackBreak.Pop());



break;
                                    }
                                case
"switch":
                                    {
                                        while
(stackSwitch.Count != 0) this.AddEdge(stack-
Pop, stackSwitch.Pop());
                                        f1 =
true;

break;
                                    }

                                case "de-
fault":
                                    {
                                        goto
case "case";
                                    }
                            }
                    }

                    //если вершина не do или
else то добавляем ее вграф
                    else if (actions[i] !=
"do" && actions[i] != "else")
                    {
                        //добавляем следующую
вершину
                        this.AddVertex(ac-
tions[i]);
                        //если флаг неребра
                        if (!f2)
                        {
                            //если он false,
то соеденяем новую вершину с последней
                            this.Add-
Edge(Edge.Count - 2, Edge.Count - 1);
                        }
                        //если он true, то
делаем его false
                        else f2 = !f2;

                        //если флаг корня
                        if (f1)
                        {
                            //добавляем в
корень все вершины
                            while (addEdge-
RootVertex.Count != 0) this.AddEdge(addEdge-
RootVertex.Pop(), Edge.Count - 1);
                            f1 = !f1;
                        }
                        //добавляем все ребра
для следующей вершины
                        while (addEdgeN-
extVertex.Count != 0) this.AddEdge(addEdgeN-
extVertex.Pop(), Edge.Count - 1);

                        if (actions[i] ==
"return")
                        {
                            this.Add-
Edge(Edge.Count - 1, 0);
                            f2 = true;

                        }
                        else if (actions[i]
== "break")
                        {
                            stack-
Break.Push(Edge.Count - 1);
                            f2 = true;
                        }
                        else if (actions[i]
== "continue")
                        {
                            stackCon-
tinue.Push(Edge.Count - 1);
                            f2 = true;
                        }
                    }

                }

                if (f1)
                {
                    //добавляем в корень все
вершины
                    while (addEdgeRootVer-
tex.Count != 0) this.AddEdge(addEdgeRootVer-
tex.Pop(), 0);
                    f1 = !f1;
                }
                if (!f2)
                {
```

```
                    //добавляем ребро из
последней вершины в вершину end
                    this.AddEdge(Edge.Count -
1, 0);
                }
                else f2 = !f2;
                this.SaveToFile(Path.GetFull-
Path(Path.Combine(Directory.GetCurrentDirec-
tory(), "..\\Example\\Graf1.txt")));
                Message-
Box.Show("Розпізнавання програмного коду
виконано!", "Програмний код", MessageBoxBut-
tons.OK, MessageBoxIcon.Information);
            }
            catch (Exception ex)
            {

                MessageBox.Show(ex.Message,
"Помилка", MessageBoxButtons.OK, MessageBoxI-
con.Error);
            }
        }

        //добавит стриговый возврат с
результатом
        public void
BuildGrafFromFlowchart(string code)//функция
построения графа на основе блок схемы
        {
            try
            {
                if (code.Length < 10)
                {
                    throw new
Exception("Невірний формат вхідних даних!");
                }
                int begin = 0;//индекс начала
                int end = 0;//индекс конца
                Stack<int> stackDelete = new
Stack<int>();
                FlowchartModel model =
JsonConvert.DeserializeObject<Flowchart-
Model>(code);//преобразуем код json в модель

                List<Blocks> blocks =
model.Blocks.ToList();//приводим множество
блоков в список
                if (blocks.Where(b => b.Type
== "Начало / конец").Count() != 2) throw new
Exception("Невірний формат вхідних
даних!"); ;//если блоков начала/конца > 2

                for (int i = 0; i <
blocks.Count; i++)//добавляем вершины в граф
                {
                    if (blocks[i].Type ==
"Начало / конец")
                    {
                        if (blocks[i].Text ==
"begin") begin = i;
                        else end = i;
                    }
                    else this.AddVertex(Pre-
processingCode.AllocateFirstTo-
kens(blocks[i].Text));
                }

                foreach (var arrow in
model.Arrows)//добавляем дуги в граф
                {
                    if ((blocks[ar-
row.Begin].Type == "Ввод / вывод" &&
blocks[arrow.End].Type == "Блок") ||
                        (blocks[ar-
row.Begin].Type == "Блок" && blocks[ar-
row.End].Type == "Ввод / вывод"))
                    {
                        if (model.Ar-
rows.Where(a => a.End == arrow.End).Count()
== 1)
                        {
                            stack-
Delete.Push(arrow.Begin + 1);
                            stack-
Delete.Push(arrow.End + 1);
                        }
                    }
                    if (arrow.Begin == begin)
AddEdge(1, arrow.End + 1);
                    else if (arrow.End ==
end) AddEdge(arrow.Begin + 1, 0);
                    else AddEdge(arrow.Begin
+ 1, arrow.End + 1);
                }
                while (stackDelete.Count !=
0)
                {
                    end = stackDelete.Pop();
                    begin = stack-
Delete.Pop();
                    this.Edge[begin] = new
List<int>(this.Edge[end]);
                    this.Edge.RemoveAt(end);
                    this.Vertex.Re-
moveAt(end);
                    for (int i = 0; i <
Edge.Count; i++)
                    {
                        for (int j = 0; j <
Edge[i].Count; j++)
                        {
                            if (Edge[i][j] >=
end) Edge[i][j]--;
                        }
                    }
                }
                Message-
Box.Show("Розпізнавання блок-схеми
виконано!", "Блок-схема", MessageBoxBut-
tons.OK, MessageBoxIcon.Information);
                this.SaveToFile(Path.GetFull-
Path(Path.Combine(Directory.GetCurrentDirec-
tory(), "..\\Example\\Graf2.txt")));
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message,
"Помилка", MessageBoxButtons.OK, MessageBoxI-
con.Error);
            }
```

```
        }
        public static double GraphCompari-
son(Graph graf1, Graph graf2)//функция
сравнения графов
        {
            bool[] used1 = default;
            bool[] used2 = default;
            try
            {
                if (graf1 == null || graf2 ==
null)
                {
                    throw new Excep-
tion("Невірний формат вхідних даних!");
                }
                Queue<int> q1 = new
Queue<int>();//очередь для обхода в ширину
графа graf1
                Queue<int> q2 = new
Queue<int>();//очередь для обхода в ширину
графа graf2

                int cur1 = 0;//перменная для
хранения вершины q1
                int cur2 = 0;//переменная для
хранения вершины q2

                used1 = Enumerable.Re-
peat<bool>(false, graf1.Vertex.Count).ToAr-
ray();//посещенность вершин graf1
                used2 = Enumerable.Re-
peat<bool>(false, graf2.Vertex.Count).ToAr-
ray();//посещенность вершин graf2

                used1[1] = used2[1] =
used1[0] = used2[0] = true;

                q1.Enqueue(1);
                q2.Enqueue(1);

                while (q1.Count != 0 &&
q2.Count != 0)
                {
                    if (q1.Count != 0) cur1 =
q1.Dequeue();
                    if (q2.Count != 0) cur2 =
q2.Dequeue();
                    if (graf1.Ver-
tex[cur1].Equals(graf2.Vertex[cur2]))
                    {
                        used1[cur1] = true;
                        used2[cur2] = true;
                    }
                    for (int i = 0; i <
graf1.Edge[cur1].Count; i++) if
(used1[graf1.Edge[cur1][i]] != true)
q1.Enqueue(graf1.Edge[cur1][i]);
                    for (int i = 0; i <
graf2.Edge[cur2].Count; i++) if
(used2[graf2.Edge[cur2][i]] != true)
q2.Enqueue(graf2.Edge[cur2][i]);
                }
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex.Message,
"Помилка", MessageBoxButtons.OK, MessageBoxI-
con.Error);
            }
            return (double)(used1.Where(x =>
x).Count() + used2.Where(x => x ==
true).Count() - 2) / (double)(used1.Length +
used2.Length - 2) * 100;
        }

        public void SaveToFile(string path)
        {
            string str;
            using (StreamWriter sw = new
StreamWriter(path, false, System.Text.Encod-
ing.Default))
            {
                for (int i = 0; i <
Edge.Count; i++)
                {
                    str = String.Empty;
                    str+= i.ToString() + " "
+ Vertex[i] +  " - ";
                    for (int j = 0; j <
Edge[i].Count; j++)
                    {
                        str +=
Edge[i][j].ToString() + " ";
                    }
                    sw.WriteLine(str);
                }

            }

        }
    }
}
```

Program.cs

```
using System;
using System.Windows.Forms;

