Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки

БАКАЛАВРСЬКА КВАЛІФІКАЦІЙНА РОБОТА

на тему:

Програмне забезпечення для візуалізації графів
08-54.БКР.019.00.000 ПЗ

Виконав: студент 2 курсу, групи 1КІ-23мс
спеціальності 123 — «Комп'ютерна інженерія»
освітня програма – «Комп'ютерна інженерія»
Чорний О.С. _____
Керівник к.пед.н., доц. каф. ОТ
Добровольська Н.В. _____
Рецензент д.т.н., проф. каф. ПЗ
Ліщинська Л.Б. _____

Допущено до захисту
Завідувач кафедри ОТ
д.т.н., проф. Азаров О. Д.

"____" _____ 2025 р.

Вінниця ВНТУ– 2025 рік

Вінницький національний технічний університет
Факультет інформаційних технологій та комп'ютерної інженерії
Кафедра обчислювальної техніки
Освітньо—кваліфікаційний рівень бакалвр
Галузь знань — 12 Інформаційні технології
Спеціальність— 123 Комп'ютерна інженерія
Освітня програма— Комп'ютерна інженерія

ЗАТВЕРДЖУЮ
Завідувач кафедри ОТ проф., д.т.н.
_____Азаров О. Д.
"___" _____ 2025 року

З А В Д А Н Н Я

НА БАКАЛАВРСЬКУ КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ

Чорному Олексію Сергійовичу

1 Тема роботи: Програмне забезпечення для візуалізації графів, керівник роботи Добровольська Наталя Вікторівна, к.т.н., доцент кафедри ОТ, затверджено наказом №97 вищого навчального закладу від «_20_» __березня_ 2025 року.

2 Строк подання студентом роботи 13.05.2025

3 Вихідні дані до роботи: опис існуючих рішень засобів для візуалізації графів, опис бібліотеки Swing, опис можливостей та застосування бібліотеки JUNG, технічний опис програмного застосунку технології розробки.

4 Зміст розрахунково-пояснювальної записки: вступ, огляд існуючих рішень і обґрунтування теми, вибір інструментів для реалізації засобу візуалізації графів, розробка алгоритму та програми для візуалізації графів, висновки.

5 Перелік графічного матеріалу: фрагмент алгоритму формування ваг для кожного з ребер.

Перелік ілюстративного матеріалу: вікно введення вхідних даних, діагностичний граф з п'яти вершин.

6 Консультанти розділів роботи приведені в таблиці 1.

Таблиця 1 — Консультанти роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв
1-3	к.т.н., доц. каф. ОТ Добровольська Н.В.	21.03.2025р.	13.05.2025р.
Нормоконтроль	ас.каф.ОТ Швець С.І.	14.05.2025р.	30.05.2025р.

7 Дата видачі завдання 21.03.2025 р.

8 Календарний план приведений в таблиці2.

Таблиця 2— Календарний план

№ з/п	Назва етапів дипломної роботи	Строк виконання	Підпис
1	Постановка задачі роботи	21.03.25	Вик.
2	Огляд існуючих рішень	21.03.25— 29.03.25	Вик.
3	Обґрунтування теми,	21.03.25— 29.03.25	Вик.
4	Вибір інструментів для реалізації засобу візуалізації графів	30.03.25— 11.04.25	Вик.
5	Розробка алгоритму для реалізації засобу візуалізації графів	12.04.25— 26.04.25	Вик.
6	Розробка програмного забезпечення для реалізації засобу візуалізації графів	27.04.25— 07.05.25	Вик.
7	Оформлення пояснювальної записки та ілюстративного матеріалу	08.05.25	Вик.
8	Перевірка якості виконання бакалаврської роботи та усунення недоліків	14.05.25	Вик.

Студент Олексій ЧОРНИЙ

Керівник роботи к.т.н., доц. Наталія **ДОБРОВОЛЬСЬКА**

АНОТАЦІЯ

УДК 004.93

Чорний О.С. Програмне забезпечення для візуалізації графів.

На укр. мові. Бібліогр.: 19 назв; рис.: 19; аркушів: 69; табл: 3.

У бакалаврській кваліфікаційній роботі розроблено програмне забезпечення для візуалізації графів у багатопроцесорних системах з використанням сучасних інструментів програмування та бібліотек.

У процесі розробки було використано мову програмування Java, разом із бібліотеками Swing для створення графічного інтерфейсу та JUNG для побудови графів. Для розробки було обрано середовище Eclipse.

Ключові слова: програмне забезпечення, візуалізація, граф, багатопроцесорні системи, цикл, вершина, ребро, Java, JUNG, Swing.

ABSTRACT

Chornyi O.S. Software for Graph Visualization.

The explanatory note contains 69 pages, 19 figures, 3 tables and 19 references.

In the bachelor's qualification work, software for graph visualization in multiprocessor systems was developed using modern programming tools and libraries.

The Java programming language was used in the development process, along with Swing libraries for creating a graphical interface and JUNG for building graphs. The Eclipse environment was chosen for development.

Keywords: software, visualization, graph, multiprocessor systems, cycle, vertex, edge, Java, JUNG, Swing.

ЗМІСТ

ВСТУП

З розвитком інформаційних технологій та розширенням використання графів у науці та техніці, їх застосування стає надзвичайно важливим для вирішення багатьох практичних задач. Теорія графів займає центральне місце у математиці та інформатиці, будучи важливим інструментом для аналізу складних систем і структурування інформації. Безліч задач у цих галузях зводяться до роботи з графами, що вимагає наявності ефективних алгоритмів для їх побудови та візуалізації.

На сьогодні існує багато бібліотек для візуалізації графів, доступних для різних мов програмування. Проте більшість із цих рішень передбачають загальне представлення графів і не враховують специфічні властивості окремих типів графів. Тому виникає потреба в розробці спеціалізованих інструментів для візуалізації окремих класів графів, що забезпечить кращу інтерпретацію та аналіз даних.

Особливо важливою є розробка рішень для візуалізації графів у специфічних умовах, таких як багатопроцесорні системи. Потреба в цьому обумовлена необхідністю спрощення процесу тестування і аналізу багатопроцесорних систем, де графи є ефективним інструментом для представлення взаємодії процесів і ресурсів В зв'язку з цим ткма роботи є **актуальною**.

Метою дослідження є створення програмного забезпечення для візуалізації графів-циркулянтів, що дозволяє наочно демонструвати процеси тестування в умовах багатопроцесорних систем.

Завдання дослідження:

— дослідження сучасних підходів до візуалізації графів та їх використання в багатопроцесорних системах;

— аналіз наявних бібліотек та інструментів для побудови графів і їх візуалізації;

— створення алгоритмів для побудови та редагування графів, а також для збереження графічних зображень;

— розробка програмного забезпечення для візуалізації графів-циркулянтів.

— тестування та оцінка розробки.

Об’єктом дослідження є процеси побудови та візуалізації графів, зокрема циркулянтних графів у багатопроцесорних системах.

Предметом дослідження є методи та інструменти для візуалізації графів та їх застосування в багатопроцесорних системах.

Для досягнення поставленої мети **у** роботі використовуються такі **методи дослідження**, як теоретичний аналіз і синтез, програмування, а також методи комп'ютерного моделювання для розробки алгоритмів та програмного забезпечення для візуалізації графів.

Практичне значення. Розроблене програмне забезпечення для візуалізації графів-циркулянтів має значний потенціал для використання в різних галузях, де необхідне дослідження і тестування складних систем. Програмне забезпечення дозволяє ефективно візуалізувати графи та взаємодії між компонентами багатопроцесорних систем, що сприяє покращенню процесів тестування та оптимізації роботи таких систем.

1 ОГЛЯД ІСНУЮЧИХ РІШЕНЬ І ОБҐРУНТУВАННЯ ТЕМИ

Рішення задач візуалізації є важливою складовою комп'ютерної інженерії та математики, зокрема в контексті теорії графів і методів представлення даних. Теорія графів, яка є основою вивчення зв’язків між об’єктами та їх взаємодії, набуває все більшого застосування завдяки розвитку комп'ютерних технологій та Інтернету. Зараз ця теорія активно використовується в різноманітних галузях, таких як соціальні мережі, картографія, лінгвістика, біоінформатика, а також в комп’ютерній інженерії для розв'язання складних задач оптимізації та аналізу даних. Важливу роль у цих процесах відіграє візуалізація графів, яка допомагає відобразити зв’язки та структури у зручному для сприйняття вигляді.

Суть графічного представлення графа полягає в тому, що воно дозволяє зрозуміти структуру і взаємозв’язки між різними елементами. Для побудови графа достатньо знати основні характеристики: кількість вершин та їхні взаємозв'язки, які виражаються через ребра. Однак важливо розрізняти сам граф як структуру даних і його візуальне зображення, оскільки одному і тому ж графу може відповідати кілька різних способів відображення. Зображення графа, як правило, має на меті показати напрямки зв'язків і взаємозалежності між вершинами. У реальних умовах часто важко однозначно визначити, чи є два різні зображення одним і тим же графом, оскільки вибір способу візуалізації залежить від специфіки завдання та потреб користувача.

Вибір оптимального способу відображення графа часто виявляється критичним, оскільки різні типи зображень можуть надати різні рівні інтуїтивного розуміння зв'язків. У зв'язку з цим готові рішення не завжди задовольняють конкретні потреби користувача або задачу. Тому дослідження в цій сфері постійно рухаються вперед, намагаючись знайти універсальні рішення для конкретних галузей, таких як комп'ютерна інженерія, де необхідно моделювати складні мережі і оптимізувати їх структуру для досягнення високої ефективності.

Існуючі системи для візуалізації графів можна умовно поділити на два основних типи. Перший тип включає вузькоспеціалізовані системи, які орієнтовані на певні моделі графів, що мають конкретну семантику та топологію. Такі системи, як правило, є частинами більш великих проектів або досліджень, виконуючи роль візуалізаторів специфічних даних для вирішення конкретних завдань, зокрема в галузях комп'ютерної інженерії та обробки даних. Однак універсальність таких систем часто обмежена, і їхнє застосування в інших сферах може бути або неможливим, або значно ускладненим.

Другий тип — це універсальні системи для візуальної обробки графічних моделей, наприклад, такі програми, як daVinci, GraphEd, Graphlet, GraVis, VCG та бібліотеки LEDA, AGD, ffGraph, Graph Layout Toolkit, Graph Editor Toolkit. Хоча ці системи досягли значного прогресу у створенні інструментів для роботи з графами, у них є й кілька суттєвих недоліків. Одним із основних обмежень є їх орієнтація на операційну систему UNIX, що може створювати труднощі для користувачів, які звикли працювати в середовищі Windows. Більшість таких систем розроблені в іноземних університетах, де інфраструктура більше орієнтована на робочі станції UNIX. Тому для універсальних систем візуалізації існують певні обмеження у їхньому використанні та адаптації до конкретних умов, зокрема в умовах комп'ютерної інженерії.

Важливо зазначити, що розвиток технологій у галузі візуалізації графів тісно пов'язаний з інженерією програмного забезпечення та комп’ютерною інженерією, де необхідно постійно вдосконалювати алгоритми для обробки великих обсягів даних і досягати високої швидкодії. Високий рівень оптимізації алгоритмів та інтеграція з іншими системами дає змогу ефективно працювати з графами в умовах складних комп'ютерних мереж та інших високонавантажених інфраструктур.

Це дозволяє вирішувати завдання, які раніше були б надзвичайно складними або майже неможливими, сприяючи значному прогресу в таких

сферах, як аналіз соціальних мереж, наукові дослідження та розробка нових технологій у сфері комп’ютерної інженерії.

1.1 Засоби для візуалізації графів

1.1.1 Мови програмування

Розробка мов програмування, орієнтованих на обробку та маніпулювання графами, розпочалася ще з 70-х років минулого століття. Оскільки теорія графів є важливою математичною дисципліною, яка дає змогу моделювати складні структури взаємозв'язків між елементами (вершинами), такі мови були створені для ефективного вирішення практичних завдань у ряді галузей, зокрема в комп'ютерних науках, аналізі соціальних мереж, інформатиці та навіть у біоінформатиці. Кожна з таких мов призначалася для певної групи задач і часто була розвитком вже існуючих мов програмування того часу, націлених на реалізацію алгоритмів для обробки даних у вигляді графів.

На сьогоднішній день існує значна кількість мов програмування, які використовують принципи теорії графів для виконання різноманітних прикладних задач. Вони значною мірою доповнюють і вдосконалюють традиційні мови програмування, які були популярні в той час, і забезпечують зручніші та ефективніші способи вирішення задач, пов'язаних із структурою даних, що складаються з елементів, зв'язаних між собою певними відносинами. Мови, орієнтовані на графи, надають користувачам специфічні засоби для опису, обробки та візуалізації цих відносин.

У процесі аналізу існуючих мов програмування, що спираються на теорію графів, було виділено кілька таких мов і технологій, які визначили основні напрямки розвитку цієї галузі. Нижче наведено деякі з них.

Graph Extended Algol (GEA) — одна з перших мов програмування, яка була створена в 1970 році для вирішення задач, пов'язаних з обробкою графів. GEA мала кілька значних переваг, серед яких простота реалізації. Програмістам було достатньо мати базові навички в програмуванні, щоб працювати з цією мовою. Однак великим недоліком цієї мови було те, що вона не враховувала

різноманітність прикладних завдань і тому була обмежена в використанні. В результаті GEA не стала широко застосовуваною і є застарілою на сьогоднішній день. Проте її створення стало важливим етапом у розвитку мов програмування, орієнтованих на теорію графів.

DOT (Graph Description Language) — це мова, яка дозволяє описувати графи у вигляді текстового опису і при цьому задавати параметри їх відображення. Зокрема, можна змінювати такі атрибути, як колір, шрифт тексту, стиль стрілок, рамки та інші візуальні елементи графа. Крім того, DOT дозволяє додавати гіперпосилання до вузлів і ребер. Однак значним недоліком цієї мови є те, що вона не є самостійним інструментом для візуалізації. Графи, описані за допомогою DOT, потребують використання додаткових програм для їх візуалізації, що створює певні труднощі при роботі з цією мовою. Крім того, автоматична оптимізація розташування елементів графа може призводити до того, що елементи розташовуються не так, як це очікується, тому розробники часто змушені вручну коригувати їх положення.

GraphML (Graph Markup Language) — файловий формат, який використовується для опису графів, включаючи орієнтовані та неорієнтовані графи, а також складніші структури, такі як змішані і ієрархічні графи. GraphML дозволяє зберігати в графах різні атрибути та властивості, що дає змогу ефективно описувати складні структури даних. Система Visual Graph, створена на основі GraphML, підтримує роботу з ієрархічними графами, включаючи кластерні графи, та дозволяє наглядно відображати існуючі графові моделі. Проте основним недоліком цієї системи є відсутність можливості редагувати графи безпосередньо в системі, що обмежує її практичне застосування для користувачів, які потребують активної роботи з графами.

Trivial Graph Format (TGF) — цей формат використовується для опису базових структур графів, дозволяючи призначати вузлам і ребрам унікальні ідентифікатори. Однак TGF має кілька обмежень, серед яких відсутність підтримки для зміни зовнішнього вигляду елементів графа або визначення їх

позицій у просторі. Крім того, формат не підтримує вкладені структури графів, що може бути обмеженням для більш складних проектів.

Game Maker Language (GML) — інтерпретована мова програмування, яка була розроблена для використання разом із середовищем для створення комп'ютерних ігор Game Maker. GML також підтримує обробку графів і був використаний у ряді систем, орієнтованих на графові структури. Проте GML має певні обмеження щодо підтримки атрибутів та підключення інформації для аналізаторів, через що він є менш придатним для вирішення складних завдань, пов'язаних із обробкою графів у більш загальних і універсальних системах.

Graph eXchange Language (GXL) — мова, яка була створена для розширення мови TXL, орієнтованої на роботу з деревами. GXL зберігає багато особливостей TXL, таких як строгість типізації та можливість локального впливу правил на підграфи. Однак головним обмеженням цієї мови є складність реалізації простих трансформацій, що вимагає значних зусиль для розробки.

Directed Graph Markup Language (DGML) — формат для опису орієнтованих графів, який дозволяє візуалізувати залежності і відносини між елементами програмного коду. DGML використовується для представлення графів у форматі XML, що робить його зручним для використання в різних середовищах. Однак він має вузьку спеціалізацію, і його застосування обмежене лише для візуалізації відносин у програмному коді.

1.1.2 Бібліотеки

У даному контексті також слід розглянути різні бібліотеки та фреймворки, призначені для візуалізації графів, зокрема Graphviz, JGraph та JUNG [4]. Ці інструменти дозволяють ефективно працювати з графами, створювати їх візуальні представлення та здійснювати різноманітні операції з даними.

Graphviz — потужне програмне забезпечення з відкритим вихідним кодом, яке дозволяє створювати графи за допомогою текстового опису. Однією з головних переваг Graphviz є можливість експортувати створені графи в різні

формати, що дозволяє інтегрувати їх у веб-сторінки або документи. Програмне забезпечення надає велику кількість налаштувань для зміни кольору, шрифтів, стилів зв'язків та інших параметрів, що дозволяє налаштувати графи під конкретні вимоги.

JGraph — ще один потужний фреймворк для візуалізації графів, що працює з мовами програмування Java, C# і JavaScript. Спочатку розроблений як розширення для компонента JTree в Java, JGraph дозволяє редагувати графи і надає гнучкі можливості для налаштування компонентів бібліотеки. При цьому, оскільки семантика графів визначається зовнішніми додатками, JGraph дає можливість адаптувати його під різні вимоги проекту.

JUNG — це фреймворк для візуалізації графів, написаний на мові програмування Java. JUNG дозволяє реалізувати безліч алгоритмів для аналізу графів, включаючи кластеризацію, оптимізацію, статистичний аналіз, а також аналіз соціальних мереж. Він має вбудовані алгоритми для обробки різноманітних типів графів, а також механізми для анотації графів, що дозволяє здійснювати детальний аналіз даних.

Окрім згаданих фреймворків, варто також згадати менш відомий UCINET, який активно використовується для аналізу соціальних мереж. Однак він має обмежену інтеграцію і не може бути використаний для створення динамічних графів у додатках кінцевих користувачів. Це є значним недоліком для проектів, які вимагають інтерактивності та можливості змінювати графи на льоту.

