УДК 004.422.83

Федасюк Д. В.[1], Чопей Р.С.[2]

[1]Д-р техн. наук, професор, проректор Національного університету «Львівська політехніка», Львів, Україна
[2]Аспірант кафедри програмного забезпечення Національного університету «Львівська політехніка», Львів, Україна

АЛГОРИТМ ПОБУДОВИ ГРАФУ ПОТОКУ КЕРУВАННЯ ЗА ТЕКСТОМ ПРОГРАМИ МОВОЮ С

Актуальність. Робота присвячена проблемі автоматизованої побудови графу потоку керування за текстом програми мовою С, що є важливим етапом структурного тестування вбудованих систем.

Мета роботи – створення алгоритму побудови графу потоку керування за текстом програми, що має високу швидкість роботи та дозволяє подальше застосування існуючими засобами автоматизованого тестування.

Метод. Запропоновано алгоритм побудови графу потоку керування за текстом програми мовою С, який здійснює попередню обробку вхідного тексту програми шляхом видалення коментарів та порожніх рядків, визначає кількість вершин та ребер графу потоку керування на основі синтаксичного аналізу тексту програм, а також формує, заповнює та зберігає матрицю інцидентності в окремому текстовому файлі, що дозволяє подальше використання засобами тестування, які у якості вхідних даних приймають граф потоку керування, окрім того, текстовий файл може використовуватись засобами для графічного представлення графу потоку керування.

Результати. Розроблено програмний модуль, що реалізує запропонований алгоритм, який використано при проведенні експериментів з дослідження залежності часу побудови графу потоку керування від кількості рядків тексту програми.

Висновки. Проведені експерименти підтвердили працездатність запропонованого алгоритму та розробленого на його основі програмного модуля, довели придатність результатів роботи алгоритму, що дозволяє рекомендувати його для подальшого застосування засобами автоматизованого тестування для зменшення часових витрат на тестування вбудованого програмного забезпечення.

Ключові слова: граф потоку керування, автоматизоване тестування, Keil uVision, ARM, вбудовані системи**.**

НОМЕНКЛАТУРА

ПЗ – програмне забезпечення;

ARM – покращена RISC машина;

N – кількість відповідних ідентифікаторів case та default;

edgeC – лічильник вершин;

CSV – значення, розділені комою;

$I(|V|r|E|)$ – матриця інцидентності;

V – кількість рядків у матриці інцидентності, і, відповідно, вершин у графі керування;

E – кількість стовпців у матриці інцидентності, і, відповідно, ребер у графі керування;

M – мова програмування С;

$f_i(\cdot)$ – функція програми;

n – кількість функцій;

G – граф потоку керування;

O_{En} – обчислювальна складність n-го етапу роботи алгоритму.

$O_{\text{ВК}}$ – обчислювальна складність процедури видалення коментарів;

$O_{\text{МП}}$ – обчислювальна складність процедури модифікації тексту програми;

$O_{\text{ФЛ}}$ – обчислювальна складність процедури формування списків ознак;

$O_{\text{ЗаМ}}$ – обчислювальна складність процедури заповнення матриці;

$O_{\text{ЗбМ}}$ – обчислювальна складність процедури збереження матриці.

ВСТУП

Графи потоку керування (Control flow graph, CFG) є загальноприйнятим засобом візуального представлення множини всіх можливих шляхів виконання програми. Кожен оператор представлений вузлом в графі, ребра відображають потік управління між ними. Графи потоку керування застосовуються при аналізі потоку керування та оптимізації коду компілятором [1–3], для визначення та оцінки метрик програмного коду, зокрема складності програми [4–7], а також для структурного тестування програмного забезпечення (ПЗ), що використовує граф потоку керування для генерування тестових випадків [8–11], та визначення тестового покриття [12].

Об'єктом дослідження є процес побудови графу потоку керування.

Наявність графу потоку керування, сформованого у вигляді, зручному для опрацювання комп'ютером (зокрема, матрицею інцидентності), дозволить використати існуючі засоби [8–12] для обчислення метрик розроблюваного вбудованого ПЗ, а програмна реалізація алгоритму побудови графу потоку керування може бути оформлена у окремий модуль засобів автоматизованого тестування вбудованого ПЗ, що дозволить істотно зменшити витрати часу та зусиль команди тестувальників, а також уникнути помилок через неуважність, що зазвичай супроводжує монотонну рутинну роботу.

Огляд інтегрованих середовищ розроблення коду для мікроконтролерів ARM показав, що всі вони будують граф потоку керування автоматично, однак, жоден з них не дає можливості видобути граф потоку керування та зберегти його у вигляді, придатному для застосування в якості вхідних даних засобів автоматизованого тестування ПЗ.

Предмет дослідження становлять алгоритми побудови графу потоку керування за текстом програми.

Метою роботи є розроблення алгоритму автоматизованої побудови графу потоку керування за текстом програми, написаним мовою С, а також дослідження ефективності його роботи та можливості застосування його існуючими засобами тестування ПЗ.

© Федасюк Д. В., Чопей Р. С., 2018
DOI 10.15588/1607-3274-2018-2-17

1 ПОСТАНОВКА ЗАДАЧІ

Нехай задано текст програми мовою програмування С, представлений набором функцій у вигляді $T = < C, \widetilde{F} >$, де $\widetilde{F} = \{ f_i(\cdot), i = 1,2,...n \}$.

Тоді задача побудови графу потоку керування, представленого у вигляді метриці інцидентності $G = I(|V| \times |E|)$, полягатиме в синтаксичному аналізі тексту програми для визначення кількості вершин і ребер графу та взаємозв'язків між ними.

