ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА ІНФОРМАЦІЙНИХ СИСТЕМ ТА ТЕХНОЛОГІЙ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Система підтримки прийняття рішень для вибору оптимальних маршрутів доставки в логістичних компаніях»

на здобуття освітнього ступеня магістра
зі спеціальності 124 Системний аналіз
освітньо-професійної програми «Інтелектуальні системи управління»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Артем КОНОПЛЬОВ
(підпис)

Виконав: здобувач вищої освіти групи САДМ-61

Артем КОНОПЛЬОВ

Керівник: Ігор ПАТРАКЄЄВ
к.т.н., доцент

Рецензент: _____
науковий ступінь, вчене звання *Ім'я, ПРІЗВИЩЕ*

Київ 2024

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра інформаційних систем та мереж

Ступінь вищої освіти магістр

Спеціальність 124 Системний аналіз

Освітньо-професійна програма Інтелектуальні системи управління

ЗАТВЕРДЖУЮ
Завідувач кафедрою ІСТ
_____Каміла СТОРЧАК
«_____» ___20__р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Конопльова Артема Андрійовича
(прізвище, ім'я, по батькові здобувача)

1. Тема кваліфікаційної роботи: «Система підтримки прийняття рішень для вибору оптимальних маршрутів доставки в логістичних компаніях»

керівник кваліфікаційної роботи Ігор ПАТРАКЄВ к.т.н., доцент,
(Ім'я, ПРІЗВИЩЕ, науковий ступінь, вчене звання)

затверджені наказом Державного університету інформаційно-комунікаційних технологій

від «____» _____20___р. № ____

2. Строк подання кваліфікаційної роботи «___» _____20___р.

3. Вихідні дані до кваліфікаційної роботи: Як результат моєї кваліфікаційної роботи була розроблена система підтримки прийняття рішень на C#.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
   1. Аналіз методів та підходів до побудови маршрутів у логістичних системах
   2. Визначення вимог до системи та варіанти використання
   3. Розробка системи прийняття рішень

5. Перелік ілюстративного матеріалу: *презентація*

6. Дата видачі завдання «_____» _____20____р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз методів та підходів до побудови маршрутів у логістичних системах		
2	Аналіз методів та алгоритмів для вибору оптимального маршруту		
3	Огляд існуючих програмних рішень		
4	Визначення вимог до системи та варіанти використання		
5	Моделювання основних процесів		
6	Проектування бази даних		
7	Розробка програмних компонентів		
8	Оформлення роботи до здачі		

Здобувач вищої освіти _____ Артем КОНОПЛЬОВ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Керівник
кваліфікаційної роботи _____ Ігор ПАТРАКЄЄВ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

РЕФЕРАТ

Текстова частина кваліфікаційної роботи на здобуття ступеня магістра: 103 стор., 40 рис., 15табл., 31 джерело.

Мета роботи полягає у розробці системи для моделювання та аналізу маршрутів у транспортних мережах, яка дозволяє автоматизувати процеси управління логістичними перевезеннями, підвищити ефективність маршрутної мережі та забезпечити її адаптивність до змін у транспортному середовищі.

Об'єктом дослідження є транспортні мережі в логістичних системах.

Предметом дослідження є програмна система для моделювання маршрутів у транспортних мережах, що використовується для планування та оптимізації перевезень.

У роботі проаналізовано методи побудови маршрутів у логістичних системах, зокрема алгоритми гілок і кордонів, Дейкстри та найближчого сусіда, а також оглянуто існуючі програмні рішення. Розроблено трьохрівневу архітектуру, UML-діаграми, спроєктовано базу даних і реалізовано програмні компоненти на C#.Система забезпечує додавання, оновлення та аналіз маршрутів. Функціональне і модульне тестування підтвердили її коректність, а сценарії використання показали адаптивність для задач різної складності. Наприклад, графи з 5 вершинами та 14 ребрами і 6 вершинами та 18 ребрами продемонстрували середню відстань 77,86 км і 91,67 км відповідно.