namespace ConformityAssessment
{
    static class Program
    {
        /// <summary>
        /// Главная точка входа для
приложения.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTex-
tRenderingDefault(false);
            Application.Run(new Main());
        }
    }
}
```

Міністерство освіти і науки України
Дніпровський національний університет залізничного транспорту
імені академіка В. Лазаряна

ТЕЗИ

Всеукраїнської науково-технічної конференції молодих учених, магістрантів та студентів «Науково-технічний прогрес на транспорті»

(29 березня 2021 року)

Дніпро
2021

СЕКЦІЯ 3
БУДІВЕЛЬНА ІНЖЕНЕРІЯ ТА ЕКОЛОГІЧНА БЕЗПЕКА
CIVIL ENGINEERING AND ENVIRONMENTAL SAFETY

criminal purposes and more. That is why the issue of cyberattacks is very acute in the modern world, which confirms the relevance of the topic.

One of the most popular types of attacks is network. Appropriate software and hardware are available to protect computer networks. However, there is a problem tracking authorized connections that can gain full access to network services. Intrusion detection systems are used to detect such attacks in real time. There is a standard approach - the analysis of network traffic and in case of detection of anomalies by sensors of sending of data to analyzers then the question of the further actions is solved. But this approach is not effective with a large amount of data. In the case of a large amount of data, it is advisable to use Data Mining technology.

At the present stage, neural networks are used to detect network attacks, the advantage of which is that they are capable of self-learning, they can find new network attacks. In addition, the method of detecting attacks based on neural networks is rational, as it allows you to select a large number of features, and then classify network packets. This allows you to get the following data: detect the attack in real time, set its type and characteristics. A review of scientific sources has shown that network attacks can be detected based on the following neural networks: Multi-Layer Perceptron (MLP); Radial Basis Function Network (RBF); Kohonen network or Self Organizing Maps (SOM); fuzzy network (Adaptive-Network-Based Fuzzy Inference System, ANFIS).

Since there are many types of neural networks with different capabilities, the results of their work may differ. The essence of hybrid approaches is to implement various schemes of combining basic classifiers, which allow to eliminate shortcomings in their operation separately. However, at the same time an important disadvantage of such techniques is the lack of universality of their application. Therefore, for further work it is proposed to use a hybrid approach to detecting network attacks: simultaneously based on three different neural network models.

It is known that neural network models can be created both programmatically (Python, PHP, etc.) and using neural packages (MatLAB, St Neural Networks, etc.). To create samples for the purpose of learning neural networks, the NSL-KDD database was chosen. The database presents the following categories of attacks: DoS; R2L; U2R; Probe, each of which, in turn, is served by several classes.

References:

1. Выбор технологий Data Mining для систем обнаружения вторжений в корпоративную сеть [URL] Выбор технологий Data Mining для систем обнаружения вторжений в корпоративную сеть | Инженерный журнал: наука и инновации (engjournal.ru)

B. M. Yakovenko

Research supervisor: O.S. Kuropiatnyk, Candidate of Engineering Sciences, Associate Professor

Language supervisor: A. O. Muntian, Candidate of Philological Sciences, Associate Professor

Dnipro National University of Railway Transport named after Academician V. Lazarian

RECOGNITION OF A FLOWCHART FOR CONVERSION TO A GRAPH VIEW

A flowchart is a diagram that depicts a process, system or computer algorithm. As a visual representation of data flow, flowcharts are useful in writing a program or algorithm and explaining it to others or collaborating with them on it. Therefore, flowcharts are often used in academic works as part (fragments) of program documentation.

But if the data from the flowchart is needed for further analysis, then the flowchart have to be converted to another type of data, for example in graph view. To solve this problem it is necessary to develop an algorithm that should recognize the controls and data the flowchart.

Block diagram is flowchart kind and is composed of function blocks of different shapes, linked by lines. Each block describes one or several actions. Flowchart elements can be divided into two types: graphical elements and printed characters. The graphical elements of the block diagram are: Start/End symbol, Input/Output symbol, Process symbol, Decision symbol.

To recognize a flowchart requires:

1. Conduct digital processing of the block diagram image;
2. Select the graphical elements of the flowchart in the image;
3. Recognize printed text in the flowchart elements.

Work with digital images can be divided into three stages:

1. Initial filtration and image preparation:

For image filtration, methods are used that allow identifying the required areas on the image without analyzing them. Most of these methods use a single transformation to all points in the image. Such methods include binarization of image by threshold, Fourier transform, Wavelet reinterpretation, contour and boundary detection, correlation, etc.

2. Logical processing of filtration results:

After filtering the image, we get a set of data that is suitable for further processing. The methods that allow to go from the image to the objects in the image include: morphology, contour analysis, segmentation, modelling, Fourier descriptors and others.

3. Decision making algorithms based on logical processing:

After logical image processing, it is necessary to use methods that do not work with the image directly, but allow making decisions on the basis of the previous image processing. In many cases, machine learning and decision making tasks. For example, the task of recognizing text on the image is included in the category of machine learning classification tasks.

The task of recognizing text in an image is quite popular, and there are many scientific works on this topic. Two approaches are used to create a system of text recognition: metrics and neural networks.

Today there are three main approaches for solving the problem of recognition of printed characters by means of metrics: pattern, structural, and iconic.

Current approaches to shape recognition can be divided as follows: methods based on the outline and methods based on the area, spatial domain and transformation domain; information-preserving and not-information-preserving methods. However, approaches to the vision and submission of figures are often divided, depending on the processing methods, into Onedimensional function, Polygonal approximation, Spatial interrelation feature, Moments, Scalespace methods, Shape transform domains. To define disordered shapes on the basis of contours, we often use: complex coordinates, distance function, dot-cut, contour curvature, and Fourier descriptors. All these methods (except for the Fourier descriptors) are included in the class of "one-dimensional functions of figure perception".

The recognition result can be used to convert the flowchart into a graph view.

S. S. Zhuk
Language supervisor: I.V. Shpak
Dnipro National University of Railway Transport named after Academician V. Lazarian

WHO IS A WEB DEVELOPER? WHAT DOES HE DO?

Web developer: To answer the question “What is a web developer?”, we must first look at what a web developer does and how they do it. A web developer or programmer is someone who takes a web design - which has been created by either a client or a design team - and turns it into a website. They do this by writing lines and lines of complicated code, using a variety of languages. Web developers have quite a difficult job because they essentially have to take a

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

АТ «УКРАЇНСЬКА ЗАЛІЗНИЦЯ»

ДНІПРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ЗАЛІЗНИЧНОГО ТРАНСПОРТУ ІМЕНІ АКАДЕМІКА В. ЛАЗАРЯНА

УКРАЇНСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ ЗАЛІЗНИЧНОГО ТРАНСПОРТУ

ДЕРЖАВНИЙ УНІВЕРСИТЕТ ІНФРАСТРУКТУРИ ТА ТЕХНОЛОГІЙ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ АВТОМОБІЛЬНО-ДОРОЖНІЙ УНІВЕРСИТЕТ

ІНФОРМАЦІЙНО-ТЕЛЕКОМУНІКАЦІЙНІ ТЕХНОЛОГІЇ ТА КОМП'ЮТЕРНЕ МОДЕЛЮВАННЯ

ЗБІРНИК ТЕЗ ДОПОВІДЕЙ
81 Всеукраїнської науково-технічної конференції
молодих учених, магістрантів та студентів
«НАУКА І СТАЛИЙ РОЗВИТОК ТРАНСПОРТУ»
28 жовтня 2021 року

INFORMATION-TELECOMMUNICATION TECHNOLOGY TA COMPUTER MODELING

CONFERENCE PROCEEDINGS
81th all Ukrainian Scientific and Technical Conference
of young scientists, masters and students
"SCIENCE AND SUSTAINABLE DEVELOPMENT OF TRANSPORT"
October 28, 2021

ДНІПРО
2021

ЗМІСТ

нку. Класичним прикладом структурного підходу є побудова дерева програми з наступним порівнянням дерев для різних програм. В ході аналізу існуючих алгоритмів для попарного порівняння коду програм було виділено два алгоритми: жадібного строкового замощення та методу ідентифікаційних міток.