Для оцінювання якості одержаних результатів може бути використано порівняння графу потоку керування, побудованого за одержаною матрицею інцидентності, та графом потоку керування, вручну створеного фахівцем з забезпечення якості ПЗ згідно з існуючими правилами.

2 ОГЛЯД ЛІТЕРАТУРИ

У [13] запропоновано алгоритм перетворення тексту програми у граф-схему, що може застосовуватися для структурного проектування складних інформаційно-вимірювальних системи. Наведено приклад застосування алгоритму перетворення, реалізованого у вигляді окремого програмного модуля для тексту програми мовою Object Pascal.

У [14] розглянуто проблему побудови графу потоку керування класичними алгоритмами для коду планувальника операційної системи. Представлено алгоритм перетворення асемблерного коду планувальника у граф потоку керування, що не виключає гілки з затримками. Проведені експерименти для систем на базі архітектури TMS320C6200 від компанії Texas Instruments показали, що побудовані за представленим алгоритмом графи є більш достовірними у порівнянні з графами, побудованими за класичними алгоритмами. А відтак, оптимізація обсягу вихідного коду та часу виконання програми стає більш ефективною.

У [15] представлено архітектуру засобу для тестування програмного забезпечення вбудованих систем. Запропонований засіб використовує машинний код, отриманий у процесі компілювання ПЗ, написаного мовою Java, та формальну специфікацію. Окрім архітектури засобу, представлено алгоритм побудови графу потоку керування з машинного коду та алгоритм заміни виклику функції, представленого вершиною графу, його графом потоку керування.

Головним недоліком наявних алгоритмів побудови графу потоку керування [13–15] є відсутність можливості їх застосування для ПЗ, написаного мовою С, яка на сьогодні утримує лідерство з-поміж мов для розроблення вбудованого ПЗ.

3 МАТЕРІАЛИ ТА МЕТОДИ

Для формалізації алгоритму використаємо наступні поняття:

Обчислювальна операція – найменша автономна часина мови програмування. Для будь-якої мови програмування виділяють три різновиди обчислювальних операцій – слідування, галуження, та цикли:

– слідування – виклики функцій, та прості операції (інкремент/декремент та інші);

– галуження – оператори галуження для мови С (if, else if, else, switch);

– цикли – оператори циклів для мови С ("for", "while", "do").

Граф потоку керування – орієнтований граф, що представлений у формі матиці, та містить один вхідний та один вихідний вузол. Вершинами графу представлені символьні оператори чи функції, а ребрами – зв'язок між ними.

Матриця інцидентності (Incidence matrix) – одна з форм представлення графа, в якій вказується зв'язок між інцидентними елементами графа (ребро і вершина). Стовпці матриці відповідають ребрам, рядки – вершинам.

Заповнення матриці відбувається згідно з наступними правилами:

– комірка матриці набуває значення «1», якщо ребро виходить з вершини;

– комірка матриці набуває значення «–1», якщо ребро входить у вершину;

– комірка матриці набуває значення «0», якщо ребро не пов'язане з вершиною.

Алгоритм побудови графу потоку керування базується на об'єднанні декількох складових: попередня обробка тексту програми, визначення кількості вершин графу, визначення кількості ребер графу, заповнення матриці на основі визначених взаємозв'язків між вершинами графу. Процесу побудови графу потоку керування зображено на рис. 1.

Побудова графу потоку керування передбачає чотири етапи роботи:

Етап 1. Попередня обробка тексту програми

На цьому етапі роботи алгоритму з тексту програми видаляються коментарі, порожні рядки, а також проводиться модифікація тексту програми, якщо виклик функції або умову галуження записано у декілька рядків.

Етап 2. Визначення кількості вершин графу

На цьому етапі роботи алгоритму виконується автоматична нумерація рядків тексту програми за винятком наступних випадків:

– рядок тексту програми, що містить лише спеціальний символ "{" або "}", не збільшує лічильник рядків;

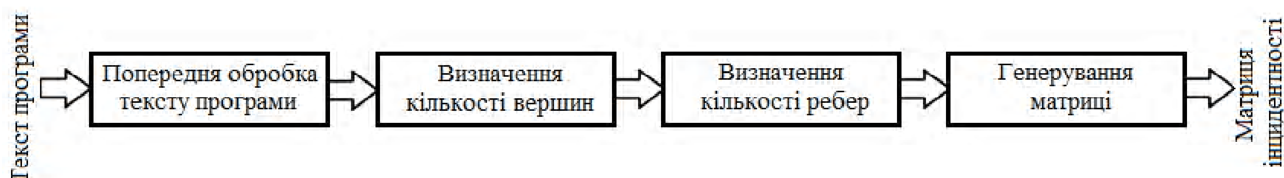

Рисунок 1 – Схема процесу побудови графу потоку керування за текстом програми, мовою С

– рядок тексту програми, що містить ідентифікатор: case, default, break, return, else, else if, не збільшує лічильник рядків;

– рядок тексту програми, що містить ініціалізацію змінної чи екземпляра структури, не збільшує лічильник рядків;

– лічильник рядків програмного коду збільшується на одиницю після останнього рядка тексту програми;

Етап 3. Визначення кількості ребер графу

На цьому етапі роботи алгоритму виконується нумерація ребер, шляхом збільшення лічильника ребер, на основі синтаксичного аналізу тексту програми, а саме пошуку у ньому ідентифікаторів галуження (if, switch, else, else if), ідентифікаторів циклу (for, do, while) та ідентифікаторів слідування (рис. 2).