Розроблена система автоматизує планування маршрутів, підвищує ефективність логістичних перевезень та адаптується до змін транспортної інфраструктури. Подальше вдосконалення включає інтеграцію параметрів вартості, часу доставки та оптимізаційних алгоритмів.

Автоматизація перевезень, аналіз даних, логістика, маршрутизація, модульне тестування, оптимізація маршрутів, побудова графів, транспортні мережі, трьохрівнева архітектур

ABSTRACT

The text part of the qualification work for obtaining a masters's degree: 103 pages, 40 figures, 15 tables, 31 sources.

The purpose of the work is to develop a system for modeling and analyzing routes in transport networks, which allows you to automate the processes of logistics transportation management, increase the efficiency of the route network and ensure its adaptability to changes in the transport environment.

The object of research is transport networks in logistics systems.

The subject of the research is a software system for modeling routes in transport networks, which is used for planning and optimizing transportation.

The work analyzes the methods of building routes in logistics systems, in particular the algorithms of branches and borders, Dijkstra and the nearest neighbor, and also reviews the existing software solutions. A three-level architecture, UML diagrams were developed, a database was designed and software components were implemented in C#. The system provides adding, updating and analysis of routes. Functional and modular testing confirmed its correctness, and usage scenarios showed adaptability for tasks of varying complexity. For example, graphs with 5 vertices and 14 edges and 6 vertices and 18 edges showed an average distance of 77.86 km and 91.67 km, respectively.

The developed system automates route planning, increases the efficiency of logistics transportation and adapts to changes in the transport infrastructure. Further improvement includes the integration of cost parameters, delivery time and optimization algorithms.

Automation of transportation, data analysis, logistics, routing, unit testing, route optimization, graph construction, transport networks, three-level architecture

ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

ПОДАННЯ
ГОЛОВІ ЕКЗАМЕНАЦІЙНОЇ КОМІСІЇ
ЩОДО ЗАХИСТУ КВАЛІФІКАЦІЙНОЇ РОБОТИ
на здобуття освітнього ступеня бакалавра (магістра)

Направляється здобувач *Конопльов А.А* до захисту кваліфікаційної роботи
(прізвище та ініціали)
за спеціальністю *124, Системний Аналіз*
(код, найменування спеціальності)
освітньо-професійної програми Інтелектуальні системи управління
(назва)
на тему: «Система підтримки прийняття рішень для вибору оптимальних маршрутів доставки в логістичних компаніях».

Кваліфікаційна робота і рецензія додаються.

Директор ННІІТ _____ Катерина НЕСТЕРЕНКО
(підпис) *(Ім'я, ПРІЗВИЩЕ)*

Висновок керівника кваліфікаційної роботи

Здобувач: Артем КОНОПЛЬОВ успішно виконав магістерську роботу на тему "Система підтримки прийняття рішень для вибору оптимальних маршрутів доставки в логістичних компаніях". У роботі він продемонстрував високий рівень професійних знань розробивши функціональну та ефективну систему прийняття рішень платформу з використанням C#. Поставлені задачі було виконано на високому рівні. Робота має практичну цінність і може бути застосована у задачах які потребують оптимізації прокладання логістичних маршрутів

Все це дозволяє оцінити виконану кваліфікаційну роботу здобувача Конопльова А.А. на оцінку «_____» та присвоїти йому кваліфікацію магістр з системного аналізу.

Керівник кваліфікаційної роботи _____ Ігор ПАТРАКЄВ
(підпис) *(Ім'я, ПРІЗВИЩЕ)*
«_____» ___20_____року

Висновок кафедри про кваліфікаційну роботу

Кваліфікаційна робота розглянута. Здобувач *Конопльов А.А.* допускається до захисту даної роботи в Екзаменаційній комісії.

Завідувач кафедрою ІСТ _____ Каміла СТОРЧАК.
(назва) *(підпис)* *(Ім'я, ПРІЗВИЩЕ)*

ВІДГУК РЕЦЕНЗЕНТА
на кваліфікаційну роботу
на здобуття освітнього ступеня магістра

здобувача(ки) вищої освіти Конопльов Артем Андрійович
(прізвище, ім'я, по батькові)

на тему «Система підтримки прийняття рішень для вибору оптимальних маршрутів доставки в логістичних компаніях»

Актуальність.