Підсумовуючи проведений аналіз, було прийнято рішення використовувати JUNG як основний фреймворк для розробки, оскільки він надає всі необхідні можливості для інтеграції з Java-додатками та підтримує створення графічного інтерфейсу користувача за допомогою Swing, що дозволяє ефективно обробляти події та взаємодіяти з графами в реальному часі.

1.2 Проблеми та обмеження існуючих рішень

Візуалізація графів є потужним інструментом для аналізу складних взаємозв'язків у багатопроцесорних системах та інших обчислювальних мережах. Однак існуючі засоби для візуалізації графів мають ряд суттєвих проблем та обмежень, що можуть обмежувати їхню ефективність і застосовність у різних сферах.

Однією з основних проблем існуючих рішень є обмежена масштабованість. При роботі з великими графами, що включають велику кількість елементів (вузлів та дуг), багато програм для візуалізації можуть відчувати значне зниження продуктивності. Це може проявлятися у вигляді затримок при відображенні графічних елементів, що негативно впливає на зручність роботи користувача. Крім того, великі графи потребують великих обсягів пам'яті, що може призводити до швидкого витрачання ресурсів системи.

Масштабованість стає особливо важливою, коли необхідно працювати з графами, що містять тисячі або навіть мільйони елементів, наприклад, при тестуванні великих багатопроцесорних систем або мереж зв'язку. У таких випадках значення мають не тільки потужність комп'ютера, але й здатність програмного забезпечення ефективно обробляти і відображати великі масиви даних.

Існуючі інструменти візуалізації часто не є достатньо адаптивними до різних типів графів. Наприклад, для складних мереж з великою кількістю вузлів і зв'язків, де важлива не лише структура, а й динаміка процесів, стандартні методи відображення можуть бути недостатньо ефективними. Вони можуть не забезпечувати необхідного рівня деталізації або, навпаки, надмірно перевантажувати інтерфейс зайвими елементами.

Для деяких специфічних типів графів, таких як дерева, циклічні графи або графи з сильно нерівномірним розподілом зв'язків, існуючі засоби візуалізації можуть не підтримувати відповідні методи відображення, що ускладнює їх аналіз і роботу з ними.

Іншою значною проблемою є підтримка великої кількості елементів у графах. У багатьох випадках сучасні системи візуалізації не здатні коректно працювати з графами, що мають велику кількість вузлів і дуг. Це обмежує можливість використання таких рішень для складних обчислювальних задач або систем, де кількість елементів може сягати десятків тисяч чи мільйонів.

Зокрема, для графів з великою кількістю елементів можуть виникати проблеми з оптимізацією відображення, що призводить до значних затримок, погіршення якості графічного відображення та зниження зручності інтерфейсу для користувача. Крім того, не всі програми здатні ефективно працювати з графами, що динамічно змінюються, де елементи можуть додаватися або видалятися в реальному часі.

Існуючі програми для візуалізації графів часто мають складний або неінтуїтивно зрозумілий інтерфейс. Це може створювати труднощі для користувачів, особливо тих, хто не є експертами в галузі програмування або графічного дизайну. Для досягнення ефективної взаємодії користувача з програмним забезпеченням важливо забезпечити простоту і доступність інтерфейсу.

Також, в багатьох випадках, відсутність гнучких налаштувань для зміни параметрів візуалізації (наприклад, стилю відображення графа, кольору, товщини ліній) може обмежувати можливості користувача щодо адаптації програми під свої конкретні потреби.

Не менш важливою проблемою є інтеграція існуючих рішень для візуалізації графів з іншими програмними засобами та системами. Багато з доступних програм не мають достатньої сумісності з іншими популярними інструментами аналізу даних або програмами для моделювання обчислювальних систем, що ускладнює їх використання в комплексних наукових або промислових проектах.

Для багатьох сучасних додатків важливим є відображення не лише статичних графів, але й динамічних, де елементи змінюються, додаються або видаляються в реальному часі. Багато існуючих рішень не забезпечують

достатньої підтримки таких динамічних змін, що обмежує їх застосування в реальних обчислювальних системах, де відбувається постійне оновлення даних.

Всі ці проблеми свідчать про необхідність розробки нових, більш ефективних рішень для візуалізації графів, які здатні вирішити зазначені обмеження та забезпечити високу продуктивність і зручність використання при роботі з великими та складними графами. Це дозволить розширити можливості аналізу і тестування обчислювальних систем, полегшивши розробку та оптимізацію сучасних технологій.

1.3 Обґрунтування вибору теми

Метою цієї дипломної роботи є розробка програмного забезпечення для візуалізації графів, що ілюструють процеси тестування в обчислювальних системах.

За останні роки область застосування обчислювальних систем значно розширилася, а потреба в ефективних інструментах для їх аналізу та моніторингу стала ще більш актуальною. Поява таких систем була викликана необхідністю вирішення складних завдань з великими обсягами обчислень і обмеженнями швидкості традиційних електронних обчислювальних машин. Використання сучасних обчислювальних систем дозволяє суттєво збільшити продуктивність, навіть при збереженні обмежень на швидкодію окремих компонентів.

З розвитком обчислювальної техніки зростають вимоги до швидкості обробки даних, і, незважаючи на зменшення часу перемикання між електронними схемами до наносекунд, швидкість передачі сигналів у з’єднувальних лініях між елементами системи все ще обмежена швидкістю світла. Таким чином, для підвищення продуктивності необхідно впроваджувати нові принципи паралельної обробки даних, створюючи багатомашинні та багатопроцесорні системи.

Важливим елементом у цьому контексті є програмне забезпечення для візуалізації, яке дозволяє зручно представляти складні структури та взаємодії в

обчислювальних системах. Візуалізація графів є потужним інструментом для аналізу та моніторингу багатопроцесорних систем, допомагаючи зрозуміти процеси паралельної обробки, виявляти вузькі місця і оптимізувати ресурси.

З розвитком багатопроцесорних систем, що активно використовуються в різних галузях, зростає необхідність у надійних інструментах для їх візуалізації та моніторингу. Виведення графів для аналізу тестових процесів дає змогу не лише контролювати роботу системи, але й ефективно реагувати на можливі відмови компонентів, замінюючи несправні елементи без порушення роботи системи.

Важливою частиною розвитку обчислювальних систем є прогрес у комп'ютерних науках і підготовка фахівців для роботи з сучасними технологіями. Створення програм для візуалізації графів, які демонструють процеси тестування та взаємодію між процесорами в багатопроцесорних системах, є важливим кроком у навчанні нових спеціалістів і забезпеченні надійності обчислювальних центрів.

Завдяки візуалізації графів, що містять взаємозв'язки між підсистемами, програмне забезпечення може надати користувачам чітке уявлення про те, як функціонують різні процесори в багатопроцесорних системах, як організовано їхнє взаємодії та обробка даних. Це дозволяє оптимізувати архітектуру обчислювальних систем і підвищити ефективність роботи пристроїв.

Програмне забезпечення для візуалізації графів також дає можливість користувачам легко моделювати різні структури, такі як матричні, деревоподібні або пірамідальні системи, що важливо для аналізу складних обчислювальних процесів. Застосування таких графічних інтерфейсів дозволяє ефективно працювати з великими обсягами даних і здійснювати аналіз інформації за допомогою простих і наочних інструментів.

Таким чином, розробка програмного забезпечення для візуалізації графів є важливою складовою частиною сучасних обчислювальних систем, дозволяючи підвищити їхню ефективність, зручність використання та забезпечити більш точне моніторинг і управління ресурсами.

1.4 Важливість та актуальність теми

Сучасні засоби візуалізації графів відіграють важливу роль у процесах прийняття рішень, зокрема, в області аналізу даних, соціальних мереж, досліджень біоінформатики, а також у технічних та інженерних системах. Завдяки швидкому розвитку графічних процесорів (GPU) та високопродуктивних комп'ютерних технологій з’явились нові можливості для обробки складних та масштабованих графів в реальному часі.

Машинне навчання та штучний інтелект для автоматичної візуалізації

Застосування технологій машинного навчання дозволяє не тільки автоматично створювати візуалізації, а й оптимізувати розташування елементів графа в залежності від контексту, що дозволяє зменшити навантаження на користувача та знизити ймовірність помилок у структурі графа. У майбутньому автоматизація цього процесу може вивести візуалізацію графів на новий рівень, що дасть змогу розв’язувати складніші задачі.

Важливим напрямком є інтерактивні засоби для візуалізації графів, які дозволяють користувачам змінювати параметри графа в реальному часі, переглядати різні аспекти взаємозв’язків, а також змінювати стиль або зовнішній вигляд елементів залежно від потреби. Інтерактивні інтерфейси, зокрема за допомогою технологій веб-браузерів або додатків з підтримкою 3D-графіки, можуть значно покращити зручність використання та доступність.

Прогрес у роботі з динамічними графами є важливим напрямком у дослідженнях. Здатність візуалізувати зміни графа в реальному часі, зокрема при роботі з великими і складними мережами, стає критичною для багатьох сучасних додатків. Це включає не лише статичні дані, але й графи, що змінюються на основі нових даних або змінюваних взаємозв'язків. Розробка нових алгоритмів, які зможуть ефективно працювати з такими графами, стане важливою частиною наукових досліджень.

У роботі з великими графами необхідно враховувати оптимізацію алгоритмів обробки даних для масштабованих систем, що дозволяє значно

зменшити затримки при відображенні графів і забезпечити стабільну роботу навіть при наявності мільйонів елементів. Для цього можуть застосовуватись нові підходи до алгоритмів пошуку, кластеризації та оптимізації, що забезпечать ефективне оброблення та візуалізацію великих графічних структур без значних втрат у продуктивності.

Для підвищення ефективності візуалізації графів важливо інтегрувати ці системи з іншими сучасними технологіями. Взаємодія з іншими платформами для аналізу даних або базами знань, наприклад, з інструментами для обробки великих даних чи штучним інтелектом, може дати додаткові можливості для автоматизованого аналізу та розпізнавання нових патернів у графах.

2 ВИБІР ІНСТРУМЕНТІВ ДЛЯ РЕАЛІЗАЦІЇ ЗАСОБУ ВІЗУАЛІЗАЦІЇ ГРАФІВ

2.1 Програмні інструменти та бібліотеки

Після аналізу доступного програмного забезпечення для реалізації дипломного проєкту було прийнято рішення використати фреймворк JUNG у поєднанні з мовою програмування Java.

Хоча Java не єдина мова, що підтримує об’єктно-орієнтований підхід, вона була створена саме з урахуванням концепцій ООП із самого початку. Це відрізняє її від таких мов, як C++ чи Object Pascal, де об'єктна модель додавалась до вже існуючих процедурних мов (C та Pascal відповідно). Завдяки цьому в Java усі компоненти — від потоків виконання до роботи з мережею, зображеннями або обробкою виключень — представлені у вигляді об’єктів [4].

Особливістю Java є відмова від класичного множинного спадкування, замість якого впроваджено механізм інтерфейсів. Крім того, мова вирізняється суворою типізацією: тип змінної або виразу визначається вже на етапі компіляції, що значно полегшує пошук помилок, адже компілятор одразу вказує на проблемні ділянки коду. Під час виконання Java-програми відбувається автоматичне виявлення виняткових ситуацій, що підвищує надійність системи.

Ще однією важливою перевагою Java є автоматичне керування пам’яттю. Розробник створює об’єкти, які використовуються в процесі виконання програми, а їхнє подальше видалення не потребує втручання програміста. На відміну від C або C++, де очищення пам’яті виконується вручну, в Java цим займається збирач сміття. Віртуальна машина JVM сама відслідковує посилання на об’єкти і, коли вони більше не використовуються, звільняє зайняту ними пам’ять.

Протягом багатьох років Java стабільно входить до переліку найпопулярніших мов програмування у світі. Її вплив не лише не зменшився з часом, а навпаки — зростав. Від моменту своєї появи вона займає провідні

позиції у створенні веб-застосунків. Кожна нова версія мови лише посилювала її позиції. Значна частина сучасного програмного забезпечення створена саме на Java, що свідчить про її ключову роль у світі розробки. Однією з головних причин її популярності є здатність до адаптації: починаючи з першої версії, Java постійно змінюється у відповідь на нові тенденції в IT-сфері. Ба більше — вона не лише пристосовується до змін, а й задає нові напрямки розвитку. Саме ця гнучкість дозволяє Java залишатися актуальною та затребуваною вже тривалий час.

2.2 Вибір мови програмування Java

Офіційною датою появи мови програмування Java вважається 23 травня 1995 року, коли компанія Sun Microsystems представила першу реалізацію Java версії 1.0.

Java має багато спільного з мовами C та C++, зокрема вона перейняла синтаксис із C, а також об’єктно-орієнтовані принципи з C++. Завдяки цьому розробники, знайомі з цими мовами, можуть швидше адаптуватися до Java. Варто зазначити, що мова має чітко визначені стандарти, які викладені в офіційній технічній документації.

Java забезпечує розробникам широкий набір стандартних бібліотек, які не лише прискорюють процес створення програм, але й підвищують їх надійність, зокрема:

— автоматичне керування пам’яттю (збирач сміття) — дозволяє уникнути витоків пам’яті без потреби вручну очищати ресурси;

— багатий вибір середовищ розробки (IDE) — спрощує створення, налагодження та тестування програмного забезпечення;

— зворотна сумісність — додатки, створені у попередніх версіях мови, як правило, сумісні з новими версіями Java;

— мультиплатформність — програми, написані на Java, можуть працювати на різних операційних системах (Windows, Linux, macOS) без модифікацій, а з незначними змінами і на Android, це дозволяє адаптувати

програмний код для мобільних платформ із мінімальними витратами. Крім того, Java підтримує розробку вебдодатків, зокрема для роботи з базами даних через інтернет;

— чітка об’єктно-орієнтована структура – дозволяє реалізовувати складні архітектурні рішення.

— гнучкість JVM — Java Virtual Machine здатна виконувати байт-код, створений не лише мовою Java, а й іншими мовами, які підтримують компіляцію до відповідного формату, це забезпечує можливість інтеграції модулів, написаних різними мовами, без втрати функціональності;

— відкритість коду — Java має реалізації з відкритим вихідним кодом, доступні під ліцензією GP;

— легка зміна бази даних — змінивши лише драйвер, можна підключити іншу СУБД без суттєвих змін у коді застосунку.

— підтримка багатопотоковості — JVM ефективно працює з потоками, що дозволяє використовувати ресурси комп’ютера максимально ефективно.

У сучасному програмному забезпеченні користувач очікує зручний графічний інтерфейс, оскільки робота з консоллю вимагає спеціальних знань. Тому Graphical User Interface (GUI) став невід’ємною складовою сучасних застосунків. Серед основних переваг GUI — зручне введення даних через елементи управління: кнопки, поля введення, списки тощо [10].

З розвитком технологій з’явилось багато бібліотек для роботи з графічними компонентами. У контексті програмування бібліотека — це набір класів та інтерфейсів, створених для виконання спеціалізованих функцій.

2.3 Порівняння графічних бібліотек AWT та Swing

Першою бібліотекою для створення графічного інтерфейсу користувача в середовищі Java була Abstract Window Toolkit (AWT). Ця бібліотека використовувала виклики до нативного коду, написаного мовою C, і взаємодіяла безпосередньо з графічними компонентами операційної системи. Таким чином, інтерфейс програми, написаної на Java з використанням AWT,

значною мірою залежав від платформи, на якій вона запускалася. Це могло призводити до відмінностей у зовнішньому вигляді елементів, таких як шрифти або компоненти управління, при запуску на різних ОС.

З метою забезпечення кращої кросплатформеності було вирішено стандартизувати інтерфейси викликів графічних компонентів у межах AWT. Однак це призвело до зниження функціональності: зменшилася кількість доступних елементів, зокрема не були реалізовані таблиці або підтримка іконок. Незважаючи на ці обмеження, пакет java.awt зберіг актуальність для виконання базових задач у розробці інтерфейсів.

Варто зазначити, що AWT не виконує обробку подій самостійно. Усі запити передаються до операційної системи, яка, власне, і виконує відповідні дії. Це обмежує можливості AWT при створенні сучасного, зручного інтерфейсу. Додатковим недоліком є те, що звільнення використаних ресурсів відбувається автоматично, що може ускладнити структуру програми та вплинути на її продуктивність.

У зв’язку з цим компанія Sun Microsystems розробила нову бібліотеку — Swing, яка суттєво розширила можливості Java у сфері графічного інтерфейсу. На відміну від AWT, компоненти Swing реалізовані повністю мовою Java, що дозволило зменшити залежність від ОС. Swing підтримує гнучку зміну стилів оформлення, має розширений набір стандартних елементів та включає так звані "легковагові" (lightweight) компоненти. Найвагомішою перевагою Swing є те, що його елементи не пов’язані з нативними компонентами операційної системи, що забезпечує стабільнішу та швидшу роботу додатків.

З урахуванням вищенаведених переваг, для реалізації графічного інтерфейсу в рамках проєкту було вирішено використовувати саме бібліотеку Swing.

2.4 Опис бібліотеки Swing

Бібліотека Swing є однією з найпопулярніших середовищ для розробки графічного інтерфейсу користувача в Java, зокрема завдяки реалізації

архітектурного шаблону Model–View–Controller (MVC). Цей шаблон дозволяє розділити відповідальність у коді: зовнішній вигляд, обробка даних та взаємодія з користувачем реалізуються окремо.

Головна ідея MVC полягає в тому, що кожен компонент взаємодіє лише зі своєю сферою відповідальності. Це дозволяє створювати гнучкий, масштабований інтерфейс, в якому візуальні елементи відповідають лише за відображення, а логіка та дані обробляються незалежними модулями.

Архітектура MVC передбачає три основні складові.

Model (Модель) — відповідає за збереження і обробку даних. Вона дозволяє взаємодіяти з інформацією, не залежачи від компонентів інтерфейсу. Наприклад, у випадку випадаючого списку (combo box) дані, які виводяться, зберігаються в окремому класі-моделі. Це дає змогу змінювати логіку роботи з даними без впливу на графічний компонент.

