МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ДОНЕЦЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ВАСИЛЯ СТУСА

ЧЕРНОВ ЄГОР АНАТОЛІЙОВИЧ

Допускається до захисту:
в.о. завідувача кафедри
інформаційних технологій
канд. техн. наук, доцент
_____О. В. Зелінська
«____»_____20__р.

ДОДАТОК ДЛЯ ПСЕВДОВИПАДКОВОЇ ГЕНЕРАЦІЇ ТЕСТОВИХ АЛГОРИТМІВ РОБОТИ ЦИФРОВИХ АВТОМАТІВ

Спеціальність 122 Комп’ютерні науки

Кваліфікаційна (бакалаврська) робота

Керівник:
Р. М. Бабаков, професор
кафедри інформаційних
технологій, д. т. н., доцент
_____

Оцінка: _____/ _____/ _____
(бали/за шкалою ЄКТС/за національною шкалою)
Голова ЕК: _____

Вінниця - 2025

АНОТАЦІЯ

Чернов Є.А. Додаток для псевдовипадкової генерації тестових алгоритмів роботи цифрових автоматів. Спеціальність 122 «Комп'ютерні науки», освітня програма «Комп'ютерні науки». Донецький національний університет імені Василя Стуса, Вінниця, 2025.

У кваліфікаційній (бакалаврській) роботі розглянуто методи побудови граф-схем алгоритмів (ГСА) та підходи до їх автоматичної генерації. Запропоновано програмний додаток для створення тестових ГСА шляхом псевдовипадкової генерації з подальшим збереженням у форматі .kiss та візуалізацією у вигляді блок-схеми. Описано реалізацію алгоритму генерації переходів, побудову мікрооперацій та логічних умов, принципи коректного формування структури ГСА, а також особливості інтерфейсу користувача.

Ключові слова: граф-схема алгоритму, генерація, цифровий автомат, Python, Graphviz, .kiss.

58 с., 28 рис., 40 джерел.

ABSTRACT

Chernov Ye. Application for Pseudorandom Generation of Test Algorithms of Operation for Digital Automata. Specialty 122 «Computer Science», educational program «Computer Science». Vasyl Stus Donetsk National University, Vinnytsia, 2025.

This bachelor's thesis explores methods of constructing algorithmic flowcharts and automatic generation approaches. A software tool is developed for generating test flowcharts using pseudorandom algorithms, with output in .kiss format and graphical visualization. The implementation covers state transitions, logical conditions, microoperations, and the interface for user interaction.

Keywords: flowchart, generation, digital automaton, Python, Graphviz, .kiss.

58 pages, 28 figures, 40 references.

ЗМІСТ

ВСТУП

Цифрові автомати є фундаментальною частиною багатьох електронних пристроїв, які забезпечують логіку прийняття рішень, контроль послідовності дій та обробку сигналів. У повсякденному житті вони зустрічаються повсюдно — від найпростіших пристроїв, як-от побутові прилади, до складних високотехнологічних систем. Зокрема, цифрові автомати є основою керування у смартфонах, робототехнічних пристроях, GPS-навігаторах, побутовій техніці та системах автоматичного регулювання. Їх присутність у сучасних технологіях обумовлює потребу в ефективних методах побудови, аналізу та тестування алгоритмів, які вони реалізують.

У зв’язку зі стрімким розвитком інформаційних технологій, актуальним постає питання формального опису алгоритмів, які лягають в основу поведінки цифрових автоматів. Одним із найзручніших і водночас універсальних способів представлення таких алгоритмів є граф-схеми алгоритмів (ГСА). Вони дозволяють наочно моделювати логіку керування, умовні переходи та операційні дії у вигляді структурованих графів, що значно полегшує процес аналізу, налагодження та тестування програмних і апаратних рішень.

Зокрема, граф-схеми алгоритмів широко використовуються у проєктуванні цифрових систем, комп’ютерних програм, мікропроцесорних контролерів, вбудованих систем, а також у навчальних курсах з алгоритмізації та інженерії програмного забезпечення. Завдяки своїй простоті, наочності та формальній строгості, ГСА дозволяють забезпечити однозначне трактування логіки виконання алгоритму, що особливо важливо при автоматизованому синтезі та тестуванні.

У сучасній інженерній практиці все більшого значення набувають методи автоматизованої побудови тестових алгоритмічних структур. Це пов’язано із зростаючою складністю цифрових систем та необхідністю ретельного тестування

їх поведінки у різноманітних умовах. Псевдовипадкова генерація граф-схем алгоритмів дозволяє автоматично створювати набір структур, що задовольняють задані параметри складності, глибини логіки та кількості операцій. Такий підхід є доцільним як у дослідницьких цілях (наприклад, при верифікації логіки), так і в навчальному процесі, де важливо забезпечити різноманітність задач для засвоєння принципів побудови алгоритмів.

Тема цієї бакалаврської роботи — «Додаток для псевдовипадкової генерації тестових алгоритмів роботи цифрових автоматів» — є актуальною в контексті автоматизації створення та тестування алгоритмічних структур. Псевдовипадкова генерація дозволяє створювати випадкові, але структурно коректні ГСА з заданими параметрами, що має практичне значення для моделювання, тестування та навчання, зокрема у галузі програмування, кібернетики та штучного інтелекту.

Об’єктом дослідження процес генерації та візуалізації алгоритмів роботи цифрових автоматів. Предметом дослідження є методи псевдогенерації ГСА та засоби їх реалізації у вигляді KISS-файлів. Методологічною основою є теорія автоматів, теорія графів, а також принципи побудови мов опису алгоритмів.

Метою роботи є розробка програмного засобу для псевдогенерації граф-схем алгоритмів, що відповідають заданим параметрам та зберігаються у форматі KISS. Для досягнення цієї мети було поставлено такі завдання:

–дослідити теоретичну базу цифрових автоматів та методи їх тестування;
–реалізувати графічний інтерфейс для задання параметрів;
–створити алгоритм генерації ГСА з урахуванням обмежень;
–реалізувати візуалізацію алгоритму у вигляді блок-схеми;
–зберігати згенеровані ГСА у форматі .kiss для подальшого використання;

Запропонована розробка має практичну цінність для студентів технічних спеціальностей, викладачів, а також фахівців, що працюють у галузях цифрової логіки, верифікації систем, моделювання та автоматизації.

РОЗДІЛ 1
АНАЛІТИЧНИЙ ОГЛЯД СУЧАСНИХ ПІДХОДІВ ДО ФОРМАЛЬНОГО ПРЕДСТАВЛЕННЯ АЛГОРИТМІВ

Алгоритм — це впорядкована послідовність дій, спрямованих на досягнення певного результату. У комп'ютерних науках алгоритм часто описується як модель обчислювального процесу, яку можна реалізувати у вигляді програми. Існує багато способів представлення алгоритмів, зокрема псевдокод, блок-схеми, графи переходів, формальні граматики, логічні формули тощо.

Залежно від мети використання, той чи інший спосіб представлення може бути зручнішим. Наприклад, псевдокод є наближеним до мов програмування, що зручно для реалізації; блок-схеми — наочні для візуального сприйняття; формальні граматики — застосовуються для синтаксичного аналізу мов. Кожен з них має свої переваги та обмеження.

Блок-схеми — це графічний спосіб опису алгоритмів із використанням стандартизованих фігур (прямокутників, ромбів, овалів), який дозволяє візуалізувати логіку дій, умов і циклів. Вони є зручним інструментом для початкового етапу проєктування та навчання, але мають обмеження щодо формальної строгості.

Граф-схеми алгоритмів (ГСА) поєднують в собі переваги блок-схем і теоретичної строгості моделей автоматів. Вони дозволяють описати як структуру алгоритму, так і його поведінку в термінах станів, переходів, логічних умов і мікрооперацій. Особливо ефективно вони застосовуються у сфері цифрової схемотехніки, мов програмування, автоматного моделювання та верифікації логіки.

У рамках формального подання алгоритмів особливу увагу приділяють моделям скінченних автоматів — Мура та Мілі. В автоматі Мура вихідна інформація залежить лише від поточного стану, тоді як в автоматі Мілі — як від

стану, так і від вхідного сигналу. Ці моделі мають математичне підґрунтя, чітко визначену структуру та зручні для програмної реалізації.

Такі автомати широко використовуються в цифровій техніці для реалізації логіки пристроїв, що реагують на вхідні сигнали і переходять між визначеними станами. Саме скінченні автомати часто лежать в основі багатьох компіляторів, контролерів, цифрових схем та тестових генераторів.

Попри наявність різних інструментів для проєктування логіки, більшість з них не передбачають автоматичну генерацію тестових алгоритмів з урахуванням випадкових факторів, обмежень або специфічних параметрів. Існуючі системи або надто складні для початкового рівня, або не дозволяють налаштовувати параметри алгоритму на гнучкому рівні.

Тому актуальним є створення програмного засобу, який би дозволяв задавати параметри — кількість станів, переходів, логічних умов, мікрооперацій — та автоматично будувати граф-схему алгоритму із візуалізацією та експортом у формат, придатний для машинної обробки, наприклад .kiss.

На основі проведеного огляду встановлено, що граф-схеми алгоритмів, є ефективним способом подання логіки цифрових пристроїв. Вони забезпечують наочність, формальну строгість і можливість автоматизації. Сучасні засоби моделювання потребують розширення функціональності в бік автоматичної генерації тестових алгоритмів. У подальших розділах буде запропоновано підходи до реалізації такого генератора з можливістю збереження результатів у форматі .kiss та виведення у вигляді блок-схеми.

Важливим аспектом аналізу алгоритмів є їхня складність, як обчислювальна, так і структурна. Обчислювальна складність визначає, скільки операцій потрібно для виконання алгоритму при збільшенні обсягу вхідних даних, тоді як структурна — наскільки складною є логіка алгоритму в плані розгалужень, глибини вкладеності, умовних конструкцій тощо. У контексті автоматичної генерації

тестових алгоритмів особливо важливо контролювати структурну складність, щоб уникнути перевантаження системи зайвими переходами або недосяжними станами.

Історично граф-схеми алгоритмів виникли як спроба поєднати машинну строгость з візуальною інтуїтивністю. Вони мають корені в теорії автоматів та формальних мов, що дозволяє застосовувати до них математичний апарат. Наприклад, такі схеми можна аналізувати з використанням методів булевої алгебри, матриць суміжності або графів станів. Це відкриває можливість використання алгоритмів оптимізації, скорочення переходів, а також перевірки еквівалентності різних алгоритмічних реалізацій.

Ще один підхід до представлення алгоритмів — це використання UML-діаграм, зокрема діаграм діяльності (activity diagrams), які також дозволяють описувати логіку процесів. Однак вони не завжди адаптовані до автоматизованої генерації та вимагають складніших засобів обробки. Тому у випадках, коли важлива саме структура переходів, ГСА зберігають перевагу завдяки своїй формальній простоті.

У контексті створення програмного забезпечення генерації ГСА важливим є не лише правильний вибір формалізму, але й врахування потреб користувача. Залежно від того, хто буде користуватись системою — студент, викладач, інженер з тестування або розробник мікросхем — інтерфейс, рівень деталізації та формат виводу мають адаптуватись під задачі користувача.

З урахуванням цього, у наступному розділі буде представлено структуру програмного засобу, що реалізує автоматичну генерацію граф-схем алгоритмів відповідно до заданих параметрів, дозволяє контролювати логічну структуру, забезпечує збереження у форматі .KISS і візуалізацію у вигляді графа або блок-схеми.

РОЗДІЛ 2
МЕТОДИЧНІ ЗАСАДИ ПОБУДОВИ ТА ГЕНЕРАЦІЇ ГРАФ-СХЕМ АЛГОРИТМІВ

2.1 Основи побудови ГСА та їх структурні елементи

Граф-схема алгоритму (ГСА) є формальним графічним способом представлення логіки виконання алгоритмічних дій у вигляді орієнтованого графа. Такий підхід дозволяє описувати як прості послідовні процедури, так і складні розгалужені алгоритми, що реалізують керовану поведінку цифрових автоматів.

Типова ГСА складається з чотирьох основних типів вершин: початкової, кінцевої, операторної та умовної. Початкова вершина не має вхідних дуг і позначає точку початку виконання алгоритму. Кінцева вершина, відповідно, не має вихідних дуг і позначає завершення виконання. Операторна вершина містить набір мікрооперацій (МО), що активуються у даному стані. Умовна вершина відповідає за перевірку однієї логічної умови (ЛУ) і має два виходи: для значень «0» та «1» відповідної умови.

Кожна вершина в ГСА асоціюється зі станом, який може бути ідентифікований символьним ім’ям. Імена станів формуються за допомогою заданого префіксу та числового індексу, наприклад: a0, a1, a2, ..., aM−1. Таке іменування дозволяє легко програмно обробляти множину станів, уникаючи конфліктів і спрощуючи генерацію.

