НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»
Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці

ДО ЗАХИСТУ ДОПУЩЕНО
В.о. завідувача кафедри
_____ Олександр КОВАЛЬ
«___»_____2025 р.

Дипломна робота

на здобуття ступеня бакалавра
за освітньо-професійною програмою «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем в енергетиці»
спеціальності 121 Інженерія програмного забезпечення

на тему: « Програмне забезпечення для обчислення параметрів функціональної стійкості інтелектуальної системи прийняття рішень в умовах деструктивних впливів в критичних інфраструктурах»

Виконала:
студентка IV курсу, групи ТВ-13
Рябець Катерина Олександрівна _____ _____
(прізвище, ім'я, по батькові) (підпис)

Керівник:
кандидат технічних наук, доцент, доцент кафедри
інженерії програмного забезпечення в енергетиці
Шуклін Г.В. _____ _____
(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Рецензент:

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

Засвідчую, що у цій дипломній роботі немає запозичень із праць інших авторів без відповідних посилань.
Студентка _____
(підпис)

Київ – 2025

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці
Рівень вищої освіти перший (бакалаврський)
Спеціальність 121 Інженерія програмного забезпечення
Освітньо-професійна програма «Інженерія програмного забезпечення інтелектуальних кібер-фізичних систем в енергетиці»

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Олександр КОВАЛЬ
«___»_____2025р.

ЗАВДАННЯ

на дипломну роботу студенту

Рябець Катерині Олександрівні
(прізвище, ім'я, по батькові)

1. Тема роботи: «Програмне забезпечення для обчислення параметрів функціональної стійкості інтелектуальної системи прийняття рішень в умовах деструктивних впливів в критичних інфраструктурах»
керівник роботи Шуклін Герман Вікторович кандидат технічних наук, доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «__» _____ 2025р. №_____
2. Строк подання студентом роботи «_9_» червня 2025р.
3. Вихідні дані до роботи: мова програмування Python, середовище розробки PyCharm
4. Зміст (дипломної роботи) пояснювальної записки (перелік завдань, які потрібно розробити): розробити програмне забезпечення для обчислення параметрів функціональної стійкості інтелектуальної системи прийняття рішень в умовах деструктивних впливів в критичних інфраструктурах.
5. Перелік ілюстративного матеріалу: структура СППР, діаграма варіантів використання, діаграма класів, алгоритм розрахунків за методом простих ланцюгів, алгоритм розрахунків за методом Монте-Карло, графічний інтерфейс програми.
6. Дата видачі завдання «31» жовтня 2024р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Строки виконання етапів роботи	Примітка
1	Отримання завдання	30.10.2024	
2	Дослідження предметної області	31.10.2024 – 01.11.2024	
3	Постановка вимог до проєктування системи	02.11.2024 – 01.12.2024	
4	Розробка алгоритму програмної системи	02.12.2024 – 12.01.2025	
5	Розробка програмного продукту	13.01.2025 – 11.05.2025	
6	Тестування	12.05.2025 – 15.05.2025	
7	Захист програмного продукту	13.05.2025	
8	Оформлення дипломної роботи	19.05.2025 – 01.06.2025	
9	Передзахист	02.06.2025 – 06.06.2025	
10	Захист	16.06.2025 – 27.06.2025	

Студент _____ Катерина Рябець
(підпис) (ім’я, прізвище)

Керівник роботи _____ Герман Шуклін
(підпис) (ім’я, прізвище)

РЕФЕРАТ

Структура та обсяг дипломної роботи. Робота містить 58 сторінок, 15 рисунків, 2 додатки та 24 посилання.

Метою роботи є розробка програмного засобу для обчислення параметрів функціональної стійкості інтелектуальної системи підтримки прийняття рішень (ІСППР) в умовах деструктивних впливів в критичних інфраструктурах.

Для досягнення мети було виконано наступні задачі;

– проаналізовано поняття функціональної стійкості та визначено її основні показники ;

– обрано математичну модель для представлення архітектури ІСППР;

– обрано методи оцінки імовірності зв’язності двох модулів системи;

– розроблено програмний застосунок, який, використовуючи обрані методи оцінки параметрів функціональної стійкості, дає можливість оцінити функціональну стійкість системи.

Практичне значення одержаних результатів полягає в отриманні оцінки функціональної стійкості системи, яка дозволяє виявляти слабкі місця в архітектурі системи та приймати обґрунтовані рішення щодо її вдосконалення.

Ключові слова: функціональна стійкість, інтелектуальна система підтримки прийняття рішень, надійність.

ABSTRACT

Structure and scope of the thesis. The work contains 58 pages, 15 figures, 2 appendices and 24 references.

The purpose of the work is to develop a software tool for calculating the functional resilience parameters of an Intelligent Decision Support System (IDSS) under destructive impacts in critical infrastructures. To achieve this goal, the following tasks were completed:

– the concept of functional resilience was analyzed, and its main indicators were identified;

– a mathematical model was selected to represent the architecture of the IDSS;

– methods for evaluating the connectivity probability between two system modules were chosen;

– a software application was developed that, using the selected methods for assessing functional resilience parameters, allows evaluating the overall functional resilience of the system.

The practical value the obtained results lies in providing an assessment of the system's functional resilience, which allows identifying weak points in the system architecture and making informed decisions regarding its improvement.

Keywords: functional resilience, intelligent decision support systems, reliability.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

СППР	Система підтримки прийняття рішень
ІСППР	Інтелектуальна система підтримки прийняття рішень
ФС	Функціональна стійкість
IDE	(англ. Integrated Development Environment) інтегроване середовище розробки
DFS	(англ. Depth-First Search) пошук у глибину
BFS	(англ. Breadth-First Search) пошук у ширину
ПЛ	Простий ланцюг
ПСМ	Проста січна множина
СПЛ	Сукупний простий ланцюг

ВСТУП

Інтелектуальні системи підтримки прийняття рішень (ІСППР) усе частіше впроваджуються в критично важливих сферах, зокрема в енергетиці, транспорті, обороні, авіації, охороні здоров’я тощо. Вони виконують функції аналізу, контролю, управління та оптимізації складних технічних процесів, забезпечуючи оперативне прийняття рішень у динамічних і часто нестабільних умовах.

Однак через зростаючу складність, вони дедалі більше піддаються впливу деструктивних факторів, причому як внутрішніх (відмови компонентів, помилки програмного забезпечення, перевантаження), так і зовнішніх (кібератаки, збої електроживлення, природні або техногенні катастрофи).

У зв’язку з цим, оцінка функціональної стійкості ІСППР до введення в експлуатацію набуває особливої важливості. Така оцінка дозволяє заздалегідь виявити вразливі місця в архітектурі системи, змоделювати поведінку при впливі деструктивних факторів, визначити мінімальний рівень надмірності, необхідний для збереження ключових функцій.

Тому метою цієї роботи є розробка програмного засобу, який дозволяє моделювати архітектуру ІСППР у вигляді графа, задавати ймовірності справності станів компонентів та аналізувати функціональну стійкість системи.

У цій роботі розглянуто інтелектуальні системи підтримки прийняття рішень, їхнє використання в енергетичній галузі, поняття функціональної стійкості , обрано модель та розглянуто спосіб оцінки функціональної стійкості ІСППР, також на основі дослідженого матеріалу, було створено програмний продукт, описано його архітектуру, функціонування та роботу з ним.

1 ОПИС ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Постановка задачі

Метою цієї роботи є розробка програмного засобу для розрахунку функціональної стійкості інтелектуальної системи підтримки прийняття рішень (ІСППР) в умовах деструктивних впливів в критичних інфраструктурах.

Розроблене програмне забезпечення призначене для оцінки функціональної стійкості (ФУ) ІСППР в галузі енергетики. Основним інструментом є редактор графів, який дозволяє моделювати структуру системи, задавати ймовірності безвідмовної роботи елементів та зв’язків між ними, будувати матриці зв’язності та виконувати розрахунок показників стійкості.

Потенційними користувачами програми є інженери та аналітики, які займаються розрахунками надійності та стійкості інформаційних систем, розробники інтелектуальних систем підтримки прийняття рішень, студенти та викладачі технічних спеціальностей для навчальних цілей.

Мовою написання програмного забезпечення обрано Python. При розробці використовувалися бібліотеки NetworkX, Matplotlib, Tkinter, Pandas. Для написання коду використано інтегроване середовище розробки PyCharm.

Програма надає можливість створення та редагування графа, що відображає ІСППР, виконання обчислень на основі введеного графа та відображення результатів розрахунків у зручній формі.

1.2 Інтелектуальні системи підтримки прийняття рішень

1.2.1 Означення та класифікація систем підтримки прийняття рішень

Системи підтримки прийняття рішень (СППР) [14] – це широкий клас інтерактивних комп’ютерних систем, які допомагають приймати рішення, використовуючи дані, моделі та знання для вирішення слабо структурованих, погано структурованих або неструктурованих задач.

Структурно СППР складаються з особи, що приймає рішення, вхідних даних, обробник та вихідні дані. Структуру СППР наведено на рисунку 1.1.

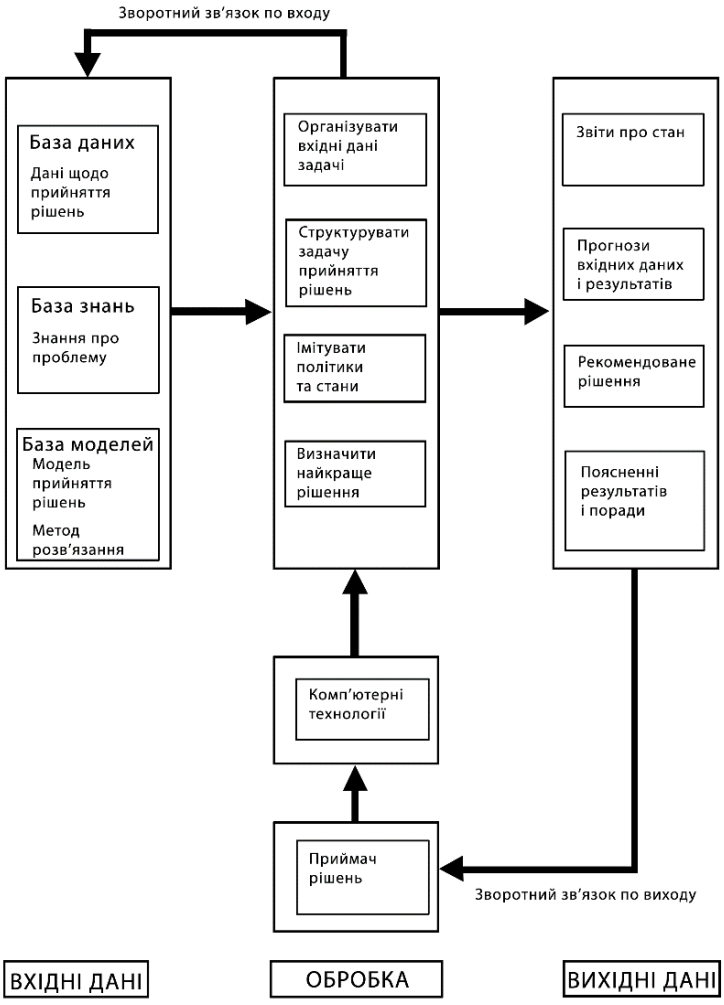

Рисунок 1.1 – Структура СППР

До вхідних даних належать бази даних, база знань та база моделей. У базі даних знаходяться дані щодо вирішуваної проблеми. База знань може містити, наприклад, рекомендації з вибору альтернативи. База моделей містить алгоритми, формальні моделі, методології для отримання рішень.

Обробка полягає у використанні моделей прийняття рішень для симуляції або аналізу різних станів та пошуку найкращого рішення з урахуванням обмежень. Результатом цього етапу можуть стати додаткові вхідні дані.

Вихідні дані можуть включати прогнози, пояснення, обґрунтування рекомендацій та, власне, самі рекомендації.

Існує багато різних класифікацій ССПР за різними ознаками[1, 8, 15]. Надалі буде їх розглянуто.

За характером взаємодії з користувачем СППР поділяються на :

– активні СППР (active DSS) – мають можливість самостійно формувати рекомендації;

– пасивні СППР (passive DSS) – не формують рекомендації, а лише надають інформацію, яка допомагає в процесі прийняття рішень;

– кооперативні СППР (cooperative DSS) – надають можливість користувачеві редагувати надані системою рекомендації і повторно їх опрацьовують.

За способом допомоги [15] СППР поділяють на :

– керовані даними СППР (data-driven DSS);

– керовані моделями СППР (model-driven DSS);

– керовані знаннями СППР (knowledge-driven DSS);

– орієнтовані на документи СППР (document-driven DSS);

– орієнтовані на комунікації СППР (communication-driven DSS);

– web-орієнтовані СППР (web-based DSS).

Системи підтримки прийняття рішень, керовані даними (data-driven DSS) – це системи, які допомагають приймати обґрунтовані рішення на основі аналізу великих обсягів структурованих даних. Вони забезпечують доступ до даних, їх

обробку, аналіз і візуалізацію для подальшого використання в управлінні та плануванні. Важливу роль у таких системах відіграють технології бізнес-аналітики (Business Intelligence, BI) – програмні інструменти, які дозволяють користувачам отримувати доступ до потрібних даних, виконувати детальний аналіз, створювати інтерактивні звіти та інформаційні панелі для ухвалення рішень у зручній візуальній формі. Разом з BI використовується технологія оперативної аналітичної обробки даних (OLAP), яка дозволяє швидко досліджувати багатовимірні дані, фільтрувати їх за різними критеріями (час, регіон, товар тощо) та формувати детальні зведення.

Керовані моделями СППР (model-driven DSS) – спеціально розроблені моделі, створені відповідно до параметрів і вимог, заданих користувачем, і націлені на аналіз конкретних ситуацій для підтримки процесу прийняття рішень. Основою таких систем є аналітичні інструменти, які базуються на математичних моделях. Ці системи не потребуються великого обсягу даних, а база даних або взагалі не потрібна, або використовується мінімально. В моделюванні прийняття рішень застосовуються, зокрема, методи Монте-Карло, агентне та мультиагентне моделювання, візуальні моделі імітації.

СППР, керовані знаннями (knowledge-driven DSS), виникли на основі інтелектуальних систем підтримки рішень і тісно пов’язані з технологіями штучного інтелекту. Головна їхня функція полягає не лише в аналізі даних, а й у наданні користувачеві обґрунтованих порад та рекомендацій на основі вже наявних знань або штучного інтелекту.

Системи підтримки прийняття рішень, орієнтовані на документи (document-driven DSS) включать багато інструментів для роботи з документами. Наприклад, пошукові системи, інтегровані з document-driven DSS, допомагають знаходити потрібні документи, що підтримують прийняття рішень. Документи в таких системах зазвичай мають неоднакову структуру, тому потрібні спеціальні методи пошуку та обробки неструктурованих даних.

СППР, орієнтовані на комунікації (communication-driven DSS) базуються на використанні мережевих технологій та електронних засобів, щоб об'єднати ключових учасників процесу прийняття рішень в єдине середовище, де доступна вся необхідна інформація, дані та ресурси. Такий підхід має назву групова система підтримки прийняття рішень (GDSS – Group Decision Support System). Він включає різноманітні інструменти, які використовуються для формування рішень, структурування проблем, планування, моделювання варіантів дій. Іншим прикладом СППР, орієнтованої на комунікацію є колаборативна система підтримки прийняття рішень (collaborative DSS, CDSS), яка поєднує в собі елементи комп'ютерних систем, сприяє взаємодії між учасниками процесу, дозволяє групі спільно працювати над вирішенням проблем.

Web-орієнтовані СППР розглядаються як складні системи, які значно розширили свої можливості завдяки використанню Всесвітньої павутини (World Wide Web) та Інтернету у процесі підтримки прийняття рішень.

Класифікація СППР за масштабом використання [8]:

– корпоративні СППР (enterprise-wide DSS) – це СППР рівня підприємства, які підтримують прийняття рішень у великих організаціях;

– настільна СППР (desktop single user DSS) – це СППР, призначені для одного користувача. Вони зручні для індивідуального прийняття рішень та зазвичай встановлюються на персональному комп'ютері й мають обмежені функції.

В залежності від орієнтації, тобто основного підходу або напряму, СППР поділяють на:

– орієнтовані на текст СППР (text-oriented DSS) – працюють з текстовими даними (наприклад, документи, звіти) ;

– орієнтовані на бази даних СППР (database-oriented DSS) – отримують інформацію з баз даних;

– орієнтовані на правила СППР (rule-oriented DSS) – базуються на правилах і використовують ШІ для формування рішень;

– орієнтовані на обчислювачі СППР (solver-oriented DSS) – використовують математичні рішення для фінансів, прогнозування, оптимізації;

– табличноорієнтовані СППР (spreadsheet-oriented DSS) – орієнтовані на таблиці, дозволяють створювати моделі аналізу вручну;

– комбіновані СППР (compound DSS) – поєднують кілька типів орієнтацій у єдиній системі.

Класифікація за типом та частотою прийняття рішень враховує, наскільки часто використовують СППР:

– інституційні СППР (institutional DSS) – для регулярних, повторюваних рішень;

– ситуаційні СППР (ad-hoc DSS) – для разових, нестандартних ситуацій, де дані можуть бути неповними або нестабільними.

Залежно від кількості користувачів СППР поділяють на:

– індивідуальні СППР (individual DSS) – для одного користувача;

– групові СППР (group DSS) – для групи осіб, часто із функціями співпраці, координації в реальному часі.

За метою розробки виділяють:

– індивідуально-розроблені СППР (custom-made DSS) – індивідуально розроблені під конкретну задачу, з урахуванням особливостей бізнесу;

– типові СППР (vendor-ready-made DSS) – готові рішення, розроблені для стандартних задач.

За ступенем процедурності:

– процедурні СППР (procedural DSS) – працюють за чітко визначеними алгоритмами і процедурами;

– непроцедурні СППР (non-procedural DSS) – допускають гнучкість у використанні моделей і доступі до даних, користувач може самостійно налаштовувати логіку аналізу.

1.2.2 Інтелектуальні системи підтримки прийняття рішень

Останнім часом в системах підтримки прийняття рішень почали використовувати функції штучного інтелекту для розширення можливостей, звідси і пішла назва – інтелектуальні системи підтримки прийняття рішень (ІСППР, або IDSS – Intelligent Decision Support Systems). Такі системи використовують [14] еволюційні обчислення (Evolutionary Computing), прецедентне моделювання (Case-based Reasoning) штучні нейронні мережі (Artificial Neural Networks), нечітку логіку (Fuzzy Logic) та інтелектуальних агентів (Intelligent Agents), що значно підвищує ефективність і забезпечує допомогу в розв'язанні складних прикладних задач. Такі задачі зазвичай вимагають прийняття рішень, які пов'язані з обробкою великих обсягів даних, в реальному часі.

Для ІСППР є характерними навчання на основі досвіду, розуміння неоднозначної та суперечливої інформації, адекватна та своєчасна реакція на нові ситуації, робота з заплутаними та складними ситуаціями, використання логічного судження ті раціонального висновку для розв'язання проблем, застосування знань для розуміння довкілля, розпізнавання відносної важливості різних факторів при прийнятті рішення.

Штучні нейронні мережі в ІСППР здатні знаходити закономірності і узагальнювати на основі попередніх випадків, класифікувати дані. На відміну від послідовних, заснованих на логіці підходів, які передбачають певний тип взаємозв'язку між вхідними та вихідними даними, нейронні мережі навчаються виявляти приховану функцію в даних.

Нечітка логіка надає можливість представляти вхідні дані так, як їх сприймає людина. На відміну від булевої логіки, яка заснована на двійковій системі, де 0 – false, а – true, нечітка логіка допускає, що вхідне значення знаходиться в проміжку від 0 до 1. Такий підхід забезпечує гнучкість у врахуванні несподіванок, можливість моделювання сценаріїв типу «що, якщо», а також підтримує задачі з невизначеністю, які важко описати за допомогою суворих математичних моделей.

Крім того, нечітка логіка дозволяє використовувати інтуїтивні оцінки, наприклад: «можливо», «дуже добре» тощо. З появою нової інформації значення можна легко оновлювати та коригувати. Нечітка логіка може бути об’єднаною зі штучними нейронними мережами, утворюючи нечіткі нейронні мережі. Такий підхід робить процес прийняття рішень більш зрозумілим для людини.

Експертна система – комп’ютерна система, яка намагається вирішувати задачі, які зазвичай вирішує людина-експерт. Ця система слугує для збору, збереження та інтерпретації знань від експерта і передачі цих знань користувачу, який приймає рішення. Розробник такої системи вивчає, як експерт приймає рішення, і інтегрує ці знання в систему. Користувач такої системи може звертатися до бази знань, щоб отримати доступ до попередніх випадків, отримувати висновки за допомогою механізму логічного виведення, звертатися до модулю пояснення, щоб зрозуміти, як було зроблено висновок.

Щодо еволюційних обчислень, найчастіше використовується генетичний алгоритм. Суть цього алгоритму полягає в тому, що покоління взаємодіють, обмінюються інформацією і впливають одне на одного, щоб краще відповідати заданій цілі.

Серед усіх методів інтелектуальні агенти виявились найбільш універсальними у задачах прийняття рішень. Існують також мультиагентні системи (Multi-Agent Systems, MAS). В таких системах агенти працюють у командах для вирішення складних задач, причому вони не мають повного уявлення про дії інших агентів. Кожен агент може мати власні характеристики і взаємодіяти з іншими агентами. MAS особливо добре показують себе у динамічних та невизначених середовищах, оскільки агенти можуть сприймати його, адаптуватися, обмінюватися знаннями, досягати узгоджених планів дій.

1.2.3 Огляд існуючих ІСППР в галузі енергетики

Критична інфраструктура – сукупність об’єктів критичної інфраструктури.

Об’єкти критичної інфраструктури – об’єкти інфраструктури, системи, їх частини та їх сукупність, які є важливими для економіки, національної безпеки та оборони, порушення функціонування яких може завдати шкоди життєво важливим національним інтересам [22].

До об'єктів критичної інфраструктури, відповідно до того ж Порядку, належать підприємства та установи наступних галузей: енергетика, хімічна промисловість, банки та фінанси, охорона здоров'я, транспорт, інформаційні технології та телекомунікації, продовольство, комунальні господарства, що є стратегічно важливими для функціонування економіки та безпеки держави, суспільства чи населення.

Наразі, задля підвищення ефективності та оптимізації праці, в критичних інфраструктурах надалі частіше застосовують штучний інтелект та ІСППР у тому числі. Зокрема в енергетиці, такі системи можуть оптимізувати роботу енергосистем, прогнозувати режими роботи інфраструктури, виявляти та вчасно реагувати на кібератаки та навіть взаємодіяти з клієнтом через чат-боти [23].