View (Вигляд) — відповідає за виведення даних на екран. Відокремлення вигляду від моделі дозволяє візуалізувати одну і ту ж інформацію в різних форматах. Наприклад, HTML-документ можна представити як відформатовану сторінку або як сирцевий код із тегами та текстом.

Controller (Контролер) — визначає, як система повинна реагувати на дії користувача. Контролер обробляє події, які надходять із вигляду, і оновлює модель або вид відповідно. Такий підхід дозволяє використовувати одну модель з кількома виглядами або контролерами. Наприклад, список може мати окремий контролер для звичайного відображення і ще один — для редагування.

На рисунку 2.1 зображено загальну схему взаємодії між цими компонентами, що ілюструє логіку шаблону MVC.

Кожен графічний інтерфейс користувача базується на трьох ключових елементах:

— компоненти інтерфейсу — це елементи, з якими безпосередньо працює користувач (кнопки, поля вводу, списки тощо);

— макет (layout) — визначає спосіб розміщення компонентів на екрані;

— поведінка — це реакції інтерфейсу на дії користувача (наприклад, натискання кнопок, введення тексту).

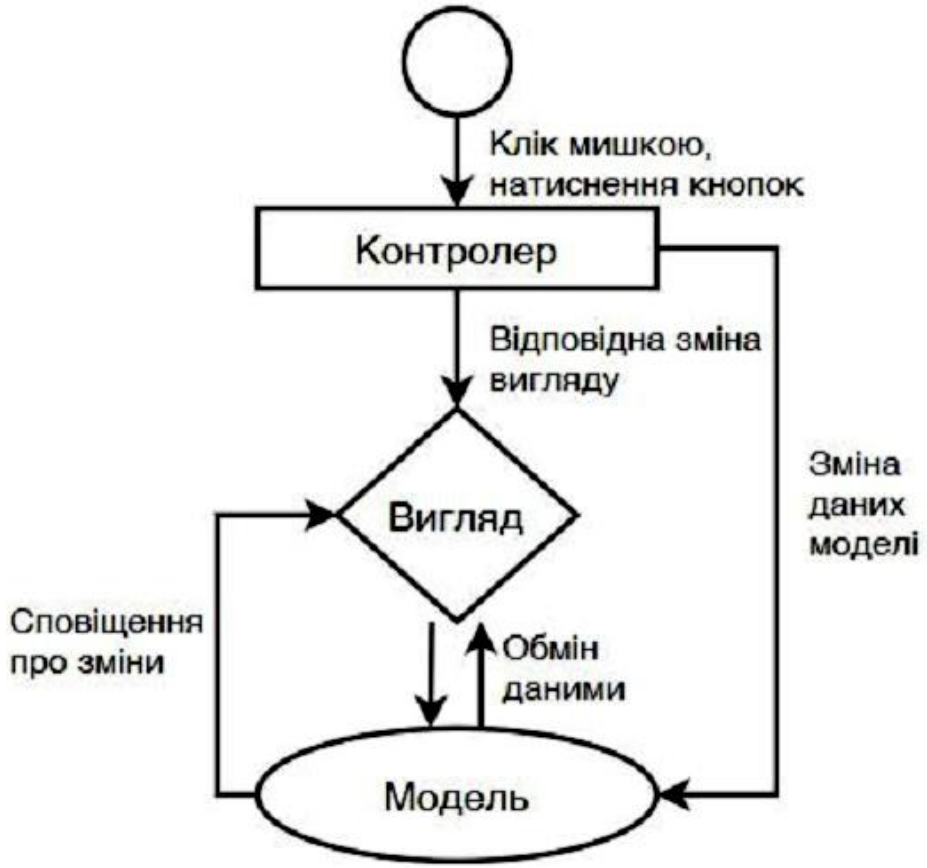


Рисунок 2.1 — Логіка шаблона MVC

Усі програми, створені з використанням Swing, повинні мати хоча б один контейнер верхнього рівня, з якого починається побудова інтерфейсу. До таких контейнерів належать:

- JFrame — основне вікно додатка;
- JApplet — аплети для використання в браузерах;
- JWindow — вікна без заголовка;
- JDialog — діалогові вікна.

Ці компоненти розташовані на вершині ієрархії елементів керування, як це показано на рисунку 2.2. У межах цих контейнерів можна розміщувати інші елементи — легковагові компоненти, які і складають інтерфейс додатка.

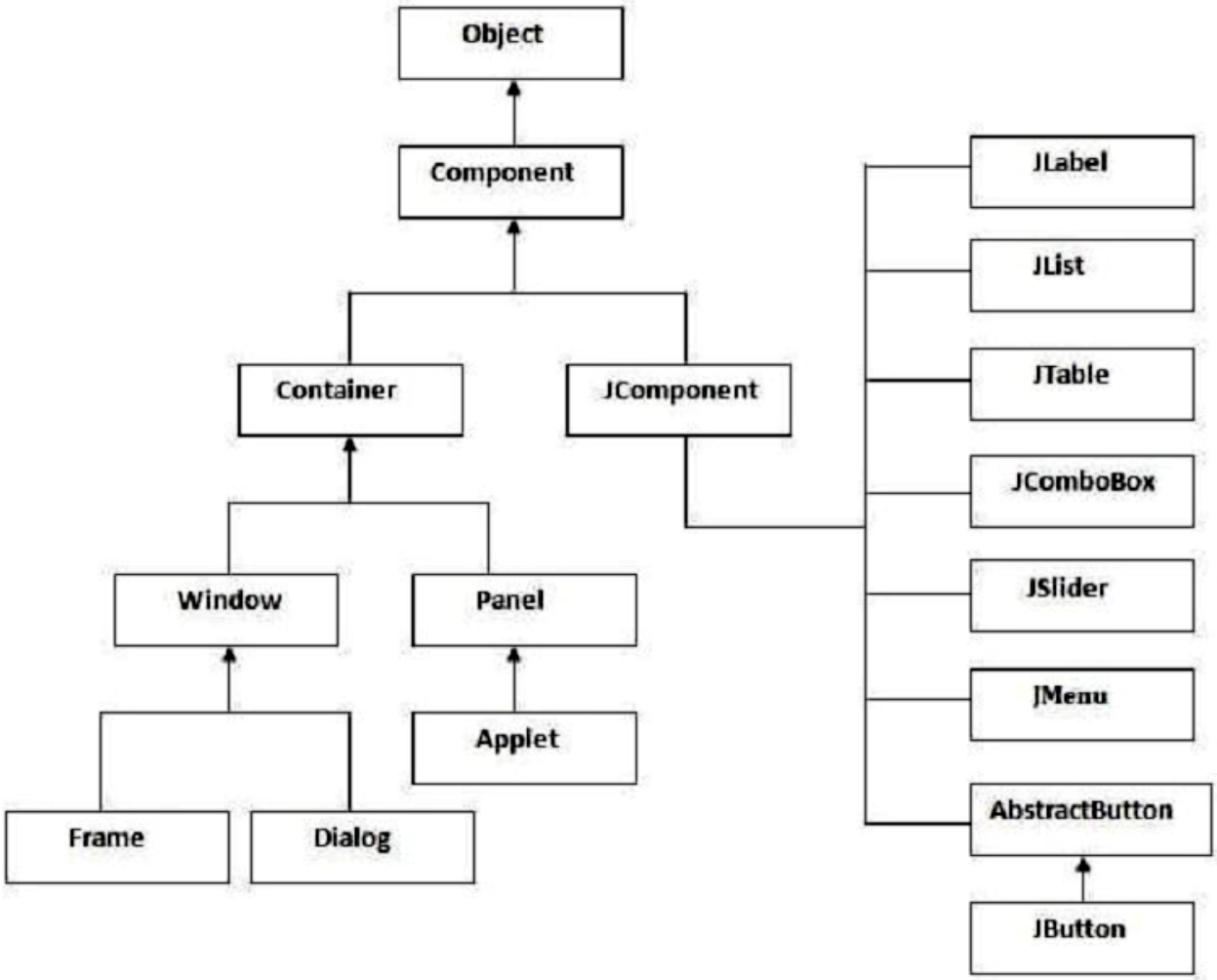


Рисунок 2.2 — Взаємозвязок контейнерів бібліотеки Swing

2.4.1 Компоненти та контейнери

У графічному інтерфейсі Java всі елементи поділяються на два основних типи.

Компоненти (JComponent) — це базові елементи інтерфейсу користувача, такі як кнопки (JButton), підписи (JLabel), текстові поля (JTextField), прапорці (JCheckBox), списки (JComboBox) та інші. Ці об’єкти є частиною класу JComponent, який успадковується більшістю візуальних елементів. Компоненти виконують специфічні функції взаємодії з користувачем, наприклад, прийом вводу, відображення інформації чи виконання подій.

Кожен компонент має вбудований метод add(Component comp), який дозволяє вкладати інші елементи, хоча на практиці це частіше реалізується саме через контейнери, які спеціально створені для організації компонування.

Контейнери (Container) — це спеціальні об’єкти, призначені для зберігання і управління розміщенням компонентів. Вони реалізують логіку компоновки елементів за допомогою менеджерів макету (layout managers), таких як FlowLayout, BorderLayout, GridLayout, BoxLayout тощо. Найпоширенішими контейнерами є JFrame, JPanel, JDialog та JWindow.

Контейнери можуть містити як окремі компоненти, так і вкладені контейнери, що дозволяє створювати складні ієрархії інтерфейсу. Для додавання компонентів до контейнера використовують метод add(Component comp), або його перевантажені варіанти, які дозволяють вказати додаткові параметри, зокрема положення компонента в певному макеті.

На рисунку 2.3 показано приклад побудови інтерфейсу з використанням різних типів контейнерів.

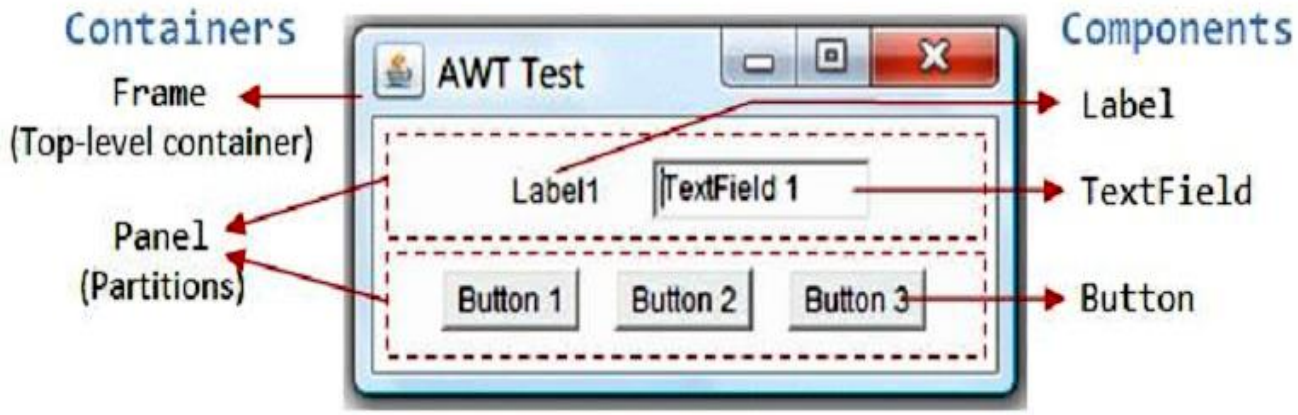


Рисунок 2.3 — Приклад побудови інтерфейсу з використанням різних типів контейнерів

У цьому прикладі контейнер верхнього рівня — це JFrame, який виступає основним вікном програми. Він має заголовок, значки керування (мінімізація, розгортання, закриття), а також внутрішню область, куди додаються інші елементи. Всередині JFrame розміщено дві JPanel, які слугують для групування елементів інтерфейсу.У панелі компонуються п’ять базових елементів: підпис (JLabel), текстове поле (JTextField) і три кнопки (JButton) для взаємодії з користувачем.

Всі елементи GUI мають бути розміщені в межах контейнера. Без цього вони не зможуть коректно відображатися або взаємодіяти з іншими компонентами. Під час створення інтерфейсу необхідно обрати відповідний контейнер і визначити менеджер розміщення для правильного позиціонування візуальних елементів.

У Swing рекомендовано уникати абсолютного позиціонування (тобто вказування координат вручну), натомість варто користуватися системою layout-менеджерів, яка гарантує адаптивність інтерфейсу до різних розмірів екрана та систем.

2.4.2 Головні складові

Одним із найпоширеніших елементів інтерфейсу в бібліотеці Swing є JLabel — компонент, призначений для виведення тексту або графічного зображення. JLabel виконує важливу роль у формуванні зручного та інформативного користувацького інтерфейсу, оскільки забезпечує пояснення або супровід інших елементів (наприклад, текстових полів, кнопок). Його часто використовують не лише як статичний текстовий блок, але і як носій піктограм чи логотипів.

До основних методів класу JLabel належать:

— getText() / setText(String text) — для зчитування або встановлення тексту, що відображається;

— getIcon() / setIcon(Icon icon) — для роботи з графічними зображеннями (наприклад, форматів .png або .jpg);

— getHorizontalAlignment() / setHorizontalAlignment(int alignment) — встановлює вирівнювання тексту по горизонталі;

— getVerticalAlignment() / setVerticalAlignment(int alignment) — аналогічно, по вертикалі;

— getDisplayedMnemonic() / setDisplayedMnemonic(char key) — дозволяє задати гарячу клавішу (mnemonic), яка активує пов’язаний компонент при натисканні [Alt + клавіша].

Іншим ключовим компонентом у Swing є JButton — інтерактивна кнопка, яка ініціює певну дію після натискання. На відміну від JLabel, кнопка є активним елементом керування. Її поведінка визначається через слухача подій (ActionListener). Методика роботи з JButton подібна до JLabel, що спрощує освоєння бібліотеки.

Окрім стандартних методів для тексту й іконок, JButton підтримує стани, які відображають, чи кнопка доступна, натиснута, обрана тощо. Для кожного стану можна призначити окреме зображення (setPressedIcon(), setDisabledIcon()), що покращує візуальний відгук інтерфейсу.

Основний елемент введення тексту в Swing — це JTextField. Він дозволяє користувачу вводити однорядковий текст, редагувати його, виділяти, видаляти тощо. Основний метод взаємодії з цим компонентом — getText() / setText(String text), які дозволяють програмно отримати чи задати вміст поля.

Для організації простору інтерфейсу та розміщення згаданих компонентів використовується JFrame — контейнер верхнього рівня, що представляє вікно графічного застосунку. JFrame є основою будь-якого GUI-додатку, оскільки надає набір функціональних можливостей, аналогічних вікнам операційної системи: заголовок, кнопки згортання/розгортання/закриття, можливість зміни розмірів і переміщення.

Ось кілька ключових методів для налаштування JFrame:

— getTitle() / setTitle(String title) — для задання заголовка вікна;

— getState() / setState(int state) — управління станом фрейма (нормальний, згорнутий, розгорнутий);

— isVisible() / setVisible(boolean flag) — визначає, чи вікно відображається на екрані;

— getLocation() / setLocation(int x, int y) — координати появи вікна;

— getSize() / setSize(int width, int height) — розміри вікна;

— add(Component comp) — додавання елементів до вікна.

Зазвичай компоненти JLabel, JButton та JTextField розміщуються безпосередньо в JFrame, або опосередковано через додаткові контейнери типу JPanel, які забезпечують гнучке компонування за допомогою layout-менеджерів.

2.4.3 Обробка подій в Swing

Зміна стану об'єкта в Java називається подією. Подія відображає зміну стану джерела і генерується в результаті взаємодії користувача з компонентами інтерфейсу. Це важлива концепція в розробці графічних інтерфейсів, оскільки дозволяє системі реагувати на дії користувача, такі як натискання кнопок, введення тексту або переміщення миші.

Події зазвичай поділяють на дві основні категорії.

Події переднього плану — ці події є результатом безпосередньої взаємодії користувача з елементами графічного інтерфейсу. Прикладами таких подій є натискання кнопки, переміщення миші, введення символів через клавіатуру, вибір елемента зі списку або прокручування сторінки.

Фонові події — ці події не вимагають прямої взаємодії користувача, але можуть бути важливими для роботи програми. Це можуть бути події, що виникають через переривання операційної системи, збій апаратного або програмного забезпечення, завершення операцій або закінчення часу таймера.

Java використовує модель делегування подій, яка визначає стандартний механізм для генерування та обробки подій. Ця модель має два основних учасники:

— Source (Джерело події) — об'єкт, в якому виникає подія. Він відповідає за генерування події та передачу її обробнику (слухачу), джерело інформує слухача про настання події та передає відповідну інформацію для подальшої обробки;

— Listener (Слухач події) — об'єкт, який чекає на події від джерела і реагує на них, після отримання події слухач виконує відповідну дію, наприклад, змінює стан інтерфейсу, виконує обчислення або запускає іншу операцію.

Слухач може бути призначений на конкретну подію, таку як натискання кнопки або зміна тексту в полі введення.

Основною перевагою цієї моделі є відокремленість логіки взаємодії з користувачем (графічного інтерфейсу) від логіки, що генерує події. Це дозволяє розробникам писати більш чистий та гнучкий код, в якому елементи інтерфейсу можуть делегувати обробку подій окремим фрагментам коду, що спрощує тестування і підтримку програми.

У Java компоненти графічного інтерфейсу є об'єктами, які успадковують властивості від класів Component та Window. Клас Component є предком усіх графічних елементів, що дозволяє виконувати основні операції з ними. Наприклад, методи для зміни розмірів та форм фреймів або компонентів зазвичай розташовані в цих класах:

Метод show() знаходиться в класі Window, який є суперкласом для Frame і відповідає за відображення фреймів на екрані.

Метод setLocation(), що знаходиться в класі Component, дозволяє змінювати розташування компонента в межах контейнера або на екрані.

Координати, які задаються методами setLocation() і setBounds(), обчислюються відносно екрана, тоді як координати елементів всередині контейнера визначаються відносно цього контейнера. Це важливо для коректної організації виведення та взаємодії елементів на різних рівнях інтерфейсу.