Визначення кількості ребер відбувається згідно з правилами:

– лічильник ребер збільшується на одиницю, якщо рядок тексту програми відповідає обчислювальному процесу – слідування;

– лічильник ребер збільшується на одиницю, якщо рядок тексту програми містить ідентифікатори if, для яких є відповідний ідентифікатор else;

– лічильник ребер збільшується на одиницю, якщо рядок тексту програми містить ідентифікатор else або else if;

– лічильник ребер збільшується на два, якщо рядок тексту програми містить ідентифікатори if, для яких відсутні відповідні ідентифікатори else та else if;

– лічильник ребер збільшується на два, якщо рядок тексту програми містить ідентифікатор циклу – for або while;

– лічильник ребер збільшується на *N*, якщо рядок тексту програми містить ідентифікатор галуження – switch.

Етап 4. Формування матриці інцидентності

На цьому етапі роботи алгоритму створюється матриця з розмірами, що були визначені на попередніх етапах, та виконується її заповнення. Цей етап роботи є найбільш трудомістким, адже передбачає визначення взаємозв’язків у програмному коді. Для цього створюється чотири списки:

List1 – містить номер вершини, якій відповідає рядок тексту програми.

List2 – містить номер вершини, яка є наступною після ідентифікатора галуження або ідентифікатора циклу.

List3 – містить маску, що відповідає типу рядка, та може набувати таких значень:

– functionName – рядок, що містить ім’я функції;

– returnLine – рядок, що містить ідентифікатор return;

– followLine – рядок, що містить виклик іншої функції або присвоєння значення у змінну;

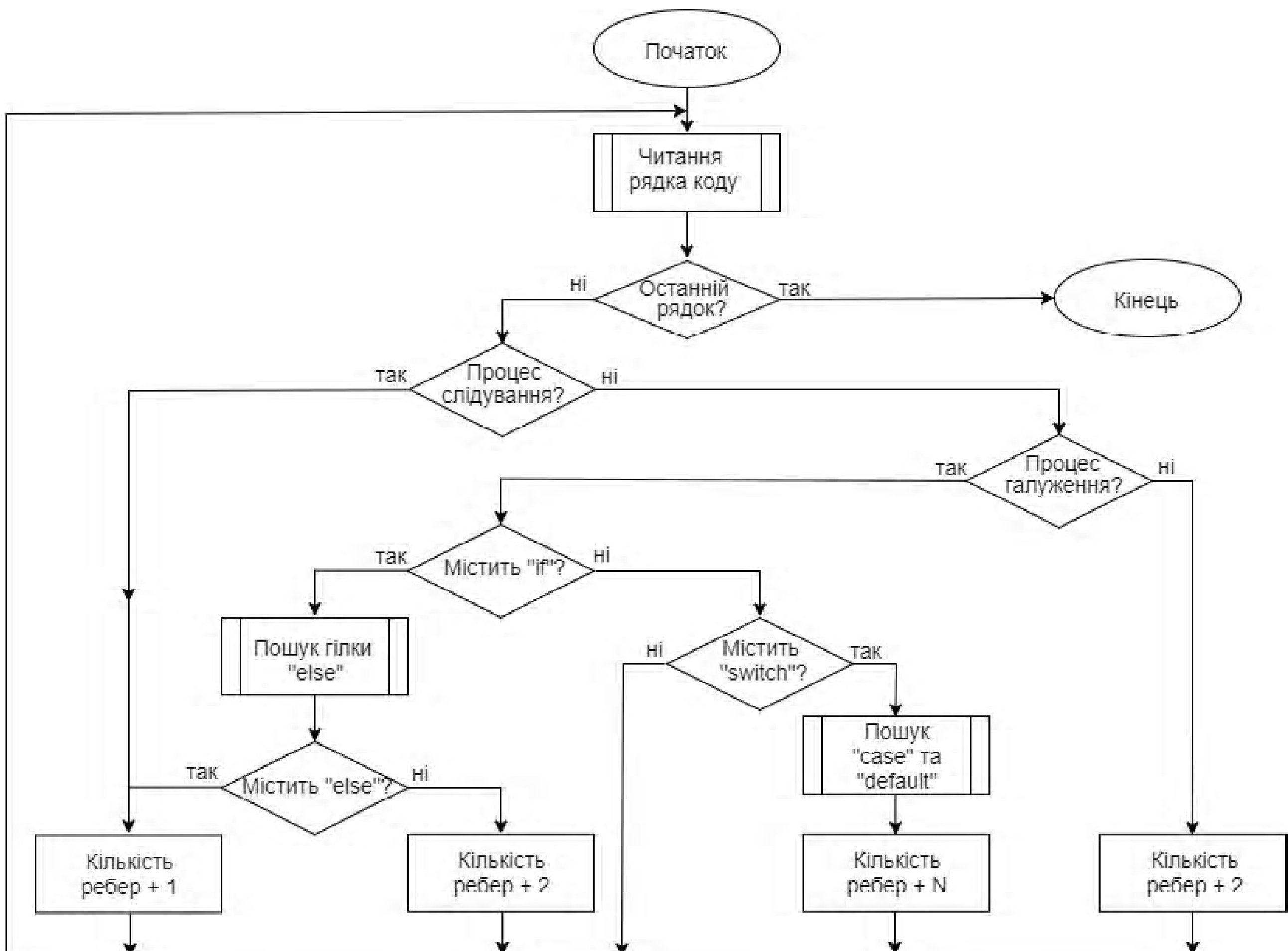

Рисунок 2 – Блок-схема етапу визначення кількості ребер у графі

– lastLineInsideIF – останній рядок вихідного коду під конструкцією if;

– lastLineInsideCycle – останній рядок вихідного коду під конструкцією for або while;

– lastLineInsideCase – останній рядок вихідного коду під конструкцією case;

List4 – містить номер вершини ідентифікатора if, switch, for або while, під якою знаходиться рядок тексту програми.