Орієнтований граф, що формує ГСА, повинен бути зв’язним. Це означає, що між будь-якою парою станів існує маршрут переходів. Додатково важливою є вимога повної досяжності кінцевого стану: кожен маршрут має мати можливість завершення, тобто досягнення стану завершення a0.

Зміст кожного переходу в ГСА задається четвіркою параметрів:

1. Шаблон логічних умов (записується як x1,x2,…,xL);
2. Початковий стан — поточне положення автомата;

3. Кінцевий стан — наступний стан після виконання переходу;
4. Вихідні дії — набір активованих мікрооперацій (записується як y1,y2,…,yN).

При генерації ГСА особливу увагу слід приділяти логічній послідовності переходів, унікальності імен станів, повноті розгалужень та коректності зв'язків між вершинами. Саме ці аспекти визначають функціональну правильність побудованого алгоритму.

2.2 Параметри генерації граф-схеми алгоритму

Генерація ГСА потребує завдання ряду параметрів, які визначають як структурну, так і функціональну складову майбутнього автомата. На прикладі графа G1 (рис.1) далі будуть описуватися основні базові параметри:

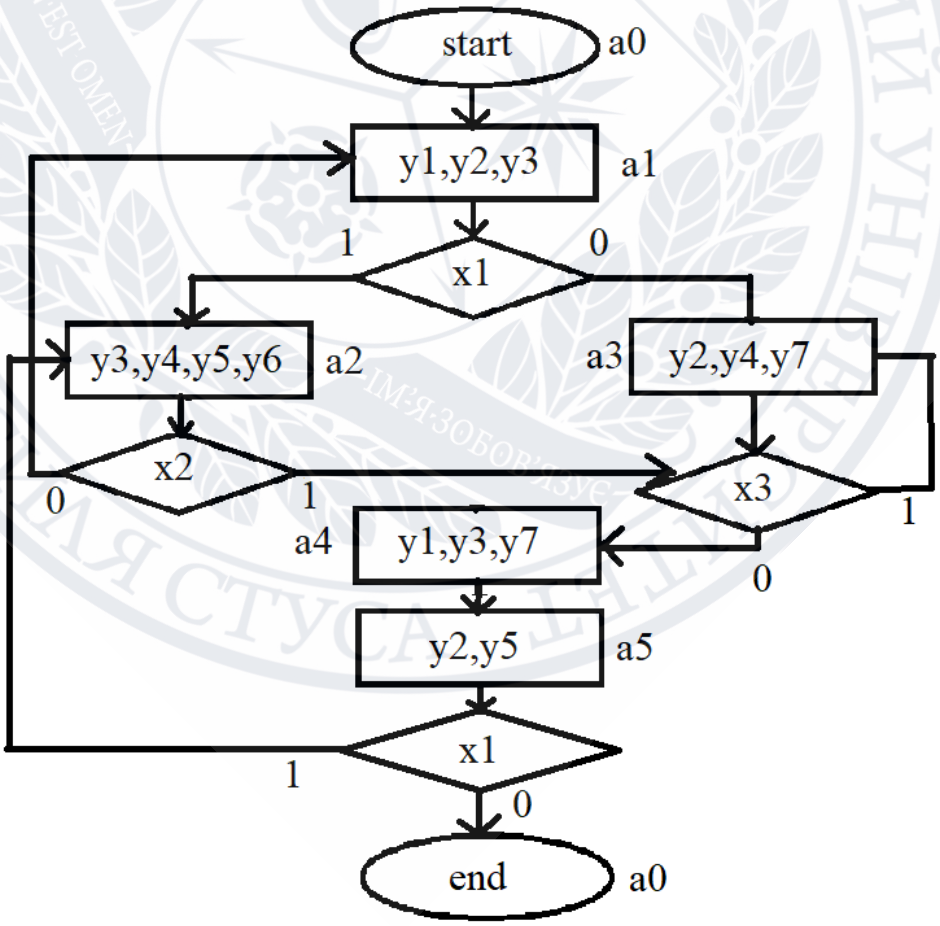

Рисунок 1 – Граф G1

Основні параметри ГСА:

1. К**ількість станів (M)** - визначає загальну кількість іменованих операторних або службових вершин у графі. В наведеному вище прикладі ГСА G1(рис.1) має M = 6 станів. Чим більше станів, тим складнішою буде ГСА. Якщо задати M = 10, то в ГСА будуть стани a0, a1, a2, ..., a9. Стану a10 не буде, оскільки індекси станів починаються з нуля, так як це зручно пов’язувати із індексами масивів. Якщо стан записаний в масиві з індексом 15, то назва стану буде дорівнювати «a» плюс число 15, перетворене на текстовий рядок.
2. **Кількість мікрооперацій (N)**. - визначає кількість входів, які перевіряються в умовних вершинах. Мікрооперації yn вважаються вихідними сигналами автомата і керують певним пристроєм або цифровою системою. З урахуванням того, що в цій роботі позначається загальна структура ГСА, поясненням значення цих сигналів можна знехтувати. Для генерації ГСА важлива лише кількість таких сигналів. Нехай цей параметр буде позначен символом N. У випадку ГСА G1 (рис.1) значення N = 7, оскільки останньою мікрооперацією є y7. На відміну від станів, мікрооперації прийнято нумерувати з одиниці, хоча це й не принципово. Мінімальна кількість МО дорівнює 1, хоча теоретично операторна вершина може взагалі бути порожньою.

   При записі до текстового файлу мікрооперації кодуються як бітовий вектор із символів «0» та «1», де одиниця означає активну операцію в даному стані.
3. **Кількість логічних умов (L)**. Логічні умови – це вхідні сигнали, які здатен проаналізувати автомат. Увесь аналіз полягає в тому, що в залежності від значення логічної умови автомат переходить в різні стани. А різні стани – це різні мікрооперації, тобто різні команди, які будуть поступати від нашого автомата в об’єкт керування.

4. **Мінімальна і максимальна кількість мікрооперацій в одній операторній вершині (N_{min} і N_{max})**. Ці параметри впливають на кількість одиниць в шаблоні кожного переходу. Наприклад, при N=10, N_min=2, N_max=5 можливі такі шаблони мікрооперацій:

   0110010001

   1100000000

   0000100011

   0111100010

   0101010101

5. **Мінімальна і максимальна кількість ЛУ, що перевіряються в одному переході (L_{min} і L_{max})** - ці параметри задають діапазон кількості логічних умов, які можуть бути одночасно перевірені при одному переході. Умови повинні задовольняти: $1 \leq Lmin \leq Lmax \leq L$.
6. **Кількість переходів (B)** — загальна кількість автоматних переходів у ГСА. Мінімальне значення дорівнює кількості станів ($B \geq M$), що відповідає повністю лінійній схемі. Максимальна кількість переходів визначається як:

$$B_{max} = M * 2^{L_{max}} \quad (2.1)$$

   де B_{max} – максимальна кількість переходів;

   M – кількість станів;

   L_{max} – максимально допустима кількість умов, що перевіряються у межах одного переходу.

7. **Ступінь лінійності ГСА (B1)** є допоміжною величиною, яка дозволяє регулювати співвідношення між умовними та безумовними переходами у згенерованій граф-схемі. Цей параметр приймає значення в діапазоні від 0

до 1 і фактично задає рівень "простоти" або "розгалуженості" побудованого алгоритму.

- Значення B1 = 1 відповідає повністю лінійній ГСА, де всі переходи є безумовними, тобто граф є простим ланцюгом без умовних розгалужень.
- Значення B1 = 0 означає повністю розгалужену схему, де всі переходи є умовними, тобто кожна операторна вершина завершується умовною перевіркою.

Зазвичай, в реальних алгоритмах існує певний баланс між умовними та безумовними переходами. Наприклад, при $B_1 = 0.5$ половина всіх переходів формуються безумовно, а інша половина — із перевіркою логічних умов. Це дозволяє керувати складністю згенерованої ГСА та її придатністю для подальшого тестування систем.

На етапі генерації переходів значення B1 використовується для обчислення кількості безумовних переходів (B1 × B) та умовних переходів ((1 – B1) × B), де B — загальна кількість переходів. Надалі ці значення використовуються для вибірки відповідного типу шаблонів при побудові кожного переходу.

2.3 Формальне представлення ГСА у текстовому форматі

Для подальшої автоматизованої обробки граф-схем алгоритмів (ГСА) необхідне чітко структуроване текстове подання переходів між станами. Формалізований опис дозволяє реалізувати читання, перевірку коректності та експорт до стандартних форматів збереження, зокрема KISS. У такому поданні кожен перехід описується впорядкованим набором параметрів, що включають:

- шаблоном логічних умов;
- початковим станом;
- кінцевим станом;

- шаблоном мікрооперацій.

Кожен з перелічених елементів є важливою складовою інформаційного опису функціонування переходу між вершинами ГСА.

Шаблон логічних умов (умова переходу) формується як послідовність символів довжиною L (де L — кількість логічних умов). Допустимими є символи «0», «1» і «–», які означають відповідно логічне значення «нуль», «одиниця» або "байдуже" (don't care). Наприклад, шаблон 1–1 вказує на те, що перехід активується за умови $x_1 = 1$, x_2 — не має значення, $x_3 = 1$.

Початковий та кінцевий стани позначаються символьними іменами, які складаються з буквено-цифрового префікса та індексу. Імена повинні бути унікальними в межах однієї ГСА. Наприклад: a0, a1, a2. Не допускається використання пробілів та спеціальних символів, що можуть порушити парсинг текстового файлу.

Маска мікрооперацій представляє собою двійковий вектор довжиною N, де кожна позиція відповідає одній мікрооперації y_i. Значення «1» у позиції вказує на активність відповідної мікрооперації в поточному стані, значення «0» — на її відсутність. Зміст маски є сталим для всіх переходів з одного стану, що узгоджується з моделлю автомата типу Мура.

В якості прикладу можна використати ГСА G_2 (рис. 2):

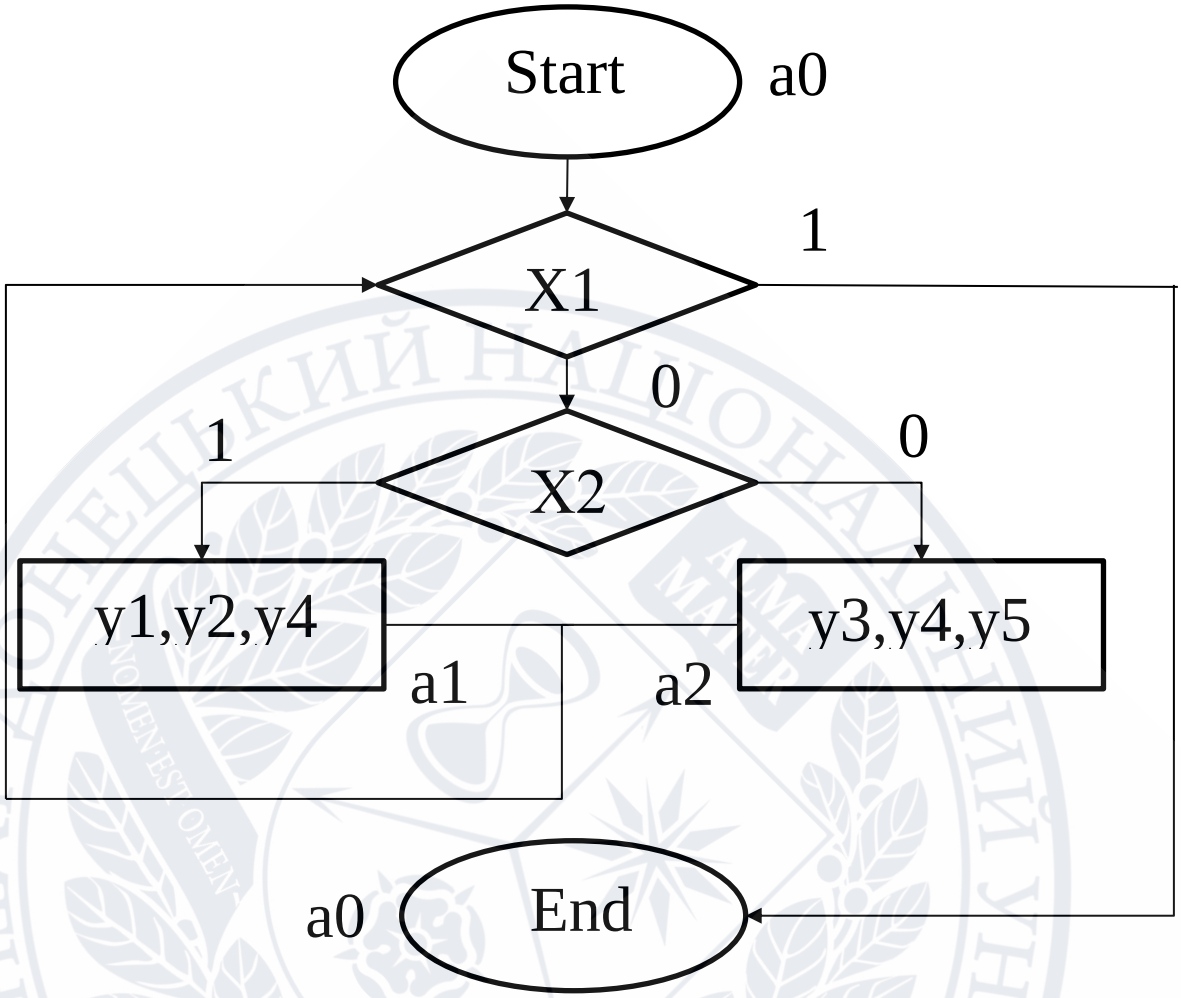

Рисунок 2 – ГСА G_2

Для цього графу опис усіх переходів буде виглядати таким чином:

1- a0 a0 00000

01 a0 a1 00000

00 a0 a2 00000

01 a1 a1 11010

00 a1 a2 11010

1- a1 a0 11010

01 a2 a1 00111

00 a2 a2 00111

1- a2 a0 00111

Такий формат опису дозволяє легко зберігати переходи у текстовому представленні, яке є проміжною формою між візуальною блок-схемою та файлом типу KISS. Більше того, за допомогою такого тексту можна виконувати автоматизовану верифікацію, синтаксичний аналіз і перетворення у внутрішні структури даних, що використовуються у середовищах моделювання.