Однією з ключових сфер використання ІСППР є прогнозування та планування. Такі системи дозволяють передбачати попит на електроенергію на основі погоди, поведінки споживачів, сезонності тощо.

Також ІСППР допомагають визначити оптимальний час для технічного обслуговування чи заміни обладнання, тим самим зменшуючи ризики аварій.

ІСППР у сфері оперативного управління в реальному часі забезпечують підтримку рішень стосовно дотримання балансу генерації та споживання, підключення або відключення обладнання, реакції на аномальні режими роботи.

Важливою функцією ІСППР є моніторинг і діагностика відхилень у роботі енергосистеми, виявлення несправностей, крадіжок енергії та інших аномалій.

Також ІСППР активно застосовуються для підтримки керування мікромережами (Microgrid), віртуальними електростанціями (Virtual Power Plants) та розподіленими джерелами енергії (DER).

Окрім того, ІСППР використовуються для забезпечення кібербезпеки. Вони аналізують трафік в енергосистемах, виявляють підозрілу активність та реагують на потенційні загрози.

Британська компанія Octopus Energy [7] використовує власну інтелектуальну систему підтримки прийняття рішень Kraken для скорочення викидів CO_2 та збільшення вигоди для споживання. Kraken керує даними про енергетичну систему в реальному часі і автоматично приймає рішення, які спрямовані на балансування навантаження і підвищення стійкості мережі. Система відслідковує коливання в виробці електроенергії з відновлюваних джерел, і в залежності від цього пропонує клієнтам змінити своє споживання. Наприклад, коли виробляється багато зеленої енергії, Octopus платить клієнтам за використання енергії. А коли в роботу вступають викопні джерела енергії, компанія винагороджує тих, хто знижує вживання. Так, за словами компанії, за одну зиму вдалося запобігти викидам в розмірі біля 435 тонн CO_2.

Іншим прикладом є компанія Arm [17], яка використовує ІСППР для управління енергоспоживанням будівель. Ця компанія використовує рішення від стартапу Verdigris Technologies. Verdigris використовує розумні сенсори, підключені до електричних ланцюгів будівель, які збирають дані і передають їх у хмару. Там штучний інтелект аналізує енергоспоживання, передбачає несправності, формує автоматично рекомендації для управління будівлею. Таким чином, будівлі, що використовують технологію, отримують зелений сертифікат. За статистикою, будівлі з зеленим сертифікатом мають на 10% більшу ринкову вартість та на 5% нижчі операційні витрати.

1.3 Поняття функціональної стійкості

У наш час інтелектуальні системи підтримки прийняття рішень є надзвичайно важливим інструментом в процесі прийняття управлінських рішень у багатьох галузях, включно з критичними інфраструктурами. Для запобігання збоям ІСППР повинні бути функціонально стійкими, тобто мати здібність зберігати або оперативно відновлювати свої функції при впливі деструктивних факторів, забезпечуючи безперебійне ефективне функціонування та вирішення задач. Ключовою особливістю функціонально стійких систем є здатність структурно деградувати до повного виходу з ладу, тобто виключати елементи, які відмовили, перебудовувати свою архітектуру і перелаштовувати параметри для адаптації до умов експлуатації, які змінилися.

Функціональна стійкість тісно пов’язана з поняттями, стійкості, живучості, надійності та відмовостійкості [19, 24].

Стійкість ІСППР полягає в їхній спроможності зберігати координати руху у мінливих та ворожих середовищах. Це дозволяє системі ефективно працювати в умовах нестабільності, протистояти атакам, адаптуватися до нових обставин та продовжувати виконання своїх задач без критичних збоїв. Щодо основних характеристик стійкості ІСППР, це, по-перше, стійкість до різноманітних вхідних даних – система повинна правильно обробляти як нормальні, так і потенційно шкідливі дані, не припускаючи серйозних збоїв або спотворень висновків. По-друге, важливою характеристикою стійкості є адаптивність, тобто вміння швидко підлаштовуватись під зміни в навколишньому середовищі та у вимогах користувачів. По-третє, система повинна бути стійкою до різних типів атак, а також мати механізми виявлення та запобігання подібним загрозам. Також важливою характеристикою є спроможність до відновлення після збоїв та атак з мінімумом витрат. Окрім того, повинно проводитись регулярне тестування та оцінка стійкості у різних сценаріях. Важливою характеристикою є також неперервне оновлення безпеки системи, удосконалення в процесах прийняття рішень.

Важливо не плутати стійкість функціонування та функціональну стійкість [19], які хоч і пов’язані між собою, відображають різні аспекти поведінки системи в умовах збурення. Стійкість функціонування описує, як система реагує на невеликі зміни в початкових умовах. Формально ця властивість описується наступним чином: для будь-якого наперед заданого додатного числа θ, яке характеризує допустиме розходження траєкторій, існує число δ, таке що, якщо відстань між початковими умовами незбуреного і збуреного руху менше δ, то на всій часовій осі відстань між відповідними траєкторіями системи залишиться менше θ. На відміну від стійкості функціонування, функціональна стійкість характеризує стабільність не усієї траєкторії, а деякої функції від стану системи. Важливим є те, щоб, навіть при зміні траєкторії, значення цієї функції залишалося в допустимих межах. Формально ця властивість описується наступним чином: для будь-якого наперед заданого додатного числа θ, яке характеризує допустиме відхилення значень деякої функції від стану системи, існує таке число δ, що, якщо значення цієї функції в початкових точках незбуреного і збуреного рухів відрізняється менше ніж на δ, то у будь-який момент часу розходження значень функції від відповідних траєкторій не перевищить θ.

Живучість ІСППР визначається їхньою здатністю зберігати працездатність та виконувати критично важливі функції в умовах збоїв, атак і інших несприятливих ситуацій. Живучість також включає запобігання збоям, своєчасне виявлення загроз та мінімізацію їхніх наслідків для функціонування системи. Однієї з характеристик живучості є відмовостійкість та наявність механізмів резервування, тобто система повинна бути спроможною перемикатися на запасні ресурси (резервні сервери, канали зв’язку тощо), якщо основні виходять з ладу. Також важливою характеристикою є здатність до автоматичного відновлення, яка полягає в тому, що ІСППР повинна сама усувати наслідки збоїв, відновлюючи дані з резервних копій чи переходячи на альтернативні канали зв’язку, без участі людини. Іншою ключовою характеристикою є динамічне навчання та адаптація: система повинна не просто реагувати на обставини, а й навчатися на власному досвіді,

підлаштовуючись під мінливі обставини. Важливою характеристикою також є моніторинг та виявлення аномалій, що дозволяють швидко виявити відхилення в роботі системи і усунути їх. Окрім того, живуча ІСППР характеризується гнучкістю та масштабованістю, необхідними для ефективного керування мінливими вимогами та обсягом даних. Живучість системи підвищується за рахунок регулярного тестування на готовність до нештатних ситуацій.

Надійність інтелектуальних систем підтримки прийняття рішень – ймовірність того, що система буде виконувати своє призначення без відмов протягом заданого проміжку часу і при заданих умовах. До ключових характеристик відносять точність, тобто система повинна забезпечувати максимально коректні прогнози та відповіді. Також важливою характеристикою є стабільність при різних типах вхідних даних і умовах експлуатації, без сильної чутливості до змін у даних чи середовищі. Іншою характеристикою є спроможність до узагальнення, тобто використання наявних знань до нових ситуацій. Також важливою є прозорість та зрозумілість – користувачі повинні мати розуміння, на основі чого було прийняте те чи інше рішення. Окрім того система повинна бути здібною керувати невизначеністю, тобто працювати навіть тоді, коли дані неповні, нечіткі або суперечливі. Також важливою характеристикою є безпека, яка включає захищеність від атак і зовнішніх втручань.

Відмовостійкість – це здатність системи відновлюватися та продовжувати функціонувати при збоях окремих компонентів. Основним способом досягнення цієї якості є резервування ключових елементів системи при її розробці. Ключовий показник відмовостійкості Q дозволяє оцінити, наскільки ефективно система здатна продовжувати працювати при виникненні відмов, без переходу в стан повного руйнування. Цей показник визначається на основі марківської моделі функціонування системи як відношення сумарного часу, який система провела у не фатальних станах відмов, до повного часу її функціонування до повної відмови.

Для того, щоб мати можливість оцінити функціональну стійкість ІСППР, важливо визначити функції, які система повинна виконувати під впливом

деструктивних факторів протягом певного заданого часу. У контексті ІСППР будемо вважати, що основними функціями є, по-перше, збір, зберігання, обробка та візуалізація інформації; по-друге, передача інформації між модулями.

Звідси, основними вимогами до ІСППР є :

– Працездатність усіх модулів протягом експлуатації

$$\forall v_i \in \mathrm{V} \Leftarrow \omega_{\mathrm{i}}(\tau) = 1, \tau \in [0, \mathrm{t}), \tag{1.1}$$

де $\mathrm{V} = \{\nu_{\mathrm{i}}\}$ – множина всіх модулів;

$\omega_{\mathrm{i}}(\tau)$ – працездатність модуля ν_{i} (1 при працездатності, інакше 0);

τ, t – час експлуатації.

– Наявність маршрутів зв'язку між модулями

$$\forall v_i, v_j \Leftarrow \exists v_j \in \Gamma_{\mathrm{i}}, \mathrm{i}, \mathrm{j} = 1, 2 \ldots \mathrm{n}\ , \tag{1.2}$$

де Γ_{i} – множина досяжних вершин графа з вершини ν_{i};

n – число модулів.

Одним з показників функціональної стійкості ІСППР є ймовірність того, що кожна задача буде вирішена в межах допустимого часу і з заданою точністю.

$$\mathrm{P}\{\mathrm{t}(\mathrm{q_{ji}}) \le T_{\text{макс}}(\mathrm{q_{ij}})\}, \mathrm{q_{ij}} \in \mathrm{Q_j}, \mathrm{Q_j} = \{\mathrm{q_{j1}}, \mathrm{q_{j2}}, \ldots \mathrm{q_{jm}}\}, \tag{1.3}$$

де $\mathrm{Q_j}$ – множина задач, які вирішуються в j-ому модулі;

$\mathrm{q_{ij}}$ – і-та задача в j-ому модулі;

$T_{\text{макс}}$ – максимально припустимий час виконання;

$\mathrm{t}(\mathrm{q_{ji}})$ – фактичний час вирішення задачі $\mathrm{q_{ij}}$.

При оцінці функціональної стійкості ІСППР будемо вважати, що канали взаємодії між модулями мають достатню пропускну здатність, що означає забезпечення своєчасну надійну передачу інформації:

$$\lambda_{ij}(t) \leq \sigma_{ij}, \forall\, t \in [0, T], \tag{1.4}$$

де σ_{ij} – пропускна здатність каналу між модулями ν_i та ν_j;

$\lambda_{ij}(t)$ – інтенсивність потоку даних.

Отже, функціональна стійкість ІСППР забезпечується за умови, що існує така конфігурація її елементів і зав'язків між ними, за якої система здатна виконувати щонайменше критично важливі функції, навіть у разі відхилення від штатного режиму. При цьому вона має містити достатній рівень надмірності, щоб компенсувати або обійти наслідки несправностей чи збоїв. Така умова є необхідною і достатньою для функціональної стійкості ІСППР.

1.4 Висновки до розділу

У цьому розділі було окреслено задачу дипломної роботи, яка полягає у створенні програмного продукту для обчислення параметрів функціональної стійкості інтелектуальної системи прийняття рішень в умовах деструктивних впливів в критичних інфраструктурах. Також було розглянуто основні поняття, що стосуються СППР загалом, та класифікацію СППР. Після цього було конкретніше розглянуто саме інтелектуальні СППР, їхні види та прикладі реального використання таких систем в галузі енергетики. Окрім того було проаналізовано поняття функціональної стійкості, стійкості, живучості, відмовостійкості та надійності; визначено вимоги до функціонально стійкої ІСПП

2 МЕТОДИКА РОЗРАХУНКУ ФУНКЦІОНАЛЬНОЇ СТІЙКОСТІ

2.1 Математична модель ІСППР

Надалі будемо описувати ІСППР за допомогою математичної моделі [19] у вигляді графу $G = \{D,Ф,P\}$. $D = \{d_i\}$, $D = V \cup L$ – множина елементів графа, де $V=\{v_i\}$ – множина вершин графа, $L=\{l_{ij}\}$ – множина ребер. Вершини відповідатимуть функціональним модулям системи, а ребра – зв'язки між модулями, які забезпечують передачу даних. $Ф(l_{ij}) = v_i \& v_j$ – відображення інцидентності і суміжності елементів. $P = \{p_i\}$ – імовірності справного стану елементів.

Граф можна описати кількома способами:

– матриця суміжності – квадратна матриця розміром n×n, в якій кожна комірка a_{ij} дорівнює 1, якщо ребро між вершинами v_i та v_j існує, і 0 – якщо такого ребра не існує;

– список зв'язності – список довжиною m (де m – кількість ребер), в якому перелічені усі пари з'єднаних вершин;

– список суміжності – список, в якому для кожної вершини v_i формується окремий список, який містить всі вершини, з якими v_i з'єднана ребром;

– матриця інцидентності – матриця розміром n×m, в якій рядки відповідають вершинам, а стовбці – ребрам. Кожна комірка вказує, чи інцидентна вершина цьому ребру.

Показниками функціональної стійкості структури вважатимемо числа реберної $\lambda(G)$ та вершинної $\chi(G)$ зв'язностей, та імовірність зв'язності $P_{ij}(t)$.

Число реберної зв'язності $\lambda(G)$ – мінімальна кількість ребер, видалення яких порушить зв'язність графа, тобто зробить його незв'язним.

Число вершинної зв'язності $\chi(G)$ – мінімальна кількість вершин, видалення яких разом інцидентними ребрами призведе до того, що граф стане незв'язним або буде складатися лише з однієї вершини.

Імовірність зв'язності $P_{ij}(t)$ – це імовірність, що повідомлення з модуля i буде доставлено до модуля j за час, який не перевищуватиме t.

За структурним критерієм структура буде вважатися функціонально стійкою, якщо число вершин зв'язності та число реберної зв'язності задовольняють умову:

$$\chi(G) \geq 2 \cup \lambda(G) \geq 2, \tag{2.1}$$

де $\chi(G)$ – це число вершин зв'язності;

$\lambda(G)$ – це число реберної зв'язності.

За імовірнісним критерієм структура буде вважатися функціонально стійкою, якщо імовірність зв'язності між кожною парою вершин буде не менше заданої:

$$P_{ij}(t) \geq P_{ij}^{зад}, \ i \neq j, \ i, j = 1, 2 \dots n, \tag{2.2}$$

де n – це число вершин графа G (V, L).

Імовірність зв'язності $P_{ij}(t)$ є ключовим імовірнісним показником функціональної стійкості ІСППР. На відміну від структурних характеристик, таких як вершини і ребра графа, цей показник відображає імовірність успішної і своєчасної передачі інформації між функціональними модулями з урахуванням реальних факторів.

Подія зв'язності між модулями системи $E_{x,y}$ задається фактом існування між ними принаймні одного простого ланцюга μ_i або $\bar{\bar{\mu}}_i$ (ПЛ) – послідовності ребер та

вершин графа між ν_x та ν_y без петель і паралелей. Відсутність такого ланцюга робить взаємодію неможливою.

Оскільки передача інформації по ланцюгу неможлива при виході з ладу принаймні одного з його елементів, вважається, що ланцюг не існує, якщо хоча б один з його компонентів несправний.

Настанню події незв'язності $\bar{E}_{x,y}$ відповідає наявність принаймні однієї простої січної множини (ПСМ).

Подія зв'язності $E_{x,y}$ та незв'язності $\bar{E}_{x,y}$ складають повну групу подій.

Зазвичай вважається, що між вузлами має існувати принаймні один простий ланцюг, щоб зв'язок вважався встановленим ($\alpha = 1$). На основі цього вводиться імовірність $P_{x,y}$, тобто імовірність того, що існує не менше α простих ланцюгів між вузлами:

$$P_{x,y}^{\alpha=1} = P\left(E_{x,y}\right) = P(\exists \mu_i \in M_{x,y}\left[\bar{\mu}_i / \forall d_i \in D[\bar{\bar{d}}_j \bar{\vee} \bar{d}_j \bar{\vee} \dot{d}_j]\right]) \qquad (2.3)$$

Вимогою на незв'язність є не існування жодного простого ланцюга між ν_x та ν_y. Імовірність незв'язності $\bar{P}_{x,y}$ тоді обчислюється наступним чином:

$$\bar{P}_{x,y}^{\beta=1} = P\left(\bar{E}_{x,y}\right) = P(\exists s_i \in S_{x,y}[\bar{s}_i / \forall d_i \in D[\bar{\bar{d}}_j \bar{\vee} \bar{d}_j \bar{\vee} \dot{d}_j]]) \qquad (2.4)$$

Як вже згадувалося, у моделі оцінки функціональної стійкості ІСППР особливу роль відіграють так звані прості січні множини (ПСМ) – мінімальні за складом сукупності елементів структури системи (вершини і/або ребра графа), одночасна відмова яких призводить до повного руйнування усіх простих ланцюгів між двома заданими вузлами і настанню події $\bar{E}_{x,y}$. При цьому жоден елемент не має бути надлишковим, тобто не повинно існувати елементів, відмова від яких не

впливає на наявність з'єднання. Якщо всі елементи деякої ПСМ s_i знаходяться в несправному стані, вважається, що подія зв'язності між вузлами не наступить.

У структурі ПСМ розрізняють прості перерізи (ПП), які містять тільки вершини; прості розрізи (ПР) – містять тільки ребра; змішані ПСМ – містять і вершини і ребра.

Прості січна множина розділяє граф на дві незв'язні компоненти $Д_1$ та $Д_2$. Це означає, що якщо всі елементи ПСМ вийдуть за ладу, то вершини ν_x та ν_y, між якими раніше існував зв'язок, опиняться у різних незв'язних частинах структури.

2.2 Критерій функціональної стійкості

Оскільки імовірність зв'язності P_{ij} розглядається тільки для двох вузлів, всю систему можна охарактеризувати за допомогою матриці імовірності зв'язків P.

$$P = \begin{bmatrix} 0 & P_{12} & P_{13} & \dots & P_{1n} \\ P_{21} & 0 & P_{23} & \dots & P_{2n} \\ . & . & . & . & . \\ P_{n1} & P_{n2} & P_{n3} & \dots & 0 \end{bmatrix}, \quad (2.5)$$

де кожен елемент P_{ij} – це імовірність наявності зв'язку між парою вершин ν_i та ν_j.

Одним із способів оцінки функціональної стійкості є згортка цієї матриці. Такий підхід дає можливість оцінити загальну зв'язність системи, проте не враховує структурні особливості мережі. Одним з варіантів, як доповнити розрахунки, є введення матриці [21]:

$$H = \begin{bmatrix} 0 & h_{12} & h_{13} & \dots & h_{1n} \\ h_{21} & 0 & h_{23} & \dots & h_{2n} \\ . & . & . & . & . \\ h_{n1} & h_{n2} & h_{n3} & \dots & 0 \end{bmatrix}, \quad (2.6)$$

де кожен елемент h_{ij} обчислюватиметься наступним чином:

$$h_{ij} = \begin{cases} \frac{N_{ij}^{min}}{(|i-j|+1)^2}, i \neq j \\ 0, \quad i = j \end{cases}, \quad (2.7)$$

де N_{ij}^{min} – мінімальна кількість ребер на найкоротшому шляху між ν_i та ν_j.

Таким чином, критерій функціональної стійкості K будемо розраховувати за формулою [21]:

$$K = \sum_{\substack{i,j=1 \\ i \neq j}}^{n} N_{ij}^{min} \cdot P_{ij} \cdot (|i-j|+1)^{-2} \quad (2.8)$$

Тепер виникає необхідність скласти матрицю P. Методи розрахунку P_{ij} для кожної пари вершин буде розглянуто далі. Таких методів існує багато і у цій роботі будуть розглянуті лише деякі.

2.3 Метод розрахунку імовірності зв'язності двох вершин на основі аналізу простих ланцюгів

Цей метод базується на припущенні, що дві вершини графа вважаються зв'язними (наступає подія зв'язності $E_{x,y}$), якщо існує принаймні один простий ланцюг μ_i між ними. Тому для обчислення імовірності зв'язності між ними $P_{x,y}$

використовується повний перебір усіх можливих ПЛ, які з'єднують задані вершини. Після цього обчислюється імовірність існування хоча б одного з них.

Так, подію зв'язності $E_{x,y}$ можна описати з використанням формули включень-виключень наступним чином:

$$E_{x,y} = \sum_{v=1}^{m_{x,y}} (-1)^{v+1} \sum_{z=1}^{C_{m_{x,y}}^{v}} \prod_{\substack{\xi=1 * \\ i \epsilon z}}^{v} \bar{\bar{\mu}}_{i\xi}, \quad (2.9)$$

де $m_{x,y}$ – загальна кількість прости ланцюгів між вершинами ν_x та ν_y;

$C_{m_{x,y}}^{v}$ – кількість комбінації по v ланцюгів із $m_{x,y}$;

v – розмір підмножини простих ланцюгів;

$\bar{\bar{\mu}}_{i\xi}$ – подія справності ПЛ;

символ * (логічне множення) передбачає, що елементи перемножених ПЛ включаються лише 1 раз.

Результатом такого об'єднання ν простих ланцюгів є сукупний простий ланцюг (СПЛ) μ_j^v – структура, яка охоплює всі унікальні ребра, що належать хоча б одному з ланцюгів комбінації.

Оскільки вважається, що СПЛ існує тільки у випадку, коли всі його ребра знаходяться в працездатному стані, імовірність його існування визначається добутком імовірностей безвідмовної роботи кожного ребра:

$$p(\mu_j^v) = \prod_{\sigma=1}^{m_j} p(d_{i\sigma}) \,, \quad (2.10)$$

де m_j – кількість ребер в СПЛ μ_j^v;

$p(d_{i\sigma})$ – імовірність безвідмовної роботи ребра $d_{i\sigma}$.

Таким чином, на основі аналізу всіх можливих комбінацій простих ланцюгів, які зв'язують вершини ν_x та ν_y, можна розрахувати точну імовірність їхньої зв'язності. Для цього використовується формула включень-виключень, яка враховує імовірність існування СПЛ, сформованих з різних комбінацій простих шляхів. Формула для розрахунку $P_{x,y}$ має наступний вигляд [19]:

$$P_{x,y} = \sum_{v=1}^{m_{x,y}} (-1)^{v+1} \sum_{z=1}^{C_{m_{x,y}}^{v}} p(\mu_{j_z}^{v}), \tag{2.11}$$

де $m_{x,y}$ – загальна кількість простих ланцюгів між вершинами ν_x та ν_y;

$C_{m_{x,y}}^{v}$ – кількість комбінації по v ланцюгів ;

$p(\mu_{j_z}^{v})$ – імовірність існування СПЛ.