Перший алгоритм потребує токенізації вхідного програмного коду (виділення найбільш значущих лексичних конструкцій). Таким чином код програми буде перетворений на рядок із символів, відповідних вибраним словам. Такі символи називають токенами, а рядок – рядком токенів, чи токенізованим представленням програмного коду. Результатом застосування жадібного алгоритму для таких рядків буде набір їх спільних підрядків, що не перетинаються. Застосування другого алгоритму полягає у хешуванні отриманого рядка символів і порівняння в подальшому їх наборів. Таким чином спрощується задача аналізу отриманих результатів.

Описані вище алгоритми мають суттєвий недолік: їх неможливо використати для визначення саме структурної схожості програм на рівні проекту – з точки зору архітектури. Для вирішення цієї проблеми пропонується графічно відобразити архітектурну структуру програми за допомогою орієнтованих навантажених графів за дугами та вершинами. Навантаження дуг – тип зв'язку (узагальнення, асоціація, залежність, реалізація). Навантаження вершин – структура класу.

Аналіз схожості програм буде виконано на основі побудованих графів із застосуванням апарату конструктивно-продукційного моделювання. Розробка системи конструкторів для формалізації та зіставлення класової структури програми (конструктор породження для графового перетворення коду програми, конструктор-аналізатор для порівняння орієнтованих графів) дозволить зробити висновок чи є дві програми структурно схожі.

Окремою задачею в рамках дослідження можна виділити визначення метрик схожості, які можна застосувати саме для оцінки програм.

В рамках задачі доцільно провести експеримент над різними архітектурними рішеннями між класами та спробувати визначити, які саме структурні особливості та в якій мірі впливають на схожість між конкретними програмами.

Практичне застосування розробки, полягає у використанні програмного забезпечення в задачах виявлення запозичень, а також при формуванні критеріїв оцінювання навчальних проектів на основі аналізу їх структурної складності та міри схожості з іншими проектами.

ВИЗНАЧЕННЯ ВІДПОВІДНОСТІ ТЕКСТУ ПРОГРАМИ ГРАФІЧНОМУ ПРЕДСТАВЛЕННЮ АЛГОРИТМУ

Автор: Яковенко Б.М., студент групи ПЗ2021
Науковий керівник: к. т. н., доцент Куроп'ятник О.С.
Дніпровський національний університет залізничного
транспорту імені академіка В. Лазаряна

В академічному середовищі останнім часом дуже гостро постає проблема неправомірних запозичень у навчальних та наукових роботах. Вже існують розроблені моделі та методи для визначення запозичень у документах, в тому числі структурованих. Вони орієнтовані переважно на природомовні тексти, проте наукові роботи можуть також містити частини, написані штучними мовами, а саме фрагменти програмного коду, формули тощо, а також зображення (графічні представлення алгоритмів, UML-діаграми та ін.).

Фрагменти документів, написані природними та штучними мовами, не можна обробляти однаковими методами. Окремою задачею є порівняння між собою фрагментів, представлених різними штучними мовами: програмування, графічними мовами (наприклад, uml). Тому доцільним є розробка методу для порівняння програмного коду та графічних елементів між собою (як штучномовних конструкцій), оскільки існуючі методи виявлення

запозичень у наукових роботах працюють лише з фрагментами написаними однією мовою.

Для розробки методу визначення відповідності тексту програми графічному представленню алгоритму необхідно вирішити наступні задачі:

1. Лексичний аналіз програмного коду та побудова графу потоку керування.
2. Розпізнання блок-схеми на зображенні та побудова графового представлення, аналогічного до графу керування.
3. Зіставлення графів для виявлення відповідності.

Лексичний аналіз тексту програми перетворює програмний код в набір лексем, які мають вид *номер класу:номер в класі*. При побудові графу потоку керування в програмному коді виділяються такі конструкції:

– розгалуження: умовний оператор (if-else). оператор розгалуження (switch);

– цикли з лічильником (for), з передумовою (while), з постумовою (do-while).

Виділення керуючих конструкцій в тексті програми здійснюється за ключовими словами, характерними для конкретної мови програмування.

Розпізнавання блок-схеми на зображенні є задачею цифрової обробки зображення, а саме класифікацією. Необхідно провести попередню обробку зображення та виділити елементи блок-схеми, а саме: блоки початку, введення/виведення, умови та дії, лінія та лінія з стрілкою.

Після визначення блоку початку необхідно поступово розпізнавати кожний елемент блок-схеми, паралельно з цим будуючи граф потоку керування.

Зіставлення графів виконано на основі методу пошуку в ширину. Перегляд вершин є паралельним на обох графах, а вершини помічаються, якщо вони співпадають в обох графах.

Далі схожість графів виражається в числовому еквіваленті за формулою:

$$Match(A,B) = \frac{|A'| + |B'| - 2}{|A| + |B| - 2} \times 100,$$

де A, B – досліджувані графи, $|A|, |B|$ – кількість вершин у відповідних графах, $|A'|, |B'|$ – кількість помічених верши. В результаті отримуємо число в діапазоні $0 \ldots 100$, яке виражає схожість графів в процентному співвідношенні.

Для формалізації методу з метою комп’ютерної реалізації, і придатному для подальшої модифікації буде використано апарат конструктивно-продукційного моделювання.

Застосування розроблюваного методу є перспективним, адже дозволяє виявляти запозичення у академічних роботах, які містять фрагменти різними штучними мовами, при цьому розглядаючи роботу, подану на перевірку, як єдиний документ, що не потребує окремого програмного забезпечення в залежності від мови вміщуваного тексту.

ДОСЛІДЖЕННЯ ХАРАКТЕРИСТИК ІЄРАРХІЧНОГО КРАУДСОРСИНГУ В РОЗРОБЦІ ПРОГРАМ

Автор: Смірнов В. О. , студент групи 951м
Науковий керівник: к.т.н., доцент Андрющенко В. О.
Дніпровський національний університет залізничного транспорту імені академіка В. Лазаряна

Краудсорс системи набирають популярність останні п'ятнадцять років, і все частіше в сучасному світі широко використовується в сферах маркетингу, розробки програмного забезпечення, тестування. Особливістю краудсорс є те що плата за виконану роботу, на відміну від аутсорс відсутня або зовсім малооплачувана.

ISSN 2307–3489 (Print), ISSN 2307–6666 (Online)

Наука та прогрес транспорту. Вісник Дніпропетровського національного університету залізничного транспорту, 2021, № 4 (94)

ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНІ ТЕХНОЛОГІЇ ТА МАТЕМАТИЧНЕ МОДЕЛЮВАННЯ

УДК 004.622:[004.94:519.17]

О. С. КУРОП'ЯТНИК[1*], Б. М. ЯКОВЕНКО[2*]

[1*]Каф. «Комп'ютерні інформаційні технології», Дніпровський національний університет залізничного транспорту імені академіка В. Лазаряна, вул. Лазаряна, 2, Дніпро, Україна, 49010, тел. +38 (056) 373 15 35,
ел. пошта olena.kuropiatnyk@gmail.com, ORCID 0000-0003-2286-884X
[2*]Каф. «Комп'ютерні інформаційні технології», Дніпровський національний університет залізничного транспорту імені академіка В. Лазаряна, вул. Лазаряна, 2, Дніпро, Україна, 49010, тел. +38 (056) 373 15 35,
ел. пошта bohdanyakovenko98@gmail.com, ORCID 0000-0001-6174-0027

Визначення відповідності тексту алгоритму програми на основі конструктивно-продукційної моделі графа керування

Мета. Основною метою статті є розробка та програмна реалізація методу визначення відповідності тексту алгоритму програми, представленого у вигляді блок-схеми. **Методика.** Для зіставлення тексту програми та блок-схеми побудовано математичну модель їх перетворювачів у графове представлення з використанням апарату конструктивно-продукційного моделювання та його методів: спеціалізації, конкретизації, інтерпретації та реалізації. Графове представлення (графи) тексту будується з урахуванням операторів керування, блок-схеми – за json-файлом, що містить опис елементів схеми та їх зв'язків. Для порівняння графів застосовано метод пошуку в ширину з підрахунком кількості однакових вершин. Для програмної реалізації розробленого методу й моделей була застосована технологія об'єктно-орієнтованого програмування та CASE-технології, в основі яких лежить уніфікована мова моделювання UML. **Результати.** Запропоновано метод, що дозволяє представити текст та блок-схему програми в єдиному форматі орієнтованого графа (графа керування) та виконати оцінку їх відповідності за кількістю однакових вершин. Для його формалізації та автоматизованого використання розроблено конструктивно-продукційні моделі перетворювачів вхідних даних. На основі моделей та методу створено програмний додаток. **Наукова новизна.** Отримали подальший розвиток методи конструктивно-продукційного моделювання в задачах обробки текстів, написаних штучними мовами. Побудована система конструкторів, що виконує перетворення тексту програм мовою C++ у граф керування. **Практична значимість.** Результати роботи мають значення для розв'язання таких задач, як зіставлення текстів програм із метою виявлення запозичень, визначення відповідності алгоритмів програм їх програмним реалізаціям із метою поліпшення навичок кодування. Графове представлення, яке продукує розроблена система конструкторів, може бути застосоване для дослідження впливу оптимізації та рефакторингу коду на складність програм із використанням метрик МакКейба.

Ключові слова: конструктивно-продукційне моделювання; конструктор; графове представлення тексту; граф керування програми; алгоритм; відповідність алгоритму