У наступному підрозділі буде розглянуто принципи побудови коректної множини переходів, зокрема методи забезпечення досяжності всіх станів, уникнення зацикленості та формування повного набору логічних гілок у відповідності до обмежень параметрів L, N, M і B.

2.4 Побудова коректної множини переходів

Формування множини переходів є ключовим етапом побудови ГСА. Саме переходи визначають динаміку зміни станів, логіку виконання алгоритму, а також коректність функціонування моделі в цілому. Відповідно до заданих параметрів (M — кількість станів, L — кількість логічних умов, N — кількість мікрооперацій, B — загальна кількість переходів) необхідно згенерувати такий перелік переходів, який задовольняє умови досяжності, відсутності зациклення, повноти та структурної узгодженості.

1. Умови досяжності

Для кожного стану ГСА повинна існувати принаймні одна вхідна дуга, тобто має бути хоча б один перехід з іншого стану, що веде до нього. Це забезпечує досяжність стану, дозволяючи підсистемі переходити у будь-який стан із деякої початкової конфігурації.

Алгоритмічно це реалізується підрахунком кількості вхідних переходів до кожного стану. У разі, якщо вхідні переходи відсутні, такий стан вважається ізольованим та потребує додавання хоча б одного входу, що веде до нього з іншого досяжного стану. Наявність ізольованих станів порушує логічну замкненість ГСА та унеможливлює досягнення деяких частин алгоритму.

2. Виходи зі стану

Кожен стан повинен мати хоча б один вихідний перехід. В іншому разі, якщо алгоритм потрапляє в такий стан, подальше виконання припиняється. Як правило, при побудові ГСА передбачається наявність як мінімум одного безумовного переходу (в найпростішому випадку), або умовного — з однозначно визначеною логікою.

Для забезпечення гнучкості та варіативності структури, в більшості випадків кількість вихідних переходів зі стану задається випадково в межах дозволеного діапазону, з урахуванням параметрів L_min і L_max. Якщо необхідно реалізувати умовні переходи, використовується генерація унікальних шаблонів логічних умов, які не перетинаються.

3. Запобігання зацикленням

У деяких випадках можлива ситуація, коли всі переходи з певного стану ведуть або до нього самого, або до інших станів, з яких не існує шляху до кінцевої вершини (a0 або аналогічної). Такий фрагмент вважається замкненим циклом і може порушити правильне завершення алгоритму.

Щоб виключити появу таких циклів, застосовуються методи обходу графа (наприклад, пошук у глибину чи в ширину), які перевіряють, чи існує шлях від кожного стану до фінального. У випадку, якщо такий шлях відсутній, структура переходів коригується з додаванням додаткового переходу в інший, гарантовано досяжний стан.

4. Збалансований розподіл мікрооперацій

При генерації масок мікрооперацій необхідно забезпечити, щоб усі N мікрооперацій зустрічалися в межах принаймні одного стану. Недопустимо, щоб певна МО взагалі не з’являлась у жодному з операторних вузлів, оскільки це призводить до зниження покриття вхідних та вихідних умов тестового алгоритму.

Це досягається через відстеження використання кожної мікрооперації у процесі генерації, з подальшим примусовим включенням відсутніх МО у випадково обрані стани наприкінці генерації, якщо це не порушує обмеження N_{max}.

5. Приклад побудови

Розглядається ГСА з параметрами:

* M = 4 (стани: a0 — початковий, a1, a2, a3),

* L = 2 (логічні умови: x1, x2),

* N = 5 (мікрооперації: y1–y5),

* B = 7 (загальна кількість переходів).

Допустимо значення L_{min} = 1, L_{max} = 2. У процесі побудови випадковим чином вибираються шаблони умов, імена кінцевих станів, кількість переходів на стан, а також унікальні маски мікрооперацій.

Список переходів (умовно):

1- a0 a1 00000

0- a0 a2 00000

11 a1 a3 10101

10 a1 a2 10101

00 a2 a0 11000

01 a2 a1 11000

-- a3 a0 01111

Ця множина задовольняє всім переліченим критеріям:

* усі стани мають входи та виходи,

* усі МО використовуються,

* забезпечено повернення до a0.

Завдяки цьому побудована ГСА є функціонально замкненою та структурно коректною. Візуальне представлення ГСА також сприяє виявленню логічних

помилок і повторів. Однак це вже стосується етапу реалізації програмного засобу, що стане предметом розгляду у наступному розділі.

2.5 Висновки до розділу 2

У другому розділі було розглянуто основні теоретичні принципи побудови граф-схем алгоритмів (ГСА) для подання тестових алгоритмів роботи цифрових автоматів. Визначено ключові параметри, що впливають на структуру алгоритму: кількість станів, логічних умов, мікрооперацій, шаблони умов та структура переходів. Проаналізовано обмеження, вимоги до коректності, особливості побудови умовних і безумовних переходів, а також варіанти структурної складності з урахуванням конфігурацій ГСА.

Наголошено на важливості уникнення некоректних конструкцій — таких як зациклення, недосяжні або надлишкові стани, дублювання логічних умов і неініціалізовані мікрооперації. Надано приклади побудови переходів і розглянуто структуру текстового представлення ГСА у форматі, придатному для збереження та подальшого аналізу.

Накопичена теоретична база створює передумови для розробки програмного засобу, що дозволяє здійснювати автоматизовану генерацію ГСА за заданими параметрами, контролювати коректність побудови та забезпечувати збереження результатів у стандартизованому вигляді. Практичні аспекти розробки такого додатку, а також вибір інструментів реалізації та принципи візуалізації будуть розглянуті у наступному розділі.

РОЗДІЛ 3
РОЗРОБКА ПРОГРАМНОГО ЗАСОБУ ПСЕВДОГЕНЕРАЦІЇ ГСА

Після формалізації основних понять граф-схем алгоритмів та визначення параметрів, які впливають на їхню структуру та функціональність, постає завдання реалізації інструментального засобу, що дозволяє здійснювати автоматизовану генерацію таких схем. Метою цього розділу є опис вибору відповідних технологій, реалізації програмного додатку, що враховує задані параметри, а також візуалізації результатів у вигляді граф-схеми з подальшим збереженням у форматі .kiss.

У цьому розділі розглянуто ключові компоненти програмної реалізації: вибір мови та бібліотек, структура інтерфейсу користувача, модуль генерації, модуль перевірки коректності, принципи візуалізації, а також процес створення вихідного KISS-файлу.

3.1 Вибір інструментальних засобів реалізації

Для реалізації програмного додатку, що здійснює псевдовипадкову генерацію граф-схем алгоритмів (ГСА), було обрано мову програмування Python. Це високорівнева мова загального призначення, яка поєднує читабельність коду, широке поширення у середовищі освітніх та науково-дослідних проєктів, а також багату екосистему бібліотек для роботи з графами, інтерфейсами, візуалізацією та файлами.

Основними критеріями вибору Python є:

- відкритий вихідний код і кросплатформеність;
- наявність численних бібліотек, що охоплюють необхідні функціональні області;
- підтримка модульного програмування;
- активна спільнота та хороша документація;
- можливість швидкого прототипування.

Для побудови графічного інтерфейсу користувача (GUI) використано вбудовану бібліотеку **tkinter**, яка є частиною стандартної поставки Python. Вона забезпечує базові інструменти створення вікон, текстових полів, кнопок, міток, полів введення, випадаючих меню тощо. Tkinter дозволяє швидко формувати зручний інтерфейс для введення параметрів ГСА, запуску процесу генерації, перегляду логів.

Для генерації псевдовипадкових значень параметрів, вибору шаблонів, логічних умов, цільових станів використовується стандартний модуль **random**. Він забезпечує генерацію цілих чисел у заданих діапазонах, перемішування списків, випадковий вибір елементів, що необхідно при побудові умовних переходів.

Для візуального представлення графів у вигляді блок-схем використовуються бібліотека **graphviz**. Ця бібліотека використовується для побудови схем, де підтримується автоматичне розміщення вузлів, підписування вершин, орієнтація стрілок та експорт у графічні формати.

3.2 Принципи візуалізації граф-схем алгоритмів

Одним із ключових елементів програмного додатку є можливість візуального представлення згенерованої ГСА у вигляді зрозумілої блок-схеми. Візуалізація дозволяє користувачу оцінити структуру алгоритму, перевірити логіку переходів, наявність циклів, умовних розгалужень та кінцевих станів. Вона також сприяє виявленню помилок, таких як зациклення, відсутність переходів до певного стану, надмірне дублювання тощо.

Для візуалізації побудованих ГСА використовуються принципи блок-схем, орієнтованих зверху вниз. Таке розміщення дозволяє забезпечити природну інтерпретацію алгоритму: початок угорі, виконання зверху вниз, розгалуження вправо та вліво.

У програмі передбачено три типи вузлів:

- Вузли у вигляді кола — початковий та кінцевий стани;
- прямокутні — операторні вершини, що містять шаблони мікрооперацій;
- ромбовидні — умовні вершини, в яких перевіряються логічні умови.

Для кожного стану візуалізація передбачає підпис із його унікальним ім’ям. Для умовних переходів зазначаються маски логічних умов (наприклад, 1–0–), які визначають значення логічних змінних, при яких виконується перехід.

Розміщення вузлів реалізовано з урахуванням унікальності координат, щоб уникнути накладення блоків і забезпечити достатні відступи для читабельності. Структура побудови графа динамічно змінюється в залежності від кількості станів, рівня розгалуження, типу переходів.

З допомогою бібліотеки graphviz схема автоматично компілюється у формат PNG та зберігається у папці разом із відповідним .kiss-файлом.

Візуалізація ГСА є важливим етапом перевірки коректності побудови та корисним інструментом для подальшого аналізу тестових алгоритмів. Наявність інтегрованої графіки в програмі покращує зручність роботи з великою кількістю згенерованих алгоритмів.

3.3 Реалізація генерації та візуалізації граф-схем алгоритмів

У межах бакалаврської роботи було розроблено програмний засіб для генерації та візуалізації граф-схем алгоритмів (ГСА), реалізований мовою програмування Python із використанням графічної бібліотеки Tkinter, інструменту для побудови графів Graphviz та базових структур даних мови.

Структура програми побудована у вигляді двох основних класів: GSAGenerator – клас генерації ГСА, та GSAApp – клас, який реалізує графічний інтерфейс користувача.