2.5 Бібліотека JUNG: можливості та застосування

JUNG (Java Universal Network/Graph Framework) — це потужна бібліотека програмного забезпечення, розроблена для моделювання, аналізу та візуалізації даних, що можуть бути представлені у вигляді графів або мереж. Оскільки вона написана на мові програмування Java, JUNG надає розробникам доступ до широких можливостей Java API, а також дозволяє інтегрувати сторонні бібліотеки Java для додаткової функціональності. Це забезпечує велику гнучкість при розробці додатків для роботи з графами та мережами.

Архітектура JUNG розроблена таким чином, щоб підтримувати різноманітні типи графів і їх елементи: орієнтовані та неорієнтовані графи, графи з паралельними ребрами та гіперграфи. Ці типи графів можуть бути використані для моделювання складних взаємозв'язків і залежностей між об'єктами у системах різного рівня складності. Бібліотека дозволяє додавати метадані до вершин і ребер, що полегшує створення аналітичних інструментів для дослідження таких систем. Наприклад, можна вивчати не лише відносини між об'єктами, але й додаткову інформацію, що зберігається разом з кожним елементом графа.

Однією з важливих можливостей є підтримка візуалізації даних. JUNG включає механізми для створення інтерактивних інструментів, що дозволяють досліджувати багатопроцесорні системи або складні мережі. Це дозволяє розробникам ефективно аналізувати великі обсяги даних, що постійно змінюються, а також надає гнучкість в управлінні процесами відображення даних. Додатково передбачено фільтрацію даних, що дозволяє зосередити увагу на певних частинах графа, відсіваючи менш важливі елементи під час виконання аналізу.

Однією з головних переваг JUNG є масштабованість. Лише обмеженнями ресурсів комп'ютера, на якому працює Java Virtual Machine (JVM), визначаються розміри графа, кількість вершин та ребер. Це дозволяє працювати з надзвичайно великими графами, що можуть містити мільйони вершин і ребер, що є важливою характеристикою для великих систем і комплексних моделей.

JUNG реалізує типи графів через інтерфейси, абстрактні класи та конкретні реалізації, що дозволяє користувачам легко створювати й використовувати графи відповідно до своїх потреб.

Бібліотека підтримує різні типи графів, кожен з яких має свій набір інтерфейсів і класів:

— інтерфейси ArchetypeGraph, ArchetypeVertex та ArchetypeEdge визначають загальну поведінку для графів, вершин та ребер, дозволяючи

розробникам створювати різні варіанти графів, включаючи орієнтовані, неорієнтовані, гіперграфи та графи з паралельними ребрами.

— інтерфейси Graph, Vertex та Edge успадковуються від базового інтерфейсу і дозволяють реалізувати методи для графів, де кожне ребро з'єднує тільки дві вершини, це забезпечує ефективне управління зв'язками між елементами графа.

Абстрактні класи: AbstractSparseGraph, AbstractSparseVertex та AbstractSparseEdge орієнтовані на створення розріджених графів, де кількість ребер значно менша за кількість вершин. Ці класи менш ефективні для графів з великою кількістю зв'язків.

Конкретні реалізації: класи DirectedSparseGraph, DirectedSparseEdge, DirectedSparseVertex та їх аналоги для неорієнтованих графів є найбільш оптимізованими для використання в конкретних сценаріях, де необхідно працювати з орієнтованими або неорієнтованими графами.

JUNG надає потужні інструменти для візуалізації графів. Візуалізація складається з кількох основних елементів:

— макет — визначає розташування вершин на площині;

— компонент (Swing) — використовує VisualizationViewer, який є компонентом на основі JPanel, для відображення графа;

— рендерер — займається малюванням вершин та ребер на основі розташування, заданого макетом.

Ці елементи працюють разом для створення інтерактивного перегляду графів. Крім того, в JUNG реалізовано GraphDraw, який спрощує інтеграцію макету, компонента та рендерера для створення зручних візуалізацій, що можуть бути адаптовані під конкретні потреби користувача.

JUNG також включає утиліти, що полегшують налаштування та покращення візуалізації. Наприклад, FadingVertexLayout дозволяє створювати ефекти затухання для вершин, що дає змогу створювати динамічні візуалізації, де елементи можуть поступово зникати або відновлюватися. Це робить

взаємодію з графом більш інтуїтивною і дає можливість візуалізувати зміни в реальному часі.

Завдяки цьому набору можливостей, JUNG є потужним інструментом для розробки аналітичних інструментів, що працюють з великими та складними графами, і є дуже корисним для дослідження, аналізу та візуалізації складних мережевих структур.

2.6 Основні поняття теорії графів

За останні роки, на тлі розвитку теорії графів, на передній план виходять проблеми аналізу багатопроцесорних обчислювальних систем. Нехай V — непуста множина, наприклад $\{v_1, v_2, v_3, v_4, v_5\}$. Для цього випадку множина всіх його двоелементних підмножин позначається як $V^{(2)}$:

$$V^{(2)} = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_1, v_5\}, \{v_2, v_3\}, \{v_2, v_4\}, \{v_2, v_5\}, \{v_3, v_4\}, \{v_3, v_5\}, \{v_4, v_5\}\}.$$

Розглянемо довільне елемент E, яке належить множині V(2):

$$E = \{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, v_4\}, \{v_2, v_4\}, \{v_1, v_5\}, \{v_3, v_4\}, \{v_4, v_5\}\}.$$

Пару V, E називають неорієнтованим графом G, де V — це множина вершин, а E — множина ребер, що є підмножиною множини $V^{(2)}$. У більш компактній формі це визначення зазвичай формулюється так: пара V, E є неорієнтованим графом, якщо V — непуста множина елементів, що називаються вершинами, а E — множина невпорядкованих пар різних елементів з V, що називаються ребрами.

Для позначення різноманітних відношень у теорії графів використовуються позначення VG або V(G) для множини вершин і EG або E(G) для множини ребер графа G.

Наочне представлення графа здійснюється через рисунок (діаграму), де вершини зображаються точками, колами або іншими фігурами, а ребра — лініями, що з'єднують зображення вершин відповідної пари. Форма та розміри цих зображень не мають значення. Важливо, щоб вони відповідали множинам V та E. На рисунку 2.4 наведено варіанти такого зображення для описаного графа.

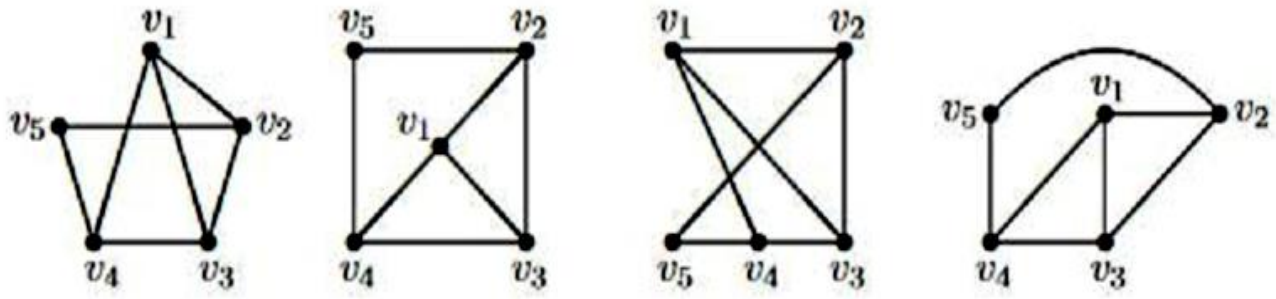


Рисунок 2.4 — Приклади графів

Дві вершини, які утворюють пару vi,vj, називаються його кінцями, а ребро, яке з'єднує ці вершини, називають ребром, що з'єднує vi та vj. Вершини vi і vj називаються суміжними. Суміжними також є ребра, які мають спільну вершину. Ребро і вершина, яка є його кінцем, називаються інцидентними. Наприклад, в графі на рисунку 2.4 вершини v1 і v2 є суміжними, а вершини v1 і v3 не є суміжними. Ребра {v1,v2}і {v1,v3} є суміжними, а ребра {v4,v5} і {v2,v3} не є суміжними. Вершина v3 і ребро {v2,v3} є інцидентними.

Число ребер, що інцидентні вершині v, визначається як степінь вершини, позначений deg(v). Таким чином, в графі на рисунку 2.4 deg(v1)=deg(v2)=deg(v3)=deg(v4)=3, а deg(v5)=2. Вершину v називають ізольованою, якщо deg(v)=0, і кінцевою, якщо deg(v)=1. Ребро, інцидентне

кінцевій вершині, називається кінцевим. Список степенів усіх вершин називається степеневою послідовністю графа.

Множина вершин, які суміжні з вершиною v, позначається як adj(v). Наприклад, в графі на рисунку 2.4 adj(v1)={v2,v3,v4}.

Важливими кількісними характеристиками графа є: число вершин n=|V|n = |V|n=|V|, що визначає порядок графа, і число ребер m=|E|. Граф з n вершинами та m ребрами називається (n, m)-графом.

Теоремо теорії графів базується на твердженні Ейлера, що зв'язує суму ребер, вершин і їх степенів.

Теорема 1.

Сума степенів вершин графа (n, m) дорівнює подвоєному числу його ребер:

$$\sum_{i=1}^{n} \deg(v_i) = 2|E| = 2m$$

Доведення:

Оскільки будь-яке ребро інцидентне двом вершинам, кожне ребро враховується двічі при підсумовуванні степенів всіх вершин графа. Таким чином, сума степенів всіх вершин дорівнює подвоєному числу ребер.

Як наслідок, у будь-якому графі кількість вершин непарної степені є парною. Нехай множини вершин парної і непарної степені позначені відповідно як V1 і V2. Очевидно, що сума степенів усіх вершин дорівнює 2m. Маємо дві складові суми:

— перша складова — сума степенів вершин з парною степеню, що є парним числом;

— друга складова — сума степенів вершин з непарною степеню, що також має бути парним числом.

Тому, при додаванні непарних чисел сума може бути парною лише в тому випадку, якщо їх кількість також парна.

Для визначення, чи є різні зображення графа відображеннями одного і того ж графа, використовується концепція ізоморфізму графів.

Ізоморфізм — це взаємно однозначна відповідність між множинами вершин двох графів G1 і G2, яка зберігає відношення суміжності. Графи G1 і G2 називаються ізоморфними, якщо їхні структури суміжності однакові, тобто для кожної пари вершин з одного графа існує відповідна пара вершин в іншому графі, і вони з'єднані відповідними ребрами.

Позначення для ізоморфізму виглядає так:

$$G_1 \cong G_2 \text{ або } G_1 = G_2$$

Для визначення, чи є два графи ізоморфними, необхідно скласти списки вершин кожного з графів, вказуючи для кожної вершини її сусідів (множину adj(v)). Потім порівнюємо ці списки для обох графів. Якщо списки для всіх вершин збігаються, то графи є ізоморфними. У разі різних нумерацій вершин потрібно змінювати нумерацію одного з графів до досягнення однакових списків.

Приклад: два ізоморфні графи G1 і G2, що зображені на рисунку 2.5.

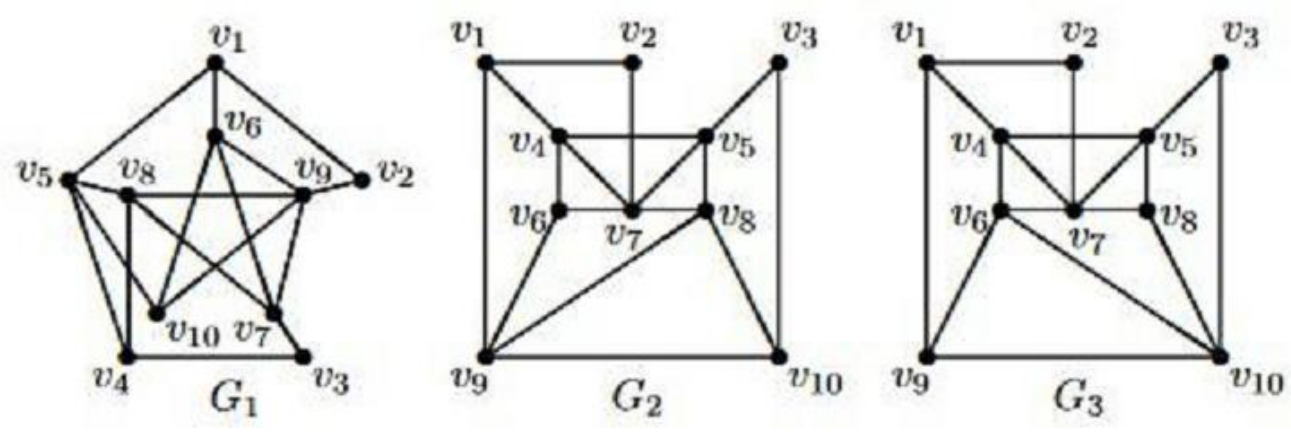


Рисунок 2.5 — Приклади ізоморфних графів

Для цих графів необхідно пройти описану вище процедуру, змінюючи нумерацію вершин, поки не буде знайдено ізоморфізм.

Для доведення того, що два графи G1 і G3 не є ізоморфними, треба виконати всі перевірки і продемонструвати, що їхні списки вершин не можуть бути однаковими, навіть при зміні нумерації.

Інваріанти — це характеристики графа, що не змінюються при ізоморфізмі. Вони можуть допомогти значно зменшити кількість перевірок на ізоморфізм. Інваріанти включають:

— число вершин і число ребер графа;

— число вершин парної і непарної степені;

— степенева послідовність;

— властивості графа, такі як зв'язність, дводольність, наявність чи відсутність циклів.

Інваріанти дозволяють спростити задачу ідентифікації ізоморфізму між графами.

На множині з nnn вершин можливе утворення $2^{C_n^2}$ різних невпорядкованих пар, що відповідають всім можливим ребрам графа. Тому кожному nnn-вершинному графу можна поставити у відповідність C_n^2 — розрядний двійковий код, де кожен розряд відповідає певному ребру: якщо розряд дорівнює 1, то граф містить це ребро, якщо ж 0 — то не містить. Звідси випливає, що кількість графів порядку nnn можна визначити як:

$$2^{C_n^2}$$

Однак, якщо не враховувати розмітку вершин (яка принципово значення не має), а лише дозволяє описати зв'язки між вершинами, то не всі ці графи є різними. Наприклад, існують лише 8 різних трьох вершинних графів, оскільки графи з однаковими структурами, але різними позначеннями вершин, є ізоморфними. Це означає, що деякі з цих графів є еквівалентними, і тому їх не слід рахувати окремо.

Наприклад, якщо ми розглянемо всі можливі графи з трьох вершин, то їх буде 8, навіть якщо при їх зображенні ми використовуємо різні мітки вершин. Всі ці графи зображені на рисунку 2.6.

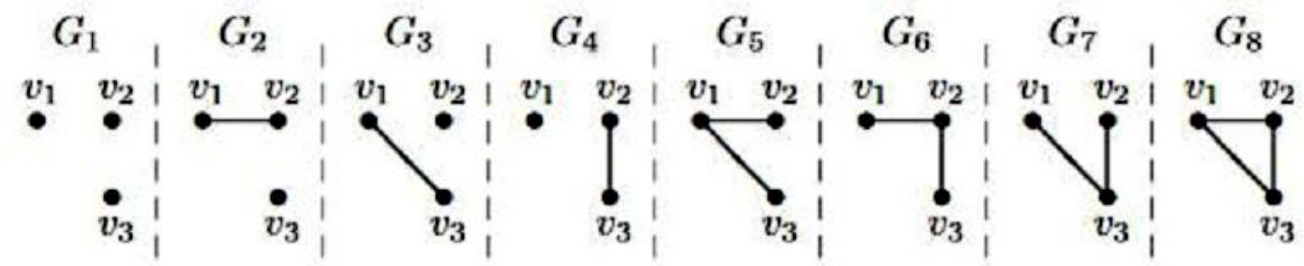


Рисунок 2.6 — Приклад графів з трьома вершинами

Далі з'являється поняття абстрактного графа, що описує структуру зв'язків між вершинами без урахування їх конкретних міток чи позначень. Це дозволяє уникнути дублювання графів, які є ізоморфними, але мають різні позначення вершин.

У даному контексті, абстрактний граф означає, що нас цікавить тільки зв'язність між вершинами, без прив'язки до конкретних імен чи позначень самих вершин.

Незв’язний граф — це граф, у якому множина його вершин може бути розділена на два або більше неперетинних підмножин, причому між цими підмножинами не існує жодних ребер. Кожна з таких підмножин називається компонентом зв’язності графа. Якщо граф розділяється на кілька компонент, то він є незв’язним. Наприклад, на рисунку 7 графи G1, G2, G3, G4, G5 є незв’язаними, а графи G6, G7, G8, G9, G10, G11 — зв’язаними.

Зв’язний граф — це граф, у якому існує шлях між будь-якими двома його вершинами, тобто, можна дістатися від будь-якої вершини до будь-якої іншої, переміщаючись по ребрах. На рисунку 7 графи G6-G11 є зв’язаними.

Пустий граф — це граф, у якому немає жодних ребер. Наприклад, граф G1 на рисунку 7 є пустим графом, він складається лише з вершин і не містить жодних ребер. Такі графи часто позначають як O_n , де n — це кількість вершин графа.

Повний граф — це граф, в якому кожна пара різних вершин з’єднана ребром. Повний граф на рисунку 2.7 — це граф G11, який має максимальну кількість ребер для графа четвертого порядку.