Вступ

Невід'ємною частиною навчального процесу студентів спеціальності «Інженерія програмного забезпечення» є вивчення, побудова та реалізація алгоритмів. Важливо правильно інтерпретувати позначення, які використовують під час запису алгоритму, оскільки це дозволяє коректно його реалізувати у вигляді програмного коду. Студентам для самоконтролю, а викладачам для оцінювання якості виконаної роботи необхідна перевірка відповідності написаного програмного коду розробленому алгоритму.

doi: https://doi.org/10.15802/stp2021/

ISSN 2307–3489 (Print), ISSN 2307–6666 (Online)

Наука та прогрес транспорту. Вісник Дніпропетровського національного університету залізничного транспорту, 2021, № 4 (94)

Процес перевірки ускладнюється такими факторами як: складність алгоритму, що зумовлено великою кількістю складових (блоків) та їх вкладеністю; довжина реалізації алгоритму; можлива зміна та доповнення позначень змінних; великі обсяги даних (для викладача) тощо.

Автоматизація цього процесу дозволить усім його учасникам заощадити час. Для її реалізації необхідна розробка та формалізація методу визначення відповідності тексту й алгоритму програми, що передбачає побудову відповідних математичних моделей обробки вхідних даних. Вхідними даними є текст (код) програми, написаний мовою високого рівня, та алгоритм, представлений у вигляді блок-схеми.

Також сферою застосування зазначеного методу є перевірка документації, яка містить представлення алгоритмів та програм, на академічний плагіат. Така перевірка потребує зіставлення алгоритмів між собою та аналогічне зіставлення текстів. Наявні наукові роботи присвячені розв'язанню задачі виявлення плагіату в природомовних текстах [7, 8, 9], у тому числі у структурованих документах [10], у програмному коді [4, 11, 6]. Також програмні коди зіставляють для визначення алгоритмічної схожості [5]. Верифікація програм дозволяє визначити відповідність програми її специфікації [1, 2]. Проте задача зіставлення й визначення відповідності програмного коду алгоритму є невирішеною.

Мета

Основною метою даної роботи є розробка методу визначення відповідності тексту (програмного коду) алгоритму програми на основі їх конструктивно-продукційних (КП) моделей. Метод передбачає виконання алгоритмів перетворення вхідних даних у графи керування та їх подальше зіставлення.

Методика

Для визначення відповідності тексту й алгоритму програми пропонується метод на основі КП-моделювання. Метод має такі складові кроки:

1. Попередня обробка тексту програми, що передбачає видалення незначимих частин коду (коментарі, пробіли тощо) та його розбиття на лексеми.

2. Проміжне представлення тексту програми, отриманого на попередньому кроці, у вигляді списку керуючих елементів (реалізують алгоритмічні структури вибору та циклу, а також операції передачі керування) та елементів процесу (відповідають алгоритмічній конструкції слідування).

3. Побудова графа керування програми за списком керуючих елементів.

4. Побудова графа керування за алгоритмом програми, що представлений блок-схемою.

5. Зіставлення графів керування програми та алгоритму шляхом обходу в ширину [3].

Для моделювання побудови графа керування програми та його проміжного представлення у вигляді списку застосуємо апарат конструктивно-продукційного моделювання [13, 14], в основі якого лежить поняття узагальненого конструктора:

$$C = \langle M, \Sigma, \Lambda \rangle, \qquad (1)$$

де M – неоднорідний носій, який містить конструктивні елементи з атрибутами та може поповнюватися; Σ – сигнатура операцій (і відповідних відношень) зв'язування, підстановки й виведення, операцій над атрибутами; Λ – множина тверджень інформаційного забезпечення конструювання (ІЗК). ІЗК (конструктивна аксіоматика) містить онтологію, мету, правила, обмеження, початкові умови та умови завершення конструювання.

Призначення конструктора полягає у формуванні множин конструкцій за допомогою операцій сигнатури, які задають правилами ІЗК.

Для формування конструкцій необхідно виконувати ряд уточнюючих перетворень конструктора [13, 14]:

– спеціалізацію ($_S\mapsto$) – визначення предметної області застосування конструктора;

– інтерпретацію ($_I\mapsto$) – зв'язування операцій сигнатури з алгоритмами виконання деякої алгоритмічної структури;

– конкретизацію ($_K\mapsto$) – розширення ІЗК множиною правил продукцій, завдання конкретних множин нетермінальних і термінальних символів із їх атрибутами і, за необхідності,

doi: https://doi.org/10.15802/stp2021/

ISSN 2307–3489 (Print), ISSN 2307–6666 (Online)

Наука та прогрес транспорту. Вісник Дніпропетровського національного університету залізничного транспорту, 2021, № 4 (94)

значень атрибутів;

– реалізацію ($_R\mapsto$) – послідовне виконання операції виведення для побудови конструкцій.

Результати

Конструктор списку. Спеціалізований конструктор для перетворення програмного коду в проміжне представлення у вигляді списку має вигляд:

$$C=\langle M,\Sigma,\Lambda\rangle\ {}_S\mapsto C_T=\langle M_T,\Sigma_T,\Lambda_T\rangle, \quad (2)$$

де M_T – неоднорідний розширюваний носій, що складається з термінальних і нетермінальних елементів; Σ_T – множина операцій і відношень на елементах M_T; Λ_T – ІЗК, що містить онтологію, мету, обмеження, правила, умови початку й завершення конструювання.

Онтологія конструктора C_T: розширюваний носій складається з множини термінальних і нетермінальних елементів $M_T=\{T_T\cup N_T\}$. Термінальними є $T_T=\{txt,list,\{P_i\},\{O_i\},\{Ob_i\},\{L_i\},\varepsilon\}$, де $txt\supset\{Ob_i\}\cup\{O_i\}\cup\{P_i\}$ – послідовність операторів програмного коду; *list* – конструкція, що містить послідовність операторів програмного коду та допоміжних лексем; P_i – позначення процесу, що відповідає будь-якій послідовності операторів, що реалізує алгоритмічну структуру «слідування»; $\{O_i\}=\{return,\ break,continue\}$ – множина керуючих операторів без тіла; $\{Ob_i\}=\{if,else,do,while,\ for,switch,$ $case,default\}$ – множина керуючих операторів, що можуть мати власне тіло; $\{L_i\}=\{"++","--"\}$ – множина допоміжних лексем для заповнення списку, ε – порожній елемент (символ).

Розглянемо сигнатуру:

$$\Sigma_T=\langle\Theta_T,\{\to\},\{+\}\rangle\cup\Psi_G, \quad (3)$$

де $\Theta_T=\{\Rightarrow,|\Rightarrow,||\Rightarrow\}$ – множина операцій виведення; $\to$ – відношення підстановки; Ψ_G – множина правил продукції виведення $\psi_i=\langle\hat{s}_i,\breve{s}_i\rangle$, де i – номер правила, $\hat{s}_i$ – правило підстановки для розпізнавання тексту програмного коду, $\breve{s}_i$ – правило підстановки для побудови стеку; $+(el,list)$ – операція додавання нової вершини до списку, де el – значення нової вершини, $list$ – список.

Метою конструювання є побудова проміжного представлення програмного коду у вигляді списку.

Обмеження конструктора C_T накладають конструкцією програмного коду.

Початкова умова конструювання: η – нетермінал, із якого починається побудова конструкції списку; α – нетермінал, із якого починається розпізнавання тексту програми.

Умова завершення конструювання: форма не містить нетерміналів, побудована конструкція списку відповідає програмному коду.

У результаті конкретизації конструктора $C_{T\ K}\mapsto_K C_T$ маємо такі правила підстановки:

$$\hat{s}_0=\langle\alpha\to\gamma\alpha\,|\,\beta\alpha\rangle; \quad (4)$$

$$\hat{s}_1=\langle\beta\to P_1\,|\,P_2\ldots P_n\rangle, \quad (5)$$

$$\breve{s}_1=\langle\eta\to+(p,list)\eta\rangle;$$

$$\hat{s}_2=\hat{s}_3=\langle\gamma\to C_1\,|\,C_2\ldots C_n\rangle, \quad (6)$$

$$\breve{s}_2=\langle\eta\to+(Ob_i,list)+ +("++",list)\eta+("--",list)\eta\rangle;$$

$$\breve{s}_3=\langle\eta\to+(O_i,list)\eta\rangle; \quad (7)$$

$$\breve{s}_4=\langle\eta\to\varepsilon\rangle. \quad (8)$$

У ході інтерпретації проведемо зв'язування операцій сигнатури Σ_T з алгоритмами їх виконання:

$$\langle C_T=\langle M_T,\Sigma_T,\Lambda_T\rangle,C_A=\langle M_A,\Sigma_A,\Lambda_A\rangle\rangle\ {}_I\mapsto\ {}_{I,C_A}\mapsto C_T=\langle M_T,\Sigma_T,\Lambda_1\rangle, \quad (9)$$

де $M_A\supset V_A,V_A=\{A_i^0\,|_{X_i}^{Y_i}\}$ – множина базових алгоритмів; X_i,Y_i – множини визначення та значень алгоритму $A_i^0\,|_{X_i}^{Y_i}$;

doi: https://doi.org/10.15802/stp2021/

ISSN 2307–3489 (Print), ISSN 2307–6666 (Online)

Наука та прогрес транспорту. Вісник Дніпропетровського національного університету залізничного транспорту, 2021, № 4 (94)

$\Lambda_A = \left\{ M_A = \bigcup_{A_i^o \in V_A} \left(X\left(A_i^0\right) \subset Y\left(A_i^0\right)\right) \cup \Omega\left(C_T\right) \right\}$ – неоднорідний носій; $\Omega\left(C_T\right)$ – множина конструкцій списків, які задовольняють C_T; $\Lambda_1 = \Lambda_T \bigcup \Lambda_A \bigcup \Lambda_2$.

Конструктор $_{I,C_A} C_T$ містить алгоритми виконання операцій: $\Lambda_2 = \left\{ \left(A_1^0 \mid_{A_i, A_j}^{A_i \cdot A_j} \hookleftarrow \cdot \right), \left(A_2^0 \mid_S^{A_i} \hookleftarrow : \right), \left(A_3 \mid_{el, list}^{list} \hookleftarrow + \right), \left(A_4 \mid_{l_h, l_q, f_i}^{f_i} \hookleftarrow \Rightarrow \right), \left(A_5 \mid_{f_i, \Psi}^{f_j} \hookleftarrow \mid \Rightarrow \right), \left(A_6 \mid_{\sigma, \Psi}^{\bar{\Omega}} \hookleftarrow \| \Rightarrow \right) \right\}$.

Результатом реалізації конструктора (9) є множина конструкцій списків, які є проміжним представлення тексту програми та відповідного йому графа керування.