Програмна реалізація генерації граф-схем алгоритмів (ГСА) здійснюється за допомогою спеціального класу GSAGenerator, який виконує ініціалізацію станів,

генерацію мікрооперацій та побудову переходів відповідно до заданих параметрів. Окремий графічний інтерфейс користувача забезпечує зручне введення параметрів генерації та відображення результатів у вигляді тексту та графа. (Рис. 3)

```
class GSAGenerator:
    def __init__(self, M, N, L, N_min, N_max, L_min, L_max, B=None, B1=None, prefix="a"):
        self.M = M
        self.N = N
        self.L = L
        self.N_min = N_min
        self.N_max = N_max
        self.L_min = L_min
        self.L_max = L_max
        self.B = B
        self.B1 = B1
        self.prefix = prefix

        self.states = [f"{prefix}{i}" for i in range(M + 1)]
        self.operations = {}
        self.transitions = []

        self.generate_operations()
        self.generate_structured_transitions()
```

Рисунок 3 – Клас FSMGenerator

Головним елементом для побудови моделі є метод generate_operations(), який відповідає за формування бітових рядків мікрооперацій для кожного стану. У початковому стані (a0) усі значення дорівнюють нулю, що символізує відсутність операцій. Для решти станів випадковим чином визначається набір операцій, активованих у відповідному стані. (Рис. 4)

```
def generate_operations(self):
    # Генерація випадкових мікрооперацій для кожного стану
    for state in self.states:
        if state == self.states[0]:
            self.operations[state] = "0" * self.N
        else:
            k = random.randint(self.N_min, min(self.N_max, self.N))
            indices = random.sample(range(self.N), k)
            bitstring = ''.join('1' if i in indices else '0' for i in range(self.N))
            self.operations[state] = bitstring
```

Рисунок 4 – Клас generate_operations

Наступним ключовим методом є generate_structured_transitions. У цьому методі реалізовано логіку створення автоматних переходів між станами. Початковий стан з'єднується з першим операторним станом безумовним переходом. Далі, в залежності від параметра B_1 або заданого B, автоматично обирається структура переходів – умовна або безумовна. Для умовних переходів генеруються додаткові вершини з відповідною умовою (типу cond_node), які зв'язуються з подальшими станами двома виходами – для значень 0 та 1 відповідної логічної умови (Рис.5).

```
if use_cond:
    cond_node = f"cond_{src}_{cond_counter}"
    cond_index = random.randint(0, self.L - 1)
    cond_bits = ['-' for _ in range(self.L)]
    cond_bits[cond_index] = str(random.randint(0, 1))
    self.transitions.append(("".join(cond_bits), src, cond_node, "", "cond_node"))
    outgoing[src] += 1
    incoming[cond_node] += 1
    cond_counter += 1
    count += 1

    for val in [0, 1]:
        if available:
            tgt = available.popleft()
        else:
            tgt = end
        cond = ['-' for _ in range(self.L)]
        cond[cond_index] = str(val)
        self.transitions.append(("".join(cond), cond_node, tgt, "", f"cond_edge_{val}"))
        outgoing[cond_node] += 1
        incoming[tgt] += 1
        next_nodes.append(tgt)
        count += 1
else:
```

Рисунок 5 – Фрагмент методу generate_structured_transitions

Для завершення побудови графа усі стані без вихідних переходів з'єднуються з кінцевим станом. Окрім цього, якщо жоден перехід не веде у кінцевий стан, виконується примусове створення такого зв'язку для запобігання утворенню неорієнтованих підграфів. (Рис. 6)

```
# Додаємо переходи до кінця для всіх вершин без виходів
for state in self.states:
    if outgoing[state] == 0 and state != end:
        self.transitions.append(("-" * self.L, state, end, self.operations.get(state, ""), "fix"))
        outgoing[state] += 1
        incoming[end] += 1
# Якщо до кінця немає жодного переходу — додаємо примусово
if incoming[end] == 0 and len(self.states) > 2:
    fallback = self.states[-2]
    self.transitions.append(("-" * self.L, fallback, end, self.operations.get(fallback, ""), "fix"))
```

Рисунок 6 – Фрагмент коду для додавання переходів в кінцевий вузел

Метод to_kiss реалізує перетворення згенерованої ГСА у текстове представлення формату KISS. Для цього здійснюється перенумерація станів відповідно до шаблону a0, a1, a2 тощо. Створюється службова частина файлу, яка включає заголовок з іменем графа, кількість входів, виходів, переходів, станів та ім’я початкового стану. Далі додаються рядки переходів, відповідно до структурованого масиву self.transitions, що був сформований у попередньому методі. (Рис. 7)

```
cond_links = {dst: (src, cond) for cond, src, dst, _, t in self.transitions if t == "cond_node"}

for cond, src, dst, ops, t_type in self.transitions:
    if t_type.startswith("cond_edge") and src in cond_links:
        orig_src, _ = cond_links[src]
        lines.append(f"{cond} {state_labels[orig_src]} {state_labels[dst]} {self.operations[orig_src]}")
    elif not src.startswith("cond_") and not dst.startswith("cond_"):
        if state_labels[src] != state_labels[dst]:
            lines.append(f"{cond} {state_labels[src]} {state_labels[dst]} {ops}")

return "\n".join(lines)
```

Рисунок 7 – Фрагмент класса to_kiss

На наступному малюнку (Рис.8) можна бачити як вигладає текстовий .kiss файл отриманий після псевдогенерації граф- схеми алгоритму GSA_1.

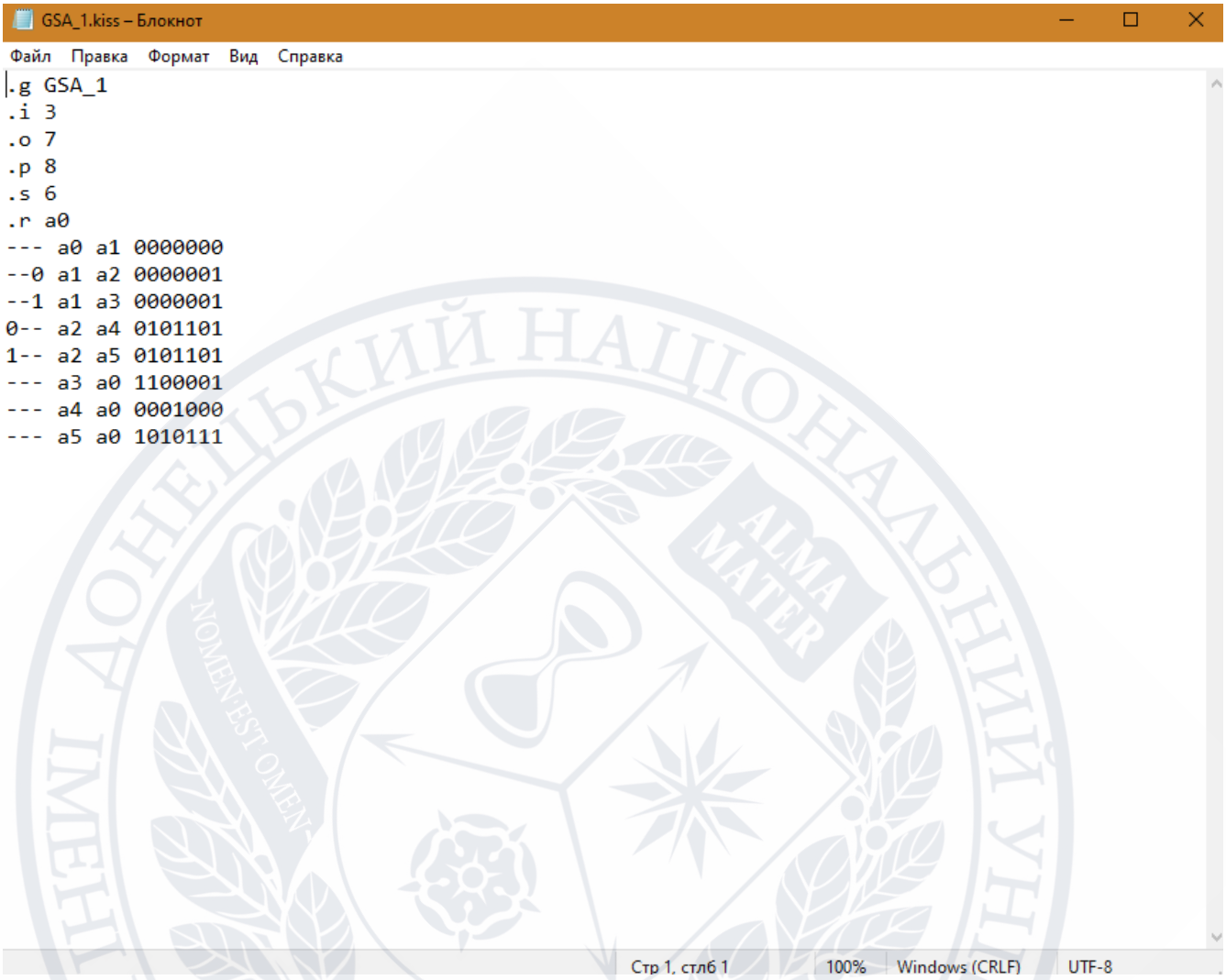

Рисунок 8 – Згенерований файл .kiss

Для візуалізації граф-схеми реалізовано метод render_graph, у якому використовується бібліотека Graphviz. Вказується орієнтація графа згори донизу (опція rankdir='TB'), а також налаштовуються відступи між вузлами та шарами. Стани автомата зображуються як прямокутники, умовні вершини – як ромби, початковий та кінцевий стани – овали зі спеціальним маркуванням. Всі вузли підписуються за допомогою параметра xlabel, який містить відповідну назву стану (наприклад, a2). Мікрооперації для кожного стану виводяться у вигляді багаторядкового підпису. (Рис. 9)

```
def render_graph(self, output_path="gsa", view=True):
    dot = Digraph(format='png')
    dot.attr(rankdir='TB', splines='ortho', nodesep="0.1", ranksep="0.3")
    start = self.states[0]
    end = self.states[-1]

    dot.attr('node', shape='ellipse', style='filled', fillcolor='lightgreen')
    dot.node(start, label="Початок", xlabel="a0", labelloc="c", width="1.2")

    dot.attr('node', shape='doublecircle', style='filled', fillcolor='lightblue')
    dot.node(end, label="Кінець", xlabel="a0", labelloc="c", width="1.2")
```

Рисунок 9 – Фрагмент класа render_graph

Зв’язки між окремими вузлами у граф-схемі алгоритму будуються відповідно до інформації, що міститься у структурованому масиві переходів. Кожен елемент цього масиву описує автоматний перехід від одного стану до іншого із зазначенням умов переходу та відповідних мікрооперацій. У разі, якщо перехід має умовний характер, тобто виконується за певного значення логічної умови (або групи умов), у візуальному представленні він реалізується як розгалуження з умовної вершини (що зображується у вигляді ромба). З такої вершини виходять дві дуги — одна з міткою “0”, інша з міткою “1”. Ці дуги вказують на різні напрями подальшого руху алгоритму залежно від результату перевірки відповідної логічної умови. Такий підхід дозволяє чітко та однозначно представити логіку прийняття рішень у рамках побудованої ГСА.

У випадку, коли перехід є безумовним, тобто не залежить від значень жодної з логічних умов, він зображується як пряма орієнтована стрілка, що з’єднує дві вершини без додаткових розгалужень. Це забезпечує зручність сприйняття структури алгоритму та дозволяє легко ідентифікувати фрагменти, які виконуються лінійно без логічних перевірок. Операторні вершини (стани) візуалізуються у вигляді прямокутників з відповідними підписами — або з набором активних мікрооперацій, які формуються у цьому стані, або з умовним позначенням «—», якщо мікрооперації відсутні. Початкова вершина графа (тобто

старт алгоритму) має спеціальне оформлення у вигляді еліпса світло-зеленого кольору з підписом «Початок», тоді як кінцева вершина має подвійну обвідку і виділяється світло-блакитним кольором, що візуально підкреслює завершення виконання алгоритму.

Кожен зв’язок у графі супроводжується стрілкою, яка чітко вказує напрям переходу від одного стану до іншого. Це дозволяє легко простежити логіку функціонування кожної окремої гілки алгоритму, а також оцінити загальну топологію побудованої граф-схеми. Залежно від типу переходу — умовного чи безумовного — стрілки можуть бути позначені відповідними логічними умовами або залишатись без підпису. Таке представлення дає змогу користувачеві не лише побачити загальний вигляд згенерованого алгоритму, а й зрозуміти, за яких умов система переходить від одного стану до іншого.

Крім того, розташування вузлів на площині виконується автоматично з урахуванням оптимального вертикального напрямку (зверху вниз), що робить схему читабельною навіть при великій кількості станів і переходів. Такий підхід особливо корисний при візуальній перевірці складних структур, оскільки дозволяє швидко ідентифікувати як лінійні фрагменти, так і циклічні або розгалужені ділянки алгоритму.