Кваліфікаційна робота є актуальною та відповідає сучасним вимогам до розробки програмного забезпечення. Тема роботи обрана вдало, а запропоноване рішення має високу практичну цінність, що підтверджується можливістю його використання у корпоративних середовищах. Здобувач(ка) демонструє глибоке розуміння теми, належний рівень технічних знань та здатність до вирішення складних завдань. Робота заслуговує на позитивну оцінку.

Позитивні сторони.

1. Практична значущість: Розроблений чат-бот має високу практичну цінність.
2. Економічна ефективність: Рішення дозволяє знизити витрати.
3. Комплексний підхід до розробки: У роботі детально описані всі етапи — від аналізу вимог до впровадження у реальні умови використання системи, що демонструє системний підхід здобувача до вирішення проблем.

Недоліки.

1.Відсутне візуалізація маршруту на карті.

2.Не було впроваджено роботу з реальними картами для розрахунку маршрутів

Відзначені зауваження не впливають на загальну позитивну оцінку кваліфікаційної роботи магістерської.

Висновок: *кваліфікаційна робота на здобуття ступеня магістра заслуговує оцінку "_____", а здобувач Артем КОНОПЛЬОВ заслуговує присвоєння кваліфікації Магістр з системного аналізу*

Рецензент:
науковий ступінь, вчене звання _____ *підпис* _____ *Ім'я, ПРІЗВИЩЕ*

ЗМІСТ

ВСТУП

Актуальність теми. У сучасних умовах глобалізації та швидкого розвитку електронної комерції логістичні компанії стикаються зі зростаючими вимогами щодо оперативності, точності та економічності доставки товарів. Вибір оптимальних маршрутів доставки стає одним із ключових завдань, від якого залежить ефективність діяльності компаній, їх здатність залишатися конкурентоспроможними на ринку. Зокрема, для України, де логістична інфраструктура потребує модернізації, розробка систем підтримки прийняття рішень (СППР) для оптимізації маршрутів доставки має важливе значення. Це пов’язано з необхідністю підвищення продуктивності транспортних систем, зниження витрат на перевезення та зменшення екологічного впливу.

Критичний аналіз існуючих підходів до вирішення цієї проблеми свідчить, що більшість сучасних рішень фокусуються на загальних математичних моделях та алгоритмах оптимізації, проте часто ігнорують специфічні особливості локальних умов, таких як стан дорожньої інфраструктури, погодні умови, та динамічні зміни у транспортному трафіку. Крім того, в Україні актуальною є проблема адаптації таких систем до умов нестабільного економічного середовища, що ускладнює застосування стандартних рішень.

Проблематика оптимізації маршрутів доставки є предметом численних досліджень у галузях операційного аналізу, математичного моделювання та інформаційних технологій. Серед основних підходів слід відзначити використання алгоритмів комівояжера, евристичних методів (наприклад, генетичних алгоритмів, мурашиних колоній) та методів машинного навчання. Значний внесок у розвиток цих напрямків зробили такі науковці, як Дантціг Г., Беллман Р. та інші [1].

Проте у більшості робіт основна увага приділяється математичним моделям, тоді як інтеграція цих моделей у реальні інформаційні системи для підтримки прийняття рішень у логістичних компаніях часто залишається недостатньо дослідженою. В Україні ця проблема є ще менш розробленою, що створює

об’єктивну потребу в розробці ефективної СППР, яка враховує локальні реалії та специфіку логістичних операцій.

Вихідними даними для цієї роботи є сучасні теоретичні та практичні напрацювання в галузі оптимізації транспортних систем, а також потреби логістичних компаній у вдосконаленні процесів маршрутизації. Підґрунтям для дослідження є аналіз практичного досвіду застосування існуючих систем в умовах української логістики, зокрема, з урахуванням обмежень інфраструктури, доступності даних та вартості їх реалізації.