Наступним кроком після створення списків є заповнення цих списків та створення змінної – *edgeC*, що буде використано при оновленні інформації у матриці. Оновлення матриці відбувається згідно з правилами:

– якщо елемент *List1* містить 0 і елемент *List3* містить маску functionName, встановлюємо у комірці [0,0] значення «1», що відповідає виходу ребра з вершини 0.

– якщо елемент *List3* містить маску followLine, встановлюємо значення «-1» у комірці [*List1[n], edgeC*], збільшуємо лічильник вершин на одиницю та встановлюємо значення «1» у комірці [*List1[n], edgeC*];

– якщо елемент *List3* містить маску lastLineInsideIF встановлюємо значення «–1», у комірці [*List1[n], edgeC*], збільшуємо лічильник вершин на 1 та встановлюємо значення «1», у комірці [*List1[n], edgeC*]. У комірці [*List2[n], edgeC*] встановлюємо значення «–1» та встановлюємо значення «1», у комірці [*List4[n], edgeC*];

– якщо елемент *List3* містить маску lastLineInsideCycle встановлюємо значення «–1», у комірці [*List1[n], edgeC*], збільшуємо лічильник вершин на 1 та встановлюємо значення «1» у комірці [*List1[n], edgeC*], встановлюємо значення «–1» у комірці [*List4[n], edgeC*] і збільшуємо лічильник вершин на 1, встановлюємо значення «1» у комірці [*List4[n], edgeC*];

– якщо елемент *List3* містить маску lastLineInsideCase встановлюємо значення «–1» у комірці [*List1[n], edgeC*], збільшуємо лічильник вершин на 1 та встановлюємо значення «1» у комірці [*List1[n], edgeC*]. У комірці [*List2[n], edgeC*] встановлюємо значення «–1», а також встановлюємо значення «1» у комірці [*List4[n], edgeC*];

– якщо елемент *List3* містить маску returnLine встановлюємо значення «–1» у комірці [*List1[n], edgeC*];

Останнім кроком роботи цього алгоритму є збереження матриці у окремому текстовому файлі з форматом даних CSV.

4 ЕКСПЕРИМЕНТИ

Розглянемо застосування розробленого алгоритму для функції читання значення температури, що містить основні ідентифікатори мови С (дод. А).

У відповідності до представленого алгоритму виконуємо процедуру попереднього опрацювання тексту програми, а також визначаємо кількість вершин та ребер. Текст програми після процедури попереднього опрацювання та визначення кількості вершин наведений у дод. Б.

На основі аналізу тексту програми виявлено:

– 14 рядків, що відповідають обчислювальному процесу слідування – 14 ребер;

– 2 рядки, що містять ідентифікатор циклу for та while, для кожного з яких необхідно 2 ребра – 4 ребра для обох;

– 3 рядки, що містять ідентифікатор галуження if, else if та else, – 3 ребра;

– 1 рядок, що містить ідентифікатор галуження switch, для якого є два ідентифікатори case та один ідентифікатор default – 3 ребра;

– загальна кількість ребер – 24.

– загальна кількість вершин – 19.

Створюємо матрицю інцидентності $I(|19|\times|24|)$ та чотири списки для 19 вершин, які будуть заповнені згідно з наведеними вище правилами. Результати аналізу взаємозв'язків у тексті програми наведені у табл. 1.

Таблиця 1 – Списки ознак для функції ReadTemperature

List1	List2	List3	List4
0	–	functionName	–
1	6	followLine	–
2	6	followLine	1
3	6	followLine	1
4	6	followLine	1
5	6	lastLineInsideCycle	1
6	9	followLine	–
7	9	followLine	6
8	9	lastLineInsideCycle	6
9	–	followLine	–
10	14	followLine	–
11	14	lastLineInsideIF	10
12	14	lastLineInsideIF	10
13	14	lastLineInsideIF	10
14	18	followLine	–
15	18	lastLineInsideCase	14
16	18	lastLineInsideCase	14
17	18	lastLineInsideCase	14
18	18	returnLine	–

Матриця, заповнена в результаті роботи алгоритму, має вигляд:

$$I=\begin{bmatrix}
1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
-1&1&0&0&0&-1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&-1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&-1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&-1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&-1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&-1&1&0&-1&1&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&-1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&-1&1&0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&-1&1&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&-1&1&0&1&0&1&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&-1&1&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&-1&1&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&-1&1&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&-1&0&-1&0&-1&1&0&1&0&1&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&-1&1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&-1&1&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&-1&1\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&0&-1&0&-1&0&-1
\end{bmatrix} \quad (1)$$

Отримана матриця буде збереженою для подальшого використання. Також граф потоку керування, представлений у вигляді матриці інцидентності, можемо представити графічно з допомогою Інтернет-сервісу [16]. Побудований за матрицею (1) граф наведений на рис. 3.

5 РЕЗУЛЬТАТИ

Враховуючи модульну структуру розробленого алгоритму обчислювальна складність алгоритму є сумою складності для кожного етапу роботи. Перший етап передбачає видалення коментарів та модифікацію вихідного коду при необхідності, складність якого визначається згідно (2):

$$O_{E1} = O_{\text{ВК}}(1) + O_{\text{МП}}(N). \quad (2)$$

Другий етап роботи алгоритму передбачає визначення кількості вершин у графі. Обчислювальна складність цього етапу роботи залежить лише від кількості рядків тексту програми, що рівна $O_{E2}(N)$.