Отже, метод оцінки імовірності зв'язності на основі перебору простих ланцюгів є одним з точних підходів, який забезпечує високу точність результатів без використання апроксимацій. У той же час, через необхідність повного перебору усіх простих ланцюгів, обчислювальна складність зростає експоненціально зі збільшенням кількості вершин. Тому такий метод доцільно застосовувати тільки для невеликих структур, де важлива максимальна точність.

2.4 Оцінка імовірності зв'язності двох вершин методом Монте-Карло

Оскільки попередньо описаний метод має високу обчислювальну складність, було вирішено розглянути також пошук імовірності зв'язності двох вершин методом Монте-Карло.

Згідно з Кембриджським словником статистики, метод Монте Карло [5] – це метод розв'язання математичних та статистичних задач за допомогою симуляції,

які застосовуються, коли аналітичний розв’язок задачі або неможливий, або потребує занадто багато часу.

Симуляція – штучне відтворення випадкових процесів (зазвичай з використанням псевдовипадкових чисел і\або комп’ютерів) для імітації поведінки конкретних статистичних моделей.

Цей метод дозволяє отримати оцінки імовірнісних характеристик системи шляхом моделювання її випадкової поведінки. Такий підхід широко застосовується для імітації процесів у системах, де присутній випадковий фактор.

Алгоритм [6] реалізується наступним чином. Так, для обчислення імовірності зв’язності між вершинами v_x та v_y, повторимо симуляцію N разів. У межах однієї симуляції для кожного ребра випадковим чином визначається його стан: з імовірністю p ребро вважається справним і залишається в графі, а з імовірністю q=-p – несправним та тимчасово виключається зі структури. Після цього формується тимчасовий підграф, який включає лише працездатні елементи. Далі за допомогою алгоритму обходу графу перевіряється, чи існує шлях між вершинами v_i та v_j. Якщо шлях знайдено, результат моделювання вважається успішним. Після виконання усіх N випробувань кількість успішних випадків S ділиться на загальну кількість симуляцій, і таким чином отримуємо P_{xy}:

$$P_{xy} = \frac{S}{N} \tag{2.12}$$

Отже, метод Монте Карло дозволяє здійснити чисельне моделювання поведінки системи в умовах випадкових відмов ребер, не потребуючи явного переліку всіх маршрутів або комбінацій. Його обчислювальна складність зростає лінійно зі збільшенням кількості симуляцій.

2.5 Висновки до розділу

У цьому розділі було розглянуто методику оцінки функціональної стійкості ІСППР на основі представлення системи у вигляді неорієнтованого графа без петель та кратних ребер, в якому вершини відповідають модулям системи, а ребра – зв’язкам між ними. Також було визначено основні показники функціональної стійкості: число реберної зв’язності, число вершинної зв’язності, імовірність зв’язності між парами модулів та критерій функціональної стійкості. Окремо увагу було приділено саме методам розрахунку імовірності зв’язності між двома модулями: методу простих ланцюгів та методу Монте-Карло, кожен з яких було детально розглянуто.

3 ЗАСОБИ РОЗРОБКИ

3.1 Мова програмування Python

Python – це високорівнева, інтерпретована, об’єктно-орієнтована мова програмування загального призначення. Вона створювалась з упором на простоту читання коду, простий синтаксис та високу продуктивність розробки.

Python [10] використовую практично в усіх галузях, зокрема у системному програмуванні, веброзробці, Data Science, AI, ML, аналізі даних, кібербезпеці, розробці ігор, 3-D анімації, робототехніці, при написанні банківських та фінансових додатків.

Мова програмування Python набула своєї популярності завдяки її численним перевагам. Зокрема, через її простоту та зрозумілість коду, швидкість написання програм цією мовою, кросплатформеність, масштабованість, гнучкість, безкоштовність та відкритість.

Серед відомих компаній Python використовують такі гіганти як Google, YouTube, NASA, IBM, Intel, Pixar та Dropbox.

У контексті цієї роботи було обрано саме Python, оскільки ця мова є, мабуть, найзручнішою для роботи з графами, для виконання математичних обчислень та для візуалізації даних.

3.2 PyCharm

Для написання було використано інтегроване середовище розробки(IDE) під назвою PyCharm розробників JetBrains. PyCharm є зручним та надає велику кількість інструментів.

З переваг PyCharm варто виділити наступний функціонал:

– підтримка автодоповнення коду не лише вбудованих елементів мови, а й зовнішніх бібліотек, що значно прискорює розробку;

– інтеграція з Git, візуальна взаємодія та можливість бачити останні зміни;

– контроль покриття коду, який дає можливість оцінити, яку частину коду вже було виконано;

– вбудований менеджер керування пакетами, завдяки якому можна легко оновлювати, встановлювати та переглядати бібліотеки;

– відстеження змін у файлах і можливість повернутися до попереднього стану;

– наявність вбудованого терміналу та підтримка віртуальних середовищ;

– зручна навігація у проєкті за допомогою візуального представлення.

3.3 Використані бібліотеки

Tkinter [2] [79] – стандартний модуль мови Python, який забезпечує інтерфейс до бібліотеки Tcl/Tk, призначений для створення користувацького графічного інтерфейсу (GUI). Tcl (Tool Command Language) – це мова сценаріїв, а Tk – це інструментарій для побудови GUI. І, оскільки, Tcl/Tk не є частиною Python, для взаємодії як раз і використовується Tkinter. Tkinter приховує складність роботи з Tcl/Tk, дозволяє створювати графічні програми простими засобами Python. Весь GUI будується за принципом об’єктно-орієнтованого програмування, де кожен елемент інтерфейсу розглядається як об’єкт. Надалі буде описано, які елементи Tkinter було використано під час реалізації програми.

Головне вікно застосунку створюється за допомогою tk.Tk. У цьому вікні розміщуються контейнери Frame, які організовують простір інтерфейсу.

Для побудови графа використовується Canvas, на якому за допомогою методів create_oval (create_line для ребер) та create_text створюються вершини, ребра та їхні підписи. Canvas також використовується для створення кнопок.

Для відображення іконок кнопок інструментів застосовується стандартний механізм Tkinter для роботи з графікою PhotoImage.

Для виведення заголовків, підписів або результатів із заданим шрифтом та вирівнюванням використовується Label (як tk.Label, так і ttk.Label). Для введення числових значень в редакторі матриці використовується Entry, які вирівнюються у таблицю за допомогою grid.

Також для редагування матриці суміжності відкривається окреме вікно реалізоване за допомогою Toplevel.

Menu, що створює контекстне меню для експорту/імпорту графа.

Matplotlib [11] – це бібліотека для візуалізації даних, яка дозволяє створювати графіки та діаграми на основі даних. У роботі ця бібліотека використовується для візуалізації графіку залежності критерію функціональної стійкості від імовірності зв’язності ліній зв’язку.

Pandas [13] – бібліотека для аналізу та обробки табличних даних (DataFrame), яка підтримує фільтрацію, агрегацію, об’єднання тощо. У роботі pandas використовується для зручної обробки та представлення матриць.

NetworkX [12] – бібліотека для побудови, візуалізації та аналізу графів і мереж. У програмі використовується для відображення графу, заданого за допомогою матриці зв’язності.

Collections [3] представляє спеціальні структури даних. Двостороння черга deque використовується для зручного виконання обходу графа в ширину.

Json [9] – вбудований модуль для серіалізації в структуру JSON і навпаки. Використовується для збереження створеного графу та завантаження збереженого графу.

Random [16] – модуль для генерації псевдовипадкових чисел і випадкового вибору елементів з послідовностей.

3.4 Висновки до розділу

У цьому розділі було розглянуто основні засоби, які були використані під час розробки програмного продукту. Було обґрунтовано вибір використання мови програмування Python як гнучкого та зручного інструменту для обчислень, візуалізації даних та роботи з графами. Також було описано інтегроване середовище розробки PyCharm, у якому відбувалося написання коду. Крім того було розглянуто використані бібліотеки, зокрема Tkinter для інтерфейсу, Matplotlib для побудови графіків, Pandas для обробки табличних даних, NetworkX для роботи з графами.

4 ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

4.1 Use-Case діаграма

Для кращого розуміння того, який функціонал буде реалізовано у цій роботі, було побудовано діаграму варіантів використання (Use-Case діаграму). На цій діаграмі відображено основні сценарії взаємодії користувача з системою. Діаграму варіантів використання наведено на рисунку 4.1.

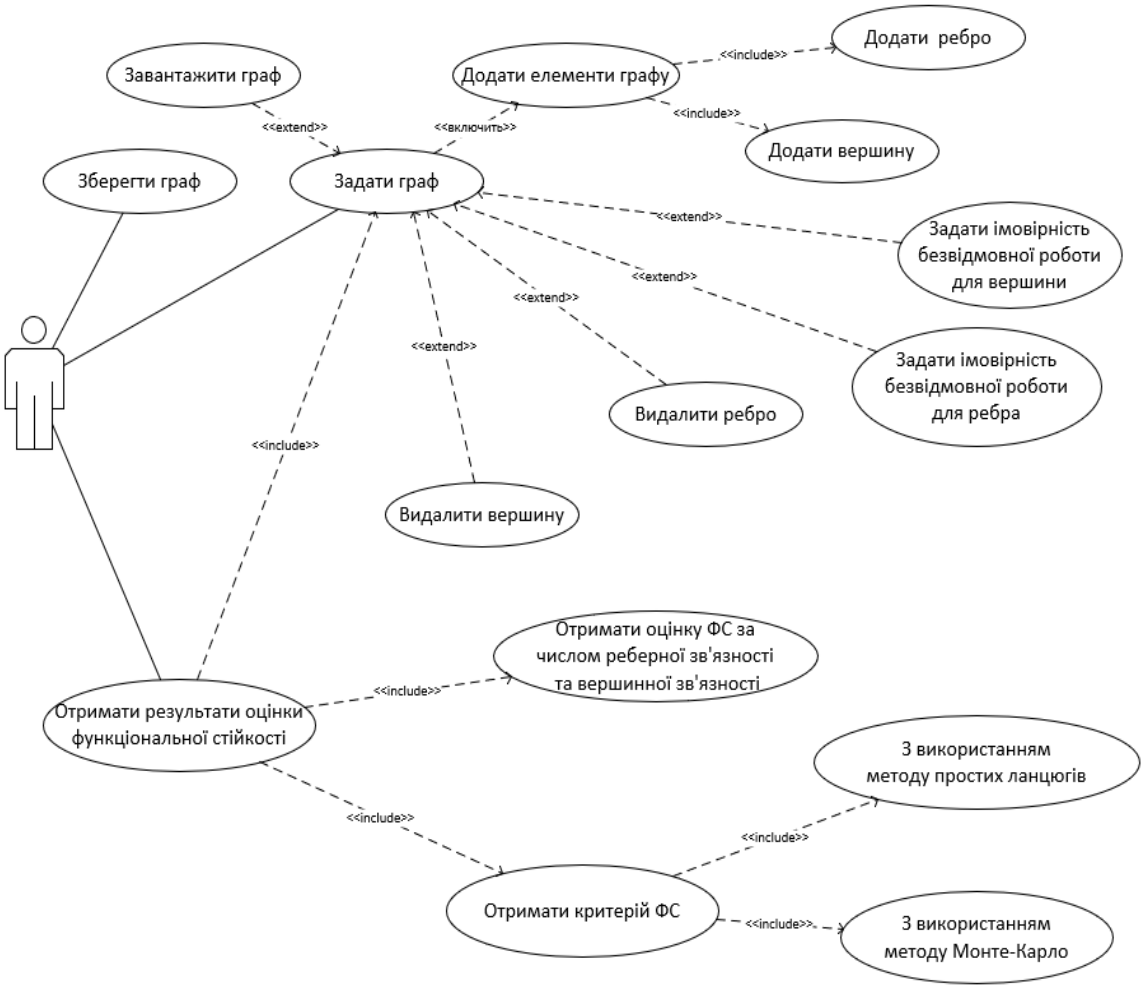

Рисунок 4.1 – Діаграма варіантів використання

Отже, користувач повинен мати можливість задати у вигляді графу ІСППР, функціональну стійкість якої він хоче оцінити. З можливостей задання графу

необхідно було передбачити можливість додавання вершин, ребер. Також користувач має можливість видалити вершину чи ребро. Крім того користувач має можливість задати імовірність безвідмовної роботи модуля (вершини) та каналу зв’язку (ребра). Для зручності передбачено можливість зберегти створений граф та потім завантажити його з файлу. Сам розрахунок провести без задання системи неможливо. Задавши граф, користувач може отримати оцінку функціональної стійкості системи за числом реберної зв’язності та вершинної зв’язності. Також користувач отримує оцінку критерію функціональної стійкості, для розрахунку якого потрібна матриця імовірностей зв’язності всіх вершин. Ця матриця, в залежності від розміру системи, формується або за допомогою методу простих ланцюгів або за допомогою методу Монте-Карло.

4.2 Діаграма класів

Для більш детального проєктування архітектури було створено діаграму класів, на якій представлено класи програмного продукту, їх відношення між собою та їх атрибути, а саме методи та змінні. Пунктирні лінії з зафарбованими стрілками показують залежність одного класу від іншого. Суцільна лінія без стрілки використана показує відношення асоціації між класами. Лінії з незафарбованими стрілками означають, що клас наслідує батьківський. Діаграму класів програмного продукту наведено на рисунку 4.2.

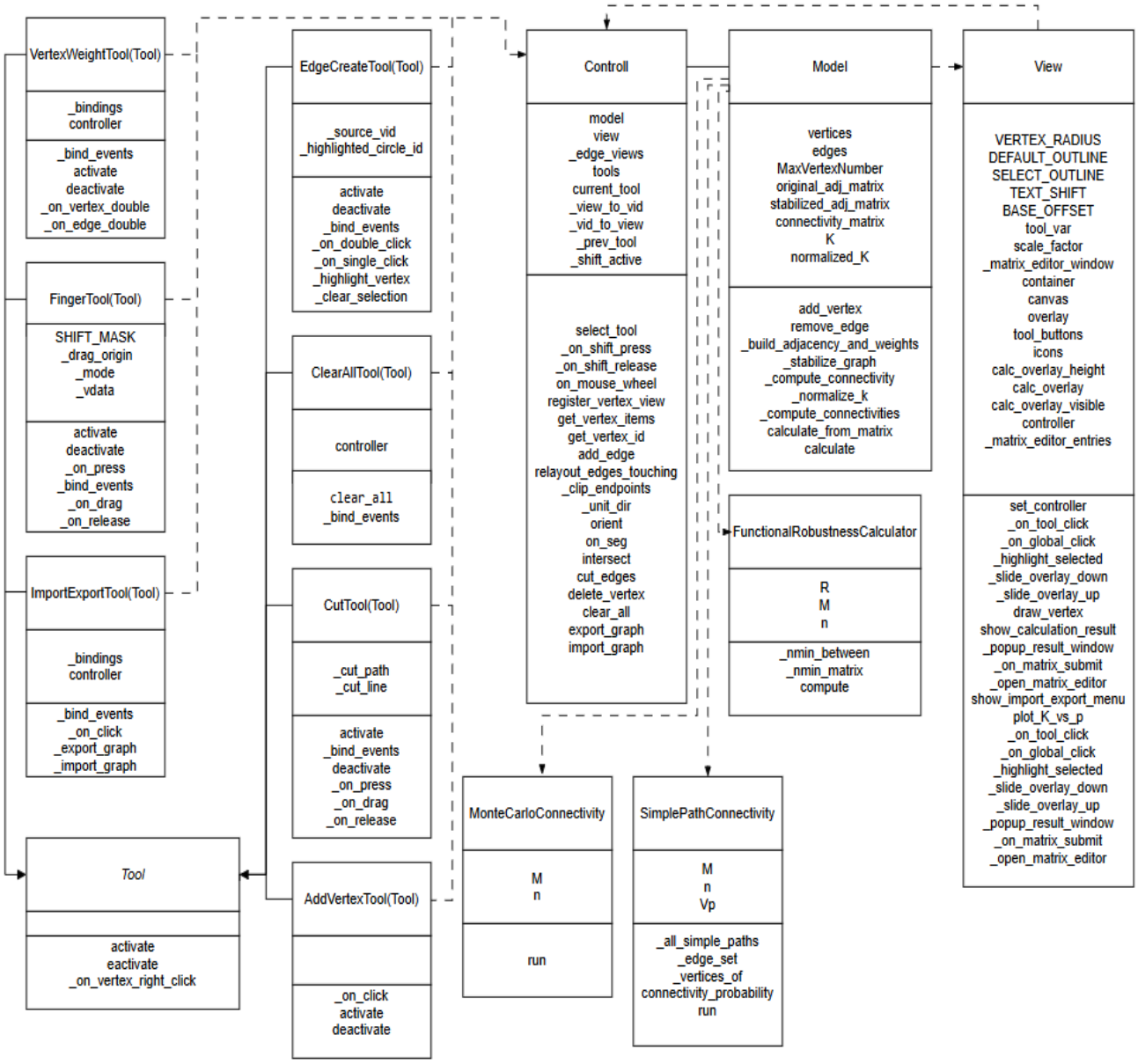

Рисунок 4.2 – Діаграма класів

На діаграмі можна побачити, що класи EdgeCreateTool, ClearAllTool, CutTool, AddVertexTool, EdgeDirectionTool, ImportExportTool, FingerTool, VertexWeightTool наслідують абстрактний клас Tool. На те, що клас є абстрактним вказує назва написана курсивом.

Всі інструменти залежні від класу Controller. Цей клас є найбільшим і центральним компонентом програми.

4.3 Алгоритми виконання розрахунків

4.3.1 Реалізація оцінки імовірності зв’язності двох вершин на основі аналізу простих ланцюгів

Метод простих ланцюгів дозволяє обчислити зв’язність між двома заданими вершинами шляхом аналізу всіх простих ланцюгів між ними. Підхід базується на поелементному розгляді множин ребер, які утворюють кожен із таких шляхів, та подальшому застосуванні формули включення-виключення для оцінки загальної ймовірності зв’язності. Алгоритм у вигляді блок-схеми наведено на рисунку 4.3.

Рисунок 4.3 – Алгоритм розрахунків за методом простих ланцюгів

Спочатку виконується обхід графа в глибину (DFS) за допомогою стека й повертає всі прості шляхи між вершинами i та j, тобто такі, що не містять повторних вершин. Кожен знайдений шлях представляється у вигляді впорядкованого списку вершин.

Пошук у глибину (Depth-First Search, DFS) [4] – базовий алгоритм теорії графів, призначений для дослідження вершин та ребер графу. Алгоритм послідовно досліджує суміжні вершини від деякої початкової, просуваючись вглиб, доки не досягне тупика, після чого повертається назад і досліджує інші доступні напрямки.

Далі для кожного шляху, знайденого на першому етапі формується множина неорієнтованих ребер, що його утворюють. Усі ці множини ребер збираються в окремий список для подальшої обробки.

Після цього розглядаються всі можливі комбінації знайдених шляхів. Для кожної комбінації шляхів обчислюється об'єднання всіх ребер, які входять до неї. На основі цього об’єднання обчислюється ймовірність того, що всі ці ребра будуть працездатними одночасно. Ця імовірність отримується шляхом перемноження індивідуальних імовірностей для кожного ребра, які беруться з матриці надійності ребер.

Оскільки різні шляхи можуть мати спільні ребра, звичайне додавання імовірностей може призвести до дублювання і завишення оцінки, тому застосовується формула включення-виключення відповідно до формули (2.11). Для кожної підмножини простих шляхів між вершинами і та j обчислюється ймовірність одночасної працездатності всіх елементів, після чого результат додається або віднімається із загальної суми залежно від парності розміру підмножини. Якщо в підмножині парна кількість елементів, імовірність віднімається від загальної суми, а якщо непарна – додається.

4.3.2 Реалізація оцінки імовірності зв’язності двох вершин методом Монте-Карло

Раніше у розділі 2 було описано, як можна застосувати метод Монте-Карло для розрахунку імовірності зв’язності двох вершин. У реалізації методу Монте-Карло ключовим завданням є моделювання багатьох варіантів випадкової

структури графа та перевірка виникнення зв’язку між вершинами в кожній з таких симуляцій. Алгоритм у вигляді блок-схеми наведено на рисунку 4.4.

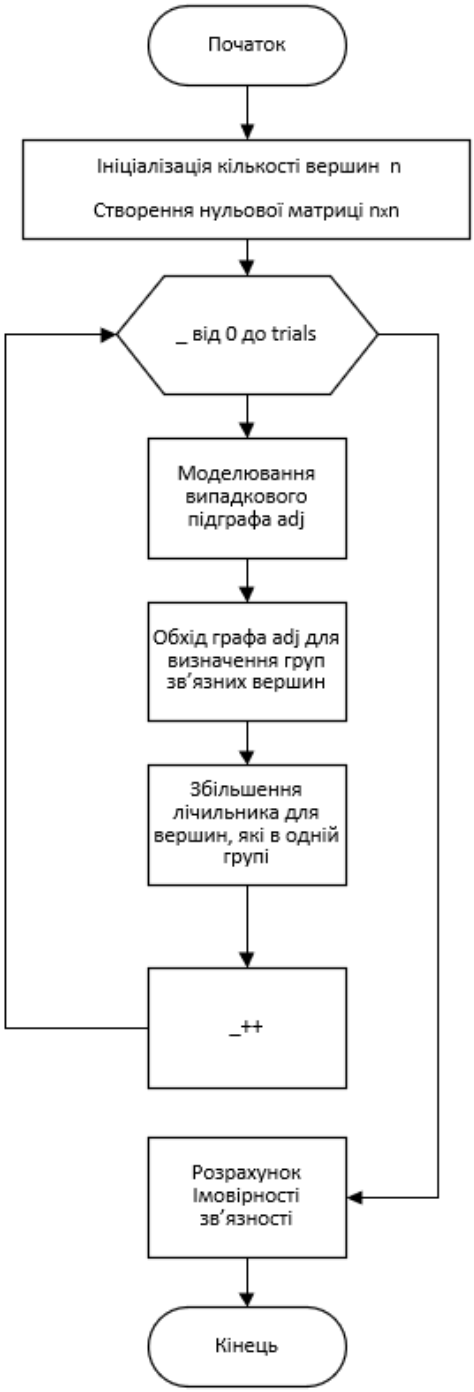

Рисунок 4.4 – Алгоритм розрахунків за методом Монте-Карло

Перед запуском основного циклу здійснюється ініціалізація: створюється змінна n, яка зберігає кількість вершин у графі. Далі ця змінна використовується

для побудови квадратної матриці counts розміром n×n, яка виконує роль накопичувача статистики: кожен її елемент відповідає кількості симуляцій, у яких вершини i та j потрапляли до однієї компоненти зв’язності. Спочатку матриця заповнюється нулями, оскільки жодного моделювання ще не відбулося.