Конструктор графа. Спеціалізований конструктор для побудови графа керування програми за її проміжним представлення у вигляді списку керуючих операторів буде таким:

$$C = \langle M, \Sigma, \Lambda \rangle \ {}_S\mapsto C_G = \langle M_G, \Sigma_G, \Lambda_G \rangle, \quad (10)$$

де M_G – неоднорідний розширюваний носій; Σ_G – множина операцій і відношень на елементах M_G; Λ_G – ІЗК.

Онтологія конструктора C_G: розширюваний носій складається з множини термінальних і нетермінальних елементів $M_G = \{T_G \bigcup N_G\}$. Терміналами є конструкції графів та їх складові, проміжне представлення програмних кодів у вигляді списків та допоміжних конструкцій:

$$T_G = \{G \bigcup V \bigcup E \bigcup list \bigcup stack\_temp \bigcup \quad (11)$$
$$\bigcup stack\_root \bigcup stack\_break \bigcup stack\_next$$
$$\bigcup stack\_continue \bigcup stack\_switch \bigcup$$
$$\bigcup temp \bigcup l\_vertex\},$$

де $G = \langle V, E \rangle$ – конструкція графа; $V = \{v_i\}$, $E = \{e_i\}$ – множина вершин і дуг відповідно; *list* – побудована конструкція C_T; допоміжні конструкції-стеки вершин графа з навантаженням: у вигляді назв керуючих операторів – *stack_temp*, які є коренем підграфа, що утворюється під час роботи з операторами $\{if, else, while, for, case\}$ – *stack_root*; "*break*" – *stack_break*; "*continue*" – *stack_continue*; які мають спільні дуги з вершиною, що буде додана – *stack_next*; "*case*" – *stack_switch*; *temp* зберігає вершину допоміжних стеків, *l_vertex* зберігає вершину *list*. Допоміжні конструкції стеків є впорядкованими наборами елементів, що формуються за принципом LIFO.

Вершина графа має атрибути $w_v \hookleftarrow v = \langle index, name \rangle$, де *index* – індекс вершини, набуває цілочислових значень, *name* – навантаження вершини. Атрибути дуги $w_e \hookleftarrow e = \langle start, finish \rangle$, де *start*, *finish* – цілочислові індекси інцендентних вершин.

Граф має атрибути $w_G = \langle begin, end, current_v, current_e \rangle$, де *begin* – початкова вершина графу, *end* – кінцева вершина графа, $current_v = \langle name, index, prev \rangle$ – поточна вершина під час формування графа, де *prev* – попередня вершина графа, $current_e = \langle start, finish \rangle$ – поточна дуга під час формування графа.

Атрибути вершини проміжного представлення програмного коду *l_vertex* – $w_l = \langle name, prev \rangle$, де *name* – навантаження вершини списку, *prev* – навантаження попередньої вершини списку.

Атрибути вершини допоміжного стеку *temp* – $w_{ST} = w_v$.

Розглянемо сигнатуру Σ_G:

$$\Sigma_G = \langle \Xi_G, \Theta_G, \Phi_G, \{\rightarrow\} \rangle \bigcup \Psi_G, \quad (11)$$

де $\Xi_G = \left\{ \tilde{\cup}, \cup \right\}$ – множина операцій зв'язування; $\Theta_G = \{\Rightarrow, \mid\Rightarrow, \|\tilde{\Rightarrow}\}$ – множина операцій виведення; $\Phi_G = \{\div, :=, \%, \#, +, -, —, \times\}$ – множина операцій над атрибутами; $\rightarrow$ – відношення підстановки.

Розглянемо операції над атрибутами:

doi: https://doi.org/10.15802/stp2021/

ISSN 2307–3489 (Print), ISSN 2307–6666 (Online)

Наука та прогрес транспорту. Вісник Дніпропетровського національного університету залізничного транспорту, 2021, № 4 (94)

– $\div(c,n,L)$ – виконання n операцій зі списку L, якщо $c = true$;

$a \coloneqq b$ – присвоєння, копіює значення операнду b в a;

– $\%(index,V)$ – знаходження навантаження вершини, індекс якої дорівняє $index$, у множині вершин V;

– $\#Q$ – обчислення потужності множини, визначає число, яке дорівнює кількості елементів у Q;

– $+(el,stack)$ – додавання елемента el до допоміжного стеку $stack$;

– $-(el,stack)$ – вилучення вершини допоміжного стеку $stack$ в el за умови, що $stack$ не порожній;

– $--(el,list)$ – вилучення вершини списку $list$ та збереженні її в el за умови, що $list$ не порожній;

– $\times(a,b)$ – множення двох чисел, передбачає знаходження третього числа, що є їхнім добутком.

Операція об'єднання графів $w_G \leftarrow\!\lrcorner G = \tilde{\cup}(w_1 \leftarrow\!\lrcorner G_1, w2 \leftarrow\!\lrcorner G_2)$ передбачає формування нового графа $w_G \leftarrow\!\lrcorner G$, що містить об'єднані множини вершин і дуг вихідних графів $w_G \leftarrow\!\lrcorner G = \langle V, E\rangle$, де $V = V_1 \cup V_2$, $E = E_1 \cup E_2$, $w_1 \leftarrow\!\lrcorner G_1 = \langle V_1, E_1\rangle$, $w_2 \leftarrow\!\lrcorner G_2 = \langle V_2, E_2\rangle$, при цьому $\cup$ – традиційна операція об'єднання.

Відношення підстановки має вигляд:

$$\psi_i = \langle s_i, g_i\rangle,\ s_i = \langle \bar{s}_i, \tilde{s}_i\rangle,\ g_i = \langle \bar{g}_i, \tilde{g}_i\rangle, \quad (12)$$

де $\bar{s}_i, \tilde{s}_i$ – відношення підстановки для роботи зі списком і побудови конструкції графа відповідно; $\bar{g}_i, \tilde{g}_i$ – операції над атрибутами списку та графа, його вершин і дуг відповідно. Якщо операцію над атрибутами не виконують, відношення підстановки має вигляд $\psi_i = \langle s_i, \varepsilon\rangle$.

Операція повного виведення $\|\tilde{\Rightarrow}(\Psi, w_l \leftarrow\!\lrcorner l)$ та більш детальна інформація щодо інших операцій наведена в роботах [13, 14]. Результатом операції виведення є конструкції-граф.

Метою конструювання є побудова графа керування програми за проміжним представленням у вигляді $list \in \Omega(C_T)$.

Обмеження конструктора C_G накладають навантаженням вершин у списку.

Початкова умова конструювання: σ – нетермінал, із якого починається побудова конструкції графа керування; ρ – нетермінал, із якого починається вилучення вершин із $list \leftarrow\!\lrcorner C_T$.

Умова завершення конструювання: форма не містить нетерміналів, конструкція списку порожня.

У результаті конкретизації конструктора $C_G\ {}_K\!\mapsto {}_K C_G$ маємо нижченаведені правила підстановки.

Правило ініціалізації графа має вигляд:

$$\bar{s}_1 = \varepsilon,\ \bar{g}_1 = \varepsilon; \quad (13)$$

$$\tilde{s}_1 = \langle \sigma \to G\alpha\rangle;$$

$$\tilde{g}_1 = \langle index \leftarrow\!\lrcorner v_1 \coloneqq 1, name \leftarrow\!\lrcorner v_1 \coloneqq "begin",$$

$$index \leftarrow\!\lrcorner v_2 \coloneqq 0, name \leftarrow\!\lrcorner v_2 \coloneqq "end", begin \leftarrow\!\lrcorner G \coloneqq v_1,$$

$$end \leftarrow\!\lrcorner G \coloneqq v_2, current \leftarrow\!\lrcorner G \coloneqq v_1, V = \{v_1, v_2\}\rangle.$$

Побудова графа передбачає вилучення вершини конструкції $list$ ($\bar{s}_2$), тому далі для правил $G\alpha$ та $G\beta$, якщо $\bar{s}_i$ або $\bar{g}_i$ не вказано, відповідно $\bar{s}_i = \bar{s}_2$, $\bar{g}_i = \bar{g}_2$, а для правил Gf, $G\beta$ та $G\sigma$, якщо $\bar{s}_i$ або $\bar{g}_i$ не вказано, $\bar{s}_i = \varepsilon$, $\bar{g}_i = \varepsilon$:

$$\bar{s}_2 = \langle \rho \to --(l\_vertex, list)\rangle; \quad (14)$$

$$\bar{g}_2 = \varepsilon.$$

Правило для додавання нових вершин до графа має вигляд:

$$\tilde{s}_2 = \left\langle G\alpha\ {}_{d_1}\!\to \tilde{\cup}(G, G^*)\alpha \right\rangle; \quad (15)$$

$$\tilde{g}_2 = \langle \div(name \leftarrow\!\lrcorner l\_vertex = P \mid el \in \{Cb_i\} \setminus$$

$$\setminus \{"do","else"\}, 9, d_1 \coloneqq true, G^* = V^* \cup E^*,$$

$$V^* = \{v\}, E^* = \{e\}, name \leftarrow\!\lrcorner v \coloneqq name \leftarrow\!\lrcorner l\_vertex,$$

doi: https://doi.org/10.15802/stp2021/

ISSN 2307–3489 (Print), ISSN 2307–6666 (Online)

Наука та прогрес транспорту. Вісник Дніпропетровського національного університету залізничного транспорту, 2021, № 4 (94)

$index \leftarrow\! \lrcorner v := \#G, start \leftarrow\! \lrcorner e := index \leftarrow\! \lrcorner current \leftarrow\! \lrcorner G,$

$finish \leftarrow\! \lrcorner e := index \leftarrow\! \lrcorner v, current \leftarrow\! \lrcorner G := v)\rangle.$

Далі наведемо правила для додавання вершин до конструкції $stack\_temp$:

– вершина "do":

$$\tilde{s}_3 = \langle G\alpha_{\ d_2} \to G\alpha\rangle; \quad (16)$$

$\tilde{g}_3 = \langle \div(name \leftarrow\! \lrcorner l\_vertex = "++" \& name \leftarrow\! \lrcorner prev \leftarrow\!$

$\leftarrow\! \lrcorner l\_vertex = "do", 2, d_2 := true,$

$+(-1\times(\#G+1), stack\_temp)\rangle.$

– вершина "$else$":

$$\tilde{s}_4 = \langle G\alpha_{\ d_3} \to G\alpha\rangle; \quad (17)$$

$\tilde{g}_4 = \langle \div(name \leftarrow\! \lrcorner l\_vertex = "++" \& name \leftarrow\! \lrcorner prev \leftarrow\!$

$\leftarrow\! \lrcorner l\_vertex = "else", 2, d_3 := true,$

$+(-1, stack\_temp)\rangle;$

– інші вершини:

$$\tilde{s}_5 = \langle G\alpha_{\ d_4} \to G\alpha\rangle; \quad (18)$$

$\tilde{g}_5 = \langle \div(name \leftarrow\! \lrcorner l\_vertex = "++" \& name \leftarrow\! \lrcorner prev \leftarrow\!$

$\leftarrow\! \lrcorner l\_vertex! = el \in \{"do", "else"\}, 2, d_4 := true,$

$+(index \leftarrow\! \lrcorner current \leftarrow\! \lrcorner G, stack\_temp)\rangle.$

Правила для обробки різних вершин конструкцій мають вигляд:

– "$return$" конструкції $list$:

$$\tilde{s}_6 = \left\langle G\alpha_{\ d_5} \to \tilde{\cup}(G, G^*)\alpha \right\rangle; \quad (19)$$

$\tilde{g}_6 = \langle \div(name \leftarrow\! \lrcorner l\_vertex = "return", 5, d_5 := true,$

$G^* = E^*, E^* = \{e\}, start \leftarrow\! \lrcorner e_i := index \leftarrow\! \lrcorner current \leftarrow\! \lrcorner G,$