Третій етап роботи алгоритму передбачає визначення кількості ребер у графі. Складність цього етапу роботи збільшується за умови наявності у програмному коді операторів галуження “if” та “switch”. Тому вважаємо, що складність цього етапу $O_{E3}(N\hat{}2)$.

Етап генерування матриці є найбільш трудомістким. Складність цього етапу рівна (3):

$$O_{E4} = O_{\text{ФЛ}}(N\hat{}2) + O_{\text{ЗаМ}}(N) + O_{\text{ЗбМ}}(1). \quad (3)$$

Загальна складність алгоритму побудови графу потоку керування визначається:

$$O_{E1} + O_{E2} + O_{E3} + O_{E4} = O_{\text{ВК}}(1) + O_{МП}(N) + O_{E2}(N) + $$
$$+ O_{E3}(N\hat{}2) + O_{\text{ФЛ}}(N\hat{}2) + O_{\text{ЗаМ}}(N) + O_{\text{ЗбМ}}(1) = O(N\hat{}2). \quad (4)$$

Згідно з проведеним розрахунком складності алгоритму визначаємо час побудови графу потоку керування, для тексту програми обсягом 50, 100 та 120 тис. рядків. Розрахунок буде виконаний за умови, що комп’ютер виконує 1 млрд. операцій в секунду. Результати розрахунку наведені у табл. 2.

Таблиця 2 – Час побудови графу потоку керування для різного обсягу вхідних даних

Загальна к-ть рядків тексту програми, тис.	К-ть рядків коду після попередньої обробки, тис.	Час побудови, с
50	49	2,5
100	98	10
120	118	14,4

Отримані теоретичні результати розрахунку часу роботи алгоритму, свідчать про придатність результатів та доцільність реалізації алгоритму для проведення експериментальних досліджень.

Реалізація алгоритму здійснювалась у вигляді окремого програмного модуля, з використанням мови програмування C#. Окрім того, для проведення експериментів з великим обсягом вхідних даних реалізовано програмний модуль, який почергово передає текст функцій, з проектів розроблених у середовищі програмування Keil uVision. Експериментальні дослідження проведені на персональному комп’ютері Lenovo W520, з процесором Intel Core i5, що працює з частотою 2,2 ГГц та обсягом оперативної пам’яті 8 ГБайт. У якості вхідних даних використано три комерційні продукти, що містять 56, 90, та 150 тисяч рядків тексту програми. Результати експерименту наведені у табл. 3.

На основі отриманих теоретичних та експериментальних даних побудовано графік залежності часу побудови від кількості рядків тексту програми (рис. 4).

Як видно з рис. 4, теоретичні результати розрахунку та експериментальні дані дещо відрізняються, що зумовлено витратами часу на послідовне виконання алгоритму для кожної функції комерційного проекту.

Відзначимо, що для зменшення часу побудови графу потоку керування можливе паралельне виконання другого та третього етапів.

6 ОБГОВОРЕННЯ

Запропонований алгоритм побудови графу потоку керування порівняно з алгоритмом побудови на основі асемблерного коду [14] забезпечує більшу наочність перетворення, адже граф потоку керування будується напряму з тексту програми мовою С, в той час як граф потоку керування, побудований за асемблерним кодом, може дещо відрізнятись, що зумовлено процесом перетворення тексту програми у асемблерний код. Також запропонований алгоритм потребує більших часових витрат на побудову графу, ніж алгоритм побудови графу потоку керування за асемблерним кодом [14].

Таблиця 3 – Час побудови графів потоку керування для комерційних продуктів

Загальна к-ть рядків тексту програми, тис.	К-ть рядків коду після попередньої обробки	Час побудови, с
56	54 390	23
90	87 690	30
150	147 300	50,2

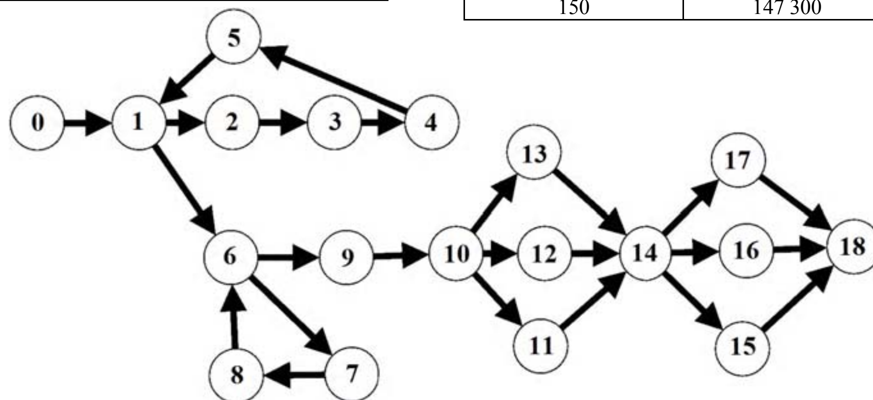

Рисунок 3 – Граф потоку керування функції “ReadTemperature”