Після цього починається основний цикл, кожна ітерація якого відповідає одній симуляції. Спочатку відбувається побудова випадкового неорієнтованого графа. Для кожної пари вершин (i,j), відповідно до заданої матриці надійності вирішується, чи утворювати ребро: генерується випадкове число з інтервалу [0;1) та порівнюється з відповідною ймовірністю з матриці надійності. Якщо випадкове число менше за імовірність безвідмовної роботи, утворюється ребро між вершинами i та j у поточному підграфі. Таким чином симулюється відмова різних ребер початкового графа. У результаті формується підграф, який кожного разу відрізняється завдяки використанню генератора випадкових чисел.

Перед початком пошуку компонент зв’язності здійснюється ініціалізація службового списку, у якому кожній вершині присвоюється початкове значення -1, що означає «не відвідано». Цей список використовується для фіксації належності кожної вершини до певної компоненти зв’язності. Також ініціалізується змінна cid (component ID), яка використовується у якості лічильником компонент. Кожна нова компонента отримує ідентифікатор, який збільшується після завершення обходу кожної групи зв’язаних вершин.

Наступний етап полягає у визначенні груп зв’язних вершин у підграфі, який було сформовано на першому кроці алгоритму. Для цього реалізовано алгоритм обходу в ширину (BFS), за допомогою якого всі вершини, що знаходяться у спільній компоненті, маркуються однаковим ідентифікатором.

Обхід графа в ширину (Breadth-First Search, BFS) [4] – базовий алгоритм пошуку в графах, суть якого полягає в тому, щоб послідовно «відвідувати» всі досяжні від заданої стартової вершини. Спочатку відвідуються всі вершини, які знаходяться на відстані 1 від початкової, потім всі на відстані 2 і так далі, поки весь граф не буде пройдено. В результаті формується дерево обходу в ширину, в якому

збережено найкоротші шляхи від початкової вершини до інших. Алгоритм використовує чергу (FIFO), яка гарантує, що вершини обробляються у правильному порядку.

Цей етап дозволяє визначити групи вершин, між якими гарантовано існує шлях.

Далі здійснюється накопичення статистики щодо пар вершин. Для кожної пари (i, j) перевіряється, чи потрапили ці вершини до однієї компоненти зв’язності у поточній симуляції. Якщо так, то лічильник у матриці counts збільшується на одиницю. Таким чином, з кожною симуляцією формується статистика про те, наскільки часто дві вершини опиняються зв’язаними.

Після завершення заданої кількості симуляцій відбувається розрахунок за формулою (2.12).

Отримана у результаті розрахунку матриця перетворюється на структуру типу DataFrame для подальшого аналізу. Таким чином, отримуємо матрицю P, кожен елемент якої представляє собою оцінку імовірності зв’язності між відповідними вершинами графа.

4.4 Висновки до розділу

У цьому розділі було описано програмну реалізацію результатів дослідження. Було представлено діаграму прецедентів, яка ілюструє основні сценарії взаємодії користувача з програмним продуктом.

Також було розглянуто діаграму класів, яка показує архітектуру програми, демонструючи класи.

Окремо було описано реалізацію методів розрахунку імовірності зв’язності між двома вершинами. Було реалізовано обидва підходи: метод простих ланцюгів та метод Монте-Карло. Описано роботу обох алгоритмів, а також пояснено суть алгоритмів обходу графа в ширину і в глибину.

5 РОБОТА КОРИСТУВАЧА З ПРОГРАМНОЮ СИСТЕМОЮ

5.1 Робота користувача з програмним продуктом

При запуску програм користувач бачить перед собою графічний інтерфейс для редагування графів, який складається з робочої області та панелі інструментів зліва. Зовнішній вигляд інтерфейсу наведено на рисунку 5.1.

Рисунок 5.1 – Графічний інтерфейс програми

На панелі інструментів представлено наступні інструменти: «вказівник», «додати вершину», «додати ребро», «видалити ребро», «задати ваги», «розрахувати», «видалити граф», «зберегти/завантажити граф». Щоб обрати інструмент потрібно натиснути на його іконку. Іконку обраного інструменту буде виділено іншим кольором. Розглянемо роботу кожного інструменту далі.

Почнемо з інструменту «додати вершину». Обравши цей інструмент, можна одним кліком додавати вершини графа. Додані вершини будуть нумеруватися в порядку їхнього додавання, починаючи з 1. Результат використання цього інструменту наведено на рисунку 5.2.

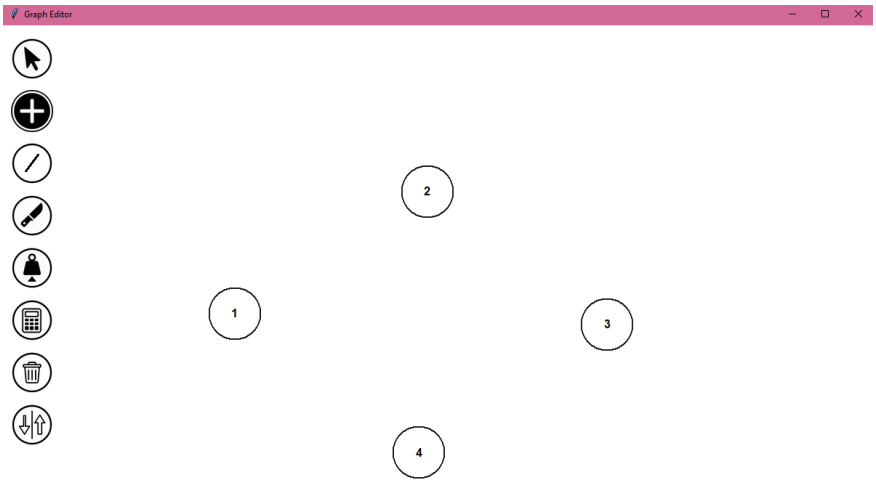

Рисунок 5.2 – Результат використання «додати вершину»

Тепер створимо ребра між вершинами за допомогою інструменту «додати ребро». Для цього необхідно обрати інструмент та по черзі натиснути на кожну з двох вершин, між якими має з’явитися ребро. При додаванні нових ребер імовірність їхньої безвідмовної роботи за замовчуванням встановлюється 1.0. Результат наведено на рисунку 5.3.

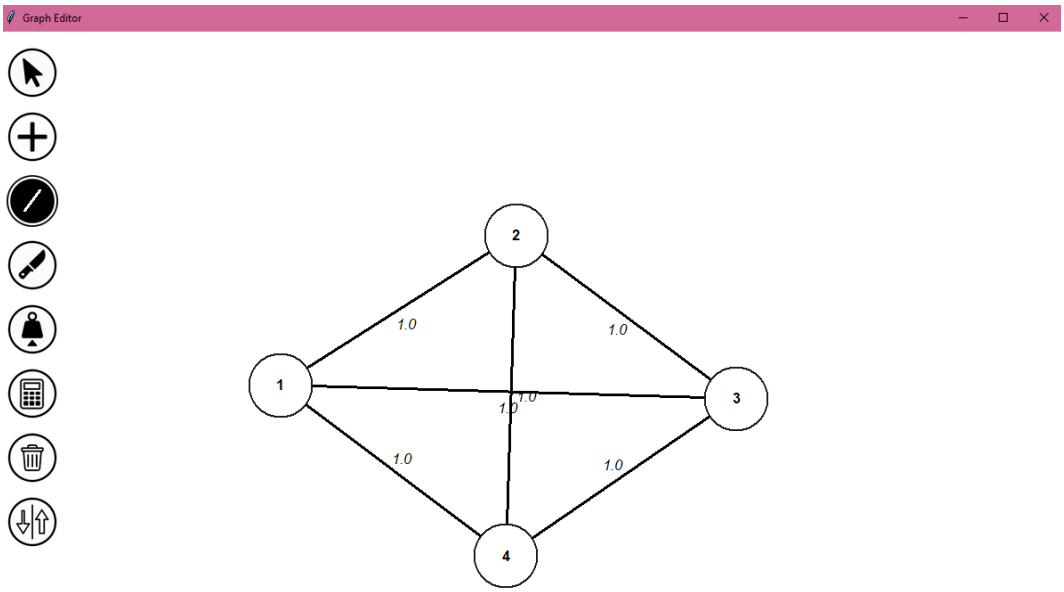

Рисунок 5.3 – Результат використання «додати ребро»

При вибраному інструменті «вказівник» можна пересувати вершини графа.

З будь-яким обраним інструментом при затисканні клавіши SHIFT можна за допомогою колеса миші збільшувати та зменшувати масштаб робочої області. Також, якщо зажати SHIFT та ліву кнопку миші, можна пересувати робочу область.

За допомогою інструменту «задати ваги» можна встановити імовірність знаходження у справному стані елементу графа. Для цього треба обрати інструмент та двічі клацнути по відповідному елементу. Це стосується як ребер, так і вершин. З'явиться віконце, в якому потрібно ввести імовірність (рис. 5.4).

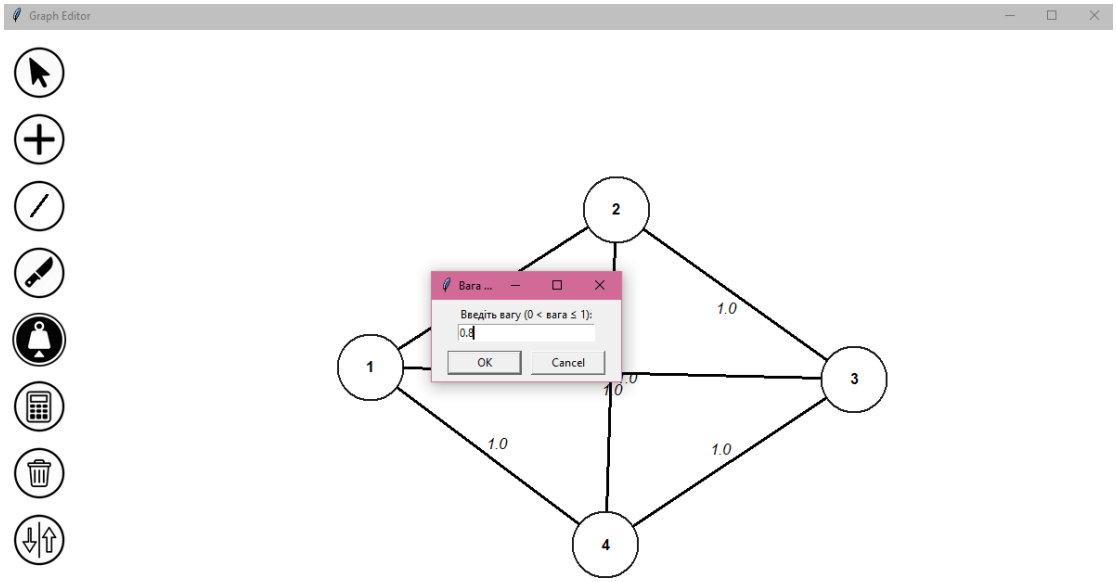

Рисунок 5.4 – Процес використання «задати вагу»

У разі хибного вводу з'явиться попередження з проханням повторити спробу. Після введення значення воно відобразиться поряд з ребром. Результат цієї дії, а також пересування вершин за допомогою інструменту «вказівник» наведено на рисунку 5.5.

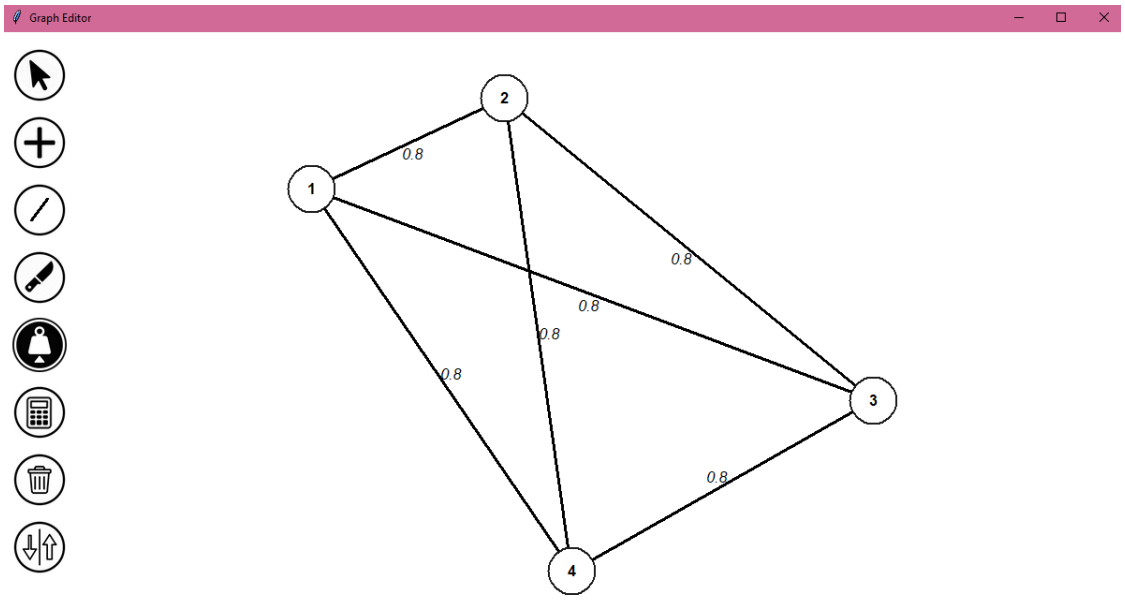

Рисунок 5.5 – Результат використання «задати вагу» та переміщення вершин

Тепер, коли архітектуру задано, виконаємо розрахунки за допомогою інструменту «розрахувати», іконка якого виглядає як калькулятор. Отримані результати приведено на рисунку 5.6.

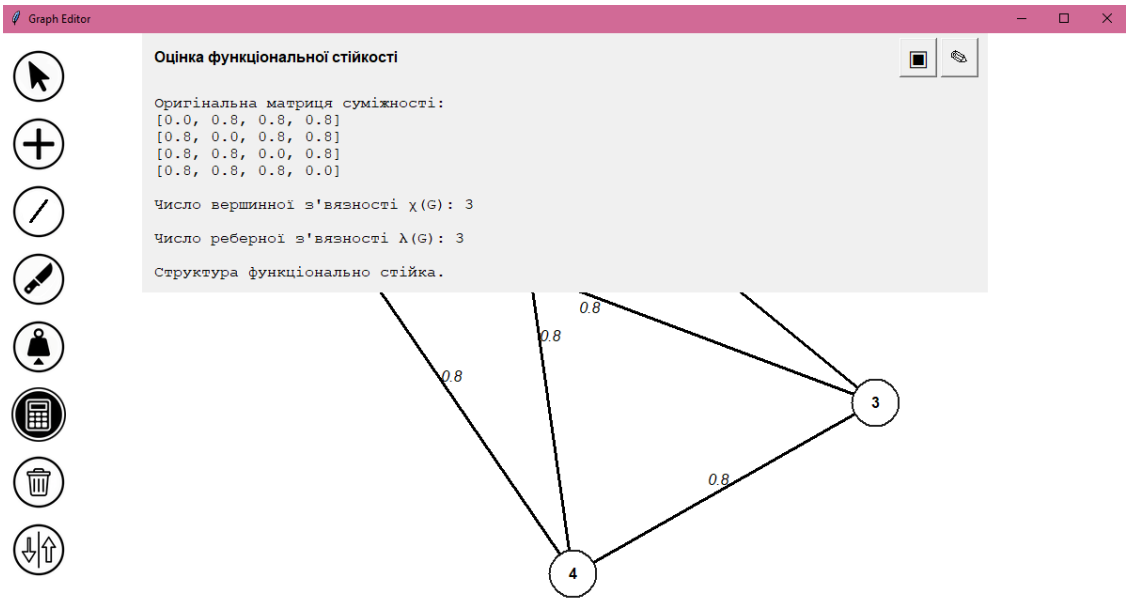

Рисунок 5.6 – Результат використання «розрахувати»

З'являться оверлей, у якому стисло виведені результати оцінки ФС. Виводиться матриця суміжності, в якій замість просто одиниць вказано імовірність безвідмовної роботи кожного ребра. Нижче знаходиться число вершинної зв’язності і число реберної зв’язності. Ще нижче – висновок щодо функціональної стійкості системи. У правому верхньому кутку також знаходяться 2 кнопки – перша дозволяє виконати повторно розрахунку для інших параметрів надійності ребер, друга – відкрити більш докладні результати у новому вікні. Відкриті у новому вікні результати наведено на рисунку 5.7.

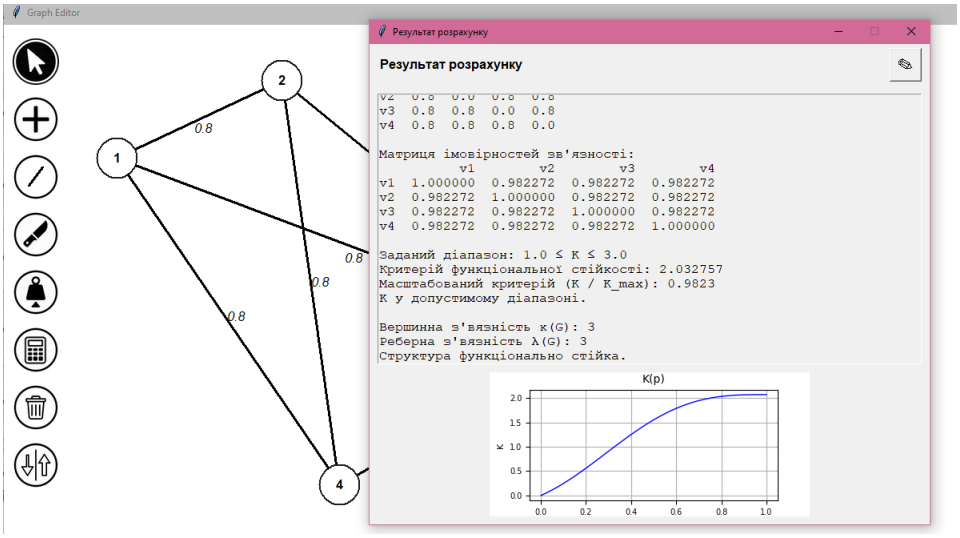

Рисунок 5.7 – Відкриті у новому вікні результати

У докладних результатах також виводиться матриця, яка утворюється після перетворення ненадійних вершин у надійні з ненадійними ребрами. У конкретному випадку вона не відрізняється від матриці суміжності, оскільки вершини надійній. Також виводиться матриця імовірностей зв’язності та розрахований на її основі критерій функціональної стійкості. Оскільки в цьому прикладі розглядаємо структуру, яка складається з 4 вершин, для розрахунку імовірностей зв’язності було використано метод простих ланцюгів. Якби вершин було п’ять або більше, було б використано метод Монте-Карло. Це зроблено для того, щоб розрахунки відбувалися швидше. Ще нижче знаходиться графік залежності критерію функціональної стійкості від імовірності зв’язності. Також у правому верхньому

кутку вікна наявна кнопка для виконання повторних розрахунків з іншими показниками надійності ребер. При натисканні на цю кнопку відкривається віконце з матрицею, де можна змінювати поточні значення. На рисунку 5.8 наведено матрицю, в якій можна міняти параметри надійності.

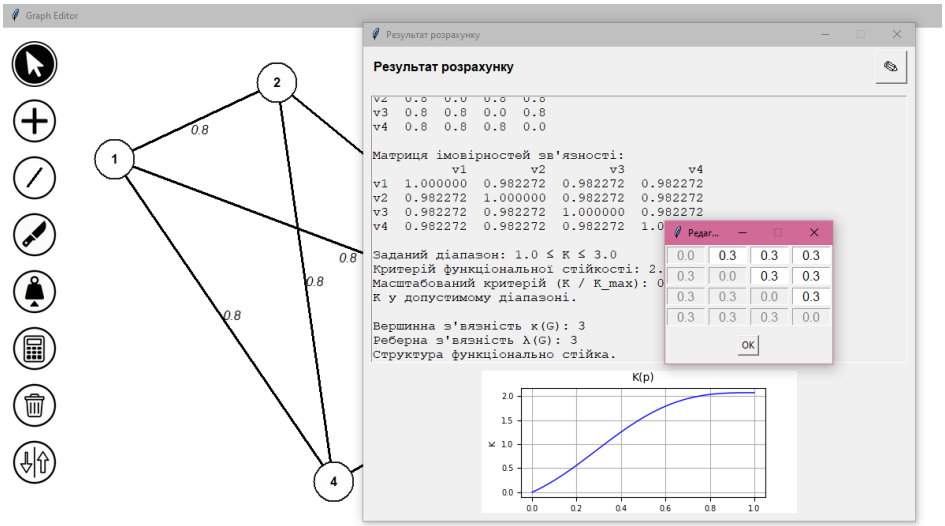

Рисунок 5.8 – Матриця для зміни надійності ребер

У цій матриці доступна для редагування лише половина над головною діагоналлю, оскільки граф є неорієнтованим. Значення для другої половини заповнюються автоматично в залежності від першої. Значення головної діагоналі також залишаються незмінними.

При натисканні «ОК» з'явиться новий оверлей з результатом (рисунок 5.9).

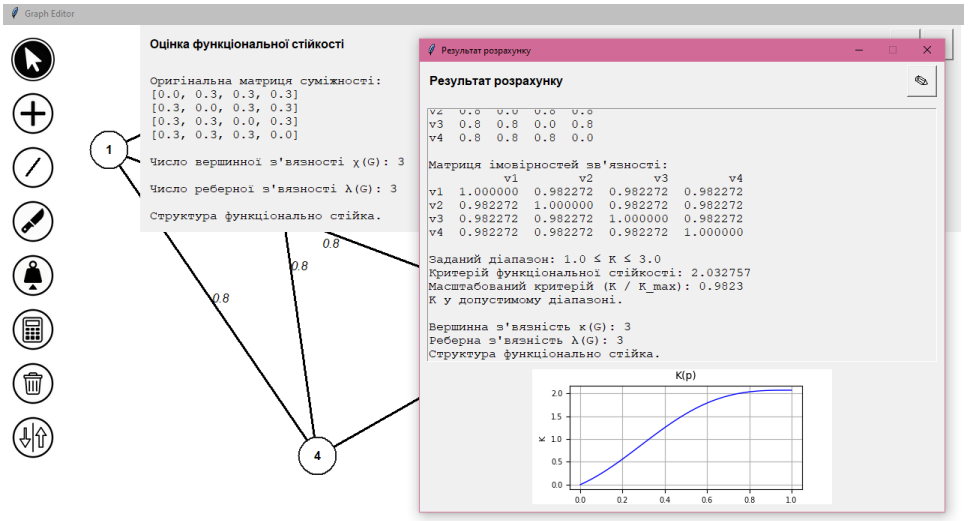

Рисунок 5.9 – Порівняння результатів

Так користувачеві буде зручно порівнювати оцінки функціональної стійкості при різних значеннях імовірностей знаходження ребер у справному стані.