Обґрунтуванням розробки системи підтримки прийняття рішень для вибору оптимальних маршрутів доставки є необхідність автоматизації процесів формування та оптимізації маршрутів, що забезпечує ефективний розрахунок найкоротших шляхів між пунктами доставки в реальному часі. Використання програмного забезпечення, інтегрованого з базами даних і динамічним генератором маршрутів, дозволяє адаптувати систему до умов логістичних компаній і враховувати унікальні особливості інфраструктури України.

Мета роботи полягає у розробці системи для моделювання та аналізу маршрутів у транспортних мережах, яка дозволяє автоматизувати процеси управління логістичними перевезеннями, підвищити ефективність маршрутної мережі та забезпечити її адаптивність до змін у транспортному середовищі.

Для досягнення поставленої мети визначено наступні основні задачі:

– проаналізувати особливості транспортних перевезень, існуючі методи і алгоритми побудови маршрутів, а також програмні рішення для оптимізації логістичних процесів;

– визначити вимоги до системи, розробити архітектуру програмного продукту та побудувати модель основних процесів маршрутизації;

– реалізувати програмний продукт, включаючи компоненти для роботи з базою даних, генерації маршрутів та їхньої візуалізації;

– провести тестування системи, розробити сценарії використання та виконати аналіз отриманих результатів для підтвердження її ефективності.

Об’єктом дослідження є транспортні мережі в логістичних системах.

Предметом дослідження є програмна система для моделювання маршрутів у транспортних мережах, що використовується для планування та оптимізації перевезень.

Для досягнення мети дослідження системи підтримки прийняття рішень для вибору оптимальних маршрутів доставки використовуються наступні методи: аналіз наукової літератури та існуючих програмних рішень, математичне моделювання процесів маршрутизації, об’єктно-орієнтоване проєктування програмного забезпечення, а також тестування розробленої системи на основі симуляційних сценаріїв.

Практичне значення отриманих результатів полягає у створенні системи, яка дозволяє автоматизувати процес вибору маршрутів доставки, знижуючи витрати на транспортування, оптимізуючи час доставки та покращуючи використання логістичних ресурсів. Розроблена система може бути інтегрована у діяльність логістичних компаній для підвищення їхньої операційної ефективності та конкурентоспроможності в умовах динамічної транспортної інфраструктури.

РОЗДІЛ 1 АНАЛІЗ МЕТОДІВ ТА ПІДХОДІВ ДО ПОБУДОВИ МАРШРУТІВ У ЛОГІСТИЧНИХ СИСТЕМАХ

1.1 Особливості транспортних перевезень і побудови маршрутної мережі

Транспортні перевезення є ключовим компонентом логістики, що забезпечує фізичне переміщення товарів від місця їх виробництва або складування до кінцевого споживача. У структурі логістичних витрат транспорт займає значну частину, часто перевищуючи 50%. Це обумовлює важливість ефективного управління маршрутами доставки, які мають забезпечити оптимальне використання ресурсів, таких як паливо, час та технічні засоби. Відмінною рисою транспортних перевезень є необхідність врахування різних факторів: типів вантажів, специфіки транспорту, дорожньої інфраструктури, нормативних обмежень та вимог клієнтів щодо строків доставки.

Транспортні перевезення поділяються на внутрішні, міжміські та міжнародні. У кожному з цих випадків виникають специфічні задачі, наприклад, обмеження на в’їзд у певні зони міста, часові обмеження на завантаження/розвантаження, а також потреба дотримуватися міжнародних стандартів безпеки вантажів [2]. Для побудови ефективних маршрутів необхідно враховувати типи транспорту (вантажівки, фургони, мультимодальні перевезення) та особливості кожного виду, такі як вантажопідйомність, витрати пального і швидкість пересування.