$finish \leftarrow\! \lrcorner e_i := index \leftarrow\! \lrcorner end \leftarrow\! \lrcorner G\rangle;$

"$break$" конструкції $list$:

$$\tilde{s}_7 = \langle G\alpha_{\ d_6} \to G\alpha\rangle; \quad (20)$$

$\tilde{g}_7 = \langle \div(name \leftarrow\! \lrcorner l\_vertex = "break", 2, d_6 := true,$

$+(index \leftarrow\! \lrcorner current \leftarrow\! \lrcorner G, stack\_break)\rangle;$

"$continue$" конструкції $list$:

$$\tilde{s}_8 = \langle G\alpha_{\ d_7} \to G\alpha\rangle; \quad (21)$$

$\tilde{g}_8 = \langle \div(name \leftarrow\! \lrcorner l\_vertex = "continue", 2,$

$d_7 := true, +(index \leftarrow\! \lrcorner current \leftarrow\! \lrcorner G, stack\_continue)\rangle;$

"$--$" конструкції $list$:

$$\tilde{s}_9 = \langle G\alpha_{\ d_8} \to G\phi\alpha\rangle; \quad (22)$$

$\tilde{g}_9 = \langle \div(name \leftarrow\! \lrcorner l\_vertex = "--", 1, d_8 := true)\rangle;$

"$else$" конструкції $stack\_temp$:

$$\tilde{s}_{10} = \langle G\phi_{\ d_9} \to G\phi\rangle; \quad (23)$$

$\tilde{g}_{10} = \langle -(temp, stack\_temp), \div(temp = -1, 2,$

$d_8 := true, +(index \leftarrow\! \lrcorner current \leftarrow\! \lrcorner G, stack\_root))\rangle;$

"do" конструкції $stack\_temp$:

$$\tilde{s}_{11} = \left\langle G\phi_{\ d_{10}} \to \tilde{\cup}(G\beta, G^*)\chi\phi \right\rangle \quad (24)$$

$\tilde{g}_{11} = \langle -(temp, stack\_temp), \div(temp < -1, 5,$

$d_{10} := true, G^* = E^*, E^* = \{e\},$

$start \leftarrow\! \lrcorner e_i := index \leftarrow\! \lrcorner current \leftarrow\! \lrcorner G,$

$finish \leftarrow\! \lrcorner e_i := temp \times -1)\rangle.$

Правило для додавання нової вершини до графа має вигляду:

$$\tilde{s}_{12} = \left\langle G\beta \to \tilde{\cup}(G, G^*)\right\rangle; \quad (25)$$

$\tilde{g}_{12} = \langle G^* = V^* \cup E^*, V^* = \{v\}, E^* = \{e\},$

$name \leftarrow\! \lrcorner v := name \leftarrow\! \lrcorner l\_vertex, index \leftarrow\! \lrcorner v := \#G,$

doi: https://doi.org/10.15802/stp2021/

ISSN 2307–3489 (Print), ISSN 2307–6666 (Online)

Наука та прогрес транспорту. Вісник Дніпропетровського національного університету залізничного транспорту, 2021, № 4 (94)

$$start\_e := index\_current\_G,$$

$$finish\_e := index\_v, current\_G := v\rangle.$$

Правило для обробки вершин "*while*" та "*for*" конструкції *stack _temp* :

$$\tilde{s}_{13} = \left\langle G\phi_{\ d_{11}} \to \tilde{\cup}\left(G, G^*\right)\chi\phi \right\rangle; \qquad (26)$$

$$\tilde{g}_{13} = \langle -(temp, stack\_temp), \div(\%(temp, V) =$$

$$= "while" | "for", 6, d_{11} := true, G^* = E^*, E^* = \{e\},$$

$$start\_e_i := index\_current\_G,\ finish\_e_i := temp,$$

$$+(temp, stack\_root))\rangle.$$

Далі подано правила для обробки вершин конструкцій *stack _root* та *stack _continue* у правилі $\tilde{s}_{13}$:

$$\tilde{s}_{14} = \left\langle G\chi \to \tilde{\cup}\left(G, G^*\right)\chi \right\rangle; \qquad (27)$$

$$\tilde{g}_{14} = \langle G^* = E^*, E^* = \{e\}, -(temp, stack\_root),$$

$$start\_e := temp,\ finish\_e := finish\_current_e\_G\rangle;$$

$$\tilde{s}_{15} = \left\langle G\chi \to \tilde{\cup}\left(G, G^*\right)\chi \right\rangle; \qquad (28)$$

$$\tilde{g}_{15} = \langle G^* = E^*, E^* = \{e\},$$

$$-(temp, stack\_continue), start\_e := temp,$$

$$finish\_e := finish\_current_e\_G |$$

$$| start\_current_e\_G\rangle;$$

$$\tilde{s}_{16} = \langle \chi \to \varepsilon \rangle. \qquad (29)$$

Правила для обробки вершини "*if*" конструкції *stack _temp* :

$$\tilde{s}_{17} = \left\langle G\phi_{\ d_{12}} \to G\phi \right\rangle; \qquad (30)$$

$$\tilde{g}_{17} = \langle -(temp, stack\_temp), \div(\%(temp, V) = "if",$$

$$3, d_{12} := true, +(temp, stack\_next),$$

$$+(index\_current\_G, stack\_root))\rangle;$$

$$\tilde{s}_{18} = \left\langle G\phi_{\ d_{13}} \to G\phi \right\rangle; \qquad (31)$$

$$\tilde{g}_{18} = \langle -(temp, stack\_temp), \div(\%(temp, V) = "if",$$

$$3, d_{13} := true, +(temp, stack\_root),$$

$$+(index\_current\_G, stack\_root))\rangle.$$

Подамо правило для обробки вершини "*case*" та "*default*" конструкції *stack _temp* :

$$\tilde{s}_{19} = \left\langle G\phi_{\ d_{14}} \to G\phi \right\rangle; \qquad (32)$$

$$\tilde{g}_{19} = \langle -(temp, stack\_temp), \div(\%(temp, V) =$$

$$= "case" | "default", 2, d_{14} := true,$$

$$+(temp, stack\_switch))\rangle.$$

Правило для обробки вершини "*switch*" конструкції *stack _temp* має вигляд:

$$\tilde{s}_{19} = \left\langle G\phi_{\ d_{15}} \to G\delta\phi \right\rangle; \qquad (33)$$

$$\tilde{g}_{19} = \langle -(temp, stack\_temp), \div(\%(temp, V) =$$

$$= "switch", 1, d_{15} := true)\rangle.$$

Правила для обробки вершин конструкцій *stack _switch* :

$$\tilde{s}_{20} = \left\langle G\delta \to \tilde{\cup}\left(G, G^*\right)\delta \right\rangle; \qquad (34)$$

$$\tilde{g}_{20} = \langle G^* = E^*, E^* = \{e\}, -(temp, stack\_switch),$$

$$start\_e := start\_current_e\_G,\ finish\_e := temp\rangle;$$

$$\tilde{s}_{21} = \langle \delta \to \varepsilon \rangle. \qquad (35)$$

Правила для обробки вершини конструкції *stack _break* :

$$\tilde{s}_{22} = \langle G\phi \to G\phi \rangle; \qquad (36)$$

$$\tilde{g}_{22} = \langle -(temp, stack\_break),$$

doi: https://doi.org/10.15802/stp2021/

ISSN 2307–3489 (Print), ISSN 2307–6666 (Online)

Наука та прогрес транспорту. Вісник Дніпропетровського національного університету залізничного транспорту, 2021, № 4 (94)

$$+(temp, stack\_next)\rangle;$$

$$\tilde{s}_{23} = \langle \phi \to \varepsilon \rangle. \qquad (37)$$

Правило для обробки вершини конструкції $stack\_next$ має вигляд:

$$\tilde{s}_{24} = \left\langle G\alpha \to \tilde{\cup}\left(G, G^*\right)\alpha \right\rangle; \qquad (38)$$

$$\tilde{g}_{24} = \langle G^* = E^*, E^* = \{e\}, -(temp, stack\_next),$$

$$start \hookleftarrow e := temp,\ finish \hookleftarrow e := index \hookleftarrow current_v \hookleftarrow G \rangle.$$

Правило для обробки вершини конструкції $stack\_root$:

$$\tilde{s}_{25} = \left\langle G\alpha \to \tilde{\cup}\left(G, G^*\right)\alpha \right\rangle; \qquad (39)$$

$$\tilde{g}_{25} = \langle G^* = E^*, E^* = \{e\}, -(temp, stack\_root),$$

$$start \hookleftarrow e := temp,\ finish \hookleftarrow e := index \hookleftarrow current_v \hookleftarrow G \,|$$

$$|\, index \hookleftarrow end \hookleftarrow G \rangle.$$

Правила для додавання дуги до останньої вершини графа:

$$\tilde{s}_{26} = \left\langle G\alpha \to \tilde{\cup}\left(G, G^*\right)\alpha \right\rangle \qquad (40)$$

$$\tilde{g}_{26} = \langle G^* = E^*, E^* = \{e\},$$

$$start \hookleftarrow e := index \hookleftarrow current_v \hookleftarrow G,$$

$$finish \hookleftarrow e := index \hookleftarrow end \hookleftarrow G \rangle.$$

Наступне правило дозволяє завершити обробку вершин конструкції $stack$:

$$\tilde{s}_{27} = \langle \alpha \to \varepsilon \rangle. \qquad (41)$$

У ході інтерпретації проведено зв'язування операцій сигнатури Σ_G з алгоритмами їх виконання:

$$\langle C_G = \langle M_G, \Sigma_G, \Lambda_G \rangle, C_{A'} = \langle M_{A'}, \Sigma_{A'}, \Lambda_{A'} \rangle \rangle_I \mapsto$$

$$_I \mapsto _{I,C_{A'}} C_G = \langle M_G, \Sigma_G, \Lambda_3 \rangle, \quad (42),$$

де $M_{A'} \supset V_{A'}, V_{A'} = \left\{ A_i^0 \,|_{X_i}^{Y_i} \right\}$ – множина базових алгоритмів; X_i, Y_i – множини визначення та значень алгоритму $A_i^0 \,|_{X_i}^{Y_i}$;

$\Lambda_{A'} = \left\{ M_{A'} = \bigcup\limits_{A_i^0 \in V_A} \left(X\left(A_i^0\right) \subset Y\left(A_i^0\right) \right) \cup \Omega\left(C_G\right) \right\}$ – неоднорідний носій; $\Omega\left(C_G\right)$ – множина конструкцій графів, які задовольняють C_G; $\Lambda_3 = \Lambda_G \cup \Lambda_{A'} \cup \Lambda_4$.

Конструктор $_{I,C_A} C_G$ містить алгоритми виконання операцій: $\Lambda_4 = \Big\{ \Big(A_1^0 \,|_{A_i, A_j}^{A_i \cdot A_j} \hookleftarrow \cdot \Big),$ $\left(A_2^0 \,|\, {}_S^{A_i} \hookleftarrow : \right),$ $\left(A_3 \,|_{l_h, l_q, f_i}^{f_i} \hookleftarrow \Rightarrow \right),$ $\left(A_4 \,|_{f_i, \Psi}^{f_j} \hookleftarrow | \Rightarrow \right),$ $\left(A_5 \,|_{\sigma, \Psi}^{\bar{\Omega}} \hookleftarrow \| \Rightarrow \right),$ $\left(A_6 \,|_{c,n,L}^{L} \hookleftarrow \div \right),$ $\left(A_7 \,|_{a,b}^{a} \hookleftarrow := \right),$ $\left(A_8 \,|_{index,V}^{name} \hookleftarrow \% \right),$ $\left(A_9 \,|_{Q}^{x} \hookleftarrow \# \right),$ $\left(A_{10} \,|_{el,stack}^{stack} \hookleftarrow + \right),$ $\left(A_{11} \,|_{el,stack}^{el} \hookleftarrow - \right),$ $\left(A_{12} \,|_{el,list}^{el} \hookleftarrow -- \right),$ $\left(A_{13} \,|_{a,b}^{c} \hookleftarrow \times \right),$ $\left(A_{14} \,|_{Q_1, Q_2}^{Q} \hookleftarrow \cup \right),$ $\left(A_{15} \,|_{G_1, G_2}^{G} \hookleftarrow \tilde{\cup} \right) \Big\}$.

Результатом реалізації конструктора (42) є множина конструкцій графів керування, побудованими за конструкціями з $\Omega\left(C_T\right)$.

Таким чином, правила (13) – (41) дозволяють перетворити текст програми, попередньо побудований за правилами (4) – (8), у граф керування. При цьому враховані оператори керування $\{C_i\} \cup \{Cb_i\}$. Інші оператори мають уніфіковане представлення у вигляді вершини-процесу.

Приклад використання моделі. Розглянемо приклад побудови графа керування для функції знаходження мінімуму з двох чисел, написаної мовою програмування С++. Програмний код функції:

```
int min(int a, int b)
{
  int _min = a;
  if(_min>b) _min = b;
  return _min;
}.
```

Наведений код було перетворено в *list*-конструкцію, побудовану за допомогою C_T (рис. 1, *а*). Head – голова списку, принцип робо-

doi: https://doi.org/10.15802/stp2021/

ISSN 2307–3489 (Print), ISSN 2307–6666 (Online)

Наука та прогрес транспорту. Вісник Дніпропетровського національного університету залізничного транспорту, 2021, № 4 (94)

ти списку – FIFO. Граф потоку керування (рис. 1, *б*) будуємо за допомогою послідовного виконання правил:

$$\sigma_1 \to G\alpha_2 \to \tilde{\cup}\left(G, G^*\right)\alpha_2 \to, \qquad (44)$$

$$_2 \to \tilde{\cup}\left(G, G^*\right)\alpha_5 \to G\alpha_2 \to \tilde{\cup}\left(G, G^*\right)\alpha_9 \to$$

$$_9 \to G\phi\alpha_{18} \to G\phi_{23} \to G\alpha_{25} \to$$

$$_{25} \to \tilde{\cup}\left(G, G^*\right)_{25} \to \tilde{\cup}\left(G, G^*\right)_{27} \to$$

$$_{27} \to \varepsilon,$$

де числові індекси стрілки є i-індексами правила $\tilde{S}_i$.

На рис. 1, *б* вершини графа містять позначки вершини списку, яким відповідають, а також номери вершини, що зазначено в дужках.

Об'єктно-орієнтована модель графа керування програми. Для зіставлення алгоритмів, представлених у вигляді блок-схем, та текстів програм подамо їх як графи керування. Граф керування будуємо за блок-схемою алгоритму, створеною в онлайн-редакторі [12]. Цей редактор дозволяє отримати схему у форматі json, розбір якого відбувається в такому порядку:

1. побудова проміжної моделі блок-схеми, що містить два класи, які відповідають за зчитування даних із файлу .json;

2. побудова графа керування на основі моделі блок-схеми, побудованої на попередньому кроці.

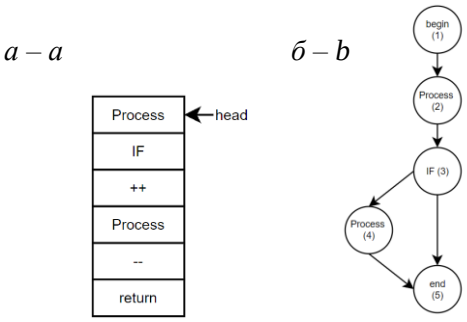


Рис. 1. Схема конструкцій, побудованих C_T і C_G: *а* – список керуючих операторів; *б* – граф керування

Fig. 1. Scheme of structures constructed by C_T and C_G: *a* – list of control operators; *b* – control graph

Граф керування програми будуємо за моделлю, описаною в (2) – (41).

Для програмної реалізації моделі графа керування програми виконаємо її об'єктно-орієнтоване (ОО) моделювання з використанням UML. Реалізацію моделі покладено на класи: PreprocessingCode, LexicalAnalysis, Graph (рис. 2).

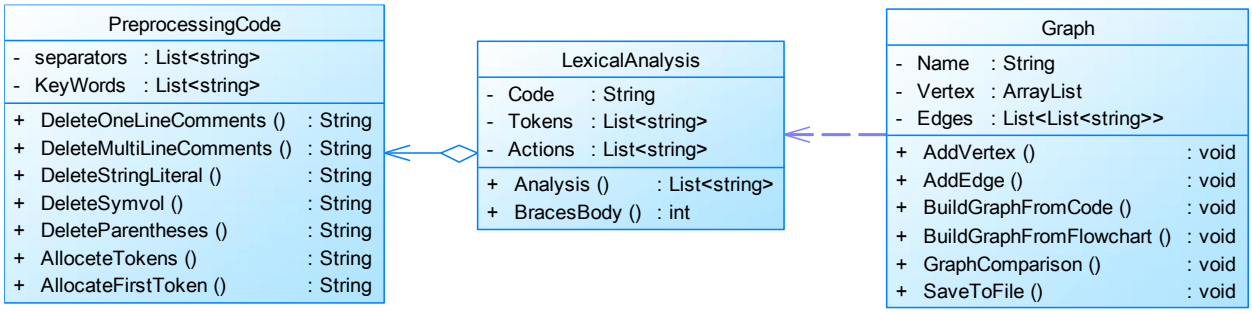


Рис. 2. Класове представлення конструкторів списку та графа керування

Fig. 2. Class representation of list and control graph constructors

Розглянемо відповідність алгоритмів конструктора стеку (9) методам класів (рис. 2):

A_1^0 – композиція алгоритмів – присутній у всіх методах, оскільки кожен із них складається з декількох підалгоритмів, таких як присвоєння, порівняння, об'єднання та інші;

A_2^0 – умовне виконання алгоритму – присутній у всіх методах, оскільки кожен із них містить оператор умовного виконання, що визначає використання тих чи інших алгоритмів;

A_3 – додавання вершини до списку – реалізовано методом LexicalAnalysis::BracesBody();

doi: https://doi.org/10.15802/stp2021/

ISSN 2307–3489 (Print), ISSN 2307–6666 (Online)

Наука та прогрес транспорту. Вісник Дніпропетровського національного університету залізничного транспорту, 2021, № 4 (94)

$A_4 - A_6$ – реалізовано методом LexicalAnalysis::BracesBody(), вони дозволяють послідовно заповнити стек.

Розглянемо відповідність алгоритмів конструктора графа (42) методам класів (рис. 1):

A_1^0, A_2^0 – відповідають алгоритмам, описаним у конструкторі списку;

$A_3 - A_5$ – реалізовано методом Graph::BuildGraphFromCode(), вони дозволяють послідовно побудувати граф;

A_6 – умовне виконання списку операцій – присутній у всіх методах;

A_7 – присвоєння операндів – присутній у всіх методах;

A_8 – пошук вершини графа за індексом – реалізовано у методі Graph::BuildGraphFromCode();

A_9 – обчислення потужності множини – реалізовано у вигляді параметрів атрибутів класів;

A_{10} – додавання елемента до допоміжного стеку – реалізовано у методі Graph::BuildGraphFromCode();

A_{11} – вилучення вершини допоміжного стеку – реалізовано у методі Graph::BuildGraphFromCode();

A_{12} – вилучення вершини списку конструкції, побудованої конструктором C_T, – реалізовано у методі Graph::BuildGraphFromCode();

A_{13} – множення двох чисел – реалізовано у методі Graph::BuildGraphFromCode();

A_{14}, A_{15} – об'єднання множин та графів відповідно – реалізовано у методі Graph::BuildGraphFromCode().

Порівняння графів, що відповідають тексту й алгоритму програми, виконуємо на основі методу пошуку в ширину: алгоритм перераховує всі досяжні з S вершини в порядку зростання відстані від S [1]. Перегляд вершин є паралельним на обох графах, а вершини помічають, якщо вони збігаються в обох графах.

Далі схожість графів виражаємо в числовому еквіваленті за формулою:

$$Match(A,B) = \frac{|A'| + |B'| - 2}{|A| + |B| - 2} \times 100, \qquad (43)$$

де A, B – досліджувані графи; $|A|$, $|B|$ – кількість вершин у відповідних графах; $|A'|$, $|B'|$ – кількість помічених верши. За порівняння графів відповідає метод Graph::GraphComparison().

Для використання побудованих графових моделей та повної реалізації методу визначення відповідності тексту й алгоритму програми було розроблено та реалізовано ОО-модель додатку з GUI (рис. 3).

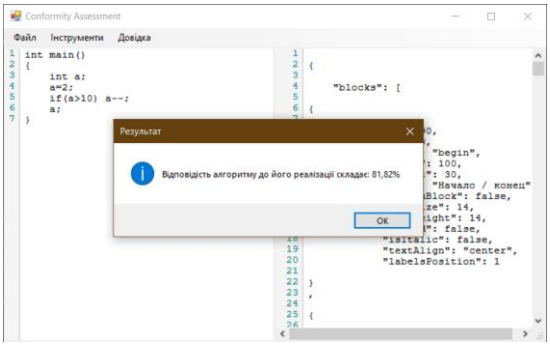


Рис. 3. Вікно додатку для зіставлення тексту й алгоритму програми

Fig. 3. Application window for comparing the text and algorithm of the program

Таким чином, метод для визначення відповідності програмного коду алгоритму, який він реалізує, передбачає використання розробленої системи конструкторів для перетворення програмного коду в граф керування. На її основі побудована об'єктно-орієнтована модель для конструювання графа керування програми та графа керування алгоритму.

Модель реалізовує побудову графів програми, алгоритму та їх порівняння, тобто повністю виконує кроки запропонованого методу.

Наукова новизна та практична значимість

У роботі отримали подальший розвиток методи конструктивно-продукційного моделювання в задачах обробки текстів, написаних штучними мовами. Запропоновано метод зіставлення алгоритму та його програмної реалізації. Метод передбачає використання розробленої системи конструкторів, що виконує перетворення тексту програм мовою С++ у граф керування для подальшого зіставлення з алгоритмом. Отримані ре-

doi: https://doi.org/10.15802/stp2021/

зультати мають значення для розв'язання таких задач, як зіставлення текстів програм із метою виявлення запозичень, визначення відповідності алгоритмів програм їх програмним реалізаціям із метою поліпшення навичок кодування. Графове представлення, яке продукує розроблена система конструкторів, може бути застосоване для дослідження впливу оптимізації та рефакторингу коду на складність програм із використанням метрик МакКейба.

Висновки

Запропоновано метод визначення відповідності тексту алгоритму програми. Використання апарату конструктивно-продукційного моделювання дозволило формалізувати процеси побудови графа керування програми, при цьому можливість розширення множини правил інформаційного забезпечення конструктора дає можливість в майбутньому врахувати класову структуру програми та модульність у цілому. Наявність поповнюваного носія в конструкторі дозволяє розширити модель для інших мов програмування.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Ивутин А. Н., Трошина А. Н. Метод формальной верификации параллельных программ с использованием сетей Петри. *Вестник Рязанского государственного радиотехнического университета*. 2019. № 70. С. 15–26. DOI: https://doi.org/10.21667/1995-4565-2019-70-15-26
2. Кондратьев Д. А., Марьясов И. В., Непомнящий В. А. Автоматизация верификации С-программ с использованием символического метода элиминации инвариантов циклов. *Моделирование и анализ информационных систем.* 2018. № 25 (5). С. 491–505. DOI: https://doi.org/10.18255/1818-1015-2018-5-491-505
3. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. *Алгоритмы. Построение и анализ.* Москва : Вильямс, 2005. 1296 с.
4. Малышев Е. В., Смелов В. В. Алгоритм распознавания плагиатов кодов программ. *Труды БГТУ.* 2018. № 1 (206). С. 135–138.
5. Никитин В. Д., Иванов А. П. Разработка методов оценки сходства алгоритмов на основе графовых моделей. *Magyar Tudományos Journal*. 2018. № 23. С. 36–41.
6. Сарвар С., Кайум З. Уль, Сафьян М., Икбал М., Махмуд Я. Выявление характерных особенностей программ для борьбы с компьютерным пиратством на основе интеллектуального анализа графов. *Труды Института системного программирования РАН*. 2019. № 31 (2). С. 171–186. DOI: https://doi.org/10.15514/ispras-2019-31(2)-12