У разі, коли структура є багатокомпонентною, вона автоматично вважається функціонально нестійкою. Тому розрахунку для неї не виконуються, а просто виводиться відповідне повідомлення (рисунок 5.10).

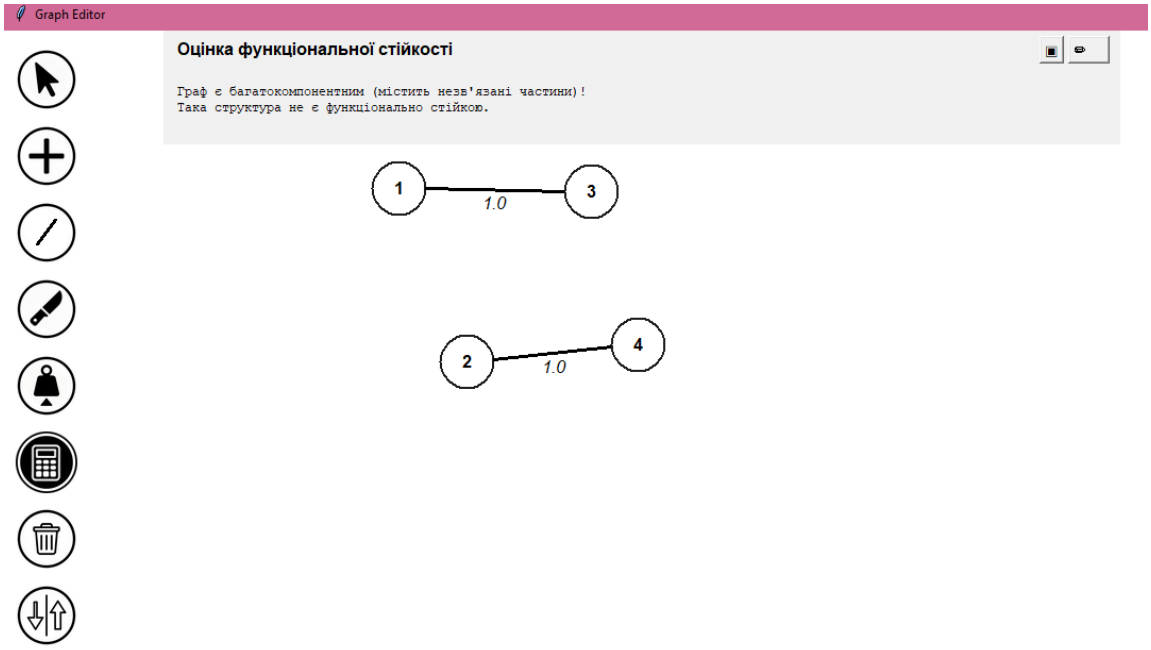

Рисунок 5.10 – Випадок багатокомпонентної структури

Також користувач може зберегти граф, або завантажити попередньо збережений. Для цього потрібно натиснути на відповідну кнопку і обрати дію, яку саме користувач хоче виконати.

Кнопка з іконкою смітника дозволяє повністю очистити робочий простір.

Також, за потреби, можна видаляти вершини. Для цього треба клікнути правою кнопкою миші по вершині. З’явиться спливаючий пункт меню, на який треба натиснути. При цьому не важливо, який інструмент буде обрано.

Щоб видалити ребро, необхідно обрати відповідний інструмент, іконка якого виглядає як ніж. Далі потрібно з затиснутою лівою кнопкою миші повести лінію, яка начебто перетинає ребро.

5.2 Системні вимоги

Мінімальними системними вимогами для роботи з розробленим програмним продуктом є:

– операційна система Windows;
– 2-ядерний процесор з тактовою частотою від 1.6 ГГц;
– мінімум 4 ГБ оперативної пам'яті;
– мінімум 250МБ вільного місця на диску;
– Python 3.8 або новіший.

5.3 Висновки до розділу

У цьому розділі було докладно описано роботу користувача з розробленим програмним продуктом. Описано графічний інтерфейс, панель інструментів та способи використання всіх інструментів. Також описано, які саме результати отримає користувач при оцінці системи. Наприкінці розділу наведено мінімальні системні вимоги, необхідні для роботи програми.

ВИСНОВКИ

У результаті написання дипломної роботи було розроблено програмний продукт для оцінки функціональної стійкості інтелектуальних систем підтримки прийняття рішень в умовах деструктивних впливів в критичних інфраструктурах. Було створено користувацький графічний інтерфейс для зручного задання архітектури ІСППР у вигляді графу з урахування імовірностей перебування компонентів у справному стані. Було реалізовано переведення архітектури системи з графічного вигляду (граф) у матричний. Також було реалізовано оцінку функціональної стійкості за числом вершинної та реберної зв’язності та за критерієм функціональної стійкості. Для вибору методу оцінки було проаналізовано багато літератури у відповідній галузі.

Серед основних проблем виявлених протягом дослідження варто виділити обмеженість вхідних даних та нестачу реальних прикладів. Так, правильність розрахунків перевірялася виключно на тестових даних, оскільки отримання реальних імовірностей перебування елементів системи у справному стані можливе тільки при наявності самої ІСППР.

З іншого боку, розроблений програмний продукт є універсальним інструментом для оцінки функціональної стійкості, у якому користувач може моделювати саме ту ІСППР, яка його цікавить.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Aqel M. J., Nakshabandi O. A., Adeniyi A. Decision Support Systems Classification in Industry // Periodicals of Engineering and Natural Sciences. – 2019. – Vol. 7, No. 2. – С. 774–785.

2. Chandrakar S., Bahadure N. B. Building Modern GUIs with Tkinter and Python: Building user-friendly GUI applications with ease. London: BPB Online, 2023. 476 с.

3. Collections – Container datatypes. URL: https://docs.python.org/3/library/collections.html (дата звернення: 05.05.2025).

4. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms. – 3rd ed. – Cambridge (Mass.) : MIT Press, 2009. – 1312 с.

5. Everitt B. S., Skrondal A. The Cambridge Dictionary of Statistics. 3rd ed. Cambridge : Cambridge University Press, 2010. – 430 с.

6. Gertsbakh I., Shpungin Y. Models of network reliability. Analysis, Combinatorics and Monte Carlo : research article // ResearchGate. January 2010. URL: https://www.researchgate.net/publication/267129765 (дата звернення: 01.05.2025).

7. Greg Jackson // Time. URL: https://time.com/7172582/greg-jackson/ (дата звернення: 30.04.2025).

8. Jain R. Decision Support Systems: an Overview // Decision Support System in Agriculture using Quantitative Analysis / за ред. Rajni Jain, S. S. Raju. Udaipur : Agrotech Publishing Academy, 2016. С. 42–50.

9. JSON – JSON encoder and decoder. URL: https://docs.python.org/3/library/json.html (дата звернення: 05.05.2025).

10. Lutz M. Learning Python. 4th ed. Sebastopol : O'Reilly Media, 2009.

11. Matplotlib: Visualization with Python. URL: https://matplotlib.org/stable/index.html (дата звернення: 05.05.2025).

12. NetworkX Documentation. URL: https://networkx.org/documentation/stable/ (дата звернення: 05.05.2025).

13. Pandas Documentation. URL: https://pandas.pydata.org/docs/ (дата звернення: 05.05.2025).

14. Phillips-Wren G. Intelligent Decision Support Systems // Multicriteria Decision Aid and Artificial Intelligence. – 2013. – С. 25–44.

15. Power D. J., Heavin C. Decision Support, Analytics, and Business Intelligence. New York : Business Expert Press, 2017.

16. Random – Generate pseudo-random numbers. URL: https://docs.python.org/3/library/random.html (дата звернення: 05.05.2025).

17. Tkinter – Python interface to Tcl/Tk. URL: https://docs.python.org/3/library/tkinter.html (дата звернення: 05.05.2025).

18. Verdigris enables ARM smart buildings // Verdigris. URL: https://www.verdigris.co/case-studies/verdigris-enables-arm-smart-buildings (дата звернення: 30.04.2025).

19. Барабаш О. В. Побудова функціонально стійких розподілених інформаційних систем : монографія. – Київ : НАОУ, 2004. – 226 с.

20. Калашник Г. А., Калашник-Рибалко М. А. Результати дослідження ефективності функціонування комплексу бортового обладнання літака DA-42 за показниками функціональної стійкості під впливом дестабілізуючих факторів // Системи обробки інформації. – 2020. – Вип. 4 (163). – С. 138–143. – DOI: 10.30748/soi.2020.163.03.

21. Макарчук А. В., Барабаш О. В., Саланда І. П. Вдосконалення ймовірнісного критерію функціональної стійкості багатомашинних інформаційних систем // Інформаційні моделюючі технології, системи та комплекси (ІМТСК-2024) : V міжнар. наук.-практ. конф., 18–19 квіт. 2024 р., м. Черкаси, Україна : зб. тез. Черкаси : Черкас. нац. ун-т ім. Б. Хмельницького, 2024. С. 51–53.

22. Про критичну інфраструктуру : Закон України від 15.12.2022 р. № 1882-IX // Відомості Верховної Ради України. – 2023. – № 5. – Ст. 13. – URL: https://zakon.rada.gov.ua/laws/show/1882-20 (дата звернення: 30.04.2025).

23. Суходоля О. М. Штучний інтелект в енергетиці : аналіт. доповідь. Київ : НІСД, 2022. 49 с. DOI: https://doi.org/10.53679/NISS-analytrep.2022.09 (дата звернення: 30.04.2025).

24. Шуклін Г. В., Барабаш О. В., Гребенніков А. Б. Властивості функціональної стійкості інтелектуальних систем під час ухвалення рішень // Проблеми програмування. – 2024. – № 4. – DOI: https://doi.org/10.15407/pp2024.04.089.

ДОДАТОК А Лістинг розробленої системи

View.py

```
import pandas as pd
import tkinter as tk


from tkinter import ttk, PhotoImage
import matplotlib.pyplot as plt
import networkx as nx
from matplotlib.backends.backend_tkagg import FigureCanvasTkAgg
from matplotlib.figure import Figure

class View(tk.Tk):
    VERTEX_RADIUS = 25
    DEFAULT_OUTLINE = "#1d1d1d"
    SELECT_OUTLINE = "#ff9500"
    TEXT_SHIFT = 12
    BASE_OFFSET = 40

    def __init__(self):
        super().__init__()
        self.title("Graph Editor")
        self.geometry("1200x800")
        self._matrix_editor_window = None
        self.tool_var = tk.StringVar(value="add_vertex")
        self.scale_factor = 1.0
        self._sliding_down = False
        self._sliding_up = False
        self.container = tk.Frame(self)
        self.container.pack(fill=tk.BOTH, expand=True)
        self.canvas = tk.Canvas(self.container, bg="white", highlightthickness=0)
        self.canvas.pack(fill=tk.BOTH, expand=True)
        self.bind_all("<Button-1>", self._on_global_click, add=True)
        self.overlay = tk.Frame(self.container, bg="white", highlightthickness=0)
        self.overlay.place(x=10, y=10)
        self.tool_buttons = {}
        self.icons = {}
        try:
            self.icons["calculate"] = PhotoImage(file="static/calculator.png")
        except Exception as e:
            print("Не вдалося завантажити static/calculator.png:", e)
            self.icons["calculate"] = None
        tools = [
            ("finger",         "cursor.png"),
            ("add_vertex",     "add.png"),
            ("edge_create",    "line.png"),
            ("cut",            "knife.png"),
            ("weight",         "weight.png"),
            ("calculate",      "calculator.png"),
            ("clear",          "clear.png"),
            ("import_export",  "import_export.png"),
        ]
        for tool_name, img_file in tools:
            # Шляхи до основної та активної версії
            img_path_default = f"static/{img_file}"
```

```
            img_path_active = f"static/{img_file.replace('.png', '_2.png')}"

            try:
                icon_default = tk.PhotoImage(file=img_path_default)
                icon_active = tk.PhotoImage(file=img_path_active)
            except Exception as e:
                print(f"Не вдалося завантажити іконку для {tool_name}:", e)
                continue

            # Зберігаємо обидві версії
            self.icons[tool_name] = {"default": icon_default, "active": icon_active}
            c = tk.Canvas(self.overlay, width=60, height=60, highlightthickness=0,
bg="white")
            x0, y0, r = 30, 30, 25
            oval = c.create_oval(x0 - r, y0 - r, x0 + r, y0 + r,
                                 fill="#e0e0e0", outline=self.DEFAULT_OUTLINE, width=2)
            c.create_image(x0, y0, image=icon_default, tags=("icon",))
            c.bind("<Button-1>", lambda e, v=tool_name: self._on_tool_click(v))
            c.pack(pady=6)
            self.tool_buttons[tool_name] = (c, oval)
        self.calc_overlay_height = 120
        self.calc_overlay = tk.Frame(
            self.container, bg="#f0f0f0", height=self.calc_overlay_height,
            bd=0, highlightthickness=0
        )
        self.calc_overlay.place(relx=0.125, y=-self.calc_overlay_height, relwidth=0.75)
        ttk.Label(self.calc_overlay, text="", font=("Arial", 16)).pack(pady=20)
        self.calc_overlay_visible = False

        self._highlight_selected(self.tool_var.get())

    def set_controller(self, controller):
        self.controller = controller

    def _highlight_selected(self, selected_val):
        for val, (canvas, oval) in self.tool_buttons.items():
            icon_img = self.icons[val]["active"] if val == selected_val else
self.icons[val]["default"]
            canvas.itemconfigure("icon", image=icon_img)

    def show_import_export_menu(self):
        menu = tk.Menu(self, tearoff=0)
        menu.add_command(label="Зберегти граф", command=self.controller.export_graph)
        menu.add_command(label="Завантажити граф", command=self.controller.import_graph)

        x = self.tool_buttons["import_export"][0].winfo_rootx()
        y = self.tool_buttons["import_export"][0].winfo_rooty() + 60
        menu.tk_popup(x, y)

    def _on_tool_click(self, val):
        self.tool_var.set(val)
        self._highlight_selected(val)
        self.controller.select_tool(val)

        if val == "calculate":
            if not self.calc_overlay_visible and not self._sliding_down:
                self._slide_overlay_down()
        else:
            if self.calc_overlay_visible and not self._sliding_up:
```

```
                self._slide_overlay_up()

    def _on_global_click(self, event):
        if not self.calc_overlay_visible:
            return
        widget = self.winfo_containing(event.x_root, event.y_root)
        if widget and (str(widget).startswith(str(self.calc_overlay)) or
                       widget in [cb for cb, _ in self.tool_buttons.values()]):
            return
        if not self._sliding_up:
            self._slide_overlay_up()

    def _slide_overlay_down(self):
        self._sliding_down = True
        y = self.calc_overlay.winfo_y()
        target = 0
        if y < target:
            y_new = min(y + 20, target)
            self.calc_overlay.place_configure(y=y_new)
            self.after(5, self._slide_overlay_down)
        else:
            self.calc_overlay_visible = True
            self._sliding_down = False

    def _slide_overlay_up(self):
        self._sliding_up = True
        y = self.calc_overlay.winfo_y()
        target = -self.calc_overlay.winfo_height()
        if y > target:
            y_new = max(y - 20, target)
            self.calc_overlay.place_configure(y=y_new)
            self.after(5, self._slide_overlay_up)
        else:
            self.calc_overlay_visible = False
            self._sliding_up = False

    def _on_close_matrix_editor(self, editor):
        editor.destroy()
        self._matrix_editor_window = None

    def draw_vertex(self, vid: int, x: int, y: int) -> tuple[int, int]:
        r = self.VERTEX_RADIUS * self.scale_factor
        circle = self.canvas.create_oval(
            x - r, y - r, x + r, y + r,
            fill="white", outline=self.DEFAULT_OUTLINE, width=2, tags=("vertex",)
        )
        text = self.canvas.create_text(
            x, y, text=str(vid), font=("Arial", 12, "bold"), tags=("vertex",)
        )
        return circle, text

    def show_calculation_result(self, result_text, original_matrix=None, weights=None,
k_vs_p=None, popup_result=None):
        for widget in self.calc_overlay.winfo_children():
            widget.destroy()

        top_frame = tk.Frame(self.calc_overlay, bg="#f0f0f0")
        top_frame.pack(fill="x", padx=10, pady=5)
```

```
        tk.Label(top_frame, text="Оцінка функціональної стійкості", font=("Arial", 12,
"bold"), bg="#f0f0f0").pack(side="left")

        if original_matrix:
            edit_btn = tk.Button(top_frame, text="✎", font=("Arial", 16), command=lambda:
self._open_matrix_editor(original_matrix, weights))
            edit_btn.pack(side="right")
            popup_btn = tk.Button(top_frame, text="🔳", font=("Arial", 16),
command=lambda: self._popup_result_window(
                popup_result or result_text,
                original_matrix, weights, k_vs_p
            ))
            popup_btn.pack(side="right", padx=4)

        label = tk.Label(self.calc_overlay, text=result_text, font=("Courier New", 12),
                         justify="left", anchor="nw", bg="#f0f0f0")

        label.pack(padx=10, pady=10, fill="both", expand=True)

        if not self.calc_overlay_visible and not self._sliding_down:
            self._slide_overlay_down()

    def _popup_result_window(self, result_text, original_matrix, weights, k_vs_p=None):
        self._slide_overlay_up()
        popup = tk.Toplevel(self)
        popup.title("Результат розрахунку")
        popup.geometry("700x600")
        popup.configure(bg="#f0f0f0")

        popup.resizable(False, False)

        top = tk.Frame(popup, bg="#f0f0f0")
        top.pack(fill="x", padx=10, pady=5)
        tk.Label(top, text="Результат розрахунку", font=("Arial", 12, "bold"),
bg="#f0f0f0").pack(side="left")
        #edit_btn = tk.Button(top, text="✎", command=lambda:
self._open_matrix_editor(original_matrix, weights))
        edit_btn = tk.Button(top, text="✎", font=("Arial", 16), command=lambda:
self._open_matrix_editor(original_matrix, weights))
        edit_btn.pack(side="right")
        content_frame = tk.Frame(popup, bg="#f0f0f0")
        content_frame.pack(fill="both", expand=True, padx=10, pady=10)
        text_box = tk.Text(content_frame, font=("Courier New", 12), bg="#f0f0f0",
wrap="none", height=18)
        text_box.insert("1.0", result_text)
        text_box.config(state="disabled")
        text_box.pack(fill="both", expand=True)
        if k_vs_p:
            ps, Ks = k_vs_p
            if ps and Ks:
                fig = Figure(figsize=(3.8, 2), dpi=100)
                ax = fig.add_subplot(111)
                ax.plot(ps, Ks, label="K(p)", color="blue", linewidth=1)
                ax.set_title("K(p)", fontsize=10)
                ax.set_xlabel("p", fontsize=8)
                ax.set_ylabel("K", fontsize=8)
                ax.tick_params(axis='both', labelsize=7)
                ax.grid(True)
                graph_frame = tk.Frame(content_frame, width=400, height=180, bg="#f0f0f0")
```

```
                graph_frame.pack_propagate(False)
                graph_frame.pack(side="bottom", pady=(10, 0))
                canvas = FigureCanvasTkAgg(fig, master=graph_frame)
                canvas.draw()
                canvas.get_tk_widget().pack(fill="both", expand=True)

    def plot_K_vs_p(self, ps, Ks, title="Графік залежності $K(p)$"):
        import matplotlib.pyplot as plt
        if not ps or not Ks:
            messagebox.showwarning("Немає даних", "Граф не має вершин або ребер для
побудови графіка.")
            return
        plt.figure(figsize=(7, 5))
        plt.plot(ps, Ks, label="K(p)", color="blue")
        plt.xlabel("Ймовірність працездатності ребра p")
        plt.ylabel("Функціональна стійкість K(p)")
        plt.title(title)
        plt.grid(True)
        plt.legend()
        plt.tight_layout()
        plt.show()
    def _on_matrix_submit(self, entries, editor, weights):
        new_matrix = []
        for row_entries in entries:
            row = []
            for e in row_entries:
                try:
                    row.append(float(e.get()))
                except ValueError:
                    row.append(0.0)
            new_matrix.append(row)
        old_edges = self.controller.model.edges[:]
        temp_edges = []
        n = len(new_matrix)
        for i in range(n):
            for j in range(i + 1, n):
                weight = new_matrix[i][j]
                if weight > 0:
                    temp_edges.append({'u': i + 1, 'v': j + 1, 'weight': weight})

        self.controller.model.edges = temp_edges
        old_vertices = self.controller.model.vertices.copy()
        for i, w in enumerate(weights):
            if i + 1 in self.controller.model.vertices:
                self.controller.model.vertices[i + 1]['weight'] = w
            else:
                self.controller.model.vertices[i + 1] = {'x': 0, 'y': 0, 'weight': w}
        self.controller.model.calculate(view=self)
        self.controller.model.edges = old_edges
        self.controller.model.vertices = old_vertices
        editor.destroy()



    def _open_matrix_editor(self, matrix, weights):
        if self._matrix_editor_window and
tk.Toplevel.winfo_exists(self._matrix_editor_window):
            self._matrix_editor_window.lift()
            return
```

```
        editor = tk.Toplevel(self)
        editor.title("Редагування матриці")
        editor.resizable(False, False)
        editor.protocol("WM_DELETE_WINDOW", lambda: self._on_close_matrix_editor(editor))
        self._matrix_editor_window = editor
        n = len(matrix)
        vars_matrix = [[tk.StringVar(value=f"{matrix[i][j]:.2f}") for j in range(n)] for i
in range(n)]
        entries = [[None] * n for _ in range(n)]

        for i in range(n):
            for j in range(n):
                var = vars_matrix[i][j]
                # Створюємо поле введення
                e = tk.Entry(editor, width=5, justify="center",font=("Arial", 12),
textvariable=var)
                e.grid(row=i, column=j, padx=2, pady=2)
                # Вимикаємо діагональ та нижній трикутник (симетрична матриця)
                if i == j or j < i:
                    e.config(state="disabled", disabledforeground="gray")
                # Додаємо обмеження для верхнього трикутника
                elif j > i:
                    def callback(var_i=i, var_j=j):
                        def on_change(*args):
                            val = vars_matrix[var_i][var_j].get()
                            try:
                                fval = float(val)
                                if fval < 0.001 or fval > 1.0:
                                    raise ValueError
                                vars_matrix[var_j][var_i].set(val)
                            except ValueError:
                                vars_matrix[var_i][var_j].set("0.00")
                                vars_matrix[var_j][var_i].set("0.00")
                        return on_change
                    var.trace_add("write", callback(i, j))
                entries[i][j] = e
        self._matrix_editor_entries = entries
        submit_btn = tk.Button(
            editor, text="OK",
            command=lambda: self._on_matrix_submit(entries, editor, weights)
        )
        submit_btn.grid(row=n, column=0, columnspan=n, pady=10)
```