Побудова маршрутної мережі спрямована на організацію ефективного транспортування вантажів між пунктами доставки. Основною метою є оптимізація витрат на перевезення, скорочення часу доставки та забезпечення максимального використання ресурсів транспортних засобів. У контексті логістичних компаній маршрутна мережа виступає основою для планування операцій, дозволяючи враховувати географічні особливості, обмеження інфраструктури та специфіку вантажів. Сучасні методи побудови маршрутної мережі забезпечують інтеграцію статичних (географія, дороги) і динамічних

(трафік, погодні умови) даних, створюючи основу для прийняття обґрунтованих рішень. У межах системи підтримки прийняття рішень побудова маршрутної мережі є важливим етапом, який включає збір, аналіз та обробку інформації для визначення оптимальних шляхів. Ефективно побудована мережа дозволяє зменшити вплив зовнішніх факторів і підвищити надійність логістичних процесів, забезпечуючи клієнтам високу якість послуг [3].

На рис. 1.1 зображено основні проблеми, які виникають у процесі побудови маршрутної мережі. Ці проблеми впливають на оперативність, точність і економічність доставки, створюючи перешкоди для реалізації логістичних процесів.

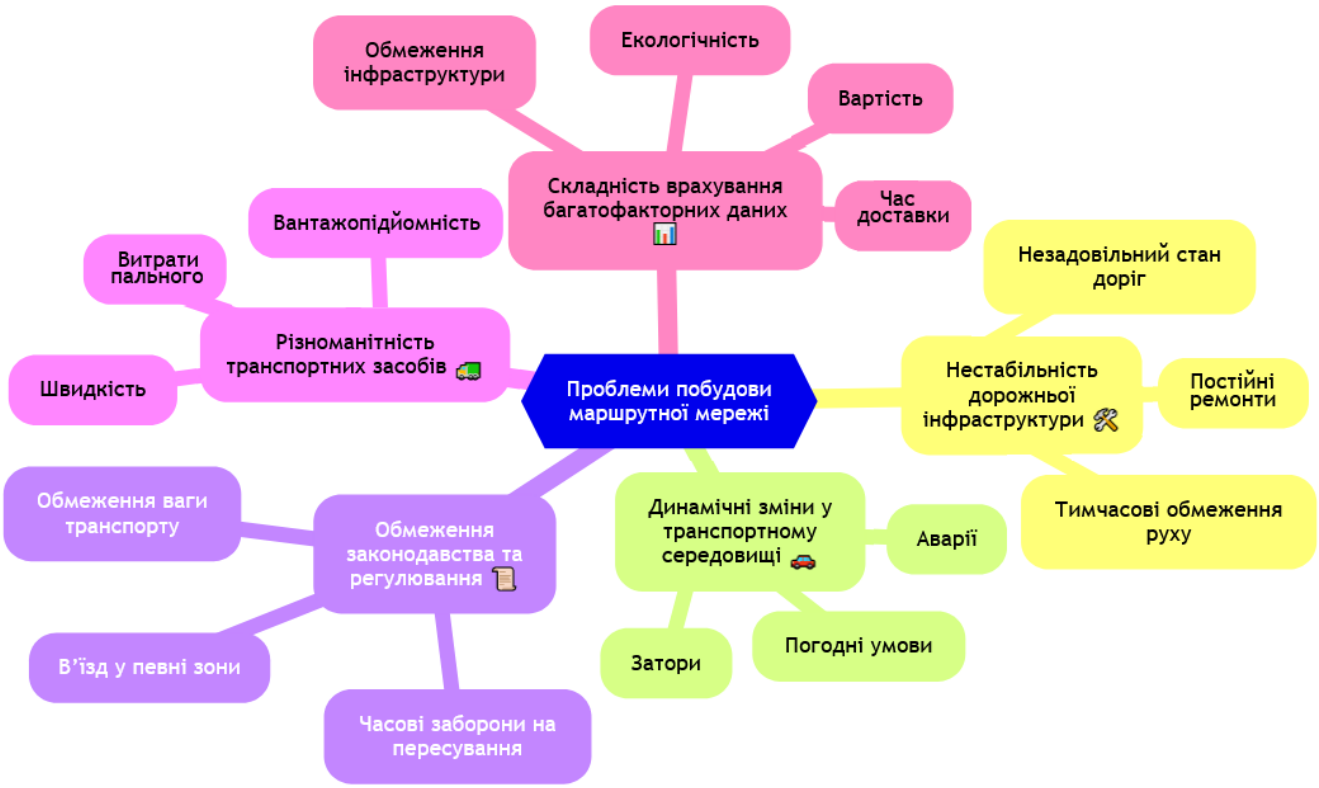


Рис. 1.1 Проблеми, які виникають у процесі побудови маршрутної мережі

До основних проблем побудови маршрутної мережі належать:

– нестабільність дорожньої інфраструктури. Наявність незадовільного стану доріг, постійні ремонти та тимчасові обмеження руху;

– динамічні зміни у транспортному середовищі. Затори, аварії та погодні умови, що змінюють доступність маршрутів у реальному часі;

– обмеження законодавства та регулювання. Обмеження ваги транспорту, часові заборони на пересування або в’їзд у певні зони;

– різноманітність транспортних засобів. Відмінності у вантажопідйомності, швидкості та витратах пального для різних типів транспорту;

– складність врахування багатофакторних даних. Необхідність одночасного врахування вартості, часу доставки, екологічності та обмежень інфраструктури.

Проблеми, що виникають при побудові маршрутної мережі, демонструють складність цього процесу та важливість використання сучасних технологій для їх вирішення. Логістичні компанії стикаються з викликами, які вимагають не лише обчислювальної ефективності, але й адаптивності до динамічних умов середовища. Розробка системи підтримки прийняття рішень, яка враховує ці проблеми, дозволяє підвищити конкурентоспроможність компаній та забезпечити високу якість логістичних послуг [4].

Для подолання зазначених проблем необхідно формувати маршрутну мережу, яка здатна враховувати складні багатофакторні умови і водночас забезпечувати оперативність та гнучкість у прийнятті рішень. Ефективно побудована мережа дозволяє інтегрувати сучасні технології та алгоритми, що сприяють оптимізації процесів транспортування та зниженню витрат. У цьому контексті вимоги до маршрутної мережі виступають ключовим елементом, який визначає її функціональні можливості та здатність адаптуватися до змінних умов логістичної інфраструктури.

Маршрутна мережа є основою для ефективного планування та виконання транспортних перевезень у логістичних системах. Її розробка передбачає створення структури, яка дозволяє визначати оптимальні шляхи доставки з урахуванням багатофакторних умов, таких як витрати, час перевезення, доступність інфраструктури та вимоги клієнтів. Вимоги до маршрутної мережі визначаються особливостями логістичних операцій, типом вантажу, географічним охопленням перевезень і технічними характеристиками транспортних засобів.

Успішне функціонування мережі залежить від здатності адаптуватися до змін у реальному часі, забезпечуючи гнучкість у виборі маршрутів і швидке реагування на непередбачувані обставини. Окрім того, сучасна маршрутна мережа повинна враховувати екологічні аспекти, сприяючи зменшенню викидів вуглекислого газу та оптимізації споживання пального [5].

На рис. 1.2 наведено ключові вимоги до побудови маршрутної мережі, які забезпечують її надійність, економічність та ефективність у динамічних умовах транспортної інфраструктури.

Рис. 1.2 Вимоги до побудови маршрутної мережі

Ефективна маршрутна мережа повинна відповідати таким основним вимогам:

– гнучкість. Можливість оперативного коригування маршрутів у разі виникнення змін у дорожній інфраструктурі, трафіку чи погодних умовах;

– економічність. Зниження загальних витрат на перевезення, включаючи паливо, обслуговування транспорту, оплату праці та логістичні витрати;

– надійність. Забезпечення стабільної роботи мережі навіть за наявності форс-мажорних обставин, таких як аварії чи технічні несправності;

– масштабованість. Підтримка роботи з великою кількістю вузлів і транспортних засобів без втрати продуктивності системи;

– екологічність. Мінімізація негативного впливу на навколишнє середовище через зниження витрат пального, вибір оптимальних маршрутів і скорочення заторів;

– доступність. Забезпечення можливості використання мережі у віддалених або географічно складних регіонах, де інфраструктура може бути обмеженою.