На наступному малюнку (Рис. 10) наведено приклад графічного зображення, сформованого у форматі PNG за результатами псевдовипадкової генерації граф-схеми алгоритму з параметрами, заданими користувачем. У цьому зображенні чітко простежується структура станів, логіка умовних переходів та послідовність виконання операцій, що підтверджує правильність і повноту реалізації генератора. Візуалізація такого типу суттєво полегшує як аналіз отриманих результатів, так і виявлення потенційних помилок або зайвих зв’язків, що особливо цінно в контексті тестування, налагодження або розширення функціональності згенерованих алгоритмів.

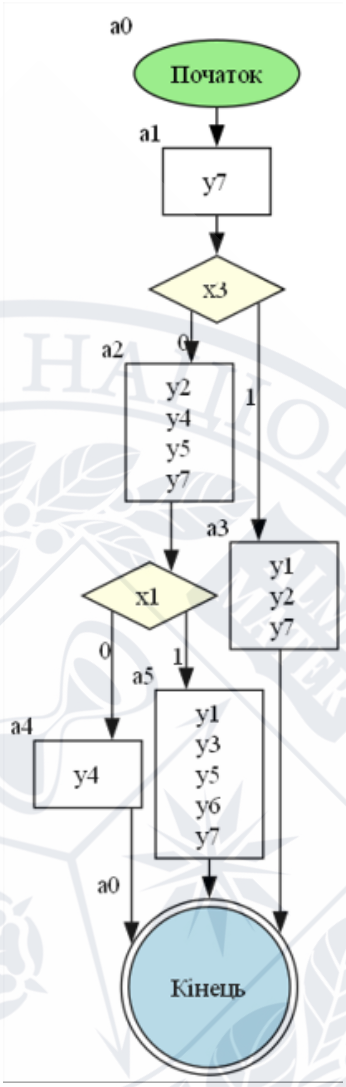

Рисунок 10 – Приклад побудованої ГСА, збереженої як зображення GSA_1.png

На основі реалізованої логіки формуються як .kiss-файли для кожного згенерованого ГСА, так і зображення у форматі PNG, які зберігаються на диску з іменами GSA_1.png, GSA_2.png тощо.

Клас GSAApp відповідає за графічний інтерфейс користувача. Він реалізує вікно з полями введення параметрів генерації та багаторядковим текстовим полем для відображення результату. Для кожного параметра створено віджет Entry, а всі введені значення зв’язуються з об’єктами типу StringVar. За замовчуванням цей клас має вже введені вхідні параметри. (Рис. 11)

```
class GSAApp(tk.Tk):
    def __init__(self):
        super().__init__()
        self.title("Генератор GSA")
        self.geometry("530x650")

        # Вхідні параметри
        self.params = {
        "M": tk.StringVar(value="6"),
        "N": tk.StringVar(value="7"),
        "L": tk.StringVar(value="3"),
        "N_min": tk.StringVar(value="1"),
        "N_max": tk.StringVar(value="5"),
        "L_min": tk.StringVar(value="1"),
        "L_max": tk.StringVar(value="2"),
        "B": tk.StringVar(value=""),
        "B1": tk.StringVar(value=""),
        "Count": tk.StringVar(value="1"),
    }

        self.create_widgets()
```

Рисунок 11 – Ініціалізація класу GSAApp

Метод create_widgets (Рис.12) відповідає за побудову графічного інтерфейсу користувача, реалізованого за допомогою бібліотеки tkinter. У цьому методі формується основна структура вікна додатку, яка включає в себе низку текстових міток (Label) та відповідних полів для введення параметрів генерації (Entry). Кожне поле відповідає одному з параметрів ГСА, таких як кількість станів (M), мікрооперацій (N), логічних умов (L), а також мінімальні та максимальні межі для відповідних підгруп параметрів. Таким чином, інтерфейс дозволяє користувачу зручно ввести початкові значення та гнучко налаштувати генератор відповідно до власних потреб.

```python
def create_widgets(self):
    frame = ttk.Frame(self, padding=15)
    frame.pack(fill=tk.BOTH, expand=True)

    # Поля введення
    labels = {
        "M": "Кількість станів (M):",
        "N": "Кількість мікроопераций (N):",
        "L": "Кількість логічних умов (L):",
        "N_min": "Мін. мікрооперацій (N_min):",
        "N_max": "Макс. мікрооперацій (N_max):",
        "L_min": "Мін. лог. умов (L_min):",
        "L_max": "Макс. лог. умов (L_max):",
        "B": "Кількість переходів (B) (опціонально):",
        "B1": "Ступінь лінійності B1 (0-1) (опціонально):",
        "Count": "Кількість ГСА:",
    }

    row = 0
    for key, text in labels.items():
        ttk.Label(frame, text=text).grid(row=row, column=0, sticky=tk.W, pady=3)
        ttk.Entry(frame, textvariable=self.params[key], width=25).grid(row=row, column=1, sticky=tk.W, pady=3)
        row += 1

    self.generate_btn = ttk.Button(frame, text="Згенерувати ГСА", command=self.generate_gsas)
    self.generate_btn.grid(row=row, column=0, columnspan=2, pady=15)

    self.output_text = tk.Text(frame, height=18, width=62)
    self.output_text.grid(row=row+1, column=0, columnspan=2, sticky=tk.W+tk.E)
```

Рисунок 12 – Клас create_widgets

Після натискання на кнопку «Згенерувати ГСА» в роботу вступає метод generate_gsas, який ініціює весь процес створення граф-схеми алгоритму. Першим кроком у цьому методі є виклик функції validate_params. Її основна мета — перевірити правильність введених користувачем даних. Зокрема, вона перевіряє, чи не виходять введені значення за допустимі межі: наприклад, кількість станів не повинна бути меншою за мінімальне значення, а ступінь лінійності B1 має знаходитись у межах від 0 до 1.

Якщо хоча б одне з введених значень не відповідає критеріям, користувачу миттєво виводиться повідомлення про помилку у вигляді вікна із поясненням, що саме введено некоректно. Це дозволяє уникнути некоректної генерації ГСА та зберегти стабільність роботи програми (Рис. 13).

```
    errors = []
    if not 2 < M < 1000:
        errors.append("M (Кількість станів) має бути не менше 2 та не більше 1000.")
    if N < 1:
        errors.append("N (Кількість мікрооперацій) має бути не менше 1.")
    if L < 1:
        errors.append("L (Кількість логічних умов) має бути не менше 1.")
    if not (0 <= N_min <= N_max <= N):
        errors.append("Параметри N_min і N_max повинні задовольняти: 0 ≤ N_min ≤ N_max ≤ N.")
    if not (0 <= L_min <= L_max <= L):
        errors.append("Параметри L_min і L_max повинні задовольняти: 0 ≤ L_min ≤ L_max ≤ L.")
    if B is not None and B < M:
        errors.append("B (Кількість переходів) має бути не менше кількості станів M.")
    if B1 is not None and not (0.0 <= B1 <= 1.0):
        errors.append("B1 має бути в діапазоні від 0.0 до 1.0.")
    if Count < 1:
        errors.append("Кількість ГСА має бути принаймні 1.")

    if errors:
        messagebox.showerror("Помилка введення", "\n".join(errors))
        return None
    return {
        "M": M, "N": N, "L": L,
        "N_min": N_min, "N_max": N_max,
        "L_min": L_min, "L_max": L_max,
        "B": B, "B1": B1, "Count": Count
    }
except ValueError:
    messagebox.showerror("Помилка", "Будь ласка, введіть коректні числові значення.")
    return None
```

Рисунок 13 – Фрагмент класу validate_params

Після успішного проходження валідації параметрів користувача розпочинається основний етап генерації. У межах циклу відбувається послідовна побудова заданої кількості граф-схем алгоритмів, кожна з яких створюється за унікальними параметрами, що було введено. Процес організовано таким чином, що для кожної ГСА формується повний список станів, переходів, умов та мікрооперацій, після чого ці дані автоматично конвертуються у формат KISS і записуються у відповідний текстовий файл.

Далі викликається метод generate_gsas, який відповідає за побудову графічного представлення відповідної ГСА. Результатом є PNG-файл, що візуалізує структуру алгоритму, включаючи всі переходи та типи вершин. (Рис. 14).

```python
def generate_gsas(self):
    params = self.validate_params()
    if params is None:
        return

    self.output_text.delete("1.0", tk.END)
    count = params.pop("Count")
    for i in range(count):
        gsa = GSAGenerator(**params, prefix=f"a")
        kiss_text = gsa.to_kiss(name=f"GSA_{i+1}")
        self.output_text.insert(tk.END, f"--- ГСА №{i+1} ---\n{kiss_text}\n\n")
        with open(f"GSA_{i+1}.kiss", "w") as f:
            f.write(kiss_text)

        # Візуалізація та зберігання
        try:
            gsa.render_graph(output_path=f"GSA_{i+1}", view=(i == 0))
        except Exception as e:
            messagebox.showwarning("Візуалізація не вдалася", f"Не вдалося візуалізувати ГСА №{i+1}: {e}")
```

Рисунок 2 – Клас generate_gsas

Таким чином, клас GSAApp забезпечує повноцінний інтерфейс користувача для управління процесом генерації ГСА. Він включає контроль введення параметрів, виклик логіки генерації, збереження результатів та зручне візуальне представлення як у текстовому, так і в графічному вигляді. Це дозволяє застосовувати розроблений додаток у навчальних, дослідницьких або тестувальних цілях без потреби в складних налаштуваннях або інтеграції з іншими інструментами.

3.4 Введення параметрів ГСА за допомогою графічного інтерфейсу користувача.

Сьогодні актуальним підходом до розробки подібних програм є кросплатформовість, тобто можливість запуску програми без перекомпіляції під різними операційними системами. З огляду на це в якості засобу розробки доречним буде використати мову програмування Python. До складу Python входить бібліотека tkinter, що дозволяє розробку графічних інтерфейсів користувача. Отже, проектування інтерфейсу програму домовимось робити за допомогою бібліотеки tkinter.

Графічний інтерфейс програми забезпечує роботу таких функцій:

– дозволяти задавати параметри ГСА у зручній для користувача формі;

– перевіряти введені дані на найбільш типові помилки (введення від’ємних значень або текстових даних замість чисел тощо);

– повідомляти користувача про невірно введені дані;

– запускати процес генерації ГСА шляхом передачі введених параметрів у відповідну функцію.

Нижче на рисунку наведено приклад графічного інтерфейсу розробленого додатку (Рис. 15).

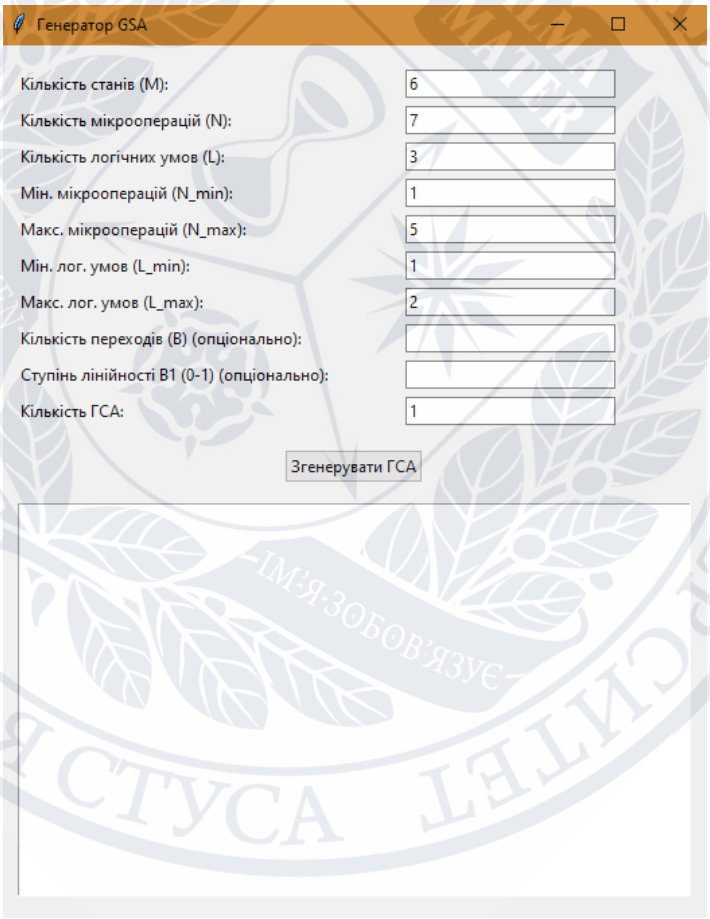

Рисунок 15 – Графічний інтерфейс програми

В наступному переліку зазначені параметри, завдяки яким програма може виконувати зазначені вище функції. Для цих параметрів був підібраний віджет

текстового поля Enter. Кожен віджет має поруч із собою підпис (віджет Label), який дозволяє зрозуміти призначення параметра.