matrix_math.py

```
import itertools
import random
from collections import deque
from typing import List, Tuple, Set, Optional
import pandas as pd

class MonteCarloConnectivity:
    def __init__(self, matrix: List[List[float]]):
        self.M = matrix
        self.n = len(matrix)
    def run(self, trials: int = 1000) -> pd.DataFrame:
```

```
        n = self.n
        counts = [[0]*n for _ in range(n)]
        for _ in range(trials):
            adj = [[False]*n for _ in range(n)]
            for i in range(n):
                for j in range(i+1, n):
                    if random.random() < self.M[i][j]:
                        adj[i][j] = adj[j][i] = True
            comp = [-1]*n
            cid = 0
            for v in range(n):
                if comp[v] == -1:
                    queue = deque([v])
                    comp[v] = cid
                    while queue:
                        u = queue.popleft()
                        for w in range(n):
                            if adj[u][w] and comp[w] == -1:
                                comp[w] = cid
                                queue.append(w)
                    cid += 1
            for i in range(n):
                for j in range(n):
                    if comp[i] == comp[j]:
                        counts[i][j] += 1

        R = [[counts[i][j]/trials for j in range(n)] for i in range(n)]
        return pd.DataFrame(R,
                            columns=[f"v{j+1}" for j in range(n)],
                            index=[f"v{i+1}" for i in range(n)])


# Клас метода простих ланцюгів
class SimplePathConnectivity:
    def __init__(self, matrix: List[List[float]], vertex: Optional[List[float]] = None):
        self.M = matrix
        self.n = len(matrix)
        self.Vp = vertex if vertex else [1.0] * self.n

    def _all_simple_paths(self, s: int, t: int) -> List[List[int]]:
        stack = [(s, [s])]
        res = []
        while stack:
            v, path = stack.pop()
            if v == t:
                res.append(path)
                continue
            for u in range(self.n):
                if self.M[v][u] > 0 and u not in path:
                    stack.append((u, path + [u]))
        return res

    @staticmethod
    def _edge_set(path: List[int]) -> Set[Tuple[int, int]]:
        return {tuple(sorted((a, b))) for a, b in zip(path, path[1:])}

    @staticmethod
    def _vertices_of(edges: Set[Tuple[int, int]]) -> Set[int]:
        vs = set()
```

```
        for u, v in edges:
            vs.update([u, v])
        return vs

    def connectivity_probability(self, s: int, t: int,
                                 include_endpoints: bool = True) -> float:
        paths = self._all_simple_paths(s, t)
        edge_sets = [self._edge_set(p) for p in paths]
        total = 0.0
        m = len(edge_sets)
        for k in range(1, m + 1):
            for combo in itertools.combinations(range(m), k):
                union_edges = set().union(*(edge_sets[i] for i in combo))
                p = 1.0
                for u, v in union_edges:
                    p *= self.M[u][v]

                total += ((-1) ** (k + 1)) * p
        return total

    def run(self, include_endpoints: bool = True) -> pd.DataFrame:
        R = [[0.0]*self.n for _ in range(self.n)]
        for i in range(self.n):
            R[i][i] = 1.0
            for j in range(i+1, self.n):
                p = self.connectivity_probability(i, j, include_endpoints)
                R[i][j] = R[j][i] = p
        return pd.DataFrame(R,
                            columns=[f"v{j+1}" for j in range(self.n)],
                            index=[f"v{i+1}" for i in range(self.n)])

# Клас критерія функціональної стійкості
class FunctionalRobustnessCalculator:
    def __init__(self, method_matrix: pd.DataFrame, adjacency: List[List[float]]):
        self.R = method_matrix
        self.M = adjacency
        self.n = len(adjacency)

    def _nmin_between(self, src: int, dst: int) -> int:
        dist = [-1] * self.n
        dist[src] = 0
        q = deque([src])
        while q:
            v = q.popleft()
            if v == dst:
                return dist[v]
            for u in range(self.n):
                if self.M[v][u] > 0 and dist[u] == -1:
                    dist[u] = dist[v] + 1
                    q.append(u)
        return 0

    def _nmin_matrix(self) -> pd.DataFrame:
        Nmin = [[0]*self.n for _ in range(self.n)]
        for i in range(self.n):
            for j in range(i+1, self.n):
                Nij = self._nmin_between(i, j)
                Nmin[i][j] = Nmin[j][i] = Nij
        return pd.DataFrame(Nmin,
```

```
                           columns=[f"v{j+1}" for j in range(self.n)],
                           index=[f"v{i+1}" for i in range(self.n)])

    def compute(self) -> float:
        N = self._nmin_matrix()
        total = 0.0
        for i in range(self.n):
            for j in range(self.n):
                if i == j:
                    continue
                weight = 1.0 / (abs(i - j) + 1)**2
                total += N.values[i, j] * self.R.values[i, j] * weight
        return total
```

Model.py

```
from matrix_math import *
from calculation_overlay import CalculationOverlay
import networkx as nx
import numpy as np

class Model:

    def __init__(self):
        self.vertices = {}
        self.edges = []
        self.MaxVertexNumber  = 5 #MaxVertexNumber перед використанням методу МонтеКарло

    def add_vertex(self, x: int, y: int) -> int: # Створення вершини з найменшим вільним
ID та заданими координатами (x, y)
        used_ids = set(self.vertices.keys())  # Множина вже зайнятих ID
        vid = 1 # Початковий кандидат для нового ID
        while vid in used_ids:
            vid += 1 # Пошук найменшого вільного ID
        self.vertices[vid] = {'x': x, 'y': y} # Додавання вершини до словника
        return vid # Повернення нового ID


    def remove_edge(self, u_vid: int, v_vid: int): # Видалення всіх ребер між вершинами
u_vid та v_vid
        self.edges = [
            e for e in self.edges
            if not (
                (e['u'] == u_vid and e['v'] == v_vid)
                or (e['u'] == v_vid and e['v'] == u_vid)
            )
        ]

    def _build_adjacency_and_weights(self): # Побудова списку ID вершин, матриці
суміжності з вагами ребер, списку ваг вершин
        ids = sorted(self.vertices.keys()) # # Відсортований список ID вершин
        id_to_index = {vid: i for i, vid in enumerate(ids)} # Відображення: ID вершини →
індекс у матриці
        n = len(ids)                                        # Кількість вершин
```

```
        mat = [[0.0 for _ in range(n)] for _ in range(n)]  # Ініціалізація n×n матриці суміжності нулями
        weights = [1.0 for _ in range(n)] # Початкові ваги вершин (за замовчуванням 1.0)

        for edge in self.edges:  # Заповнення матриці суміжності з урахуванням ваг ребер
            u = id_to_index[edge['u']]
            v = id_to_index[edge['v']]
            w = edge.get('weight', 1.0)
            mat[u][v] += w
            mat[v][u] += w
        for i, vid in enumerate(ids):  # Заповнення ваг вершин
            weights[i] = self.vertices[vid].get('weight', 1.0) # Вага вершини (або 1.0 за замовчуванням)
        return ids, mat, weights

    def _stabilize_graph(self, ids: List[int], mat: List[List[float]], weights: List[float]) -> Tuple[List[List[float]], List[float]]: #заміна кожної вершини з надійністю < 1.0 на дві надійні вершини
        new_mat = [row[:] for row in mat]          # Копія матриці суміжності
        new_weights = weights[:]                # Копія ваг вершин
        new_ids = ids[:]                     # Копія ID вершин
        n = len(weights)                      # Початкова кількість вершин
        for i in range(n):
            if weights[i] < 1.0:

                v_out_idx = len(new_mat)  # Додавання нової вершини v_out
                new_ids.append(max(new_ids) + 1)   # Призначення нового ID
                for row in new_mat:  # Розширення всіх рядків (новий стовпець)
                    row.append(0.0)
                new_mat.append([0.0] * (v_out_idx + 1)) # Додавання нового рядка в матрицю

                new_weights.append(1.0) #Присвоєння надійності новій вершині = 1.0 (надійна)
                new_weights[i] = 1.0  #Присвоєння надійності оригінальній вершині = 1.0
                new_mat[i][v_out_idx] = weights[i] #Створення з'єднання між двома надійними вершинами
                new_mat[v_out_idx][i] = weights[i]

                for j in range(n): # Перенесення всіх ребер з v_in → v_out
                    if j == i:
                        continue

                    if mat[i][j] > 0:
                        new_mat[v_out_idx][j] = mat[i][j]
                        new_mat[j][v_out_idx] = mat[i][j]
                        new_mat[i][j] = 0.0
                        new_mat[j][i] = 0.0
        return new_mat, new_weights


    def _compute_connectivity(self, mat, weights): #розрахунок матриці зв`язності
        if len(mat) > self.MaxVertexNumber:
            print("Використання методу Монте-Карло") # Кількість вершин більша заданого порогу -> Монте-Карло
            method = MonteCarloConnectivity(mat)
        else:
            print("Використання методу простих ланцюгів") # Кількість вершин менша заданого порогу -> прості ланцюги
            method = SimplePathConnectivity(mat, weights)
```

```
        return method.run()

    def _normalize_k(self, mat, weights): # Нормалізація поточного значення K
        ideal_weights = [1.0] * len(weights)  # Побудова ідеального вектора ваг (усі
вершини мають надійність 1.0)
        ideal_matrix = [[1.0 if val > 0 else 0.0 for val in row] for row in mat] #
Побудова ідеальної матриці
        method_class = SimplePathConnectivity if len(mat) <= self.MaxVertexNumber else
MonteCarloConnectivity # Вибір методу залежно від розміру графа
        R = method_class(ideal_matrix, ideal_weights).run() if method_class ==
SimplePathConnectivity else method_class(ideal_matrix).run()
        ideal_K = FunctionalRobustnessCalculator(R, ideal_matrix).compute() # Обчислення
ідеального K
        return self.K / ideal_K if ideal_K > 0 else 0.0 # Повернення нормалізованого K


    def _compute_connectivities(self, matrix): #Обчислення вершинної та реберної
зв’язності графа на основі матриці суміжності.
        G = nx.Graph() # Ініціалізація порожнього неорієнтованого графа
        n = len(matrix) # Кількість вершин
        for i in range(n):  # Побудова графа на основі матриці суміжності
            for j in range(i, n):
                if matrix[i][j] > 0: # Якщо існує з’єднання (ребро з вагою > 0)
                    G.add_edge(i, j)  # Додаємо ребро до графа
        vertex_conn = nx.node_connectivity(G) # Обчислення глобальної вершинної та
реберної зв’язності
        edge_conn = nx.edge_connectivity(G)
        return vertex_conn, edge_conn


    def _is_multicomponent(self) -> str:
        if not self.vertices:
            return "empty"
        if not self.edges:
            return "multicomponent"
        G = nx.Graph()
        for vid in self.vertices:
            G.add_node(vid)
        for edge in self.edges:
            G.add_edge(edge['u'], edge['v'])
        if G.number_of_nodes() == 0:
            return "empty"

        if not nx.is_connected(G):
            return "multicomponent"
        return None

    def calculate(self, view=None): # Обчислення функціональної стійкості графа

        multi_check = self._is_multicomponent()
        if multi_check == "empty":
            result_str = "Структуру не задано.\n"
            if view and hasattr(view, 'show_calculation_result'):
                view.show_calculation_result(
                    result_str,
                    original_matrix=[],
                    weights=[],
                    k_vs_p=([], []),
                    popup_result=result_str
```

```
                )
            return None
        if multi_check == "multicomponent":
            result_str = "Граф є багатокомпонентним (містить незв'язані частини)!\nТака
структура не є функціонально стійкою."
            if view and hasattr(view, 'show_calculation_result'):
                view.show_calculation_result(
                    result_str,
                    original_matrix=[],
                    weights=[],
                    k_vs_p=([], []),
                    popup_result=result_str
                )
            return None
        ids, mat, weights = self._build_adjacency_and_weights() # Побудова матриці
суміжності та вектору ваг
        self.original_adj_matrix = [row[:] for row in mat]      # Копія оригінальної
матриці

        old_weights = weights                                   # Збереження копії ваг для
пізнішого використання

        all_weights_one = all(w == 1.0 for w in weights) and all(   # Перевірка, чи всі
ваги дорівнюють 1
            edge.get('weight', 1.0) == 1.0 for edge in self.edges
        )
        vertex_conn, edge_conn = self._compute_connectivities(self.original_adj_matrix) #
Обчислення вершинної та реберної зв’язності
        if vertex_conn >= 2 and edge_conn >= 2:  # Формування опису стійкості графа на
основі зв’язності
            stability_str = "Структура функціонально стійка."
        elif((vertex_conn == 1 and edge_conn >= 2) or
            (edge_conn == 1 and vertex_conn >= 2) or
            (vertex_conn == 1 and edge_conn == 1)):
            stability_str = "Структура на межі функціональної стійкості."
        else:
            stability_str = "Структура не є функціонально стійкою."

        result_str = ( # Формування короткого тексту результату
            "Оригінальна матриця суміжності:\n"
            + "\n".join(str(row) for row in self.original_adj_matrix)
            + f"\n\nЧисло вершинної з'вязності κ(G): {vertex_conn}"
            + f"\n\nЧисло реберної з'вязності λ(G): {edge_conn}"
            + f"\n\n{stability_str}"
        )

        if all_weights_one:  # Якщо граф ідеально надійний (всі ваги = 1), додаткові
обчислення не потрібні
            if view and hasattr(view, 'show_calculation_result'):
                ps, Ks = self.calculate_K_vs_p()
                view.show_calculation_result(
                    result_str,
                    original_matrix=self.original_adj_matrix,
                    weights=weights,
                    k_vs_p=(ps, Ks),
                    popup_result=result_str
                )

            return None # Завершення — без подальших розрахунків
```

```
        mat, weights = self._stabilize_graph(ids, mat, weights)
        self.stabilized_adj_matrix = mat  # Збереження стабілізованої матриці
        self.connectivity_matrix = self._compute_connectivity(mat, weights) # Побудова
матриці ймовірностей зв'язності
        self.K = FunctionalRobustnessCalculator(self.connectivity_matrix, mat).compute() #
Обчислення функціонального критерію K
        self.normalized_K = self._normalize_k(mat, weights) # Обчислення нормалізованого K
(K / K_max)
        original_df = pd.DataFrame(
                self.original_adj_matrix,
                columns=[f"v{j+1}" for j in range(len(self.original_adj_matrix))], #
Побудова таблиць у вигляді DataFrame (для зручного виводу)
                index=[f"v{i+1}" for i in range(len(self.original_adj_matrix))]
            )

        stabilized_df = pd.DataFrame(
            self.stabilized_adj_matrix,
            columns=[f"v{j+1}" for j in range(len(self.stabilized_adj_matrix))],
            index=[f"v{i+1}" for i in range(len(self.stabilized_adj_matrix))]
        )
                                    # Формування повного тексту результатів
        result_str_long = (
            "Оригінальна матриця суміжності:\n"
            + original_df.to_string()
            + "\n\nМатриця після стабілізації (з додатковими вершинами):\n"
            + stabilized_df.to_string()
            + "\n\nМатриця імовірностей зв'язності:\n"
            + self.connectivity_matrix.to_string()
            + f"\n\nКритерій функціональної стійкості: {self.K:.6f}"
            + f"\nМасштабований критерій (K / K_max): {self.normalized_K:.4f}"
            + f"\n\nВершинна з'вязність κ(G): {vertex_conn}"
            + f"\nРеберна з'вязність λ(G): {edge_conn}"
            + f"\n\n{stability_str}"
        )
        if view and hasattr(view, 'show_calculation_result'): # Побудова кривої K(p) —
залежність критерію стійкості від імовірності відмови вузлів
            ps, Ks = self.calculate_K_vs_p()
            view.show_calculation_result(  #Виклик функції відображення результату у View
(інтерфейсі користувача)
                result_str,
                original_matrix=self.original_adj_matrix,
                weights=old_weights,
                k_vs_p=(ps, Ks),
                popup_result=result_str_long
            )

        return self.K
    def format_matrix(self,matrix: list[list[float]]) -> str: #Форматування матриці у
вигляді таблиці з підписами вершин
        n = len(matrix)
        header = "     " + "  ".join(f"v{j+1}" for j in range(n))
        rows = []
        for i in range(n):
            row_str = f"v{i+1} " + " ".join(f"{matrix[i][j]:.6f}" for j in range(n))
            rows.append(row_str)
        return header + "\n" + "\n".join(rows)

    def calculate_K_vs_p(self, step=0.025): # Обчислення графіка залежності K(p) —
критерію функціональної стійкості від імовірності надійності компонентів
```

```
        ids, adj, weights = self._build_adjacency_and_weights()  # Отримання списку
вершин, матриці суміжності та ваг
        if not ids:
            return [], [] # Якщо граф порожній — нічого рахувати

        n = len(adj)
        ps = [round(p, 6) for p in np.arange(0, 1.0001, step)]
        Ks = []                                                  # Список для збереження
значень критерію K(p)

        for p in ps:
            prob_matrix = [[p if adj[i][j] > 0 else 0.0 for j in range(n)] for i in
range(n)]
            ideal_weights = [1.0] * n

            if n > self.MaxVertexNumber:  # Вибір методу: точний або ймовірнісний, залежно
від розміру графа
                method = MonteCarloConnectivity(prob_matrix)
            else:
                method = SimplePathConnectivity(prob_matrix, ideal_weights)

            R = method.run()
            K = FunctionalRobustnessCalculator(R, prob_matrix).compute()
            Ks.append(K)

        return ps, Ks
```