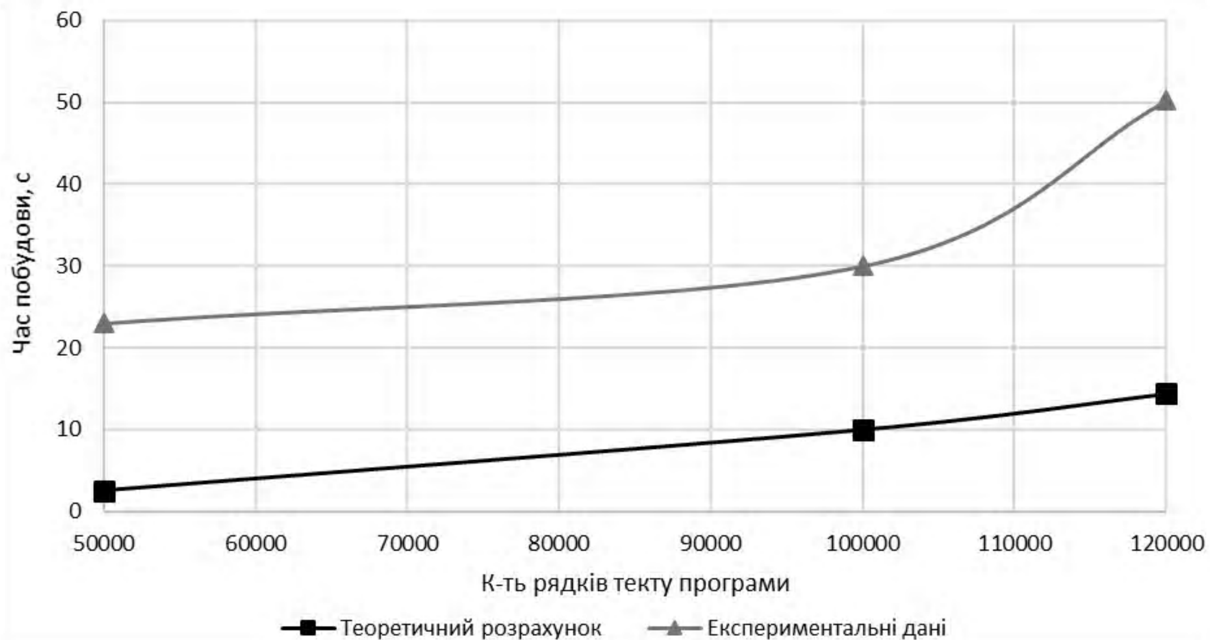

Рисунок 4 – Залежність часу побудови від кількості рядків тексту програми

Недоліком запропонованого алгоритму є його залежність від обчислювальних ресурсів, що збільшуються з кількістю рядків тексту програми.

Зменшення часу побудови графу потоку керування за рахунок паралельного виконання другого та третього етапу виконання, зростатиме з збільшенням кількості рядків тексту програми.

ВИСНОВКИ

У роботі вирішено актуальну задачу автоматизованої побудови графу потоку керування за текстом програми мовою С.

Наукова новизна роботи полягає в тому, що запропоновано алгоритм, який автоматично будує граф потоку керування з тексту програми мовою С, характеризується невисокою обчислювальною складністю, що, у свою чергу дозволяє знизити вимоги до обчислювальних ресурсів і пам'яті персонального комп'ютера, на якому він виконується.

Практична цінність отриманих результатів полягає в тому, що розроблено програмний модуль, який реалізує запропонований алгоритм та дозволяє виконувати побудову графу потоку керування, представленого у вигляді матриці інцидентності.

Перспективи подальших досліджень полягають у застосуванні запропонованого алгоритму до формування вхідних даних для алгоритму генерування тестових випадків, що інтегрований у засобі автоматизованого тестування найбільш тривалого часу виконання програми.

ПОДЯКИ

Роботу виконано в рамках госпдоговірної науково-дослідної роботи кафедри програмного забезпечення Національного університету «Львівська політехніка» на тему «Автоматизоване генерування тестових сценаріїв для забезпечення якості апаратного та мікропрограмного забезпечення, для Італійської компанії Dinamica Generale S.p.A., що є лідером в галузі розробки вбудованих систем для агропромисловості.

СПИСОК ЛІТЕРАТУРИ

1. Compilers: Principles, techniques, and tools / [eds.: A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman]. – Amsterdam: Addison Wesley, 2007. – 1038 p. ISBN: 0321486811
2. Graph-theoretic constructs for program control flow analysis : Research Report : RC-3923 / F. Allen, J. Cocke. – IBM Thomas J. Watson Research Center – IBM, 1972. – 130 p.
3. Ferrante J. The program dependence graph and its use in optimization / J. Ferrante, K. J. Ottenstein, J. D. Warren // ACM Transactions on Programming Languages and Systems. – 1987. – № 9. – P. 319–349. DOI: 10.1145/24039.24041
4. Gold R. On cyclomatic complexity and decision graphs / Robert Gold // Proceedings of the 10th International Conference of Numerical Analysis and Applied Mathematics (ICNAAM '12), 2007. – Vol. 1479. – P. 2170–2173. DOI: 10.1063/1.4756622
5. Henderson-Sellers B. The theoretical extension of two versions of cyclomatic complexity to multiple entry/exit modules / B. Henderson-Sellers, D. Tegarden // Software Quality Journal. – 1994. – Vol. 3. – P. 253–269. DOI: 10.1007/BF00403560
6. Jalote P. An integrated approach to software engineering / P. Jalote. – New York : Springer, 2005. – 566 p. ISBN: 978-0-387-28132-2
7. Sommerville I. Software Engineering / I. Sommerville. – Pearson : Addison Wesley, 2004. – 750 p. ISBN: 0321210263
8. Frankl P. G. Provable improvements on branch testing / P. G. Frankl, E. J. Weyuker. // IEEE Transactions on Software Engineering. – 1993. – Vol. 19. – P. 962–975. DOI: 10.1109/32.245738
9. Gold R. Control flow graphs and code coverage / R. Gold. // International Journal of Applied Mathematics & Computer Science. – 2010. – Vol. 20. – P. 739–749. DOI: 10.2478/v10006-010-0056-9
10. Zhu H. Software unit test coverage and adequacy / H. Zhu, P. A. Hall, J. H. May. // ACM Computing Surveys. – 1997. – Vol. 29. – P. 366–427. DOI: 10.1145/267580.267590
11. Fedasyuk D. Architecture of a tool for automated testing the worst-case execution time of real-time embedded systems' firmware / D. Fedasyuk, R. Chopey, B. Knysh. // 14th International conference, The experience of designing and application of cad systems in microelectronics. – 2017. – P. 278–282. DOI: 10.1109/CADSM.2017.7916134
12. Hossain M. I. Integration testing approach using usage patterns of global variables [Electronic resource] / M. I. Hossain, W. J. Lee, S. Youngsul. – Researchgate, 2012. Access mode: https://www.researchgate.net/publication/255173431_Integration_Testing_Approach_using_Usage_Patterns_of_Global_Variables.
13. Муха Ю. П. Алгоритм для преобразования программного кода на языке высокого уровня в граф-схему / Ю. П. Муха, В. А. Секачев // Известия волгоградского государственного технического университета. – 2009. – № 3. – С. 58–63.
14. Cooper K. D. Building a control-flow graph from scheduled assembly code [Electronic resource] / K. D. Cooper, T. J. Harvey, T. Waterman. – Researchgate, 2013. Access mode: https://www.researchgate.net/publication/2572750.