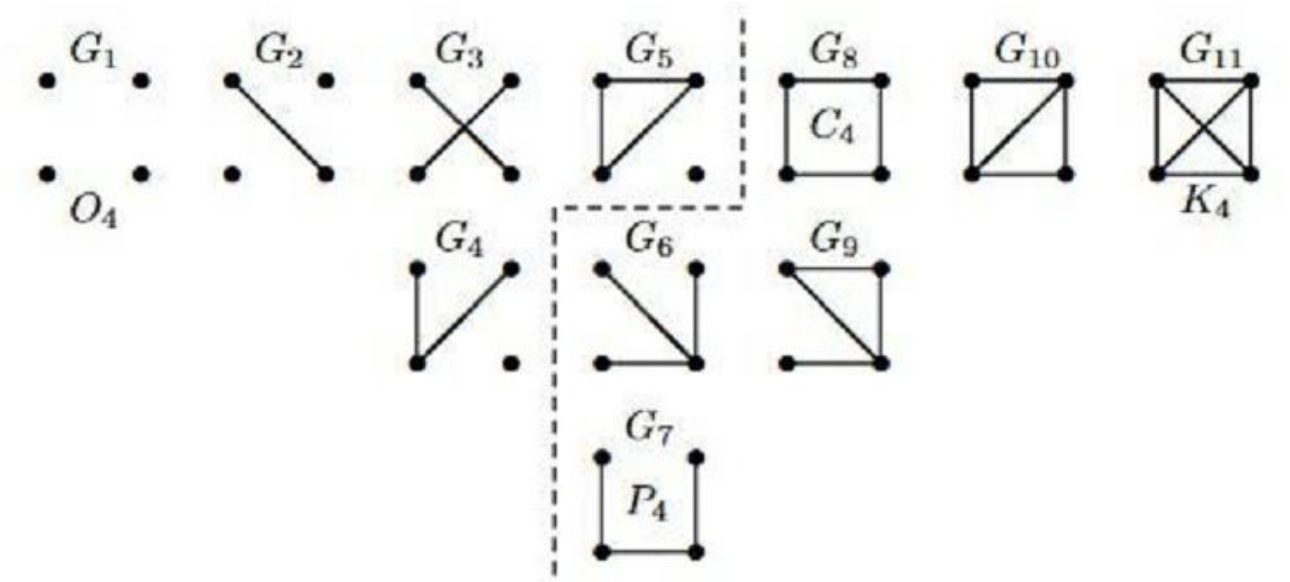


Рисунок 2.7 — Різновиди графів

Для повного графа з n вершинами кількість ребер дорівнює кількості двоелементних підмножин множини вершин, тобто:

$$m = \frac{n}{2} = \frac{n(n-1)}{2}$$

Регулярний граф — це граф, у якому всі вершини мають однаковий ступінь. Наприклад, графи G1, G3, G8, G11 на рисунку 2.7 є регулярними, і в кожному з них всі вершини мають однакові степені. Такі графи називаються однорідними або регулярними (рис. 2.8).

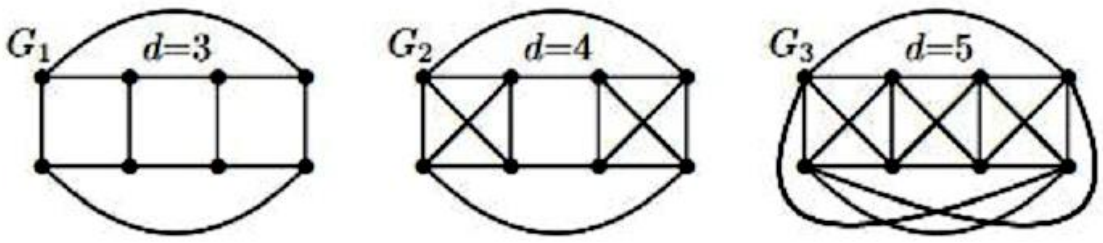


Рисунок 2.8 — Регулярні графів

Кубічний граф — це регулярний граф, в якому кожна вершина має степінь 3. Наприклад, граф G11 на рисунку 2.7 є кубічним графом. Важливо, що кількість вершин в кубічному графі завжди парна, оскільки сума степенів всіх вершин завжди парна (оскільки кожне ребро інцидентне двом вершинам).

Число ребер в регулярному графі: Якщо граф регулярний з степенем d, то кількість ребер m можна знайти за формулою:

$$m = \frac{n \cdot d}{2},$$

де n — кількість вершин;

d — степінь кожної вершини.

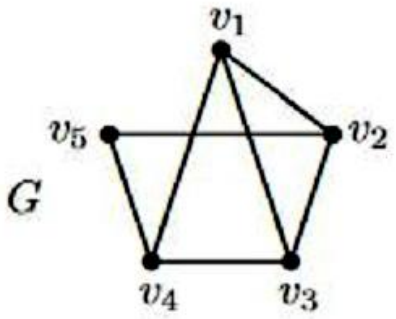
Такі графи можуть бути різними за структурою, але їх об'єднує те, що всі вершини мають однаковий ступінь.

Матриця суміжності — це квадратна матриця порядку n, де елемент a_{i,j} дорівнює 1, якщо існує ребро між вершинами v_i та v_j, і 0, якщо такого ребра немає. Така матриця симетрична для неорієнтованих графів, тобто a_{i,j} = a_{j,i} (рис. 2.9).

Властивості матриці суміжності:

— сума елементів у кожному рядку матриці суміжності дорівнює степеню відповідної вершини, тобто deg(v_i) — це кількість одиниць у рядку і матриці;

— загальна кількість одиниць у матриці дорівнює подвоєному числу ребер графа (оскільки кожне ребро інцидентне двом вершинам).



$$\mathbf{A} = \begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \\ v_5 \end{array} \begin{array}{c} \begin{array}{ccccc} v_1 & v_2 & v_3 & v_4 & v_5 \end{array} \\ \begin{bmatrix} 0 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \end{array} \begin{array}{c} \deg v_i \\ 3 \\ 3 \\ 3 \\ 4 \\ 1 \end{array}$$

Рисунок 2.9 — Граф і його матриця суміжності

Для порожнього графа матриця суміжності — це матриця, що складається з нулів: A(O_n) = 0_n.

Для повного графа матриця суміжності складається з одиниць, окрім діагональних елементів, які дорівнюють нулю: A(K_n) = 1_n - I_n, де I_n — одинична матриця.

Якщо граф складається з кількох компонент зв’язності, то матриця суміжності може бути приведена до блочно-діагонального вигляду, де кожний діагональний блок відповідає матриці суміжності окремої компоненти:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & \mathbf{A}_{22} & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{A}_{ss} \end{bmatrix},$$

У випадку к-дольного графа, матриця суміжності може бути приведена до блочного вигляду, де на головній діагоналі знаходяться нульові блоки:

$$\mathbf{A} = \begin{bmatrix} \mathbf{0} & \mathbf{A}_{12} & \dots & \mathbf{A}_{1k} \\ \mathbf{A}_{21} & \mathbf{0} & \dots & \mathbf{A}_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{k1} & \mathbf{A}_{k2} & \dots & \mathbf{0} \end{bmatrix}.$$

Блоки A_{ij}, де i ≠ j, відповідають підграфам, утвореним вершинами кожної долі графа.

Для трьохдольного графа з трьома множинами вершин, матриця суміжності складається з блоків, де зв'язки між вершинами з різних долей представлені одиницями, а всередині кожної долі немає ребер (внутрішня частина діагональних блоків містить нулі) (рис. 2.10).

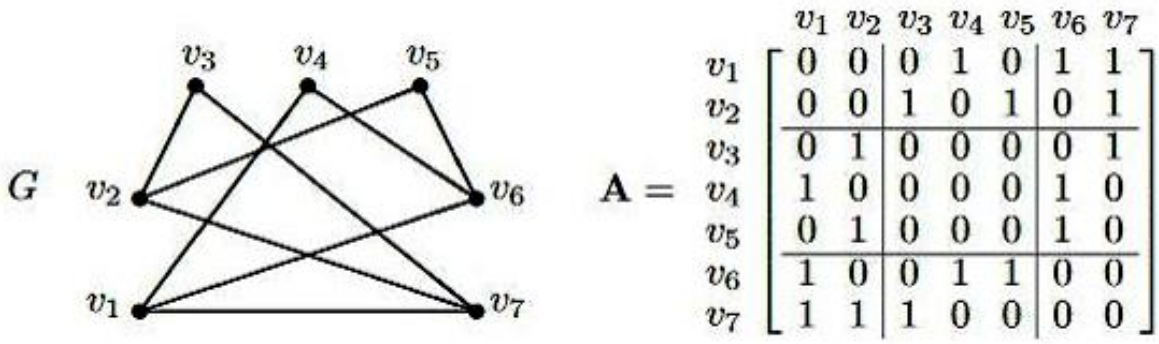


Рисунок 2.10 — Приклад трьохдольного графа та його матриці

Матриця Кірхгофа — це також квадратна матриця порядку n, але її елементи визначаються наступним чином:

— k_{ij} = 1, якщо вершини v_i та v_j суміжні,

— k_{ij} = 0, якщо вершини v_i та v_j не суміжні,

— k_{ii} = deg(v_i), тобто діагональні елементи дорівнюють степеню відповідної вершини.

Матриця Кірхгофа зазвичай позначається K = D - A, де D — діагональна матриця степенів вершин, а A — матриця суміжності графа.

Матриця Кірхгофа може бути отримана з матриці суміжності, але з урахуванням діагональних елементів, що відповідають степеням вершин.

Оскільки ізоморфні графи відрізняються лише розміткою (нумерацією) вершин, то відповідні матриці Кірхгофа можуть бути отримані одна з іншої за допомогою перестановки рядків і стовпців, що є властивістю ізоморфізму графів.

Ці матриці є основними інструментами для аналізу графів у теорії графів та використовуються для числового опису структури графа.

Матриця інцидентності є прямокутною матрицею розміру n×mn \times mn×m, де елемент bi,j=1, якщо вершина v_i інцидентна ребру e_j, і 0 у протилежному випадку. Рядки такої матриці називаються векторами інцидентності і позначаються B_i. Приклад графа, матриця інцидентності якого представлена, можна побачити на рис. 2.11.

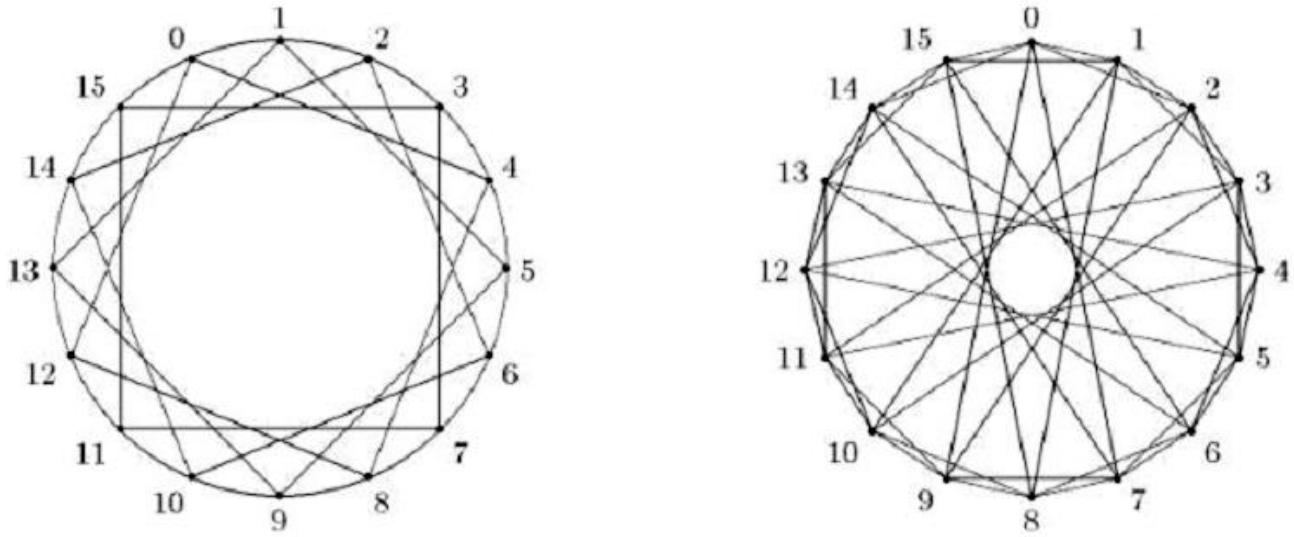


Рисунок 2.11 — Циркулянтні графи

Циркулянтні графи використовуються в різних обчислювальних системах, таких як ILLIAC-IV, MPP, Intel Paragon, Cray T3D, SONET. Вони є важливими для створення багатомодульних високошвидкісних пам'ятей в обчислювальних системах і знаходять застосування в паралельних обчисленнях. В даний час ці графи активно використовуються в оптичних мережах, моделях хімічних реакцій, нейронних мережах через їх високу надійність і зв'язність.

Визначення 1: Нехай s1,s2,...,sk — ціле число, що визначає циркулянтний граф з множиною вершин $V=\{0,1,...,n-1\}$ і множиною ребер $E=\{(i,j):|i-j|\equiv m \pmod n, m=1,...,k\}$, то такий граф називається циркулянтною мережею.

Інакше кажучи, циркулянтний граф — це граф Келі, у якого матриця суміжності є циркулянтною. Параметри n та $S=\{s1,s2,...,sk\}$ повністю визначають циркулянтний граф розмірності k. Степінь циркулянтного графа дорівнює 2k, якщо sk=n/2. Якщо n парне і sk=n/2, то циркулянт має степінь 2k−1.

Лема 1: якщо (n,s1)=1(n, s_1) = 1(n,s1)=1, то в циркулянтному графі C(n;s1,s2) існує гамільтоновий цикл, що використовує тільки хорду s1.

Циркулянт може бути зв'язним і мати гамільтоновий цикл, але не мати циклів, породжених окремими твірними, наприклад, граф C(24;3,4) C.

Лема 2: циркулянтні графи C(n;s1,...,sk) і C(n;s1,...,n−si,...,sk) є ізоморфними. Це дозволяє обмежитися розглядом циркулянтних графів з твірними, що не перевищують n/2.

Лема 3: Якщо (n,t)=1, то циркулянтні графи C(n;s1,...,sk) і C(n;ts1,ts2,...,tsk) є ізоморфними для всіх твірних ts_i, взятих по модулю n.

Метричними характеристиками циркулянтних графів є діаметр та середня відстань, що відповідають максимальним і середнім структурним затримкам у мережі.

Визначення 2: діаметром графа C називається d(n;S)=max$_{f_0}$d(i,j)де i,j∈Vi, , a d(i,j)— довжина найкоротшого шляху між вершинами i і j.

Визначення 3: середньою відстанню (середнім діаметром) графа C(n;S)називається d(n;S)=d(i,j)n(n−1), де i,j — всі пари вершин.

Теорема 1: значення M(d,k)є непарним для будь-яких d≥1 і k≥1.

Визначення 4: Множина n і S циркулянтного графа C(n;S)є оптимальними, якщо d(n;S)=D(n), і субоптимальними, якщо d(n;S)=D(n)+1.

Визначення 5: циркулянтний граф C(n;S)називається гранично оптимальним, якщо число вершин C(n;S)на відстані m від вершини 0 дорівнює S(m,k)для всіх m, де m=0,1,...,D(n)−1.

3 РОЗРОБКА АЛГОРИТМУ ТА ПРОГРАМИ ДЛЯ ВІЗУАЛІЗАЦІЇ ГРАФІВ

3.1 Алгоритм побудови циркулянтного графа

На рисунку 3.1 зображено фрагмент логіки побудови графа – додавання вершин та встановлення зв’язків відповідно до циркулянтної схеми.

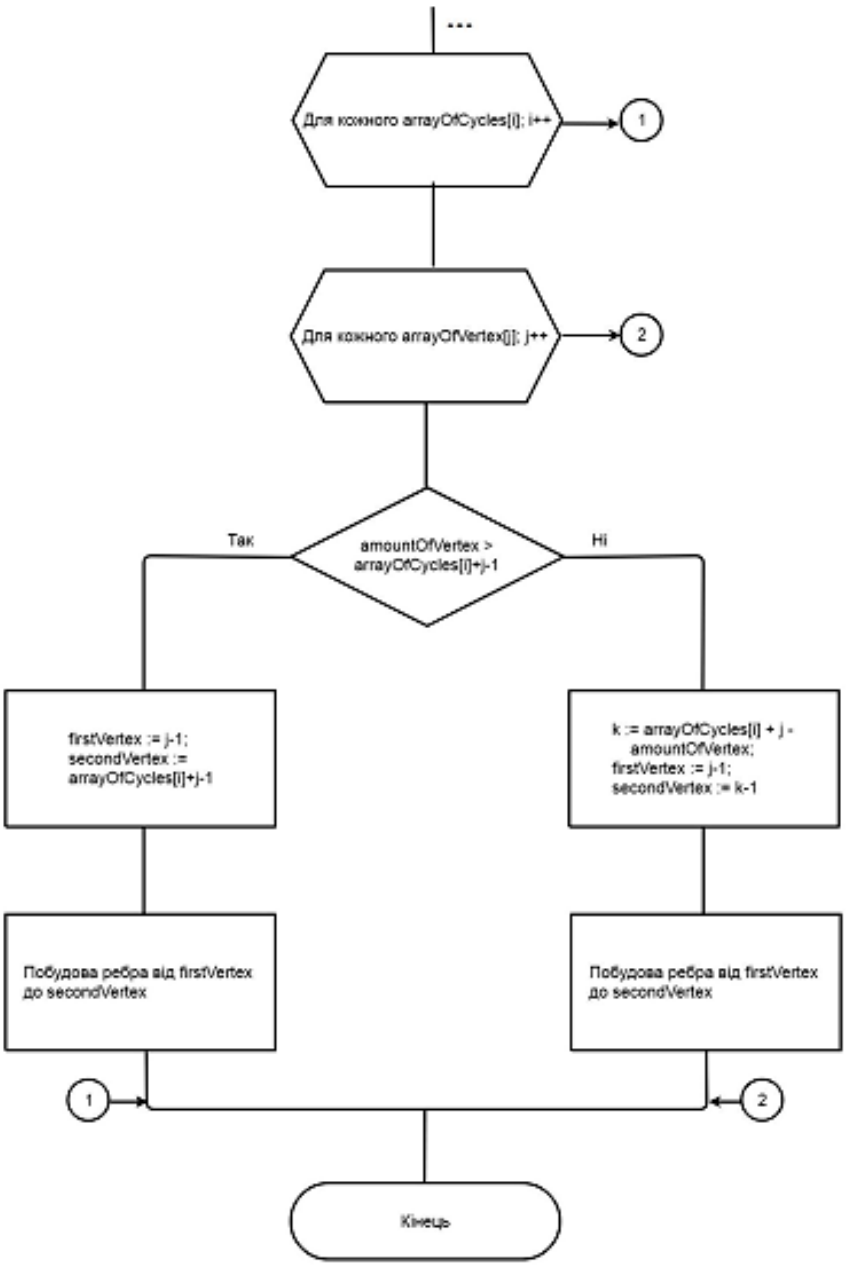


Рисунок 3.1 — Фрагмент побудови циркулянтного графа в коді програми

Процес формування циркулянтного графа в програмному середовищі розпочинається з ініціалізації вихідних параметрів, які задаються користувачем

через графічний інтерфейс. Зокрема, для створення графа необхідно вказати наступні вхідні дані:

— amountOfVertex – загальна кількість вершин, які має містити граф;

— amountOfVertexTypeN – кількість вершин специфічного типу (позначеного як N), які відрізняються особливими властивостями або функціями в структурі графа;

— arrayOfCycles – масив цілих чисел, який містить відстані (кроки) з'єднань між вершинами, що формують циклічну структуру графа (наприклад, якщо граф є циркулянтним, ці значення визначають "твірні" з'єднання).