Графічний інтерфейс розроблювальної програми забезпечує введення таких параметрів:

1) **Кількість станів M**. Генерована ГСА містить рівно *M* станів. Розроблене програмне забезпечення перевіряє, щоб кількість станів була більше нуля та не більше ніж 30 (для оптимальної читабельності).

2) **Кількість мікрооперацій N**. Це загальна кількість сигналів y_i, які можуть зустрічатись в операторних вершинах ГСА. Значення N визначає довжину текстового рядка, який відповідає шаблону операцій в кожному рядку файлу KISS. Приймають значення 0 або 1.

При введенні N має дотримуватись умова $N \geq 1$. Алгоритм генерації дозволяє використовувати будь-які значення N, більші за нуль.

3) **Кількість логічних умов L**. Кількість вхідних сигналів, що аналізуються в межах ГСА. Мінімальне значення – одиниця, максимальне – не обмежене. Значення *L* визначає кількість символів у шаблоні логічних умов в файлі KISS.

4) **Мінімальна і максимальна кількість мікрооперацій в одній операторній вершині N_min і N_max**. Ці параметри впливають на кількість одиниць в шаблоні кожного переходу. Наприклад, при N=10, N_min=2, N_max=5 можливі такі шаблони мікрооперацій:

```
0110010001
1100000000
0000100011
0111100010
0101010101
```

5) **Мінімальна і максимальна кількість ЛУ L_min і L_max**, що перевіряються в одному переході. Ці параметри стосуються тільки умовних переходів і визначають мінімально можливу і максимально можливу кількість ЛУ,

що перевіряються в одному переході (тобто мінімальну і максимальну кількість послідовно розташованих умовних вершин). Перевіряються виконання наступних умов: $1 \leq L\_min \leq L\_max \leq L$.

6) **Кількість переходів B**. Перехід – це можливість перейти з одного стану в інший. В межах ГСА перехід – це шлях, який з'єднує один стан з іншим. Кожен рядок у файлі KISS відповідає окремому переходу, а кількість рядків (за винятком кількох початкових інформаційних рядків) дорівнює числу B.

7) **Ступінь лінійності B1**. Цей параметр є кількісною характеристикою структури граф-схеми алгоритму (ГСА), яка відображає співвідношення операторних вершин до умовних. Інакше кажучи, ступінь лінійності вказує, наскільки граф наближений до прямолінійної (послідовної) моделі виконання, без складних логічних розгалужень або циклів.

8) **Кількість генерованих ГСА**. Завдяки цьому параметру користувач має змогу генерувати декілька різних ГСА, які мають однакові параметри. При введенні перевіряється, чи ввів користувач значення більше за 0. Також програма автоматично генерує різні назви для кожного файлу ГСА. Це відбувається таким чином:

GSA_{порядковий номер ГСА}.png

GSA_{порядковий номер ГСА}.kiss

Усі файли зі згенерованими ГСА створюються у тій самій папці, де знаходиться .py файл програми.

Також графічний інтерфейс має 2 таких додаткових елемента:

1. Для запуску процесу генерації використовується окрема кнопка "Згенерувати ГСА".

2. Під кнопкою "Згенерувати ГСА" знаходиться текстовий лог виконання, де користувач може бачити вміст згенерованих файлів kiss, не відкриваючи ці файли.

3.5 Формування та використання KISS-файлів

У процесі генерації граф-схем алгоритмів (ГСА) важливою задачею є збереження результатів у форматі, що є зручним для подальшого використання, інтерпретації та аналізу. Одним із найбільш поширених форматів представлення кінцевих автоматів є формат KISS (Keep It Simple, Stupid), що має широке застосування в галузі цифрової схемотехніки, автоматизації тестування та автоматного синтезу.

KISS-файл є текстовим документом, який містить структуровану інформацію про автомат, включаючи загальні параметри (кількість логічних умов, вихідних сигналів, станів), а також повний перелік переходів з відповідними умовами та мікрооперaціями. Такий формат є зрозумілим як для людини, так і для обчислювальних засобів, зокрема — для автоматизованих систем аналізу, генераторів тестів або інтерпретаторів.

Базова структура KISS-файлу включає:

- службові рядки, які описують загальні параметри автомата;
- основну частину — опис кожного переходу у вигляді чотирьох складових: вхідна умова, початковий стан, кінцевий стан, набір мікрооперацій.

Службові рядки завжди починаються з крапки та містять ключі .i, .o, .s, .p, .r. Вони мають наступне призначення:

- .i — кількість логічних умов (вхідних сигналів, L);
- .o — кількість мікрооперацій (вихідних сигналів, N);
- .p — кількість переходів (B);
- .s — кількість станів (M);
- .r — ім’я початкового стану (зазвичай це a0 або state0).

Після службових рядків слідує перелік переходів. Кожен перехід задається наступною формою:

[маска умов] [початковий стан] [кінцевий стан] [вектор мікрооперацій]

Маска умов — це послідовність символів "0", "1" або "-" (де "-" означає «будь-яке значення»), довжина якої дорівнює L. Така маска визначає, за яких умов виконується перехід з одного стану в інший. Наприклад, рядок:

01- a1 a3 1100101

означає: при x1 = 0, x2 = 1, x3 — будь-яке значення, з a1 відбувається перехід до a3, і при цьому формуються мікрооперації y1, y2, y6.

Імена станів у файлі можуть бути довільними, однак програмно зручно генерувати їх із певним префіксом (наприклад, "a") та індексом: a0, a1, ..., aM. Це дозволяє легко відстежувати та відтворювати структуру ГСА. У розробленому додатку передбачено автоматичну генерацію таких імен, а також забезпечення їх унікальності.

Цей блок формує службову частину KISS-файлу(Рис.16):

- .g — назва автомата;
- .i — кількість логічних умов (L);
- .o — кількість мікрооперацій (N);
- .p — кількість переходів (B);
- .s — кількість станів (M);
- .r — стартовий стан (a0).

```
lines += [
    f".g {name}",
    f".i {self.L}",
    f".o {self.N}",
    f".p {self.count_real_transitions()}",
    f".s {self.M}",
    f".r a0"
]
```

Рисунок 16 - Формування службової частини KISS-файлу

На наступному зображенні (Рис. 17) можна побачити приклад реального KISS-файлу, сформованого генератором у результаті виконання алгоритму псевдовипадкової побудови граф-схеми алгоритму. Цей файл містить усю необхідну службову інформацію, включно з кількістю вхідних сигналів (логічних умов), кількістю вихідних сигналів (мікрооперацій), кількістю станів, кількістю переходів, а також позначенням початкового стану. Дані у форматі KISS відображаються у вигляді простих текстових рядків, які описують автоматні переходи: кожен рядок містить маску умов переходу (наприклад, 0-1), початковий стан, кінцевий стан та бітовий вектор мікрооперацій.

```
.g GSA_1
.i 3
.o 7
.p 8
.s 6
.r a0
--- a0 a1 0000000
-0- a1 a2 0001011
-1- a1 a3 0001011
--0 a2 a4 0100000
--1 a2 a5 0100000
--- a3 a0 0010000
--- a4 a0 1100000
--- a5 a0 1000110
```

Рисунок 17 - Приклад згенерованого KISS-файлу для ГСА

Завдяки цьому опису можна не лише легко зберігати структуру графа у компактному вигляді, але й використовувати ці дані у сторонніх інструментах моделювання або аналізу.

Сформований KISS-файл зберігається автоматично після генерації кожної ГСА. У коді програми ця логіка реалізована в методі generate_gsas() класу

GSAApp. Назва файлу формується автоматично як GSA_1.kiss, GSA_2.kiss тощо, залежно від кількості згенерованих автоматів. (Рис. 18)

```
with open(f"GSA_{i+1}.kiss", "w") as f:
    f.write(kiss_text)
```

Рисунок 18 - Фрагмент коду методу generate_gsas(), де здійснюється збереження в .kiss

Таким чином, кожна згенерована граф-схема має унікальний імена, як у файлі, так і в структурі ГСА. Це дозволяє на практиці створювати колекції алгоритмів для подальшого тестування, порівняння або навчального використання.

Варто зазначити, що в рамках реалізованого програмного засобу передбачено також можливість миттєвого відображення вмісту згенерованих KISS-файлів у спеціальному текстовому полі графічного інтерфейсу користувача. Одразу після завершення процесу генерації, дані автоматично виводяться у вигляді структурованого тексту, що включає як службову інформацію (кількість станів, умов, мікрооперацій, кількість переходів, назву ГСА тощо), так і перелік усіх переходів з відповідними умовами та мікрооперераціями. Це забезпечує швидкий візуальний доступ до результатів генерації без необхідності відкривати окремий зовнішній файл (Рис. 19).

Користувач має змогу переглянути вміст кожного з переходів, оцінити правильність побудови автоматної моделі та, у разі потреби, змінити параметри генерації й повторити процес. Такий підхід є особливо зручним на етапі тестування та налагодження алгоритмів, коли потрібно оперативно перевіряти відповідність сформованої ГСА заданим параметрам. Крім того, інтеграція перегляду KISS-файлів у інтерфейс програми підвищує її зручність використання і робить весь процес побудови граф-схем алгоритмів більш наочним та доступним навіть для недосвідчених користувачів.

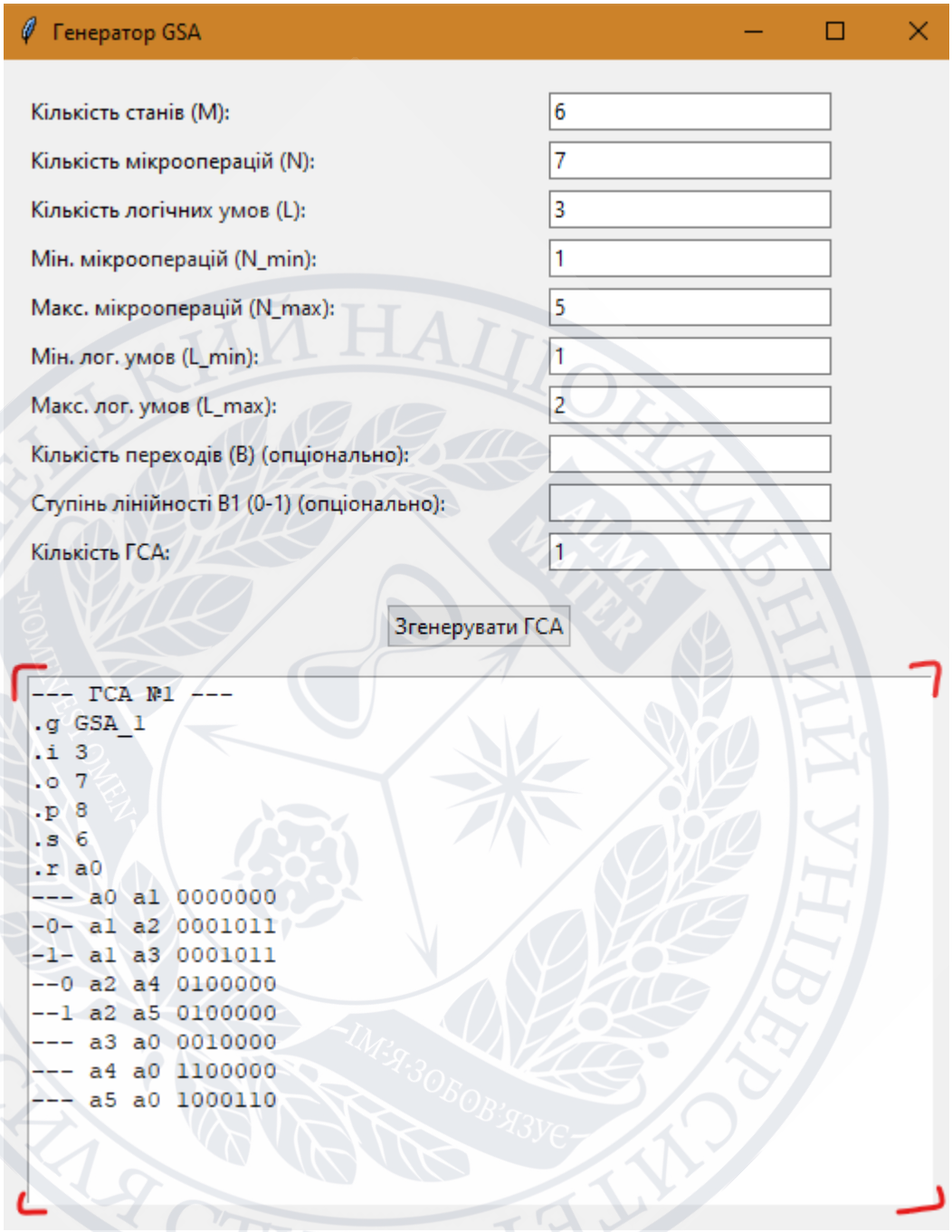

Рисунок 19 - Скриншот текстового поля у GUI після генерації

У підсумку, використання KISS-формату забезпечує:

- зручність інтеграції з іншими інструментами (як-от симулятори, транслятори);
- стандартизований формат представлення переходів;
- легкість аналізу структури алгоритмів та мікрооперацій;

- можливість повторного використання отриманих схем без потреби у візуальному інтерфейсі.