15. Amine A. Generating control flow graph from Java card byte code / A. Amine, B. Mohammed, L. Jean-Louis // Information Science and Technology (CIST). – 2014. – P. 206–212. DOI: 10.1109/CIST.2014.7016620
16. Graph Online [Electronic resource]: Creating graph from incidence matrix. Access mode: http://graphonline.ru/en/

Стаття надійшла до редакції 14.08.2017.
Після доробки 25.09.2017.

Федасюк Д. В.[1], Чопей Р. С.[2]

[1]Д-р. техн. наук, профессор, проректор Национального университета «Львовская политехника», Львов, Украина

[2]Аспирант кафедры программного обеспечения Национального университета «Львовская политехника», Львов, Украина

АЛГОРИТМ ПОСТРОЕНИЯ ГРАФА ПОТОКА УПРАВЛЕНИЯ ПО ТЕКСТУ ПРОГРАММЫ НА ЯЗЫКЕ С

Актуальность. Работа посвящена проблеме автоматизированного построения графа потока управления по тексту программы на языке С, что является важным этапом структурного тестирования встроенных систем.

Цель работы – создание алгоритма построения графа потока управления по тексту программы, что имеет высокую скорость работы и позволяет дальнейшее применение существующими средствами автоматизированного тестирования.

Метод. Предложен алгоритм построения графа потока управления по тексту программы на языке С, который осуществляет предварительную обработку исходного текста программы путем удаления комментариев и пустых строк, определяет количество вершин и ребер графа потока управления на основе синтаксического анализа текста программ, а также формирует, заполняет и сохраняет матрицу инцидентности в отдельном текстовом файле, что позволяет дальнейшее использование средствами тестирования, которые в качестве входных данных принимают граф потока управления, кроме того, текстовый файл может использоваться средствами для графического представления графа потока управления.

Результаты. Разработан программный модуль, реализующий предложенный алгоритм, который используется при проведении экспериментов по исследованию зависимости времени построения графа потока управления от количества строк текста программы.

Выводы. Проведенные эксперименты подтвердили работоспособность предложенного алгоритма и разработанного на его основе программного модуля и доказали пригодность результатов работы алгоритма, что позволяет рекомендовать его для дальнейшего применения средствами автоматизированного тестирования для уменьшения временных затрат на тестирование встроенного программного обеспечения.

Ключевые слова: граф потока управления, автоматизированное тестирование, Keil uVision, ARM, встраиваемые системы.

Fedasyuk D. V.[1], Chopey R. S.[2]

[1]Dr.Sc., Professor, Vice-Rector of Lviv Polytechnic National University, Lviv, Ukraine

[2]Post-graduate student of the Department of Software, Lviv Polytechnic National University, Lviv, Ukraine

THE ALGORITHM OF CONSTRUCTING CONTROL FLOW GRAPH BASED ON A PROGRAM WRITTEN IN C

Actuality. The work considers the problem of automated construction of the control flow graph using the text of a program written in C. The graph construction is an important step in the structural testing of embedded systems.

Objective. The work is aimed at creation of a fast algorithm for constructing the control flow graph from a source code. Such an automatically generated graph can be used by existing tools for automated testing.

Method. We propose an algorithm for constructing the control flow graph from the program written in C language, which preprocesses the source code of the program by removing comments and blank lines, determines the number of vertices and edges of the control flow graph by means of program text syntactic analysis, and then formats, fills and stores the incidence matrix in a separate text file. This file can be passed as input data for some existing automated testing tools. Moreover, the content of the text file can be visualized by existing tools for graphic representation of graphs.

Results. We've developed a software module that implements the proposed algorithm. The module can be used for experiments aimed at studying the dependence of the control flow graph's time construction on the amount of lines in a source code.

Conclusions. The conducted experiments confirmed the operability of the proposed algorithm and the software module developed on its basis. The results have proved the applicability of the algorithm, thus it can be recommended for further use together with automated testing tools to reduce the time required for testing of embedded software.

Keywords: control flow graph, automated testing, Keil uVision, ARM, embedded systems.