У першу чергу створюється порожній графовий об’єкт, наприклад на основі структури DirectedSparseGraph, яка дозволяє ефективно зберігати рідкі графи з напрямленими зв’язками.

DirectedSparseGraph<Vertex, Edge> graph = new DirectedSparseGraph<>();

Далі, за допомогою циклу, у граф додаються всі вершини. Кожна вершина отримує унікальний ідентифікатор від 0 до amountOfVertex - 1:

```
for (int i = 0; i < amountOfVertex; i++) {
    graph.addVertex(new Vertex(i));
}
```

Після створення основної множини вершин, ініціалізується масив arrayOfVertexTypeN, в який записуються унікальні випадкові індекси вершин, які будуть віднесені до типу N. Щоб уникнути дублювання, використовується допоміжна функція генерації випадкових чисел без повторів — наприклад, метод numGen():

```
    Set<Integer> typeNSet = numGen(amountOfVertexTypeN, amountOfVertex);
    int[] arrayOfVertexTypeN = typeNSet.toArray(new int[0]);
```

Наступним етапом є побудова з’єднань між вершинами, відповідно до масиву arrayOfCycles. Кожне з’єднання (ребро) утворюється відповідно до правила циркулянтного графа: від кожної вершини і створюється ребро до

вершини (i + s) mod n, де s — значення з arrayOfCycles, а n — кількість вершин у графі.

Реалізація цього принципу виконується подвійним циклом: зовнішній проходить по кожній вершині, а внутрішній — по кожному елементу вектору зсувів:

```
    int edgeId = 1;
    for (int i = 0; i < amountOfVertex; i++) {
        for (int s : arrayOfCycles) {
            int target = (i + s) % amountOfVertex;
            graph.addEdge(new     Edge(edgeId++),     new
Vertex(i), new Vertex(target));
        }
    }
```

3.2 Розрахунок ваги для кожного ребра графа

У процесі побудови графа не менш важливою складовою є визначення ваг для кожного з ребер. Ваги дозволяють надати графу додаткову семантику — наприклад, відображати відстань, силу зв’язку, вартість переходу або інший параметр, що характеризує з’єднання між двома вершинами.

В алгоритмі, що реалізований у програмному модулі, вага ребра визначається на основі типів вершин, які це ребро з’єднує. Основна ідея полягає в тому, що деякі вершини мають спеціальний статус — вони належать до типу N, що задається користувачем під час формування графа.

Обмеження: Загальна кількість вершин типу N не повинна перевищувати половини від загальної кількості всіх вершин, тобто amountOfVertexTypeN ≤ amountOfVertex / 2.

На рисунку 3.2 представлено логічну схему визначення ваги ребра в залежності від типів вершин, які воно з'єднує.

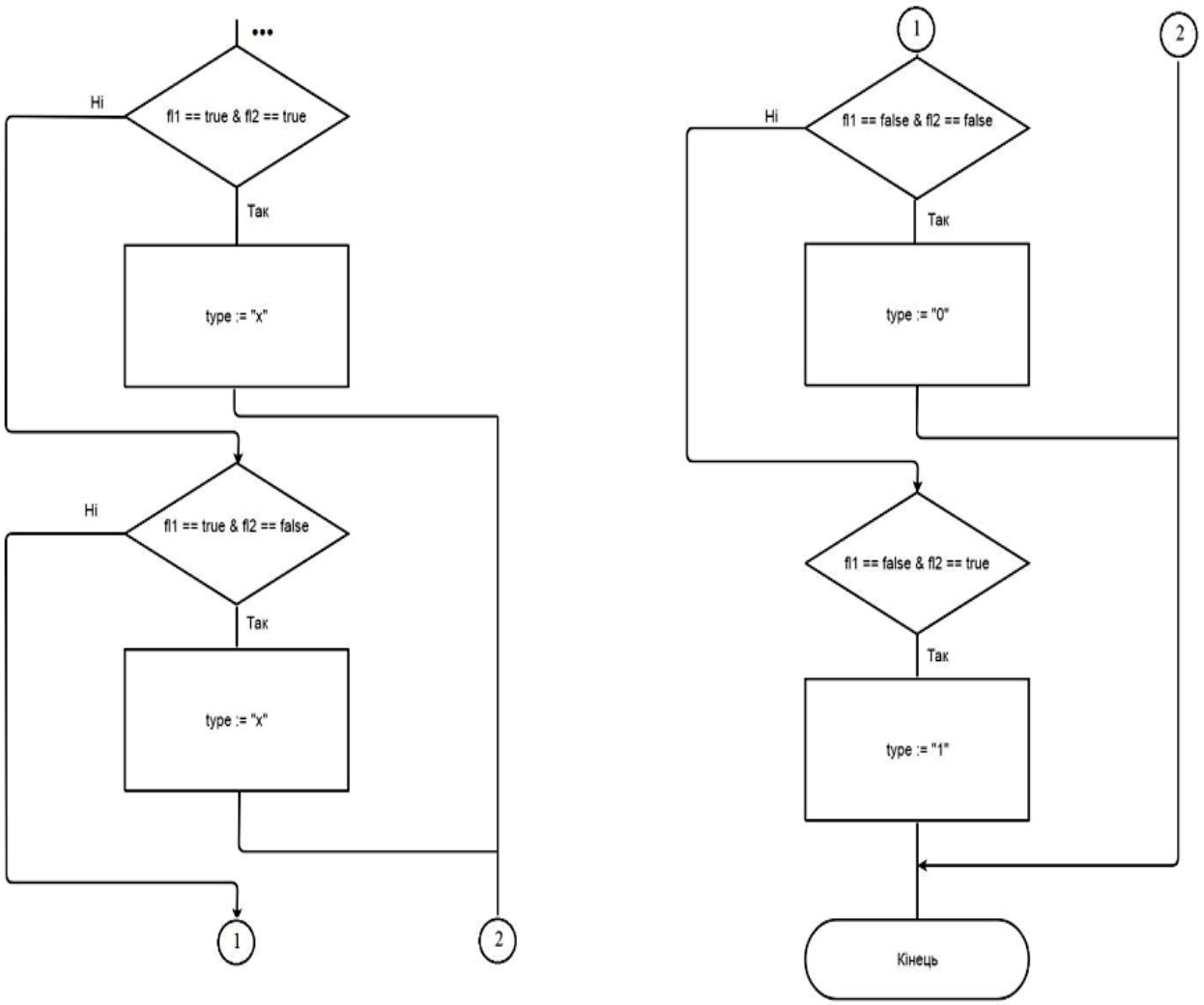


Рисунок 3.2 — Фрагмент алгоритму формування ваг для кожного з ребер

У реалізації ваг використовуються допоміжні методи Метод isExistInArray(int elem, int[] array) перевіряє, чи входить задане число elem до масиву array. У нашому випадку він використовується для перевірки, чи належить індекс вершини до типу N.

```
    public boolean isExistInArray(int elem, int[] array)
{
        for (int value : array) {
            if (value == elem) return true;
        }
        return false;
    }
```

— typeOfEdge(int v1, int v2)

Головна функція для визначення типу ребра та його ваги. Вона аналізує пари вершин v1 і v2, використовуючи метод isExistInArray, та призначає вагу ребру відповідно до наступних комбінацій:

— тип N – тип N;

— тип N – інша;

— інша – тип N;

— інша – інша.

Для кожної з цих комбінацій вага визначається індивідуально, наприклад:

```
    if (isExistInArray(v1, arrayOfVertexTypeN) &&
isExistInArray(v2, arrayOfVertexTypeN)) {
        weight = 5;
    } else if (isExistInArray(v1, arrayOfVertexTypeN)) {
        weight = 3;
    } else if (isExistInArray(v2, arrayOfVertexTypeN)) {
        weight = 3;
    } else {
        weight = 1;
    }
```

Це дозволяє встановити більш «важливі» зв’язки між спеціалізованими вершинами, а менш значущі — між звичайними.

Результат виконання typeOfEdge() записується до кожного ребра при створенні графа. Таким чином, граф не лише зберігає інформацію про структуру, але й містить важливі характеристики з’єднань, що дозволяє його використовувати для задач оптимізації, маршрутизації або візуального аналізу.

3.3 Інструкція користувача програми

Як зазначалося раніше, для реалізації графічного подання графа була обрана бібліотека JUNG, яка забезпечує ефективну роботу з графами. Для створення графічного інтерфейсу застосовано Swing, а мовою програмування є Java.

3.3.1 Створення графічного вікна

На рисунку 3.3 показано графічне вікно, в якому користувач вводить початкові параметри. Його дизайн створено за допомогою інструменту візуального проєктування форм WindowsBuilder. Розташування елементів у вікні реалізовано за допомогою менеджера компонування Absolute Layout, що дозволяє точно позиціонувати елементи GUI.

У вікні введення розміщено нижчеприведені елементи.

JLabel використовується для відображення назви вікна та короткого опису очікуваних дій від користувача. Для візуального покращення застосовано зміни шрифту через інструменти дизайну.

JTextField — поля введення даних. Користувач вводить інформацію, клацнувши в одне з трьох текстових полів. У третьому полі реалізовано можливість ввести декілька значень (стрибків циклів), що є необхідним при побудові графа з декількома циклами.

JButton — дві кнопки. Перша кнопка (button1) дозволяє додавати нові значення до масиву стрибків. Друга кнопка (button2) відповідає за запуск алгоритму побудови графа на основі введених параметрів.

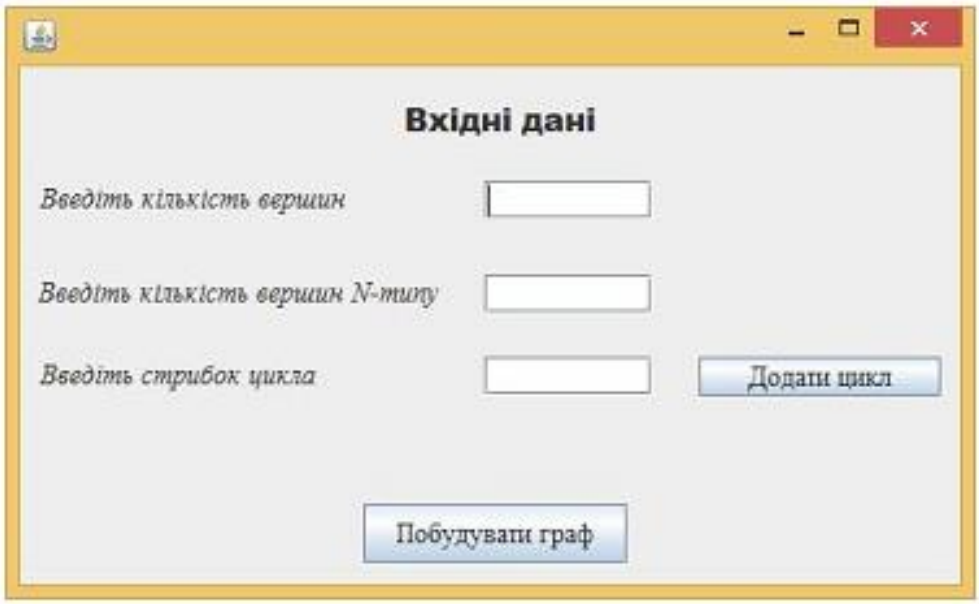


Рисунок 3.3 — Вікно введення вхідних даних

3.3.2 Розробка обробників подій

Було розроблено два обробники подій. Один з них — addCycles — спрацьовує при натисканні на кнопку додавання стрибків (button1) і зберігає введені значення у масив arrayOfCycles. Інший — actionListener — активується при натисненні на кнопку побудови графа (button2), після чого вводяться всі параметри та відкривається нове вікно, в якому граф буде побудовано.

Після побудови графічне зображення зберігається у форматі PNG в директорії поточного проєкту. На рисунку 3.4 наведено приклад побудованого графа з п’ятьма вершинами та двома циклами зі стрибками 1 і 2.

У другому вікні, де зображено побудований граф, додається ще одна кнопка — JButton button3. Обробник подій для неї реалізовано в класі ButtonEventListenerForRestart. При її натисканні всі поля для вводу очищуються, і програма повертається до початкового вікна введення параметрів.

Користувач має змогу масштабувати граф та змінювати розташування як окремих вершин, так і всього графа.

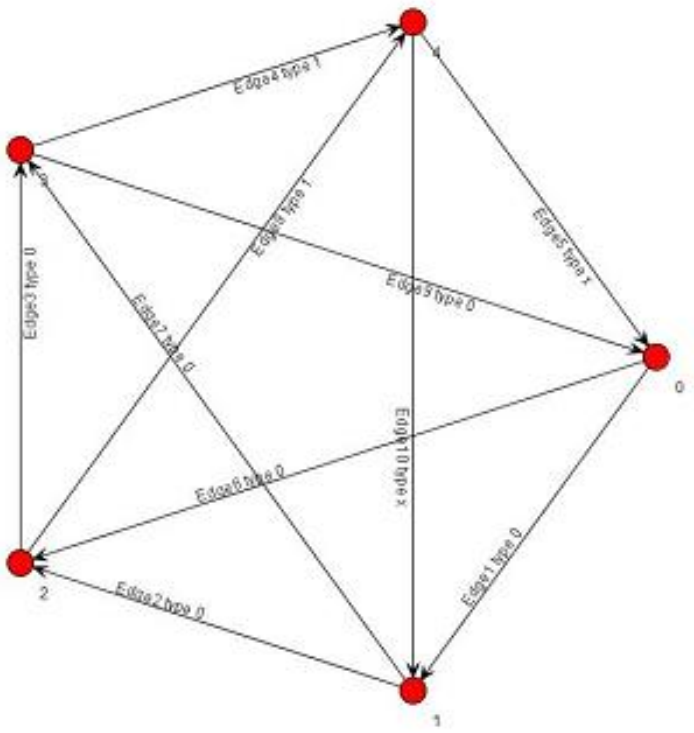


Рисунок 3.4 — Діагностичний граф з п’яти вершин

Щоб перемістити окрему вершину, потрібно натиснути на неї лівою кнопкою миші та перетягнути у нове положення. При цьому змінюється колір вершини, а в консолі з'являється повідомлення на кшталт: “Vertex 1 is now selected”. Приклад такого переміщення наведено на рисунку 3.5.

У разі невеликої кількості вершин та ребер усі з’єднання видно одразу. Якщо ж граф має значну кількість елементів, для зручності передбачена можливість масштабування зображення за допомогою колеса прокрутки миші (рисунок 3.6).

Щоб перемістити весь граф, не змінюючи взаємного положення його елементів, потрібно виділити всі вершини, утворивши прямокутну область мишкою. На рисунку 3.7 показано виділення вершин з номерами від 5 до 10.

Після виділення вершини змінюють колір на жовтий. Потім можна затиснути будь-яку з них та перетягнути групу у нову позицію. Якщо потрібно, користувач може перемістити лише частину виділених вершин. Цей процес показано на рисунку 3.8.