Вимоги до маршрутної мережі визначають її здатність адаптуватися до умов сучасної логістики, які стають дедалі складнішими через динамічний розвиток транспортної інфраструктури та підвищення стандартів обслуговування [6]. Виконання цих вимог є необхідною умовою для створення системи підтримки прийняття рішень, яка забезпечує високу ефективність логістичних операцій та дозволяє компаніям знижувати витрати і підвищувати рівень клієнтського сервісу.

В умовах України побудова маршрутної мережі ускладнюється нерівномірним розвитком інфраструктури, сезонними змінами погодних умов та обмеженою доступністю даних. Крім того, специфіка законодавства щодо вантажоперевезень та митних процедур для міжнародних маршрутів також впливає на ефективність мережі. Для адаптації до цих умов потрібні системи, здатні інтегрувати різнорідні дані та забезпечувати ефективну маршрутизацію навіть за нестабільних обставин.

1.2 Аналіз методів та алгоритмів для вибору оптимального маршруту

У логістичних системах вибір оптимального маршруту є однією з найважливіших задач, оскільки саме від неї залежить ефективність транспортування вантажів, зниження витрат і забезпечення клієнтського сервісу на високому рівні. Сучасні методи маршрутизації базуються на різних математичних і обчислювальних підходах, що дозволяють вирішувати задачі оптимізації навіть за складних умов. Серед них особливе місце займають

алгоритми, які враховують комбінаторну природу задачі, наприклад, задачу комівояжера або задачі транспортного типу. Одним із таких підходів є метод гілок і кордонів (Branch and Bound), що знаходить застосування у розв'язанні задач, які потребують знаходження глобального оптимального рішення з множини можливих варіантів [7].

Метод гілок і кордонів є ітеративним алгоритмом, що використовується для вирішення комбінаторних задач оптимізації, зокрема для вибору оптимальних маршрутів у транспортних системах. Він базується на систематичному розбитті простору допустимих рішень на підмножини (гілки) з подальшим відсіюванням підмножин, які не можуть містити оптимального розв'язку (кордони). Цей підхід забезпечує ефективний пошук оптимального маршруту, мінімізуючи необхідні обчислення шляхом виключення нерелевантних варіантів.

Нехай є задача мінімізації функції $f(x)$, де x належить до множини допустимих рішень S. Основна мета методу полягає у знаходженні:

$$x^* = \arg\min_{x \in S} f(x) \tag{1.1}$$

де x^*– оптимальне рішення, а $f(x)$ – цільова функція, яка відображає вартість маршруту, час перевезення або інші логістичні параметри.

Метод передбачає такі етапи:

1. Гілкування. Розбиття множини S на підмножини $S_1, S_2, \ldots, S_k$, де кожна з них відповідає окремому варіанту маршруту.

2. Оцінка кордонів. Для кожної підмножини S_i обчислюється нижня межа функції $f(x)$, позначена як $L(S_i)$, та верхня межа $U(S_i)$. Нижня межа визначається як найкраща можлива оцінка для елементів підмножини, а верхня – як оцінка реального маршруту.

3. Відсічення. Якщо для певної підмножини виконується умова:

$$L(S_i) \geq U^* \tag{1.2}$$

де U^* – поточне найкраще знайдене значення цільової функції, ця підмножина відсікається, оскільки не може містити оптимального рішення.

Ключовим моментом є обчислення меж. Нижня межа часто базується на евристичній оцінці, наприклад, мінімізації ваг ребер у графі, а верхня – на поточному маршруті, отриманому жадібним або іншим швидким методом.

Ітеративно алгоритм продовжує гілкування та відсічення, поки не залишиться одна підмножина або поки всі невідсічені підмножини не будуть оброблені. Таким чином, метод поступово звужує простір пошуку, зосереджуючись лише на перспективних варіантах [8].

Метод гілок і кордонів є потужним інструментом для розв’язання задач вибору оптимальних маршрутів, оскільки дозволяє систематично досліджувати простір можливих рішень і виключати нерелевантні варіанти. Його застосування забезпечує високу точність і знижує обчислювальні витрати, що робить цей метод особливо ефективним у задачах з великою кількістю можливих комбінацій.