Збереження ГСА у форматі KISS є ключовим етапом у процесі автоматичної генерації. Це дозволяє не лише зробити структуру згенерованого алгоритму доступною для аналізу, але й полегшує обробку даних сторонніми інструментами. Реалізований механізм генерації та збереження KISS-файлів забезпечує як гнучкість використання, так і розширюваність під час подальших досліджень або впровадження додаткових функцій.

3.6 Верифікація та перевірка коректності згенерованої граф-схеми алгоритму

Після завершення побудови граф-схеми алгоритму важливо переконатись у її логічній завершеності, відсутності критичних помилок у структурі та можливості її подальшого використання як тестового шаблону. Коректність ГСА безпосередньо впливає на якість її подальшої обробки, зокрема при трансляції у формат KISS або при візуалізації, тому навіть у разі псевдовипадкової генерації необхідно передбачити принаймні базові механізми перевірки її структури.

У даній програмі реалізовано кілька важливих етапів перевірки, які гарантують досягнення ГСА необхідного рівня завершеності.

Основні критерії верифікації ГСА:

1. **Перевірка досяжності кінцевого стану**

ГСА повинна гарантувати, що з будь-якого проміжного стану можна дістатися до кінцевої вершини (a0). Інакше кажучи, не повинно існувати підграфів, які є «тупиковими» — тобто частинами схеми, з яких неможливо повернутись до завершення алгоритму. Це особливо важливо для забезпечення завершеності виконання тестових алгоритмів.

У коді реалізовано наступний механізм для цього (Рис.20):

```
# Якщо до кінця немає жодного переходу — додаємо примусово
if incoming[end] == 0 and len(self.states) > 2:
    fallback = self.states[-2]
    self.transitions.append(("-" * self.L, fallback, end, self.operations.get(fallback, ""), "fix"))
```

Рисунок 20 – Код для перевірки досяжності кінця

У цьому фрагменті перевіряється, чи має кінцева вершина хоча б одну вхідну дугу. Якщо вона ізольована — створюється новий безумовний перехід із передостаннього стану до кінця.

2. **Забезпечення виходу з усіх станів**

Ще однією базовою вимогою до графа є наявність принаймні одного виходу з кожної вершини, окрім кінцевої. Якщо певний стан не має жодного виходу, це створює ризик «мертвих» гілок — алгоритм, потрапивши в такий стан, зупиняється і не може продовжити виконання. Для запобігання цьому у програмі передбачено додавання переходу до кінцевої вершини для кожного стану без виходів.(Рис. 21)

```
# Додаємо переходи до кінця для всіх вершин без виходів
for state in self.states:
    if outgoing[state] == 0 and state != end:
        self.transitions.append(("-" * self.L, state, end, self.operations.get(state, ""), "fix"))
```

Рисунок 21 – Код для перевірки наявності виходу станів

Програма послідовно перевіряє всі вершини. Якщо стан не має жодної вихідної дуги і не є кінцевим — автоматично додається безумовний перехід до кінця.

Ці дві перевірки охоплюють базовий, але важливий рівень коректності побудованої ГСА. Завдяки їм гарантовано:

- що жодна вершина не буде ізольованою (без виходів);
- що обов’язково існує хоча б один шлях до фінального завершення алгоритму.

Також, хоча не реалізовано формальну перевірку досяжності кінцевого стану з будь-якої вершини (наприклад, через DFS або BFS), на практиці обидві згадані перевірки значною мірою знижують ризик генерації некоректного автомата. Враховуючи, що генерація має псевдовипадковий характер, повноцінна валідація може призводити до значних накладних витрат. Тому реалізовані рішення є раціональним компромісом між простотою та надійністю.

У поточній реалізації програмного засобу перевірка коректності згенерованої граф-схеми базується на забезпеченні мінімальних вимог до її структури. Основна мета таких перевірок — гарантувати життєздатність автомата, уникнути зависань у «глухих» станах та забезпечити завершення виконання. У подальшій розробці можливе розширення цього функціоналу за рахунок більш складних перевірок цілісності графа, відстеження циклів, виявлення надмірних переходів та дублювань.

3.7 Генерація у форматі KISS: алгоритм побудови файлу

Для забезпечення збереження згенерованої граф-схеми алгоритму (ГСА) у зручному для подальшого аналізу вигляді реалізовано функціональність експорту до формату KISS. Даний формат, як було розглянуто в розділі 2, є стандартним текстовим способом подання автомата з логічними умовами, мікрооперaціями та переходами між станами.

У програмному модулі класу GSAGenerator для побудови такого файлу використовується метод to_kiss(). Цей метод формує перелік текстових рядків, які згодом зберігаються у .kiss-файл за допомогою стандартного виклику open(). Структура KISS-файлу генерується на основі даних, що вже були сформовані при створенні переходів та мікрооперацій у попередніх методах.

Алгоритм побудови KISS-файлу включає кілька послідовних етапів:

1. **Ініціалізація службової частини файлу**. Створюється заголовок із загальною інформацією про ГСА: кількість умов, кількість

мікрооперацій, кількість станів, кількість переходів. Усі ці значення зберігаються в окремих параметрах об’єкта та викликаються напряму: (Рис. 22)

```
lines += [
    f".g {name}",
    f".i {self.L}",
    f".o {self.N}",
    f".p {self.count_real_transitions()}",
    f".s {self.M}",
    f".r a0"
]
```

Рисунок 22 - Формування заголовку KISS-файлу

2. Приведення станів до однорідного іменування. Усі стани позначаються у форматі a0, a1, a2 тощо. Для цього використовується словник state_labels, який відображає внутрішні імена на зручні мітки, що відповідають правилам формату KISS (Рис. 23):

```
for s in self.states:
    if s == self.states[0] or s == self.states[-1]:
        state_labels[s] = "a0"
    else:
        i = 1
        while f"a{i}" in used:
            i += 1
        label = f"a{i}"
        state_labels[s] = label
        used.add(label)
```

Рисунок 23 - Генерація унікальних міток станів

3. **Обробка переходів**. Генеруються рядки переходів між станами. Для умовних переходів (через вузли типу cond_node) спочатку здійснюється пошук відповідного проміжного вузла, а далі формується остаточний перехід з відповідною маскою умов та вектором мікрооперацій. Для уникнення дублювання умов і перевірок використовується додатковий словник cond_links, у якому тимчасово зберігаються дані про умовні вузли (Рис. 24):

```
cond_links = {dst: (src, cond) for cond, src, dst, _, t in self.transitions if t == "cond_node"}
```

Рисунок 24 - Підготовка умовних вузлів для переходів

Основна генерація переходів(Рис. 25):

```
for cond, src, dst, ops, t_type in self.transitions:
    if t_type.startswith("cond_edge") and src in cond_links:
        orig_src, _ = cond_links[src]
        lines.append(f"{cond} {state_labels[orig_src]} {state_labels[dst]} {self.operations[orig_src]}")
    elif not src.startswith("cond_") and not dst.startswith("cond_"):
        if state_labels[src] != state_labels[dst]:
            lines.append(f"{cond} {state_labels[src]} {state_labels[dst]} {ops}")
```

Рисунок 25 - Основна генерація переходів

4. **Повернення сформованого тексту**. Рядки переходів об’єднуються у єдиний текстовий блок (Рис. 26):

```
return "\n".join(lines)
```

Рисунок 26 - Об'єднання всіх рядків у текст KISS-файлу

Результатом є повноцінний текстовий опис ГСА, який зберігається у файл із розширенням .kiss. У програмі передбачено автоматичне збереження файлу під назвою GSA_1.kiss, GSA_2.kiss і т.д., залежно від кількості згенерованих алгоритмів (Count).

Приклад одного з таких файлів наведено в підрозділі 2.3 (Рис.), де продемонстровано його структуру та варіант виводу переходів для конкретних параметрів M, L та N.

Завдяки цій реалізації програма забезпечує гнучкий механізм експорту та сумісність із зовнішніми системами аналізу скінченних автоматів. Форматований вивід дозволяє також візуально перевірити правильність згенерованої ГСА без додаткового ручного редагування.

3.8 Збереження та іменування файлів результатів генерації

Після завершення генерації граф-схеми алгоритму (ГСА) та побудови відповідної візуалізації, виникає потреба у збереженні результатів у зручному форматі. У розробленому додатку реалізовані автоматичне створення текстового файлу з описом ГСА у форматі KISS, а також побудова графічного зображення графа у форматі PNG.

1. **Структура збереження результатів**

Для кожної згенерованої ГСА створюються два окремі файли:

- GSA_X.kiss – текстовий файл, який містить опис ГСА у форматі KISS;
- GSA_X.png – графічне зображення, що візуалізує структуру відповідної граф-схеми.

Літера X у назві файлу відповідає порядковому номеру згенерованого алгоритму, що дозволяє генерувати одразу декілька незалежних ГСА без перезапису попередніх (Рис. 27).

```
with open(f"GSA_{i+1}.kiss", "w") as f:
    f.write(kiss_text)
```

Рисунок 27 - Створення та збереження KISS-файлу

Цей код створює файл за допомогою конструкції with, яка автоматично закриває файл після запису. Файл отримує ім’я у форматі GSA_X.kiss.

2. **Автоматичне іменування з використанням шаблонів**

Іменування реалізується на основі шаблону GSA_{номер}, що дозволяє уникнути колізій у назвах при послідовному збереженні результатів. Цей підхід дозволяє однозначно ідентифікувати кожну ГСА серед інших результатів генерації.

Окрім цього, у функції to_kiss() також зберігається заголовок .g FSM_X, де X – номер поточної ГСА. Це дозволяє зберігати у KISS-файлі не лише технічні

3. **Збереження графічної візуалізації**

Графічна частина виводиться за допомогою бібліотеки Graphviz, яка формує PNG-файл. Для кожної ГСА зображення зберігається під іменем GSA_X.png у поточній директорії. Перший згенерований граф автоматично відкривається у системному переглядачі для наочного ознайомлення (Рис. 28).

```
gsa.render_graph(output_path=f"GSA_{i+1}", view=(i == 0))
```

Рисунок 28 - Побудова та збереження PNG-файлу

У цьому фрагменті функція render_graph формує граф та зберігає його на диск.

4. **Можливі розширення**

У подальших версіях додатку можуть бути реалізовані такі функції:

- збереження згенерованих файлів у вибрану користувачем директорію;
- автоматичне створення архівів для зручного експорту;
- імпорт попередніх генерацій для подальшого редагування;
- збереження параметрів генерації у конфігураційному файлі (JSON, YAML тощо).

Такі вдосконалення сприятимуть покращенню юзабіліті та зручності використання інструменту, особливо під час багаторазових генерацій з подібними параметрами.

У межах дослідження було реалізовано програмний засіб для псевдовипадкової генерації граф-схем алгоритмів (ГСА), який забезпечує як текстову, так і графічну репрезентацію результатів. Розроблений додаток дозволяє користувачеві гнучко задавати вхідні параметри генерації, зокрема:

- кількість станів, мікрооперацій, логічних умов;
- мінімальну та максимальну кількість мікрооперацій і логічних умов у стані;
- кількість переходів або ступінь лінійності;
- префікс імен станів.

Графічний інтерфейс, побудований з використанням бібліотеки Tkinter, забезпечує зручне введення параметрів та запуск генерації, а також виведення результатів у зручному вигляді. Генерація мікрооперацій та переходів реалізована у вигляді окремих класів і функцій, що підвищує модульність і зрозумілість коду.

Особливу увагу приділено:

- генерації переходів з урахуванням параметрів коректності та уникнення помилок у побудові ГСА;
- структурованому збереженню результатів у форматі KISS, який підтримується стандартними засобами обробки скінченних автоматів;
- автоматичному створенню графічного подання з використанням бібліотеки Graphviz, яке дозволяє швидко оцінити структуру згенерованої ГСА.

Окрім цього, передбачено можливість створення одразу кількох графів для серійного аналізу або тестування.

Варто зазначити, що створений програмний додаток не лише повністю задовольняє всі заявлені функціональні вимоги, а й володіє високим рівнем гнучкості та адаптивності. Його архітектура побудована таким чином, що легко піддається модифікації та доповненню. Зокрема, існує потенціал для подальшого розвитку функціональності: можна передбачити розширення списку параметрів генерації, реалізувати збереження налаштувань у конфігураційний файл або навпаки — завантаження параметрів генерації з файлу, що полегшить повторне відтворення попередніх сценаріїв.