7. Шинкаренко В. И., Куропятник Е. С. Проблемы выявления плагиата и анализ инструментального программного обеспечения для их решения. *Наука та прогрес транспорту*. 2017. № 1 (67). С. 131–142. DOI: https://doi.org/10.15802/stp2017/94034
8. Khaled F., H., Al-Tamimi M. S. Plagiarism Detection Methods and Tools : An Overview. *Iraqi Journal of Science*. 2021. № 62 (8). P. 2771–2783. DOI: https://doi.org/10.24996/ijs.2021.62.8.30
9. Kulkarni S., Govilkar S., Amin D. Analysis of Plagiarism Detection Tools and Methods. *SSRN Electronic Journal*. 2021. № 1. P. 1–7. DOI: https://doi.org/10.2139/ssrn.3869091
10. Kuropiatnyk O., Shynkarenko V. Text Borrowings Detection System for Natural Language Structured Digital Documents. *CEUR Workshop Proceedings. 2020. Vol. 2604 : 4th International Conference on Computational Linguistics and Intelligent Systems, COLINS 2020*, Lviv, Ukraine, 23–24 April 2020. P. 294–305.
11. Pandit A. A., Toksha G. Review of plagiarism detection technique in source code. *International Conference on Intelligent Computing and Smart Communication 2019*. 2020. P. 393–405. DOI: https://doi.org/10.1007/978-981-15-0633-8_38
12. *Programforyou : редактор блок-схем*. URL: https://programforyou.ru/block-diagram-redactor
13. Shynkarenko V. I., Ilman V. M. Constructive-Synthesizing Structures and Their Grammatical Interpretations. i. Generalized Formal Constructive-Synthesizing Structure. *Cybernetics and Systems Analysis*. 2014. Vol. 50. Iss. 5. P. 665–672. DOI: https://doi.org/10.1007/s10559-014-9655-z

doi: https://doi.org/10.15802/stp2021/

14. Shynkarenko V. I., Ilman V. M. Constructive-Synthesizing Structures and Their Grammatical Interpretations. II. Refining Transformations*. *Cybernetics and Systems Analysis*. 2014. Vol. 50. Iss. 6. P. 829–841. DOI: https://doi.org/10.1007/s10559-014-9674-9

O. S. KUROPIATNYK[*1], B. M. YAKOVENKO[*2]

[1*]Dep. «Computer Information Technology», Dnipro National University of Railway Transport named after Academician V. Lazaryan, Lazaryana St., 2, Dnipro, Ukraine, 49010, tel. +38 (056) 373 15 35, e-mail olena.kuropiatnyk@gmail.com, ORCID 0000-0003-2286-884X

[2*]Dep. «Computer Information Technology», Dnipro National University of Railway Transport named after Academician V. Lazaryan, Lazaryana St., 2, Dnipro, Ukraine, 49010, tel. +38 (056) 373 15 35, e-mail bohdanyakovenko98@gmail.com, ORCID 0000-0001-6174-0027

Identification of the program text and algorithm correspondence based on the control graph constructive-synthesizing model

Purpose.The main article purpose is to develop and implement the method for identifying the correspondence between the text and the program algorithm represented in the form of a flowchart. As part of the method work conversion of the input data in the graph representation is performed by means of constructive-synthesizing modelling. **Methodology**. To compare the program text and flowchart, we constructed a mathematical model for converting the program code into a graphical representation on the basis of control structures. To build the model, the apparatus of constructive-synthesizing modeling and its methods were used: specialization, concretization, interpretation and implementation. The graph representation of the text is created taking into account the control operators; the flowcharts are created using a json file containing the description of the diagram elements and their links. To compare the graphs we use the breadth-first search algorithm with the number of identical vertices being counted. To obtain the software implementation of the developed method and models we used the technology of object-oriented programming and CASE-technologies, which are based on the unified modeling language UML. **Findings** A method is proposed to present the text and the flowchart of the program in a uniform format of the directed graph (control graph) and to evaluate their correspondence by the number of identical vertices. For its formalization and automated usage, we developed constructive-synthesizing models of input data transformers. The program application was developed based on the models and the method. **Originality**. The methods of constructive-synthesizing modeling in the tasks of processing texts written in artificial languages were further developed. We developed the system of constructors, which transforms text program in C++ into a control graph. **Practical value**. The results are significant for solving such tasks as assembling program texts for borrowings detection, determining the correspondence of the program algorithms and their software implementations to improve coding skills. The graph representation produced by the developed system of constructors can be used for investigation of influence of optimization and code refactoring on the program complexity using McCabe's metrics.

Keywords: constructive-synthesizing modelling; constructor; graph representation of the text; program control graph; algorithm; algorithm correspondence

REFERENCES

1. Ivutin, A. N., & Troshina, A. G. (2019). Petri-net based method of parallel programs formal verification. *Vestnik of Ryazan State Radio Engineering University, 70*, 15-26. DOI: https://doi.org/10.21667/1995-4565-2019-70-15-26 (in Russian)
2. Kondratyev, D., Maryasov, I., & Nepomniaschy, V. (2018). The Automation of C Program Verification by Symbolic Method of Loop Invariants Elimination. *Modeling and Analysis of Information Systems, 25*(5), 491-505. DOI: https://doi.org/10.18255/1818-1015-2018-5-491-505 (in Russian)
3. Cormen, Th., Leiserson, Ch., Rivest, R., & Stein, C. (2005). *Introduction to Algorithms*. Moscow: Vilyams. (in Russian)
4. Malyshev, Ye. V., & Smelov, V. V. (2018). Plagiarism detecting algorithms for software code. *Proceedings of BSTU, 1*(206), 135-138. (in Russian)
5. Nikitin, V. D., & Ivanov, A. P. (2018). Development of Methods for Assessing the Similarity of Algorithms Based on Graph Models. *Magyar Tudományos Journal, 23*, 36-41. (in Russian)

doi: https://doi.org/10.15802/stp2021/

6. Sarwar, S., Qayyum, Z. Ul., Safyan, M., Iqbal, M., & Mahmood, Y. (2019). Graphs Resemblance based Software Birthmarks through Data Mining for Piracy Control. *Proceedings of the Institute for System Programming of the RAS, 31*(2), 171-186. DOI: https://doi.org/10.15514/ispras-2019-31(2)-12 (in Russian)
7. Shynkarenko, V. I., & Kuropiatnyk, O. S. (2017). Plagiarism detection problems and analysis software tools for its solve. *Science and Transport Progress, 1*(67), 131-142. DOI: https://doi.org/10.15802/stp2017/94034 (in Russian)
8. Khaled, F., & H. Al-Tamimi, M. S. (2021). Plagiarism Detection Methods and Tools: An Overview. *Iraqi Journal of Science, 62*(8), 2771-2783. DOI: https://doi.org/10.24996/ijs.2021.62.8.30 (in English)
9. Kulkarni, S., Govilkar, S., & Amin, D. (2021). Analysis of Plagiarism Detection Tools and Methods. *SSRN Electronic Journal*, *1*, 1-7. DOI: https://doi.org/10.2139/ssrn.3869091 (in English)
10. Kuropiatnyk O., Shynkarenko V. (2020) Text Borrowings Detection System for Natural Language Structured Digital Documents. *CEUR Workshop Proceedings. 2020. Vol. 2604 : 4th International Conference on Computational Linguistics and Intelligent Systems, COLINS 2020, Lviv, Ukraine, 23–24 April 2020*, 294–305.
11. Pandit, A. A., & Toksha, G. (2019). Review of Plagiarism Detection Technique in Source Code. In *Algorithms for Intelligent Systems.* (pp. 393-405). DOI: https://doi.org/10.1007/978-981-15-0633-8_38
12. *Programforyou: redaktor blok-skhem.* Retrieved from https://programforyou.ru/block-diagram-redactor (in Russian)
13. Shynkarenko, V. I., & Ilman, V. M. (2014). Constructive-Synthesizing Structures and Their Grammatical Interpretations. i. Generalized Formal Constructive-Synthesizing Structure. *Cybernetics and Systems Analysis, 50*(5), 655-662. DOI: https://doi.org/10.1007/s10559-014-9655-z (in English)
14. Shynkarenko, V. I., & Ilman, V. M. (2014). Constructive-Synthesizing Structures and Their Grammatical Interpretations. II. Refining Transformations*. *Cybernetics and Systems Analysis, 50*(6), 829-841. DOI: https://doi.org/10.1007/s10559-014-9674-9 (in English)

Надійшла до редколегії: 29.03.2021
Прийнята до друку: 30.07.2021

doi: https://doi.org/10.15802/stp2021/

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЗАТВЕРДЖУЮ

Перший проректор Дніпровського

національного університету

залізничного транспорту

імені академіка В. Лазаряна

Борис БОДНАР

СИСТЕМА ВИЗНАЧЕННЯ ВІДПОВІДНОСТІ ТЕКСТУ ПРОГРАМИ ГРАФІЧНОМУ ПРЕДСТАВЛЕННЮ АЛГОРИТМУ

Робочий проект

1116130.01222-01-ЛЗ

ЛИСТ ЗАТВЕРДЖЕННЯ

Завідувач кафедри КІТ

Вадим ГОРЯЧКІН

Керівник розробки

Олена КУРОП`ЯТНИК

Виконавець

Богдан ЯКОВЕНКО

Нормоконтролер

Олена КУРОП`ЯТНИК

2021