Controller.py

```
import math
from tkinter import filedialog, messagebox
import json


from View import View           # Створюємо Model
from Model import Model         # Створюємо View

from add_vertex_tool import AddVertexTool           #Імпорт Інструментів
from CutTool import CutTool
from weight_tool import VertexWeightTool
from finger_tool import FingerTool
from clear_tool import ClearAllTool
from import_export_tool import ImportExportTool


from edge_create_tool import EdgeCreateTool


class Controller:
    def __init__(self, MVC_check=None):
        self.MVC_check = MVC_check
        self.model = Model()
        self.view  = View()
        self.view.set_controller(self)
        self._edge_views = []
```

```
        # Реєстрація інструментів
        self.tools = {
            "add_vertex": AddVertexTool(self),
            "cut":        CutTool(self),
            "weight":     VertexWeightTool(self),
            "finger":     FingerTool(self),
            "edge_create": EdgeCreateTool(self),
            "clear":      ClearAllTool(self),
        }

        self.current_tool = AddVertexTool(self)
        self._view_to_vid = {}
        self._vid_to_view = {}
        self._prev_tool = None
        self._shift_active = False
        self.view.bind_all('<KeyPress-Shift_L>',    self._on_shift_press)
        self.view.bind_all('<KeyRelease-Shift_L>',  self._on_shift_release)
        self.view.bind_all('<KeyPress-Shift_R>',    self._on_shift_press)
        self.view.bind_all('<KeyRelease-Shift_R>',  self._on_shift_release)
        self.view.canvas.bind('<MouseWheel>',      self.on_mouse_wheel)
        self.view.canvas.bind('<Button-4>',        self.on_mouse_wheel)
        self.view.canvas.bind('<Button-5>',        self.on_mouse_wheel)
        self.select_tool("add_vertex")


    def select_tool(self, name: str):
        if name == "calculate":   #Інструмент калькулятор
                self.model.calculate(view=self.view)    # Запускаємо обчислення моделі
                if self.current_tool:
                    self.current_tool.deactivate()      # Вимикаємо активний інструмент
                    self.current_tool = None          # Скидаємо поточний інструмент (щоб
випаково не змінювати канву, під час перегляду результату)
                return

        if name == "clear":     #Інструмент очищення
            self.clear_all()                    # Очищаємо граф (вершини, ребра тощо)
            if self.current_tool:
                self.current_tool.deactivate()      # Вимикаємо поточний інструмент
                self.current_tool = None
            self.select_tool("finger")          # Автоматично перемикаємося на інструмент
'finger' після очищення
            return

        if name == "import_export": # Інструмент імпорту та експорту
            self.view.show_import_export_menu()  # Відображаємо меню
            return

        if self.current_tool:       # Інші (більш звичайні) інструменти
            self.current_tool.deactivate()  # Вимикаємо попередній інструмент
        self.current_tool = self.tools.get(name)    # Отримуємо новий інструмент зі
словника
        if self.current_tool:
            self.current_tool.activate() # Активуємо новий інструмент

    def _on_shift_press(self, event):            #Обробник натискання клавіші Shift
        if not self._shift_active:
                                           # Зберігаємо назву активного інструмента (щоб
повернути його пізніше)
```

```
            self._prev_tool = self.view.tool_var.get()
            self._shift_active = True
                                        # Перемикаємося тимчасово на інструмент 'finger'
(переміщення)
            self.view.tool_var.set('finger')
            self.select_tool('finger')


    def _on_shift_release(self, event):        # повертає попередній інструмент (який був
до натискання Shift)
        if self._shift_active:
            self._shift_active = False
            if self._prev_tool:# Повертаємо попередній інструмент, якщо він збереженийт
                self.view.tool_var.set(self._prev_tool)
                self.select_tool(self._prev_tool)


    def on_mouse_wheel(self, event):
        """(zoom in/out мишею при затиснутому SHIFT)"""
        SHIFT_MASK = 0x0001                      #   Маска для перевірки натиснутої клавіші
Shift
        if not (event.state & SHIFT_MASK):  #   Якщо Shift не натиснутий — не виконуємо
зум
            return
        factor = 1.1 if event.delta > 0 else 0.9 #  використовуємо event.delta (Бо
операційна система Windows)
        x, y = event.x, event.y
        canv = self.view.canvas

        canv.scale('all', x, y, factor, factor) #   Масштабуємо всі елементи на канві
відносно точки курсора
        self.view.scale_factor *= factor  # Оновлюємо коефіцієнт масштабу


        for vid, v in self.model.vertices.items():      # Оновлюємо координати всіх вершин
у моделі
            v['x'] = x + (v['x'] - x) * factor
            v['y'] = y + (v['y'] - y) * factor

        for vid, (c_id, _) in self._vid_to_view.items():  # Перемальовуємо всі ребра
            self.relayout_edges_touching(c_id)

    def register_vertex_view(self, vid, circle_id, text_id):    # Реєструє зв’язки між
моделлю вершини та її візуальним представленням на канві
        self._view_to_vid[circle_id] = vid     # Створюємо зворотні зв’язки
        self._view_to_vid[text_id] = vid
        self._vid_to_view[vid] = (circle_id, text_id)    # Створюємо прямий зв’язок:
вершина → елементи канви (коло і текст)
        v = self.model.vertices.get(vid)       #    Додаємо ці ID в саму вершину в моделі,
якщо вона існує
        if v is not None:
            v['circle_id'] = circle_id
            v['text_id'] = text_id

    def get_vertex_items(self, item_id):                 #  За ID елемента повертає коло
та текст
        vid = self._view_to_vid.get(item_id)        # Отримуємо внутрішній ID вершини
(vid) за графічним item_id
        return self._vid_to_view.get(vid, (None, None)) #Повертаємо пов'язане з цією
```

```
вершиною коло та текст

    def get_vertex_id(self, circle_id):         #внутрішній ID вершини (vid) за графічним
ID кола (circle_id)
        return self._view_to_vid.get(circle_id)

    def _unit_dir(self, x1, y1, x2, y2): #Обчислює одиничний вектор (напрямок з довжиною
1) від точки (x1, y1) до точки (x2, y2)
        dx, dy = x2 - x1, y2 - y1
        dist = math.hypot(dx, dy) or 1.0
        return dx / dist, dy / dist

    def _clip_endpoints(self, x1, y1, x2, y2):  #Замість лінії "від центру до центру",
вона малюється "від краю до краю" кіл-вершин
        ux, uy = self._unit_dir(x1, y1, x2, y2)
        r = self.view.VERTEX_RADIUS * self.view.scale_factor  #Визначаємо радіус вершини з
урахуванням масштабу
        return (
            x1 + ux * r, y1 + uy * r, # початок обрізаного ребра
            x2 - ux * r, y2 - uy * r  # кінець обрізаного ребра
        )

    def add_edge(self, u_vid, v_vid, weight=1.0):
        """
        Додає в Model і на канву нове ребро, між вершинами u_vid і v_vid, у випадку,
        якщо такого ребра ще немає. Ребра графу малюються від зовнішньої точки кола до
такої ж точки іншого
        """

        for edge in self.model.edges:                        #Перевіряємо, чи таке ребро
вже існує у моделі
            if {edge['u'], edge['v']} == {u_vid, v_vid}:    # Якщо ребро вже є, то не
додаємо знову
                return

                                                             # Якщо ребра ще немає —
створюємо словник-опис ребра з вершинами та вагою
        edge_model = {'u': u_vid, 'v': v_vid, 'weight': weight}
        self.model.edges.append(edge_model)                   # Додаємо його до списку
ребер моделі


        u = self.model.vertices[u_vid]                       # Отримуємо координати
вершин u та v для малювання ребра
        v = self.model.vertices[v_vid]

        px1, py1, px2, py2 = self._clip_endpoints(u['x'], u['y'], v['x'], v['y']) #
Вираховуємо координати кінців ребра, обрізаючи його до країв кіл вершин


        line_id = self.view.canvas.create_line(     # Малюємо лінію (ребро) між вершинами
на канві
            px1, py1, px2, py2,
            width=3, fill="black", tags=("edge",)   #  Вказуємо товщину, колір та тег для
подальшої взаємодії
        )

        ux, uy = self._unit_dir(px1, py1, px2, py2) #Обчислюємо одиничний напрямок ребра
до іншого (вектор напряму)
```

```
        perp_x, perp_y = -uy, ux                                    #Обчислюємо
перпендикулярний напрямок до ребра, щоб розмістити текст ваги збоку
        shift = self.view.TEXT_SHIFT * self.view.scale_factor   # Зсув для позиціювання
тексту
        mx = (px1 + px2) / 2 + perp_x * shift              # Центр ребра + зсув по
перпендикуляру (X)
        my = (py1 + py2) / 2 + perp_y * shift              # Центр ребра + зсув по
перпендикуляру (Y)

        text_id = self.view.canvas.create_text(                  #Малюємо текст ваги ребра
на канві
            mx, my,
            text=str(weight),                                    #   Відображаємо вагу ребра
            font=("Arial", 12, "italic"), tags=("edge",)    #   Шрифт та тег
        )
        self._edge_views.append({ # Зберігаємо дані про візуальне представлення ребра в
окремий список
            'u':      u_vid,
            'v':      v_vid,
            'line':   line_id,          # Ідентифікатор лінії на канві
            'text':   text_id,          # Ідентифікатор тексту на канві
            'weight': weight,           # Вага ребра
        })

    def relayout_edges_touching(self, circle_id): # Оновлює положення ребер та їхніх міток
(ваг), які підключені до вершини з заданим ID кола
        vid = self.get_vertex_id(circle_id) # Отримуємо внутрішній ID вершини за графічним
ID кола

        for ed in self._edge_views: #Перебираємо всі ребра у візуальному представленні
(_edge_views)
            u, v = ed['u'], ed['v']
            if vid in (u, v):           #Якщо поточне ребро підключене до цієї вершини —
оновлюємо його
                u_coords = self.model.vertices[u]   #Отримуємо координати обох вершин
цього ребра
                v_coords = self.model.vertices[v]

                px1, py1, px2, py2 = self._clip_endpoints(  #Обрізаємо лінію ребра до
країв кіл вершин
                    u_coords['x'], u_coords['y'],
                    v_coords['x'], v_coords['y']
                )
                self.view.canvas.coords(ed['line'], px1, py1, px2, py2) #Оновлюємо
координати лінії ребра на канві

                ux, uy = self._unit_dir(px1, py1, px2, py2) #Обчислюємо одиничний вектор
напряму ребра
                perp_x, perp_y = -uy, ux                                    #Обчислюємо
вектор, перпендикулярний до ребра
                shift = self.view.TEXT_SHIFT * self.view.scale_factor   #Обчислюємо
зміщення тексту
                mx = (px1 + px2) / 2 + perp_x * shift
                my = (py1 + py2) / 2 + perp_y * shift
                self.view.canvas.coords(ed['text'], mx, my) #Оновлюємо координати тексту
на канві

    def orient(self, a, b, c):     #Обчислює орієнтацію трьох точок a, b, c
```

```
        return (b[0]-a[0])*(c[1]-a[1]) - (b[1]-a[1])*(c[0]-a[0])

    def on_seg(self, a, b, c):    # Перевіряє, чи лежить точка b на відрізку ac
        return min(a[0], c[0]) <= b[0] <= max(a[0], c[0]) and min(a[1], c[1]) <= b[1] <=
max(a[1], c[1])

    def intersect(self, p1, p2, p3, p4):          #Перевіряє, чи перетинаються два
відрізки: p1-p2 та p3-p4
        o1, o2 = self.orient(p1, p2, p3), self.orient(p1, p2, p4)
        o3, o4 = self.orient(p3, p4, p1), self.orient(p3, p4, p2)
        if o1 * o2 < 0 and o3 * o4 < 0:          # Загальний випадок: відрізки
перетинаються, якщо орієнтації протилежні
            return True
                                                 # Особливі випадки: точки лежать на одному
з відрізків
        if o1 == 0 and self.on_seg(p1, p3, p2): return True
        if o2 == 0 and self.on_seg(p1, p4, p2): return True
        if o3 == 0 and self.on_seg(p3, p1, p4): return True
        if o4 == 0 and self.on_seg(p3, p2, p4): return True
        return False

    def cut_edges(self, path: list[tuple[int, int]]):   #Видалення ребер, які
перетинаються з лінією інструмента CutTool
        canvas = self.view.canvas
        removed_pairs = []              # Список для збереження пар (u, v), які треба
видалити з моделі
        for ed in list(self._edge_views):  # Копія списку, для подальшої зміни
            u, v = ed['u'], ed['v']
            x1, y1, x2, y2 = self._clip_endpoints(  # Отримання координат кінців ребра з
урахуванням обрізання до меж кіл
                self.model.vertices[u]['x'], self.model.vertices[u]['y'],
                self.model.vertices[v]['x'], self.model.vertices[v]['y']
            )
            p1, p2 = (x1, y1), (x2, y2)
            for i in range(len(path) - 1):  # Перевірка кожну лінію з path на перетин із
ребром p1-p2
                if self.intersect(p1, p2, path[i], path[i + 1]):
                    canvas.delete(ed['line']) #Якщо є перетин — видаляємо ребро з канви
                    canvas.delete(ed['text'])
                    removed_pairs.append((u, v)) # Запис, для видалення з Model
                    self._edge_views.remove(ed)
                    break

        for u, v in removed_pairs:  # Видаляємо знайдені ребра з Model
            self.model.edges = [
                e for e in self.model.edges
                if not (e['u'] == u and e['v'] == v or e['u'] == v and e['v'] == u)
            ]

    def delete_vertex(self, vid): #Видаляє вершину з графа разом з усіма пов’язаними
ребрами та оновлює ID інших вершин і ребер для збереження послідовної нумерації.
        if vid not in self.model.vertices:  #   Перевірка, чи така вершина існує
            return
        to_remove = []         # Пошук та видалення всіх ребер з канви, які з’єднані з
цією вершиною
        for ed in list(self._edge_views):
            if vid in (ed['u'], ed['v']):
                if 'line' in ed:
                    self.view.canvas.delete(ed['line'])
```

```
                if 'text' in ed:
                    self.view.canvas.delete(ed['text'])
                to_remove.append(ed)
        for ed in to_remove:
            self._edge_views.remove(ed)
            if ed in self.model.edges:
                self.model.edges.remove(ed)
        items = self._vid_to_view.pop(vid, ()) #Видалення вершини з канви
        for item_id in items:
            self._view_to_vid.pop(item_id, None)
            self.view.canvas.delete(item_id)
        self.model.vertices.pop(vid, None) #Видалення вершини з Model
        old_ids = sorted(self.model.vertices.keys())  #Зміна ID вершини, щоб нумерації
ішли підряд
        old_to_new = {old_id: new_id for new_id, old_id in enumerate(old_ids, start=1)}
        new_vertices = {}  # Оновлення словника вершин з новими ID
        new_vid_to_view = {}
        new_view_to_vid = {}
        for old_id, new_id in old_to_new.items():
            v = self.model.vertices[old_id]
            new_vertices[new_id] = v
            v['circle_id'], v['text_id'] = self._vid_to_view[old_id] # оновлення view-
зв'язки
            new_vid_to_view[new_id] = self._vid_to_view[old_id]
            new_view_to_vid[v['circle_id']] = new_id
            new_view_to_vid[v['text_id']] = new_id
                                                        # оновлення тексту вершини
(номер)
            self.view.canvas.itemconfigure(v['text_id'], text=str(new_id))
        self.model.vertices = new_vertices
        self._vid_to_view = new_vid_to_view
        self._view_to_vid = new_view_to_vid
        new_edges = []  #Оновлюємо ребра згідно з новими ID
        for edge in self.model.edges:
            if edge['u'] in old_to_new and edge['v'] in old_to_new:
                u_new = old_to_new[edge['u']]
                v_new = old_to_new[edge['v']]
                new_edges.append({
                    'u': u_new,
                    'v': v_new,
                    'weight': edge.get('weight', 1.0) #зберігаємо вагу
                })
        self.model.edges = new_edges
                                        #Оновлення вершин в edge_views для
перемальовування
        for ed in self._edge_views:
            ed['u'] = old_to_new.get(ed['u'], ed['u'])
            ed['v'] = old_to_new.get(ed['v'], ed['v'])

    def clear_all(self): #Видалення всіх вершин
        for vid in list(self.model.vertices.keys()):  # Копія списку ключів (ID вершин)
            items = self._vid_to_view.pop(vid, [])   # Отримання (circle_id, text_id) для
вершини
            for item_id in items:
                self._view_to_vid.pop(item_id, None) # Видалення зворотного посилання
                self.view.canvas.delete(item_id)   # Видалення елементів з канви
        self.model.vertices.clear() # Повна очистка словника вершин у Model
        for edge in self._edge_views:
            self.view.canvas.delete(edge['line']) # Видалення лінії ребра
```

```
            self.view.canvas.delete(edge['text'])   # Очистка списку візуальних ребер
        self._edge_views.clear()                    # Очищаємо список ребер у Model
        self.model.edges.clear()
        self.view.canvas.delete("vertex_weight") # видалення міток ваг вершин

    def export_graph(self):   #Експортує граф у JSON-файл
        path = filedialog.asksaveasfilename(defaultextension=".json", filetypes=[("JSON
files", "*.json")]) # Відкриття вікна збереження файлу
        if not path:
            return # Користувач скасував вибір
        data = {  # Створення структури даних для збереження
            "vertices": {
                str(vid): {
                    "x": v["x"],
                    "y": v["y"],
                    "weight": v.get("weight", 1.0)
                } for vid, v in self.model.vertices.items()
            },
            "edges": [
                {
                    "u": e["u"],
                    "v": e["v"],
                    "weight": e.get("weight", 1.0)
                } for e in self.model.edges
            ]
        }
        with open(path, "w", encoding="utf-8") as f: # Записуємо структуру у файл у
форматі JSON
            json.dump(data, f, indent=2)
        messagebox.showinfo("Експорт завершено", f"Граф успішно збережено в
файл:\n{path}") # Повідомлення про успішне збереження


    def import_graph(self): #Імпортує граф із JSON-файлу
        path = filedialog.askopenfilename(filetypes=[("JSON files", "*.json")]) #
Відкриття вікна завантаження файлу
        if not path:
            return
        try:                                                # Спроба прочитати файл
            with open(path, "r", encoding="utf-8") as f:
                data = json.load(f)
        except Exception as e:
            messagebox.showerror("Помилка читання файлу", str(e))
            return
        self.clear_all()                                                # Очищення
поточного графу перед завантаженням нового
        for vid_str, v in data.get("vertices", {}).items(): # Завантаження вершин
            vid = int(vid_str)
            self.model.vertices[vid] = {
                "x": v["x"],
                "y": v["y"],
                "weight": v.get("weight", 1.0)
            }
            circle, text = self.view.draw_vertex(vid, v["x"], v["y"]) # Відображення
вершини на канві та реєстрація її
            self.register_vertex_view(vid, circle, text)
        for e in data.get("edges", []):                                 #Завантаження ребер
            u, v = e["u"], e["v"]
            self.add_edge(u, v, weight=e.get("weight", 1.0))
```

```
    def run(self):
        if self.MVC_check:
            self.MVC_check('OK')
        self.view.mainloop()
```

tool.py

```
import tkinter as tk
from abc import ABC, abstractmethod

class Tool(ABC):
    def __init__(self, controller):
        self.controller = controller

    def activate(self):
        canvas = self.controller.view.canvas
        for ev in ("<ButtonPress-1>", "<B1-Motion>", "<ButtonRelease-1>"):
            canvas.unbind(ev)
        self._bind_events()
        canvas.tag_bind("vertex", "<Button-3>", self._on_vertex_right_click)

    def deactivate(self):
        canvas = self.controller.view.canvas
        for ev in ("<ButtonPress-1>", "<B1-Motion>", "<ButtonRelease-1>"):
            canvas.unbind(ev)

    def _on_vertex_right_click(self, event):
        canvas = self.controller.view.canvas
        item = canvas.find_withtag("current")
        if not item:
            return
        item_id = item[0]
        vid = self.controller.get_vertex_id(item_id)
        if vid is None:
            return
        menu = tk.Menu(canvas, tearoff=0)
        menu.add_command(label="Видалити вершину", command=lambda:
self.controller.delete_vertex(vid))
        menu.tk_popup(event.x_root, event.y_root)

    @abstractmethod
    def _bind_events(self):
        pass
```

weight_tool.py

```
from tkinter import simpledialog
from tool import Tool

class VertexWeightTool(Tool):
    def _bind_events(self):
        canvas = self.controller.view.canvas
```

```
        self._bindings = [
            ("vertex", "<Double-Button-1>", self._on_vertex_double),
            ("edge",   "<Double-Button-1>", self._on_edge_double),
        ]
        for tag, event, handler in self._bindings:
            canvas.tag_bind(tag, event, handler)

    def activate(self):
        super().activate()
        self.controller.view.canvas.config(cursor="question_arrow")

    def deactivate(self):
        super().deactivate()
        canvas = self.controller.view.canvas
        canvas.config(cursor="")
        for tag, event, handler in getattr(self, "_bindings", []):
            canvas.tag_unbind(tag, event)
        self._bindings = []

    def _on_vertex_double(self, event):
        canvas = self.controller.view.canvas
        it = canvas.find_withtag("current")[0]
        vid = self.controller.get_vertex_id(it)
        v = self.controller.model.vertices[vid]
        current_weight = v.get("weight", 1.0)
        new_weight = simpledialog.askfloat(
            "Вага вершини", "Введіть вагу (0 < вага ≤ 1):",
            initialvalue=current_weight, minvalue=0.001, maxvalue=1.0
            )
        if new_weight is None or new_weight <= 0.0 or new_weight > 1.0:
            return
        v["weight"] = new_weight
        x, y = v["x"], v["y"]
        r = self.controller.view.VERTEX_RADIUS * self.controller.view.scale_factor
        text_x = x
        text_y = y - r - self.controller.view.TEXT_SHIFT

        if not v.get("weight_text_id"):
            v["weight_text_id"] = canvas.create_text(
                text_x, text_y,
                text=f"{new_weight:.3f}".rstrip("0").rstrip("."),
                font=("Arial", 10), fill="black",
                tags=("vertex_weight",)  # ВАЖНО
            )

        else:
            canvas.coords(v["weight_text_id"], text_x, text_y)
            canvas.itemconfigure(v["weight_text_id"],
                                 text=f"{new_weight:.3f}".rstrip("0").rstrip("."))

    def _on_edge_double(self, event):
        canvas = self.controller.view.canvas
        it = canvas.find_withtag("current")[0]

        for ed in self.controller._edge_views:
            if it in (ed["line"], ed["text"]):
                break
        else:
            return
```

```
        current_weight = ed.get("weight", 1.0)
        new_weight = simpledialog.askfloat(
            "Вага вершини", "Введіть вагу (0 < вага ≤ 1):",
            initialvalue=current_weight, minvalue=0.001, maxvalue=1.0
            )
        if new_weight is None or new_weight <= 0.0 or new_weight > 1.0:
            return
        ed["weight"] = new_weight
        canvas.itemconfigure(ed["text"], text=f"{new_weight:.3f}".rstrip("0").rstrip("."))

        for med in self.controller.model.edges:
            u, v = med['u'], med['v']
            if (u == ed['u'] and v == ed['v']) or (u == ed['v'] and v == ed['u']):
                med['weight'] = new_weight
                break
        else:
            self.controller.model.edges.append({
                'u': ed['u'],
                'v': ed['v'],
                'weight': new_weight
            })
```

add_vertex_tool.py

```
from tool import Tool
from random import randint

class AddVertexTool(Tool):


    def _bind_events(self):
        canvas = self.controller.view.canvas
        canvas.bind("<ButtonPress-1>", self._on_click)

    def _on_click(self, event):
        x, y = event.x, event.y
        vid = self.controller.model.add_vertex(x, y)
        circle_id, text_id = self.controller.view.draw_vertex(vid, x, y)
        self.controller.register_vertex_view(vid, circle_id, text_id)


    def activate(self):
        super().activate()
        self.controller.view.canvas.config(cursor="crosshair")

    def deactivate(self):
        super().deactivate()
        self.controller.view.canvas.config(cursor="")
```

calculation_overlay.py

```
import tkinter as tk

class CalculationOverlay(tk.Toplevel):
    def __init__(self, master, result_text: str):
        super().__init__(master)
        self.withdraw()
        self.overrideredirect(True)
        self.attributes("-topmost", True)
        self.attributes("-alpha", 0.0)

        self.canvas = tk.Canvas(self, bg='black', highlightthickness=0)
        self.canvas.pack(fill=tk.BOTH, expand=True)

        self.result_frame = tk.Frame(self.canvas, bg='white', bd=2, relief='ridge')
        self.result_label = tk.Label(self.result_frame, text=result_text, justify="left",
                                     bg="white", anchor='nw', font=("Courier New", 10))
        self.result_label.pack(padx=20, pady=20)
        self.result_frame.place(relx=0.5, rely=0.5, anchor="center")

        self.update_idletasks()

self.geometry(f"{master.winfo_width()}x{master.winfo_height()}+{master.winfo_rootx()}+{mas
ter.winfo_rooty()}")

    def show_with_fade(self):
        self.deiconify()
        self.lift()
        self._fade_in(0.0)

    def _fade_in(self, alpha):
        if alpha <= 0.25:
            self.attributes("-alpha", alpha)
            self.after(20, lambda: self._fade_in(alpha + 0.02))
        else:
            self.attributes("-alpha", 0.25)
```

clear_tool.py

```
from tool import Tool

class ClearAllTool(Tool):
    def _bind_events(self):
        self.controller.clear_all()
```

cutTool.py

```
from tool import Tool

class CutTool(Tool):
    CUT_COLOR    = "#ff3b30"
    DASH_PATTERN = (4, 2)
```

```
def __init__(self, controller):
    super().__init__(controller)
    self._cut_path = []
    self._cut_line = None

def _bind_events(self):

    canvas = self.controller.view.canvas
    canvas.bind("<ButtonPress-1>",   self._on_press)
    canvas.bind("<B1-Motion>",       self._on_drag)
    canvas.bind("<ButtonRelease-1>", self._on_release)

def activate(self):
    super().activate()
    canvas = self.controller.view.canvas
    try:
        canvas.config(cursor='@static/knife.cur')
    except tk.TclError:
        canvas.config(cursor="crosshair")

def deactivate(self):
    super().deactivate()
    canvas = self.controller.view.canvas
    canvas.config(cursor="")
    if self._cut_line:
        canvas.delete(self._cut_line)
        self._cut_line = None
        self._cut_path = []

def _on_press(self, event):
    self._cut_path = [(event.x, event.y)]
    self._cut_line = self.controller.view.canvas.create_line(
        event.x, event.y, event.x, event.y,
        fill=self.CUT_COLOR, width=3, dash=self.DASH_PATTERN
    )
    return "break"

def _on_drag(self, event):
    self._cut_path.append((event.x, event.y))
    coords = sum(self._cut_path, ())
    self.controller.view.canvas.coords(self._cut_line, *coords)
    return "break"

def _on_release(self, event):
    if len(self._cut_path) > 1:
        self.controller.cut_edges(self._cut_path)
    canvas = self.controller.view.canvas
    canvas.delete(self._cut_line)
    self._cut_path = []
    self._cut_line = None
    return "break"
```