REFERENCES

1. Aho A. V., Lam M. S., Sethi R., Ullman J. D. : eds. Compilers: Principles, techniques, and tools. Amsterdam, Addison Wesley, 2007, 1038 p. ISBN: 0321486811
2. Allen F., Cocke J. Graph-theoretic constructs for program control flow analysis : Research Report : RC-3923, *IBM Thomas J. Watson Research Center – IBM,* 1972, 130 p.
3. Ferrante J., Ottenstein K. J., Warren J. D. The program dependence graph and its use in optimization, *ACM Transactions on Programming Languages and Systems*, 1987, No. 9, pp. 319–349. DOI: 10.1145/24039.24041
4. Gold R. On cyclomatic complexity and decision graphs, *Proceedings of the 10th International Conference of Numerical Analysis and Applied Mathematics (ICNAAM '12),* 2007, Vol. 1479, pp. 2170–2173. DOI: 10.1063/1.4756622
5. Henderson-Sellers B., Tegarden D. The theoretical extension of two versions of cyclomatic complexity to multiple entry/exit modules, *Software Quality Journal*, 1994, Vol. 3, pp. 253–269. DOI: 10.1007/BF00403560
6. Jalote P. An integrated approach to software engineering. New York, Springer, 2005, 566 p. ISBN: 978-0-387-28132-2
7. Sommerville I. Software Engineering. Pearson, Addison Wesley, 2004, 750 p. ISBN: 0321210263
8. Frankl P. G., Weyuker E. J. Provable improvements on branch testing, *IEEE Transactions on Software Engineering*, 1993, Vol. 19, pp. 962–975. DOI: 10.1109/32.245738
9. Gold R. Control flow graphs and code coverage, *International Journal of Applied Mathematics & Computer Science*, 2010, Vol. 20, pp. 739–749. DOI: 10.2478/v10006-010-0056-9
10. Zhu H., Hall P. A., May J. H. Software unit test coverage and adequacy, *ACM Computing Surveys*, 1997, Vol. 29, pp. 366–427. DOI: 10.1145/267580.267590
11. Fedasyuk D., Chopey R., Knysh B. Architecture of a tool for automated testing the worst-case execution time of real-time embedded systems' firmware, *14th International conference, The experience of designing and application of cad systems in microelectronics*, 2017, pp. 278–282. DOI: 10.1109/CADSM.2017.7916134
12. Hossain M. I., Lee W. J., Youngsul S. Integration testing approach using usage patterns of global variables [Electronic resource], *Researchgate, 2012. Access* mode: https://www.researchgate.net/publication/255173431_Integration_Testing_Approach_using_Usage_Patterns_of_Global_Variables.
13. Muha Ju. P., Sekachjov V. A. Algoritm dlja preobrazovanija programmnogo koda na jazyke vysokogo urovnja v graf-shemu, *Izvestija volgogradskogo gosudarstvennogo tehnicheskogo universiteta*, 2009, No. 3, pp. 58–63.
14. Cooper K. D., Harvey T. J., Waterman T. Building a control-flow graph from scheduled assembly code [Electronic resource], *Researchgate, 2013*. Access mode: https://www.researchgate.net/publication/2572750.
15. Amine A., Mohammed B., Jean-Louis L. Generating control flow graph from Java card byte code, *Information Science and Technology (CIST),* 2014, pp. 206–212. DOI: 10.1109/CIST.2014.7016620
16. Graph Online [Electronic resource]: Creating graph from incidence matrix. Access mode: http://graphonline.ru/en/

ДОДАТОК А

```
short ReadTemperature (void)
{
unsigned Counter, elems,TemperatureType;
float TemperatureBuffer[10];
float Tmp = 0.0;
float summ = 0.0;
float average = 0.0;
/* the TemperatureBuffer used
for average temperature value */
for (Counter = 0; Counter < 10; Counter++)
{
StartConversionOfInternalADC_1();
osDelay(10);
TemperatureBuffer[Counter] =
ReadTemperatureADC_1();
elems++;
}
/* calculate average temperature */
while(elems > 0)
{
summ+= TemperatureBuffer[elems];
elems—;
}
average = summ/10;
/* defina th temperature type */
if (average > 40)
{
TemperatureType = 1;
}
else if ((average < 40) &&
(average > 10))
{
TemperatureType = 2;
}
else
{
TemperatureType = 3;
}
switch(TemperatureType) /* show message on
display */
{
case 1:
{
printf("Temperature type - Hot, Temperature =
%f\n", average);
}
break;
case 2:
{
printf("Temperature type - Normal, Temperature
= %f\n", average);
}
break;
default:
{
printf("Temperature type - Cold, Temperature
= %f\n", average);
}
break;
}
return ((short)average);/*return temperature
value*/
}
```

ДОДАТОК Б

```
(0)  short ReadTemperature (void)
{
unsigned Counter, elems,TemperatureType;
float TemperatureBuffer[10];
float Tmp = 0.0;
float summ = 0.0;
float average = 0.0;
 (1)      for (Counter = 0; Counter < 10;
Counter++)
{
(2)
StartConversionOfInternalADC_1();
(3)osDelay(10);
(4)TemperatureBuffer[Counter] =
ReadTemperatureADC_1();
(5)elems++;
}
(6) while(elems > 0)
{
(7)summ+= TemperatureBuffer[elems];
(8)elems—;
}
(9)average = summ/10;
(10) if (average > 40)
{
(11)TemperatureType = 1;
}
else if ((average < 40) && (average > 10))
{
(12)TemperatureType = 2;
}
else
{
(13)TemperatureType = 3;
}
(14)switch(TemperatureType)
{
case 1:
{
(15) printf("Temperature type - Hot,
Temperature = %f\n", average);
}
break;
case 2:
{
(16)printf("Temperature type - Normal,
Temperature = %f\n", average);
}
break;
default:
{
(17) printf("Temperature type - Cold,
Temperature = %f\n", average);
}
break;
}
(18) return ((short)average);
}
```