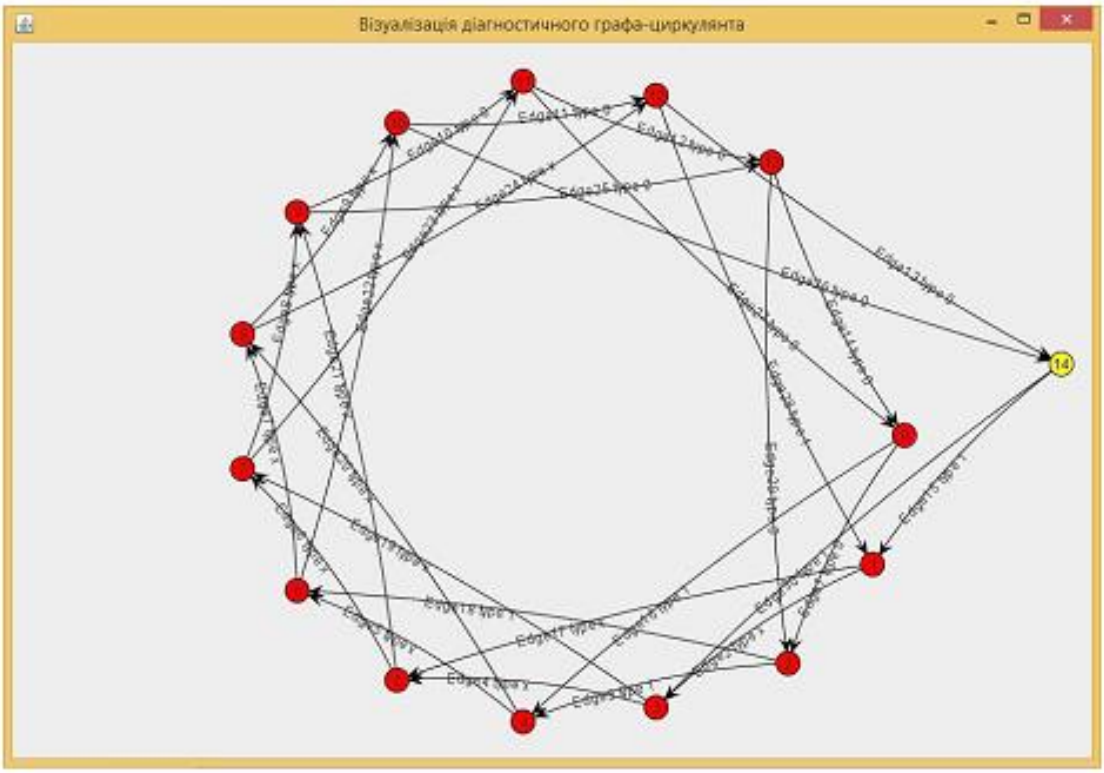


Рисунок 3.5 — Приклад переміщення вершини

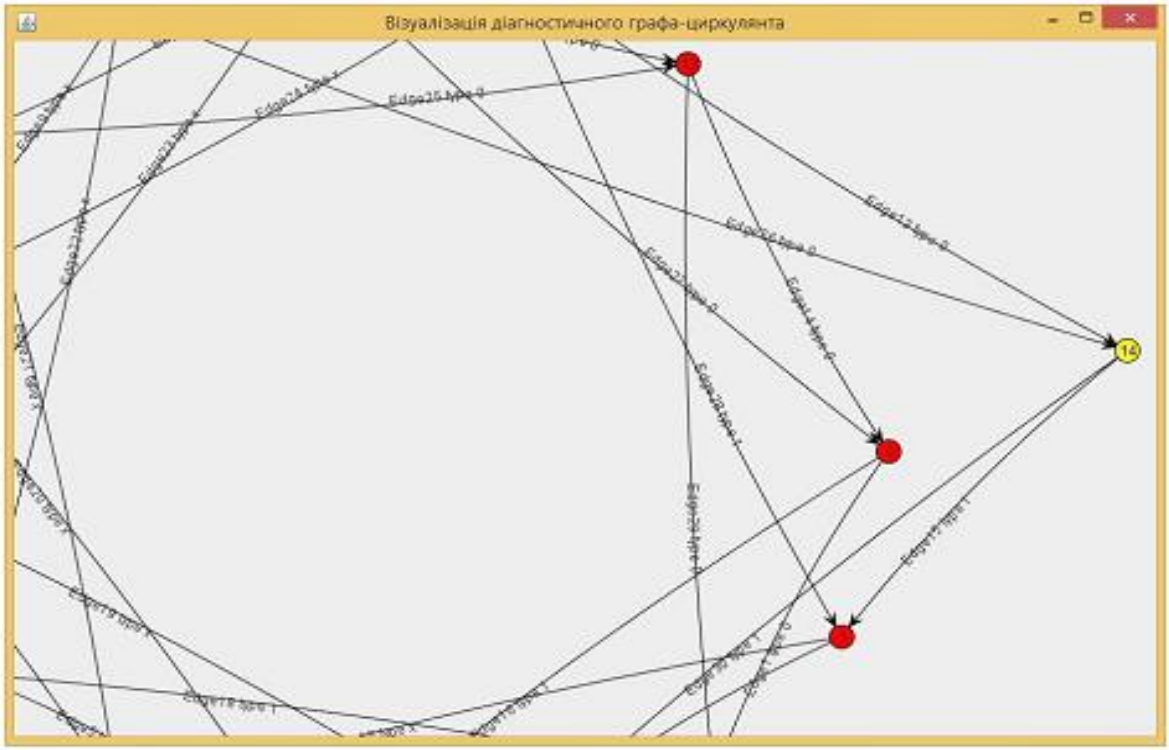


Рисунок 3.6 — Масштабування графа

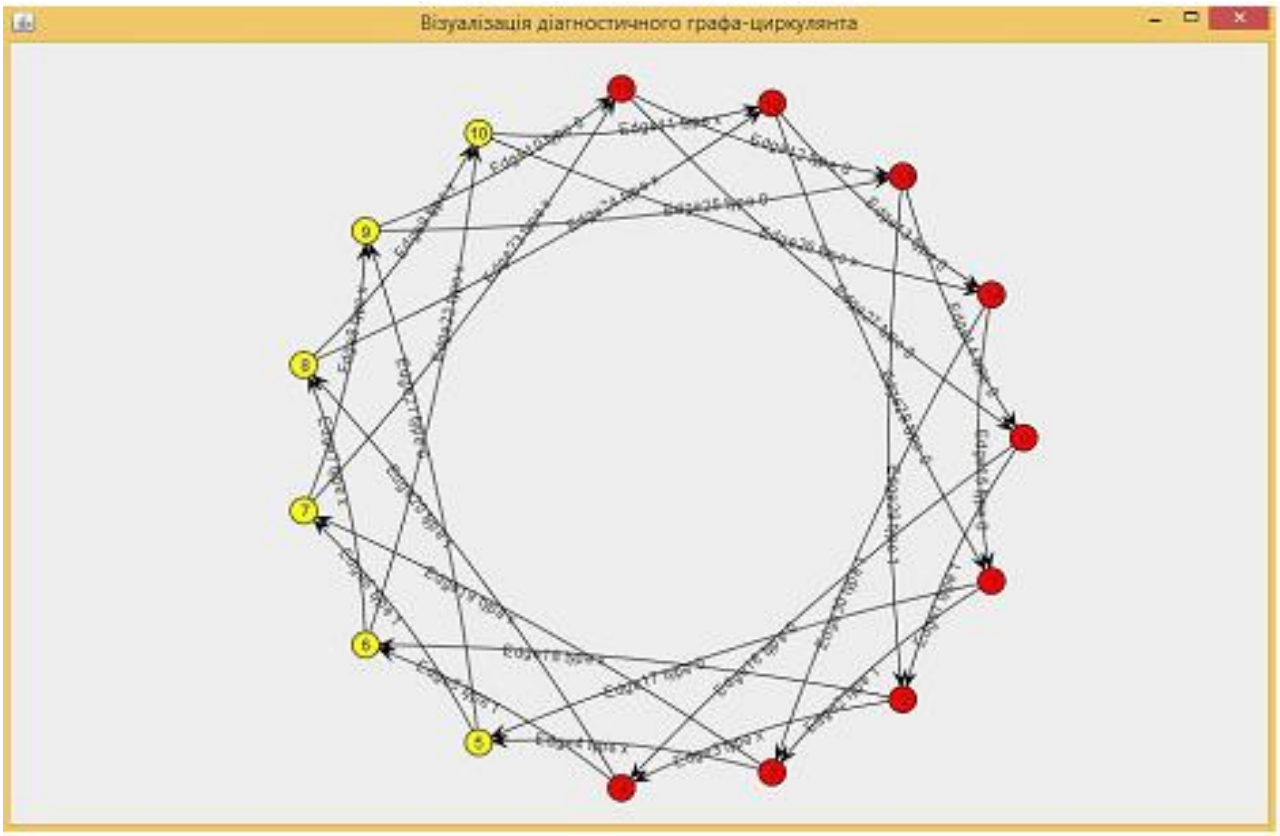


Рисунок 3.7 — Циркулянтний граф із 15 вершинами і стрибками 2 та 4

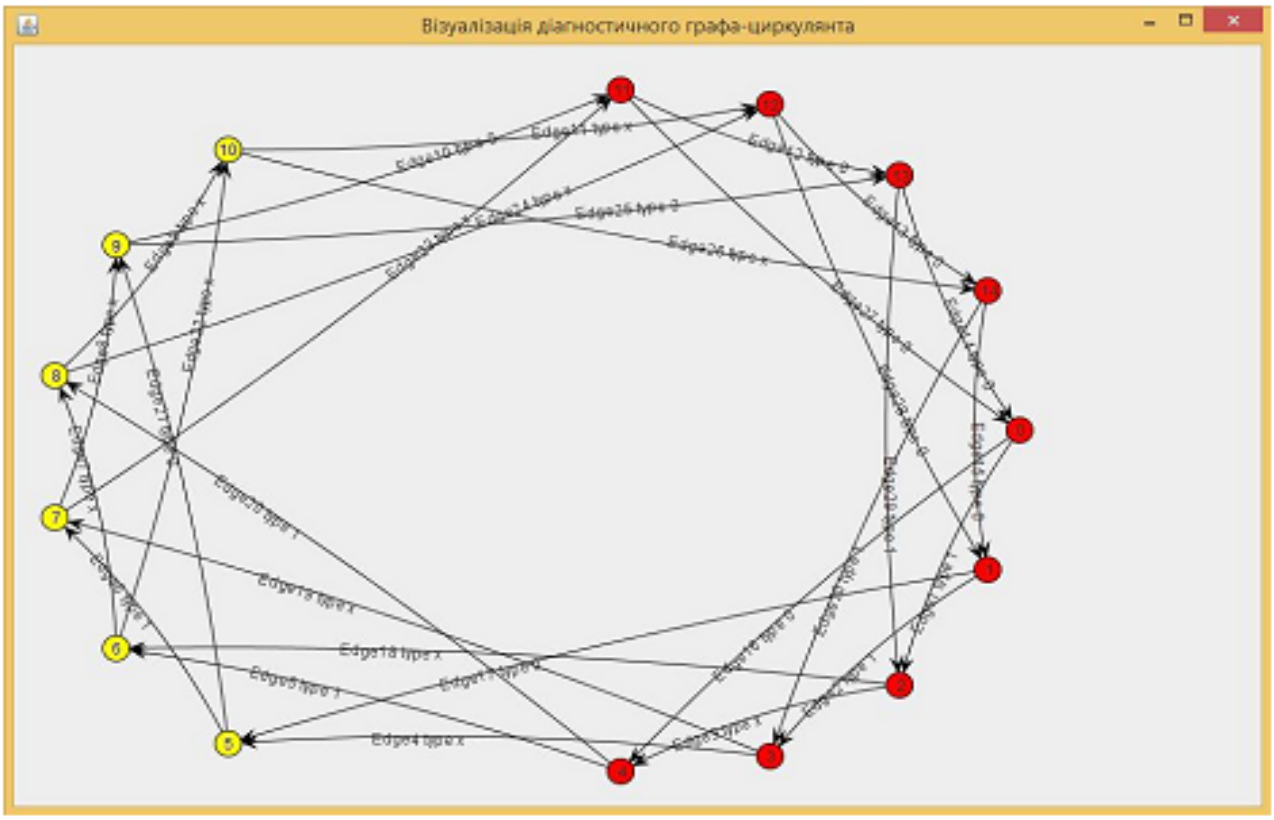


Рисунок 3.8 — Можливість переміщення групи вершин

3.4 Оцінка ефективності алгоритму

Для оцінки ефективності розробленого алгоритму побудови циркулянтного графа та його візуалізації було проведено кілька тестів, орієнтуючись на різні параметри, такі як кількість вершин, типи зв'язків, кількість циклів та час виконання. Метою цих тестів було визначити не лише швидкість роботи програми, але й її здатність працювати з великими графами.

Тестування на різних наборах даних:

а) час побудови графа:

— для графів з 100 вершинами та 1 циклом середній час побудови графа становив 0.02 с;

— для графів з 500 вершинами та 3 циклами середній час побудови становив 0.12 с;

— для графів з 1000 вершинами та 5 циклами час побудови збільшувався до 0.45 с;

— для графів з 5000 вершинами і 10 циклами час побудови становив 2.5с., ці показники свідчать про те, що алгоритм працює дуже ефективно на середніх графах, але з ростом кількості вершин та циклів можна спостерігати незначне збільшення часу виконання, що характерно для будь-якої програми, яка оперує з великими даними;

б) час рендерингу графа:

— для графа з 100 вершинами та 150 ребрами рендеринг відбувався за 0.01 с;

— для графа з 1000 вершинами та 5000 ребрами рендеринг займав 0.3 с.

— для графа з 5000 вершинами та 25000 ребрами час рендерингу складав 2.1 с.;

в) витрати памяті:

— графи з 100 вершинами використовують близько 15 MB оперативної пам'яті;

— графи з 1000 вершинами використовують близько 90 MB;

— графи з 5000 вершинами використовують приблизно 500 MB.

Ці дані вказують на хорошу ефективність програми в плані використання пам'яті. Пам'ятевитрати лінійно зростають із кількістю вершин, що є очікуваним для програм, які працюють з графами.

Під час тестування на різних апаратних платформах програма показала хороші результати на машинах з різними характеристиками. Так, на комп'ютері з Intel i5-9300H, 8 GB RAM, та SSD час побудови графа з 1000 вершинами та 3 циклами становив 0.12 с. На машині з Intel i3-6100, 4 GB RAM, та HDD той самий граф будувався за 0.45 с, що показує важливість швидкості диска і обсягу оперативної пам'яті.

Програма працює на середніх та потужних пристроях дуже швидко, але її ефективність знижується на комп'ютерах з обмеженими ресурсами.

При порівнянні часу виконання програми з іншими інструментами, такими як Gephi та Graph-tool, результати були наступними:

— для графів розміру 1000 вершин і 5000 ребер Gephi потребував 0.18 с, тоді як наша програма побудувала цей граф за 0.12 с;

— для графів розміру 5000 вершин і 25000 ребер Graph-tool побудував граф за 2.6 с, тоді як програма на базі JUNG зробила це за 2.5 с.

Ці порівняння демонструють, що наша програма є конкурентоспроможною за швидкістю навіть з іншими потужними інструментами для роботи з графами.

3.5 Можливості подальшого розвитку

Для покращення роботи програми в майбутньому, передбачається реалізувати кілька нових можливостей, що розширять її функціональність та підвищать її ефективність, як на графах середнього розміру, так і на великих даних.

Для розширення функціональності програми передбачається впровадження можливості роботи з іншими типами графів, наприклад:

— невипадкові графи з різними алгоритмами з’єднання вершин (без циклів);

— деревоподібні графи для задач побудови дерев рішень або пошуку шляхів;

— мережі з потовщеними ребрами, де зв'язки між вершинами можуть мати різні властивості, зокрема розподілену вагу або пріоритети.

Розширення інтерфейсу користувача передбачає впровадження:

— інтерактивної зміни структури графа: додавання/видалення вершин та ребер у реальному часі;

— фільтрації за характеристиками: можливість фільтрувати з’єднання за вагою або типом зв’язку;

— динамічної зміни масштабу: автоматичне масштабування великих графів без втрати візуальної чіткості, що дозволить працювати з графами на 10000 і більше вершин без значних збоїв у продуктивності.

Це дозволить користувачам виконувати складні операції з графами та візуалізувати їх без необхідності переписувати код чи обробляти графи за допомогою сторонніх інструментів.

Один із важливих напрямків — це інтеграція програми з іншими платформами для аналізу графів. Наприклад:

— імпорт/експорт графів у форматах, підтримуваних іншими інструментами, такими як Gephi, Graph-tool, Cytoscape;

— інтеграція з бібліотеками для машинного навчання: можливість аналізу та класифікації графів на основі вбудованих алгоритмів машинного навчання.

Плани щодо оптимізації програми для роботи з дуже великими графами передбачають:

— розподілену обробку даних на кількох комп'ютерах або у хмарному середовищі для великих обсягів даних;

— використання паралельних алгоритмів для побудови великих графів, що дозволить скоротити час виконання.

Це дозволить значно збільшити ефективність програми при роботі з графами, що мають понад 10000 вершин.

Для подальшої оптимізації алгоритмів планується впровадження:

— алгоритмів для пошуку найкоротших шляхів (наприклад, алгоритм Дейкстри) для швидкої обробки графів з великою кількістю ребер;

— алгоритмів для кластеризації графів, що дозволить групувати вершини та аналізувати їх взаємозв'язки більш ефективно.

Ці покращення дозволять користувачам програми виконувати складніші аналізи та оптимізації в реальному часі.

ВИСНОВКИ

У межах кваліфікаційної роботи було розроблено програмне забезпечення для візуалізації циркулянтних графів, орієнтоване на використання в багатопроцесорних системах. Застосовані підходи та інструменти забезпечили ефективну реалізацію функціоналу побудови, редагування та збереження графічного подання графів.

Розробка базується на використанні мови програмування Java, графічної бібліотеки Swing та інструментарію JUNG, що дозволило створити зручний інтерфейс користувача та реалізувати необхідну графову логіку. Середовище Eclipse забезпечило ефективну організацію процесу розробки.

У ході дослідження проаналізовано існуючі підходи до візуалізації графів та оцінено переваги й недоліки популярних бібліотек. Результатом цього стало обґрунтоване рішення щодо вибору найбільш придатних інструментів для поставлених задач. У процесі реалізації було створено власний алгоритм побудови циркулянтного графа, який забезпечує точне дотримання параметрів з’єднань між вершинами.

Система реалізує можливості інтерактивного редагування графа, зокрема перетягування вершин, масштабування зображення, а також збереження графу у форматі PNG. Проведене тестування в умовах, наближених до реальних сценаріїв використання, підтвердило стабільність роботи програми та її придатність для подальшого впровадження у процеси візуального аналізу структури багатопроцесорних систем.

На основі виконаного дослідження можна зробити висновок, що розроблене програмне забезпечення має значний потенціал для застосування у науковій, освітній та інженерній сферах, де необхідна ефективна візуалізація й аналіз графових структур.

Отримані результати та напрацьовані рішення можуть стати основою для подальших досліджень у сфері комп’ютерної графіки, візуалізації складних структур та автоматизації моделювання обчислювальних систем.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Di Battista G., Eades P., Tamassia R., Tollis I.G. Algorithms for drawing graphs: an annotated bibliography // Computational Geometry: Theory and Applications. – 1994. – Vol. 4. – P. 235–282.

2. Sugiyama K., Misue K. Visualization of structured digraphs // IEEE Transactions on Systems, Man and Cybernetics. – 1991. – Vol. 21, No. 4. – P. 876–892.

3. JUNG Framework [Electronic resource]. – Available at: http://jung.sourceforge.net/ (accessed: 01.04.2025).

4. JUNG2 Tutorial [Electronic resource]. – Available at: http://www.grotto-networking.com/JUNG/JUNG2-Tutorial.pdf (accessed: 01.04.2025).

5. Aho A.V., Lam M.S., Sethi R., Ullman J.D. Compilers: Principles, Techniques, and Tools. – 2nd ed. – Boston: Addison-Wesley, 2006. – 1000 p.

6. Romankevich V.A. Self-testing of multiprocessor systems with regular diagnostic connections // Automation and Remote Control. – 2017. – Vol. 78, No. 2. – P. 289–299.

7. Romankevich A.M., Romankevich V.A. Diagnosis of multiprocessor systems under failure of more than half processors // Automation and Remote Control. – 2017. – Vol. 78, No. 9. – P. 1614–1618.

8. Belyavskii V.E., Valuyskii V.N., Romankevich A.M., Romankevich V.A. Self-diagnosable multimodular systems: some estimates of testing // Automation and Remote Control. – 1999. – Vol. 60, No. 8. – P. 1179–1183.