edge_create_tool.py

```
from tool import Tool
import tkinter as tk

class EdgeCreateTool(Tool):

    def __init__(self, controller):
        super().__init__(controller)
        self._source_vid = None
        self._highlighted_circle_id = None

    def activate(self):
        super().activate()
        self.controller.view.canvas.config(cursor="crosshair")

    def deactivate(self):
        super().deactivate()
        canvas = self.controller.view.canvas
        canvas.unbind("<Double-Button-1>")
        canvas.unbind("<Button-1>")
        self._clear_selection()
        canvas.config(cursor="")

    def _bind_events(self):
        canvas = self.controller.view.canvas
        canvas.bind("<Double-Button-1>", self._on_double_click)
        canvas.bind("<Button-1>", self._on_single_click)

    def _on_double_click(self, event):
        canvas = self.controller.view.canvas
        items = canvas.find_overlapping(event.x, event.y, event.x, event.y)
        vertex_items = [it for it in items if 'vertex' in canvas.gettags(it)]

        if not vertex_items:
            self._clear_selection()
            return "break"

        item_id = vertex_items[0]
        vid = self.controller.get_vertex_id(item_id)
        circle_id, _ = self.controller.get_vertex_items(item_id)

        self._source_vid = vid
        self._highlight_vertex(circle_id)
        return "break"

    def _on_single_click(self, event):
        if self._source_vid is None:
            return "break"

        canvas = self.controller.view.canvas
        items = canvas.find_overlapping(event.x, event.y, event.x, event.y)
        vertex_items = [it for it in items if 'vertex' in canvas.gettags(it)]

        if not vertex_items:
            return "break"

        item_id = vertex_items[0]
        vid = self.controller.get_vertex_id(item_id)
```

```
        if vid != self._source_vid:
            self.controller.add_edge(self._source_vid, vid, weight=1.0)

        self._clear_selection()
        return "break"

    def _highlight_vertex(self, circle_id):
        if self._highlighted_circle_id:
            self.controller.view.canvas.itemconfigure(
                self._highlighted_circle_id,
                outline=self.controller.view.DEFAULT_OUTLINE,
                width=2
            )
        self._highlighted_circle_id = circle_id
        self.controller.view.canvas.itemconfigure(circle_id, outline="#ff9500", width=3)

    def _clear_selection(self):
        if self._highlighted_circle_id:
            self.controller.view.canvas.itemconfigure(
                self._highlighted_circle_id,
                outline=self.controller.view.DEFAULT_OUTLINE,
                width=2
            )
        self._highlighted_circle_id = None
        self._source_vid = None
```

finger_tool.py

```
from tool import Tool
import tkinter as tk
class FingerTool(Tool):
    """
    Інструмент «Палець»:

    """
    SHIFT_MASK = 0x0001
    def __init__(self, controller):
        super().__init__(controller)
        self._drag_origin = None
        self._mode = None
        self._vdata = None

    def _bind_events(self):
        canvas = self.controller.view.canvas
        canvas.bind("<ButtonPress-1>",    self._on_press)
        canvas.bind("<B1-Motion>",        self._on_drag)
        canvas.bind("<ButtonRelease-1>",  self._on_release)

    def activate(self):
        super().activate()
        self.controller.view.canvas.config(cursor="hand2")

    def deactivate(self):
        super().deactivate()
```

```
        canvas = self.controller.view.canvas
        canvas.config(cursor="")
        self._drag_origin = None
        self._mode = None
        self._vdata = None
    def _on_press(self, event):
        self._drag_origin = (event.x, event.y)
        canvas = self.controller.view.canvas
        if event.state & self.SHIFT_MASK:
            self._mode = "pan"
            return "break"
        items = canvas.find_overlapping(event.x, event.y, event.x, event.y)
        vertex_items = [it for it in items if 'vertex' in canvas.gettags(it)]
        if vertex_items:
            item_id = vertex_items[0]
            vid = self.controller.get_vertex_id(item_id)
            v = self.controller.model.vertices[vid]
            v['drag_origin'] = (event.x, event.y)
            self._vdata = v
            self._mode = "move"
        else:
            self._vdata = None
            self._mode = None
        return "break"
    def _on_drag(self, event):
        if not self._drag_origin:
            return "break"
        dx = event.x - self._drag_origin[0]
        dy = event.y - self._drag_origin[1]
        canvas = self.controller.view.canvas
        if self._mode == "pan":
            canvas.move("all", dx, dy)
            for vid, v in self.controller.model.vertices.items():
                v['x'] += dx
                v['y'] += dy
        elif self._mode == "move" and self._vdata:
            v = self._vdata
            canvas.move(v['circle_id'], dx, dy)
            canvas.move(v['text_id'],   dx, dy)
            if v.get('weight_text_id'):
                canvas.move(v['weight_text_id'], dx, dy)
            v['x'] += dx
            v['y'] += dy
            v['drag_origin'] = (event.x, event.y)
            if hasattr(self.controller, 'relayout_edges_touching'):
                self.controller.relayout_edges_touching(v['circle_id'])
        self._drag_origin = (event.x, event.y)
        return "break"
    def _on_release(self, event):
        if self._vdata:
            self._vdata['drag_origin'] = None
        self._drag_origin = None
        self._mode = None
        self._vdata = None
        return "break"
```

import_export_tool.py

```
from tool import Tool
import tkinter as tk
from tkinter import filedialog
import json
class ImportExportTool(Tool):
    def _bind_events(self):
        canvas = self.controller.view.canvas
        canvas.bind("<Button-1>", self._on_click)
    def _on_click(self, event):
        menu = tk.Menu(self.controller.view, tearoff=0)
        menu.add_command(label="Зберегти граф", command=self._export_graph)
        menu.add_command(label="Завантажити граф", command=self._import_graph)
        menu.tk_popup(event.x_root, event.y_root)
    def _export_graph(self):
        path = filedialog.asksaveasfilename(defaultextension=".json",
                                            filetypes=[("JSON files", "*.json")])
        if not path:
            return
        data = {
            "vertices": self.controller.model.vertices,
            "edges": [
                {k: e[k] for k in ('u', 'v', 'weight', 'direction') if k in e}
                for e in self.controller.model.edges
            ]
        }
        with open(path, "w", encoding="utf-8") as f:
            json.dump(data, f, indent=2)
        print("Граф збережено до:", path)
    def _import_graph(self):
        path = filedialog.askopenfilename(filetypes=[("JSON files", "*.json")])
        if not path:
            return
        with open(path, "r", encoding="utf-8") as f:
            data = json.load(f)
        self.controller.clear_all(redraw=False)
        for vid, v in data["vertices"].items():
            self.controller.model.vertices[int(vid)] = v
        for edge in data["edges"]:
            self.controller.model.edges.append(edge)
        self.controller.rebuild_view_from_model()
        print("Граф завантажено з:", path)
```

main.py

```
from Controller import Controller
def MVC_check(status):
    print(status)
if __name__ == "__main__":
    app = Controller(MVC_check=MVC_check)
    app.run()
```

ДОДАТОК Б Презентація

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Навчально-науковий інститут атомної та теплової енергетики
Кафедра інженерії програмного забезпечення в енергетиці

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ ДЛЯ ОБЧИСЛЕННЯ ПАРАМЕТРІВ ФУНКЦІОНАЛЬНОЇ СТІЙКОСТІ ІНТЕЛЕКТУАЛЬНОЇ СИСТЕМИ ПРИЙНЯТТЯ РІШЕНЬ В УМОВАХ ДЕСТРУКТИВНИХ ВПЛИВІВ В КРИТИЧНИХ ІНФРАСТРУКТУРАХ

Рябець Катерина Олександрівна, ТВ-13

доцент кафедри інженерії програмного забезпечення в енергетиці Шуклін Герман Вікторович,
кандидат технічних наук, доцент

2025

Актуальність теми

Інтелектуальні системи підтримки прийняття рішень (ІСППР) дедалі ширше використовуються в критично важливих сферах, зокрема в енергетиці, де вони забезпечують ефективне управління, прогнозування та оптимізацію. Однак зі зростанням їхньої складності зростає й уразливість до внутрішніх і зовнішніх деструктивних впливів, а відсутність точних інструментів для оцінки функціональної стійкості ускладнює виявлення слабких місць і підвищення надійності систем.

Розробка програмного засобу для оцінки функціональної стійкості ІСППР є важливою для створення надійних і стійких до збоїв систем, здатних працювати в умовах деструктивних впливів. Це особливо актуально для сфер, де порушення роботи можуть мати серйозні наслідки. Такий інструмент дає змогу ще на етапі розробки аналізувати архітектуру та визначати, чи здатна система зберігати працездатність при відмовах компонентів.

Мета: розробка програмного засобу для оцінки параметрів функціональної стійкості ІСППР в умовах деструктивних впливів у критичних інфраструктурах.

Об’єкт дослідження: процеси забезпечення функціональної стійкості ІСППР.

Предмет: інтелектуальні системи підтримки прийняття рішень.

Методи дослідження: теорія графів, теорія імовірності, методи математичного аналізу.

Постановка задачі

Для досягнення мети було виокремлено наступні задачі:

– проаналізувати поняття функціональної стійкості;

– визначити, за якими показниками ІСППР вважатиметься функціонально стійкою ;

– обрати математичну модель для представлення архітектури ІСППР;

– обрати методи оцінки імовірності зв'язності двох модулів системи;

– визначитися з архітектурою програмного застосунку ;

– спроектувати зручний графічний інтерфейс програмного застосунку;

– розробити програмний застосунок, який, використовуючи обрані методи оцінки параметрів функціональної стійкості, дає можливість зробити висновки щодо функціональної стійкості системи.

Проєктування системи

Математична модель ІСППР

Неорієнтований граф $G = \{D,Ф,P\}$ без петель та кратних ребер,

де $D = \{d_i\}$, $D = V \cup L$ – множина елементів графа,

де $V = \{v_i\}$ – множина вершин графа (модулів системи),

$L = \{l_{ij}\}$ – множина ребер (зв'язків між модулями).

$Ф(l_{ij}) = v_i \& v_j$ – відображення інцидентності і суміжності елементів,

$P = \{p_i\}$ – імовірності справного стану елементів.

Вершини – надійні, ребра – ненадійні, інакше:

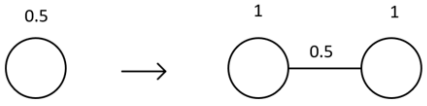

Рисунок 1. Графічне зображення надійності вершин і ненадійності ребер

Проєктування системи

Показники функціональної стійкості

ІСППР функціонально стійка, якщо існує така конфігурація її елементів і зав'язків між ними, за якої система здатна виконувати щонайменше критично важливі функції, навіть у разі відхилення від штатного режиму.

Основні показники:

- число реберної зв'язності λ (G) ;
- число вершинної зв'язності χ (G);
- імовірність переходу від i-ої вершини в j-ту $P_{ij}(t)$;
- P - матриця імовірностей переходу графа ІСППР.

$$\chi(G) \geq 2 \cup \lambda(G) \geq 2 \qquad (1)$$

$$P_{ij}(t) \geq P_{ij}^{зад}, \quad i \neq j, \quad i, j = 1, 2 \dots n, \qquad (2)$$

Критерій функціональної стійкості :

$$P = \begin{bmatrix} 0 & P_{12} & P_{13} & \dots & P_{1n} \\ P_{21} & 0 & P_{23} & \dots & P_{2n} \\ . & . & . & . & . \\ P_{n1} & P_{n2} & P_{n3} & \dots & 0 \end{bmatrix}$$

$$K = \sum_{\substack{i,j=1 \\ i \neq j}}^{n} N_{ij}^{min} \cdot P_{ij} \cdot (|i-j|+1)^{-2} \qquad (3)$$

де N_{ij}^{min} – мінімальна кількість ребер на найкоротшому шляху між вершинами i та j

Проєктування системи

Розрахунок імовірності зв'язності двох вершин на основі аналізу простих ланцюгів

Простий ланцюг– послідовність ребер та вершин графа між v_i та v_j без петель і паралелей.

$$P_{x,y} = \sum_{v=1}^{m_{x,y}} (-1)^{v+1} \sum_{z=1}^{C_{m_{x,y}}^{v}} p(\mu_{j_z}^{v}), \qquad (4)$$

де $m_{x,y}$ – загальна кількість простих ланцюгів між вершинами vx та vy;
$C_{m_{x,y}}^{v}$ – кількість комбінації по v ланцюгів ;
$p(\mu_{j_z}^{v})$ – імовірність існування СПЛ.

ПЕРЕВАГА:
Точний.

НЕДОЛІК:
Обчислювальна складність зростає експоненціально.

Рисунок 2. Алгоритм розрахунку за методом простих ланцюгів

Проєктування системи

Розрахунок імовірності зв’язності двох вершин Методом Монте-Карло

Метод Монте Карло – це метод розв’язання математичних та статистичних задач за допомогою симуляції.

$$P_{x,y} = \frac{S}{N}, \qquad (5)$$

де S– кількість успішних випадків;
N– загальна кількість випробувань;

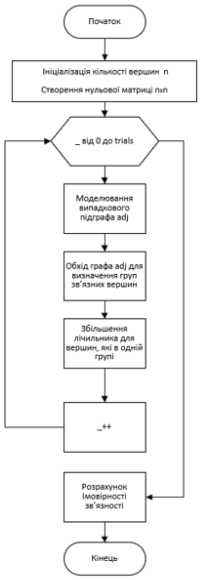

Рисунок 2. Алгоритм розрахунку за методом Монте-Карло

ПЕРЕВАГА:
Обчислювальна складність зростає лінійно.

НЕДОЛІК:
Велика похибка при обчисленні.

Проєктування системи

Рисунок 3. Діаграма варіантів використання

Проєктування системи

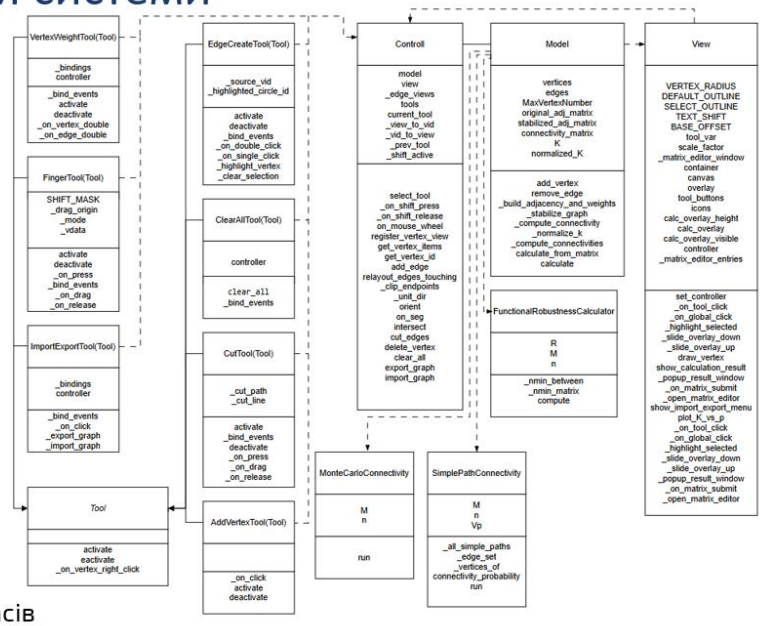

Рисунок 4. Діаграма класів

Реалізація системи

Python

PyCharm

Tkinter

matplotlib pandas NetworkX Network Analysis in Python pytest

Рисунок 5. Засоби розробки

Реалізація системи

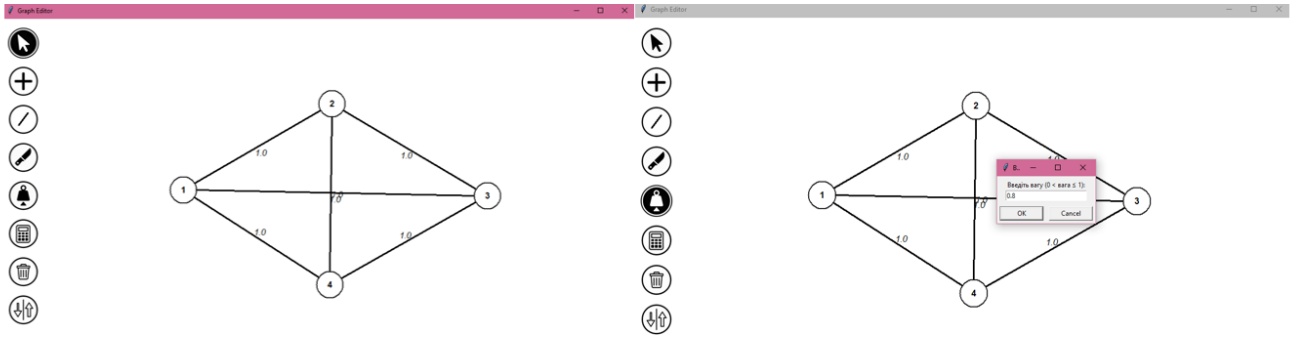

Рисунок 6. Робоча область інтерфейсу користувача

Реалізація системи

Рисунок 7. Реалізація розрахунків показників функціональної стійкості, представлене в GUI

Реалізація системи

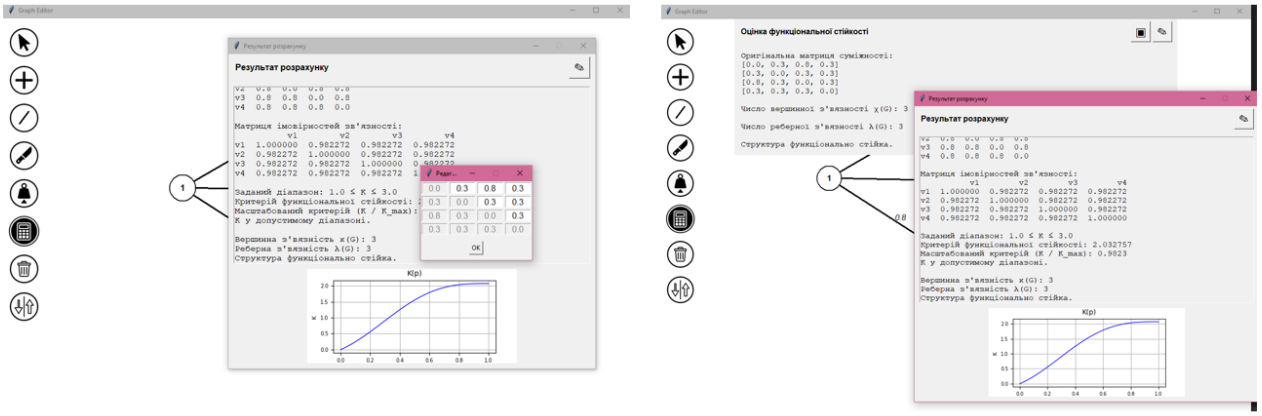

Рисунок 8. Реалізація повторних розрахунків показників функціональної стійкості, представлене в GUI

Тестування системи

Додавання вершин	Passed
Додавання ребер між вершинами	Passed
Зміна ваг ребер графу	Passed
Зміна ваг вершин графу	Passed
Автоматична нумерація вершин при створенні	Passed
Автоматична ренумерація вершин при видаленні	Passed
Підсвітка вибраного інструменту в GUI	Passed
Переміщення вершин по робочій зоні	Passed
Переміщення камери по робочій зоні	Passed
Автоматична зміна відображення ребер після руху	Passed
Візуальне приближення графу	Passed
Візуальне віддалення графу	Passed
Повна очистка робочої зони	Passed
Збереження графу як файл	Passed
Завантаження графу з файлу	Passed
Видалення зв'язків між вершинами	Passed
Візуальний ефект при видалення зв'язків	Passed
Автоматичний вибір методу розрахунку в залежності від кількості вершин	Passed
Підсвічування вершини при додаванні ребер	Passed
Відображення детальних розрахунків при натисканні кнопки	Passed
Відображення графіка	Passed
Відобрення матриці редагування ребер при натисканні кнопки	Passed
Розрахунок числа вершинної зв'язності	Passed
Розрахунок числа реберної зв'язності	Passed
Висновок щодо функціональної стійкості	Passed
Оброка випадку незв'язанної вершини	Passed
Оброка випадку пустої робочої зони	Passed
Оброка випадку введення 0 < вага ребра ≤ 1	Passed
Оброка випадку введення 0 < вага вершини ≤ 1	Passed

Рисунок 9. Чекліст, складений в ході тестування

Coverage for **Model.py**: 98%
162 statements | 159 run | 3 missing | 0 excluded

Coverage for **matrix_math.py**: 97%
119 statements | 115 run | 4 missing | 0 excluded

Рисунок 10. Покриття коду модульними тестами

Впровадження та супровід

«ЗАТВЕРДЖУЮ»
Директор товариства з обмеженою відповідальністю «ЛУЧ»
Євгеній ЧЕМЕС

АКТ

Комісія у складі:
голова: провідний інженер Лазебний Анатолій Андрійович;
членів комісії: керівник відділу інформаційно-технічного супроводу Лазебний Владислав Анатолійович;
Інженер з технічних систем, Тимо Володимир Євстратьєвич,

склала Акт, який підтверджує впровадження програмного забезпечення для обчислення параметрів функціональної стійкості інтелектуальної системи прийняття рішень в умовах деструктивних впливів в критичних інфраструктурах РЯБЕЦЬ Катерини, студентки групи ТВ-13, кафедри інженерії програмного забезпечення в енергетиці Навчально-наукового інституту атомної та теплової енергетики Національного технічного університету України «КПІ імені Ігоря Сікорського» в роботу ТОВ «ЛУЧ» (ЄДРПО 21595641).

Саме даний програмний продукт дозволяє визначити умови для зниження навантаження на мережеві карти серверу і уникнути переповнення буферної пам'яті чергами пакетів на поточну обробку, а також дає можливість не реагувати на хибні тривогу реакції інтелектуальної системи, яка здійснює моніторинг систем пожежної сигналізації.

ТОВ «ЛУЧ» прийняло до впровадження програмний продукт: **«Програмне забезпечення для обчислення параметрів функціональної стійкості інтелектуальної системи прийняття рішень в умовах деструктивних впливів в критичних інфраструктурах»** для визначення функціональної стійкості системи пожежної сигналізації критичних інфраструктур, які товариство з обмеженою відповідальністю «ЛУЧ» обслуговує.

Голова комісії _____ Анатолій ЛАЗЕБНИЙ

Члени комісії _____ Владислав ЛАЗЕБНИЙ

_____ Володимир ТИМО

Висновки

У результаті виконання дипломної роботи було виконано наступні задачі:

– проаналізовано поняття функціональної стійкості;

– визначено, за якими показниками ІСППР вважатиметься функціонально стійкою ;

– обрано математичну модель для представлення архітектури ІСППР;

– обрано методи оцінки імовірності зв’язності двох модулів системи;

– спроектовано архітектуру програмного застосунку ;

– спроектовано зручний графічний інтерфейс програмного застосунку;

– розроблено програмний застосунок, який, використовуючи обрані методи оцінки параметрів функціональної стійкості, дає можливість зробити висновки щодо функціональної стійкості системи.

ДЯКУЮ ЗА УВАГУ!