Крім того, передбачена можливість збереження результатів генерації не лише у форматі KISS, а й у більш сучасних та універсальних форматах, таких як JSON, XML чи YAML. Це значно підвищить сумісність програми з іншими системами та інструментами обробки даних. Перспективним напрямом також є розробка модуля для оберненого перетворення KISS-файлу у граф-схему, що дозволить аналізувати та редагувати вже наявні структури.

Таким чином, реалізований програмний продукт не обмежується лише основним функціоналом, а створює передумови для подальшого розвитку. Він здатен задовольнити потреби як у навчальному процесі — для ілюстрації роботи скінченних автоматів та дослідження алгоритмів керування, так і в наукових експериментах — для генерації тестових даних, моделювання та верифікації алгоритмічних структур. Програма може виступати як самостійний засіб, так і як компонент складніших середовищ, що використовуються в галузі комп’ютерних наук, системного програмування або цифрової схемотехніки.

ВИСНОВКИ

У процесі виконання бакалаврської роботи було проведено комплексне дослідження, присвячене проблемі автоматизованої генерації тестових алгоритмів для цифрових автоматів з подальшим їх поданням у вигляді граф-схем алгоритмів (ГСА) та експортом у формат KISS. Ця тема є актуальною в контексті інтенсифікації процесів тестування цифрових пристроїв, автоматизації проєктування систем керування та реалізації моделей поведінки вбудованих систем.

На першому етапі дослідження було сформовано теоретичну основу, яка включає огляд існуючих підходів до подання алгоритмів, аналіз формальних моделей представлення логіки роботи систем, зокрема скінченних автоматів, а також детальний розгляд поняття граф-схем алгоритмів. У ході аналітичного огляду були висвітлені ключові характеристики таких моделей, їх переваги у порівнянні з традиційними методами подання алгоритмів, а також виявлені обмеження, що виникають у разі потреби автоматичної генерації структур такого типу.

У другому розділі було здійснено формалізацію задачі генерації ГСА та описано основні параметри, які визначають конфігурацію графа: кількість станів (M), кількість логічних умов (L), кількість мікрооперацій (N), мінімальна і максимальна кількість логічних умов та мікрооперацій на вершину, кількість автоматних переходів (B) або ступінь лінійності (B_1). Значну увагу було приділено побудові коректної множини переходів, враховуючи необхідність уникнення помилок, пов’язаних із відсутністю виходів, входів, нескінченними циклами або надлишковими переходами. Було запропоновано приклади типових ситуацій, що можуть призводити до порушення коректності ГСА, а також розглянуто способи їх виявлення та попередження.

Окрему увагу було приділено структурі файлу формату KISS — одного з найпоширеніших текстових форматів опису скінченних автоматів, який широко використовується в наукових дослідженнях, інженерних проєктах і системах автоматичного синтезу логіки. Було описано загальну структуру такого файлу, зокрема службові рядки, систему кодування умов переходів та формат представлення мікрооперацій.

У третьому розділі безпосередньо представлено реалізацію програмного засобу генерації тестових ГСА. Застосовано мову програмування Python, що завдяки своїй гнучкості та розвиненій екосистемі дозволила реалізувати як логіку генерації, так і візуалізацію ГСА. Використання бібліотеки Graphviz дало змогу побудувати блок-схеми алгоритмів із підтримкою умовних переходів, операторних вершин, початкових та кінцевих станів. Вихідні дані користувач вводить через зручний графічний інтерфейс, побудований на основі бібліотеки Tkinter. Інтерфейс дозволяє гнучко задавати параметри ГСА, зберігати результати у KISS-файл, автоматично генерувати декілька варіантів ГСА, а також переглядати візуалізовану структуру.

Код програми було структуровано з дотриманням принципів об’єктно-орієнтованого підходу. Основна логіка генерації зосереджена в класі FSMGenerator, тоді як графічний інтерфейс реалізовано окремо, що забезпечує зручність супроводу і можливість розширення функціональності. У межах реалізації також передбачено автоматичну генерацію операцій для кожного стану, створення умовних та безумовних переходів відповідно до заданих обмежень, а також обробку крайових випадків (відсутність переходів до кінцевої вершини, надмірна кількість логічних умов тощо).

У процесі реалізації було досягнуто таких ключових результатів:

- розроблено та протестовано повнофункціональний графічний додаток для генерації тестових алгоритмів у вигляді ГСА;

- реалізовано підтримку виводу у форматі KISS з урахуванням усіх вимог до структури даних;
- забезпечено автоматичну побудову візуальної схеми алгоритму з урахуванням умовних вузлів та операторних вершин;
- створено можливість генерації множинних варіантів ГСА для подальшого аналізу або використання у тестуванні;
- забезпечено відповідність програмної реалізації заявленим вимогам до надійності, гнучкості та зручності використання.

Таким чином, сформульована мета дослідження — створення додатку для псевдовипадкової генерації тестових алгоритмів роботи цифрових автоматів — була повністю досягнута. Отримані результати мають не лише теоретичну, а й прикладну цінність. Розроблений інструмент може бути використаний для навчальних, наукових та інженерних цілей у галузях цифрової схемотехніки, програмування, моделювання систем керування, тестування програмних продуктів, а також при вивченні теорії автоматів. Крім того, реалізований підхід може бути адаптований до інших форматів представлення алгоритмів, а програмне ядро — використане в якості модуля в більших програмних системах.

За підсумками виконаної роботи доцільним є подальше вдосконалення програмного засобу шляхом:

- додавання функціоналу зворотного перетворення KISS-файлів у блок-схеми;
- підтримки більш складних типів логіки переходів (наприклад, багаторівнева перевірка умов);
- розширення налаштувань генерації для глибшого контролю над структурою ГСА;
- впровадження механізмів перевірки коректності автоматів, побудованих вручну.

СПИСОК ВИКОРИСТАНИХ ПОСИЛАНЬ

КНИГИ

1. Аверкін А.О., Бондаренко А.В. Програмування мовою Python. — Харків: ХНУРЕ, 2018. — 210 с.
2. Бондарчук В.С. Основи цифрової логіки. — Київ: КПІ ім. Ігоря Сікорського, 2018. — 180 с.
3. Глушков В.М., Ященко І.О. Основи алгоритмізації. — К.: Наукова думка, 2015. — 212 с.
4. Геєць В.М. Методи візуального програмування. — Харків: ХНУ, 2021. — 240 с.
5. Григор’єв В.Ф., Теплов В.О. Автоматизація алгоритмів. — Дніпро: ДНУ, 2016. — 228 с.
6. Кнут Д. Мистецтво програмування. Том 1: Основні алгоритми. — К.: Вільямс, 2020. — 640 с.
7. Навроцький В.І. Структурне програмування. — Львів: ЛНУ, 2019. — 246 с.
8. Панкратова Н.Д. Математичні методи та моделі в інформаційних системах. — Київ: Видавництво КНЕУ, 2020. — 294 с.
9. Сєріков А.П. Теорія скінченних автоматів та її застосування. — Харків: НТУ "ХПІ", 2014. — 188 с.
10. Шаповал О.І., Корольков С.П. Основи алгоритмізації та програмування. — Київ: Каравела, 2012. — 288 с.
11. Шилов М.М. Теория алгоритмов: учебник. — М.: БИНОМ, 2017. — 352 с.
12. Хомяков І.М. Архітектура програмного забезпечення. — Київ: Академія, 2022. — 310 с.
13. Хопкрофт Д., Ульман Дж. Вступ до теорії автоматів, мов та обчислень. — К.: Наука, 2003. — 526 с.

14. Власенко В.М. Автоматизація побудови алгоритмів. — Одеса: ОНАХТ, 2017. — 190 с.
15. Львов О.Ю., Поліщук В.В. Автоматне моделювання цифрових систем. — Тернопіль: ТНТУ, 2019. — 144 с.

СТАНДАРТИ

16. IEEE Std 830-1998. IEEE Recommended Practice for Software Requirements Specifications.
17. ISO/IEC/IEEE 12207:2017. Systems and software engineering — Software life cycle processes.
18. ISO/IEC 25010:2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models.
19. ISO/IEC 19501:2005. Information technology — Open Distributed Processing — Unified Modeling Language (UML) version 1.4.2.
20. ISO/IEC TR 19759:2005. Software Engineering — Guide to the Software Engineering Body of Knowledge (SWEBOK).

ЕЛЕКТРОННІ РЕСУРСИ

21. Chen, L. (2019). Visual FSM Generation Tools. Web of Science. — Режим доступу: https://doi.org/10.1016/j.jss.2019.04.007
22. Garfield E. More on the ethics of scientific publication: attribution abuse and citation amnesia undermine the science reward system. — URL: http://www.garfield.library.upenn.edu/essays/v5p621y1981-82.pdf
23. IEEE Computer Society. Guide to the Software Engineering Body of Knowledge (SWEBOK), Version 3.0. — Режим доступу: https://www.computer.org/education/bodies-of-knowledge/software-engineering
24. KISS File Format Specification. — Режим доступу: http://www.pldworld.com/_hdl/1/RESOURCES/kiss2.pdf

25. Matplotlib: Visualization with Python. — Режим доступу: https://matplotlib.org/
26. NetworkX — Network Analysis in Python. — Режим доступу: https://networkx.org/
27. Python Software Foundation. Python 3.11 documentation. — Режим доступу: https://docs.python.org/3/
28. Smith, A. (2019). FSM Modeling for Scalable Control Systems. ResearchGate. — Режим доступу: https://www.researchgate.net/publication/336513250
29. Tang, Y., Li, H. (2020). Finite-State Machine-Based Model for Real-Time Embedded Systems. Journal of Software Engineering and Applications, 13(10), 435–448.
30. Tkinter — Python interface to Tcl/Tk. — Режим доступу: https://docs.python.org/3/library/tkinter.html
31. W3C. Scalable Vector Graphics (SVG) Specification. — Режим доступу: https://www.w3.org/TR/SVG11/
32. Zhang, H. et al. (2022). Automatic Generation of Finite-State Machines for Embedded Applications. IEEE Access, 10, 12215–12224.
33. Gupta, P. (2021). Formal Methods in FSM Verification. International Journal of Software Engineering, 17(2), 89–103.
34. Muller, F. (2020). Modeling digital circuits using FSM. Computer Science Review, Elsevier, 35, 100234.
35. Harris, J. (2023). Finite-State Design in Game Programming. Game Dev Journal, 11(3), 201–210.
36. Rahman, S. (2021). State Machine Generation from Domain-Specific Languages. Journal of Systems and Software, 179, 111065.
37. O'Reilly, D. (2023). Automata Generation Tools for Software Testing. ResearchGate. — Режим доступу: https://www.researchgate.net/publication/372811226

38. Harel, D. (1987). Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 8(3), 231–274.
39. Pressman R. Software Engineering: A Practitioner’s Approach. — McGraw-Hill, 2020. — 944 p.
40. Sommerville I. Software Engineering. — 10th ed. — Pearson, 2015. — 792 p.

ДЕКЛАРАЦІЯ

про дотримання академічної доброчесності

Я, Чернов Єгор Анатолійович, освітня програма «Комп'ютерні науки», що нижче підписався, розуміючи та підтримуючи загальновизнані засади справедливості, доброчесності та законності,

ЗОБОВ'ЯЗУЮСЬ:

дотримуватися принципів та правил академічної доброчесності, що визначені законодавством України, локальними нормативними актами Донецького національного університету імені Василя Стуса, положеннями, правилами, умовами, визначеними іншими суб'єктами, та не допускати їх порушення.

ПІДТВЕРДЖУЮ:

що мені відомі положення статті 42 Закону України «Про освіту»;

що у даній роботі не представляла/представляв чиїсь роботи повністю або частково як свої власні. Там, де я скористалася/скористався працею інших, я зробила/зробив відповідні посилання на джерела інформації;

що дана робота не передавалась іншим особам і подається вперше, не порушує авторських та суміжних прав закріплених статтями 21-25 Закону України «Про авторське право та суміжні права», а дані та інформація не отримувались в недозволений спосіб.

УСВІДОМЛЮЮ:

що ця робота може бути перевірена університетом на плагіат або інші порушення академічної доброчесності, в тому числі з використанням спеціалізованих сервісів;

що у разі порушення академічної доброчесності, до мене можуть бути застосовані процедури, передбачені законодавством України та Кодексом академічної доброчесності та корпоративної етики Донецького національного університету імені Василя Стуса, іншими локальними нормативними актами університету, та я можу бути притягнута/притягнутий до академічної відповідальності.

_____ _____

(дата) (підпис)