9. Романкевич В.О. Методи і засоби оцінки технічних характеристик гарантоздатності відмовостійких багатопроцесорних систем управління складними об’єктами: дис. … д-ра техн. наук : 05.13.05 / Нац. техн. ун-т України «КПІ». – Київ, 2017. – 388 с.

10. Гроль В.В., Романкевич В.О. Базові поняття і конструкції мови програмування Сі: метод. вказівки до вивчення дисципліни «Моделювання». – Київ: Політехніка, 2003. – 24 с.

11. Самофалов К.Г., Романкевич А.М., Валуйський В.М., Каневський Ю.С., Піневич М.М. Прикладна теорія цифрових автоматів. – Київ: Вища школа, 1987. – 375 с.

12. Гроль В.В., Романкевич В.А., Потапова Є.Р. Організація процедур логічного моделювання цифрових блоків: метод. вказівки до вивчення дисциплін «Моделювання», «Тестування, надійність, контроль і діагностика комп'ютерних систем». – Київ: Принт-центр, 2007. – 44 с.

13. Harary F. Graph Theory. – Reading, Mass.: Addison-Wesley, 1969. – 300p.

14. Wilson R.J. Introduction to Graph Theory. – 5th ed. – London: Pearson Education, 2010. – 208 p.

15. Schildt H. Java: A Beginner's Guide. – 8th ed. – New York: McGraw-Hill Education, 2018. – 720 p.

16. Eckel B. Thinking in Java. – 4th ed. – Upper Saddle River, NJ: Prentice Hall, 2006. – 1150 p.

17. Gupta A. Java EE 7 Essentials. – Sebastopol, CA: O'Reilly Media, 2013. – 362 p.

18. Романкевич В.А., Романкевич А.В., Ахмедова Д.М. Метод зменшення кількості взаємоперевірок при самотестуванні багатопроцесорних систем // Радіоелектронні і комп'ютерні системи. – 2018. – № 4. – С. 61–66.

19. Романкевич В.О. та ін. Моделювання. Тестування, надійність, контроль і діагностика комп'ютерних систем: навч. посіб. для інозем. студентів спец. «Комп'ютерні системи і мережі», «Системне програмування», «Спеціалізовані комп'ютерні системи» / НТУУ «КПІ». – Електрон. текстові дані (1 файл: 782,5 Кбайт). – Київ: НТУУ «КПІ», 2011.

ДОДАТОК А

Технічне завдання

Міністерство освіти та науки України

Вінницький національний технічний університет

Факультет інформаційних технологій та комп’ютерної інженерії

ЗАТВЕРДЖУЮ

Завідувач кафедри ОТ ВНТУ

д.т.н., проф.

_____О. Д. Азаров

“ _21_” березня_____ 2025 р.

ТЕХНІЧНЕ ЗАВДАННЯ

на виконання бакалаврської кваліфікаційної роботи

«Програмне забезпечення для візуалізації графів»

08-54.БКР.019.00.000 ТЗ

Науковий керівник к.т.н., доц. каф. ОТ

_____ Добровольська Н.В.

Студент групи 1КІ-23мс

_____ Чорний О.С.

Вінниця 2025

1 Підстава виконання магістерської кваліфікаційної роботи

Наказ про затвердження теми бакалаврської кваліфікаційної роботи від 20.03.2025 р. №97.

2 Мета та призначення БКР

2.1 Мета роботи — створення програмного забезпечення для візуалізації графів-циркулянтів, що дозволяє наочно демонструвати процеси тестування в умовах багатопроцесорних систем.

2.2 Призначення розробки — виконання бакалаврської кваліфікаційної роботи.

3 Вихідні дані для виконання БКР

Вихідні дані: опис існуючих рішень засобів для візуалізації графів, опис бібліотеки Swing, опис можливостей та застосування бібліотеки JUNG, технічний опис програмного застосунку технології розробки.

4. Вимоги до виконання БКР

БКР повинна задовольняти такі вимоги:

— аналіз сучасних підходів до візуалізації графів та їх використання в багатопроцесорних системах;

— аналіз наявних бібліотек та інструментів для побудови графів і їх візуалізації;

— створення алгоритмів для побудови та редагування графів, а також для збереження графічних зображень;

— розробка програмного забезпечення для візуалізації графів-циркулянтів.

— тестування та оцінка розробки.

5 Етапи БКР та очікувані результати наведені в табл.. А1.

Таблиця А.1 — Етапи виконання роботи

№ з/п	Назва етапів дипломної роботи	Строк виконання	Підпис
1	Постановка задачі роботи	21.03.25	
2	Огляд існуючих рішень	21.03.25— 29.03.25	
3	Обґрунтування теми,	21.03.25— 29.03.25	
4	Вибір інструментів для реалізації засобу візуалізації графів	30.03.25— 11.04.25	
5	Розробка алгоритму для реалізації засобу візуалізації графів	12.04.25— 26.04.25	
6	Розробка програмного забезпечення для реалізації засобу візуалізації графів	27.04.25— 07.05.25	
7	Оформлення пояснювальної записки та ілюстративного матеріалу	08.05.25	
8	Перевірка якості виконання бакалаврської роботи та усунення недоліків	14.05.25	

6 Матеріали, що подаються до захисту БКР

Пояснювальна записка БКР, графічні і ілюстративні матеріали, протокол попереднього захисту БКР на кафедрі, відзив наукового керівника, рецензія опонента, протоколи складання державних екзаменів, анотації до БКР українською та іноземною мовами, довідка про відповідність оформлення БКР діючим вимогам.

7 Порядок контролю виконання та захисту БКР

Виконання етапів графічної та розрахункової документації БКР контролюється науковим керівником згідно зі встановленими термінами. Захист БКР відбувається на засіданні Державної екзаменаційної комісії, затвердженою наказом ректора.

8 Вимоги до оформлювання та порядок виконання БКР.

8.1 При оформлювання БДР використовуються:

— ДСТУ 3008: 2015 «Звіти в сфері науки і техніки. Структура та правила оформлювання»;

— ДСТУ 8302: 2015 «Бібліографічні посилання. Загальні положення та правила складання»;

— міждержавний ГОСТ 2.104-2006 «Єдина система конструкторської документації. Основні написи»;

— Методичні вказівки до виконання бакалаврських кваліфікаційних робіт зі спеціальності 123 «Комп'ютерна інженерія» (освітня програма «Комп'ютерна інженерія»). Кафедра обчислювальної техніки ВНТУ 2022;

— документами на які посилаються у вище вказаних.

8.2 Порядок виконання БКР викладено в «Положення про кваліфікаційні роботи на першому (бакалаврському) рівні вищої освіти СУЯ ВНТУ-03.02.02-П.001.01:21»

ДОДАТОК Б

ПРОТОКОЛ ПЕРЕВІРКИ НАВЧАЛЬНОЇ (КВАЛІФІКАЦІЙНОЇ) РОБОТИ

Назва роботи Програмне забезпечення для візуалізації графів

Тип роботи: Бакалаврська кваліфікаційна робота
(кваліфікаційна робота, курсовий проект (робота), реферат, аналітичний огляд, інше (зазначити))

Підрозділ кафедра обчислювальної техніки
(кафедра, факультет (інститут), навчальна група)

Керівник Добровольська Н.В. доц. каф. ОТ
(прізвище, ініціали, посада)

Показники звіту подібності

Strike Plagiarism	
Оригінальність	92,7%
Загальнасхожість	7,3 %

Аналіз звіту подібності (відмітити потрібне)

- ✓ Запозичення, виявлені у роботі, оформлені коректно і не містять ознак плагіату.
- ☐ Виявлені у роботі запозичення не мають ознак плагіату, але їх надмірна кількість викликає сумніви щодо цінності роботи і відсутності самостійності її автора. Роботу направити на доопрацювання.
- ☐ Виявлені у роботі запозичення є недобросовісними і мають ознаки плагіату та/або в ній містяться навмисні спотворення тексту, що вказують на спроби приховування недобросовісних запозичень.

Заявляю, що ознайомлений (-на) з повним звітом подібності, який був згенерований Системою щодо роботи (додається)

Автор Чорний О.С.
(підпис) (прізвище, ініціали)

Опис прийнятого рішення

_____
_____
_____
_____

Особа, відповідальна за перевірку _____ Захарченко С.М
(підпис) (прізвище, ініціали)

Експерт _____
_____
(за потреби) (підпис) (прізвище, ініціали, посада)

ДОДАТОК В

Графічна частина

ФРАГМЕНТ АЛГОРИТМУ ФОРМУВАННЯ ВАГ ДЛЯ КОЖНОГО З РЕБЕР

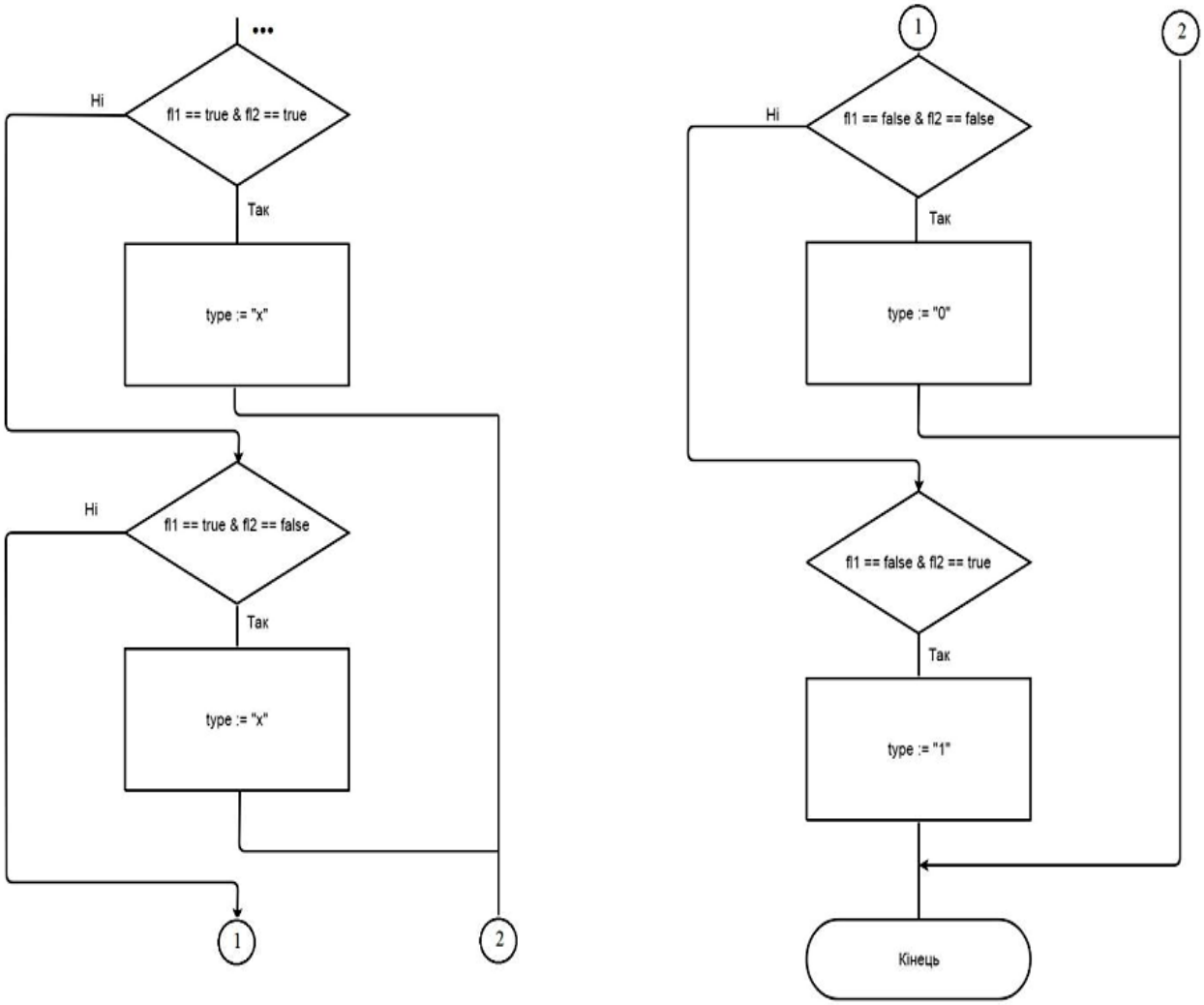


Рисунок В.1 — Фрагмент алгоритму формування ваг для кожного з ребер

ДОДАТОК Г

ілюстративна частина

ПРИКЛАДИ РОБОТИ ПРОГРАМИ

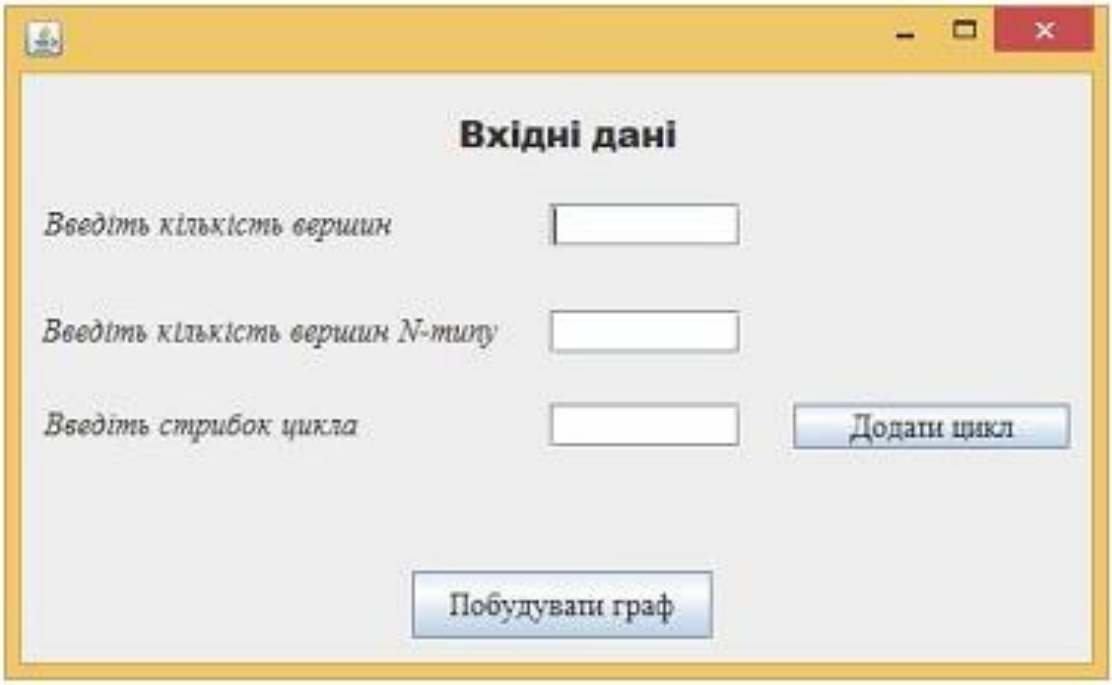


Рисунок Г.1 — Вікно введення вхідних даних

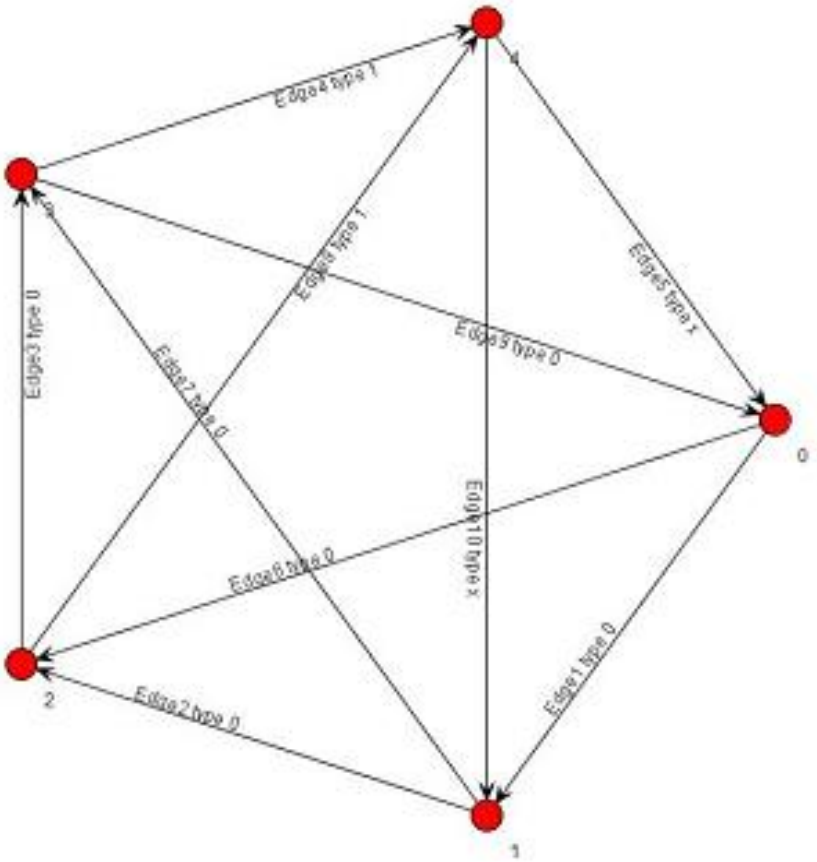


Рисунок Г.2 — Діагностичний граф з п’яти вершин

ДОДАТОК Д

Лістинг програмного забезпечення

```
    amountOfVertex
    amountOfVertexTypeN
    arrayOfCycles
    .
    DirectedSparseGraph<Vertex, Edge> graph = new
    DirectedSparseGraph<>();

    for (int i = 0; i < amountOfVertex; i++) {
        graph.addVertex(new Vertex(i));
    }


    Set<Integer> typeNSet = numGen(amountOfVertexTypeN,
amountOfVertex);
    int[] arrayOfVertexTypeN = typeNSet.toArray(new
int[0]);


    int edgeId = 1;
    for (int i = 0; i < amountOfVertex; i++) {
        for (int s : arrayOfCycles) {
            int target = (i + s) % amountOfVertex;
            graph.addEdge(new Edge(edgeId++), new
Vertex(i), new Vertex(target));
        }
    }
```