Метод Дейкстри є одним із найпоширеніших алгоритмів для знаходження найкоротшого шляху у зважених графах [9]. Його ключовою перевагою є здатність ефективно обчислювати мінімальну відстань від початкової вершини до всіх інших вершин графу, що робить його надзвичайно корисним у транспортних системах. Алгоритм використовується для побудови маршрутів у логістиці, де необхідно знайти найкоротший або найменш витратний шлях між пунктами доставки. Завдяки його детермінованому характеру та низькій обчислювальній складності, метод Дейкстри став основою багатьох систем навігації та маршрутизації.

Суть алгоритму полягає у поступовому пошуку найкоротших шляхів, починаючи з початкової вершини, шляхом оновлення відстаней до суміжних вершин. На кожному етапі вибирається вершина з мінімальною поточною відстанню, і її відстань "закріплюється", тобто більше не змінюється. Процес продовжується до тих пір, поки всі вершини не будуть оброблені або поки не буде знайдено оптимальний маршрут до потрібної вершини.

Розглянемо зважений граф $G=(V,E)$, де V– множина вершин, E– множина ребер, а $w(u,v)$ – вага ребра між вершинами u і v. Необхідно знайти мінімальну відстань $d(s,v)$ від стартової вершини s до всіх інших вершин v у графі.

Для кожної вершини v визначаємо її початкову відстань $d(s,v)$:

$$d(s,v) = \begin{cases} 0, \text{якщо } v = s \\ \infty, \text{для всіх інших вершин.} \end{cases} \quad (1.3)$$

Алгоритм працює ітеративно, поки існують невідвідані вершини. На кожній ітерації вибирається вершина u з мінімальним значенням $d(s,u)$ серед усіх невідвіданих вершин, після чого ця вершина додається до множини S. Далі для кожної сусідньої вершини v, для якої існує ребро $(u,v) \in E$, виконується оновлення значення $d(s,v)$ за формулою:

$$d(s,v) = \min(d(s,v), d(s,u) + w(u,v)) . \quad (1.4)$$

Цей крок дозволяє поступово зменшувати оцінки шляхів до суміжних вершин, знаходячи коротший шлях, якщо такий існує через поточну вершину u.

Процес завершується, коли всі вершини оброблено або коли найкоротший шлях до цільової вершини знайдено. У результаті значення $d(s,v)$ для кожної вершини v визначає мінімальну відстань від стартової вершини s.

Наприклад, для вершини u, від якої ведуть ребра до вершин $v_1, v_2, \ldots, v_k$, відстані $d(s,v_i)$ обчислюються за формулою:

$$d(s,v_i) = \min(d(s,v_i), d(s,u) + w(u,v_i)) . \quad (1.5)$$

Ця формула гарантує, що шлях через вершину u буде враховано лише в тому випадку, якщо він коротший за раніше знайдений. Це забезпечує ефективність методу Дейкстри у пошуку найкоротших шляхів у зважених графах.

Можна сказати, що метод Дейкстри є одним із найефективніших інструментів для розв'язання задач маршрутизації у зважених графах. Його надійність та швидкість забезпечують високу точність і можливість використання у реальних умовах логістичних систем, включаючи планування оптимальних маршрутів доставки. Його застосування дозволяє зменшити час і витрати на перевезення, що робить його незамінним інструментом для сучасних логістичних компаній.

Методи маршрутизації у логістиці часто вимагають не лише точності, але й обчислювальної простоти для швидкого прийняття рішень у динамічних умовах. Один із таких підходів –*метод найближчого сусіда* (Nearest Neighbor Algorithm),

який широко використовується для побудови маршрутів у задачах комівояжера, логістики та транспортування [10]. Цей алгоритм базується на ідеї жадібного пошуку, де на кожному кроці вибирається найближчий можливий варіант, зберігаючи загальну простоту реалізації та високу швидкість обчислення. У логістичних системах метод найближчого сусіда дозволяє швидко формувати маршрути, що можуть бути корисними у випадках, коли точність не є критичною або коли необхідно отримати початкове наближення до оптимального рішення.

Розглянемо зважений граф $G = (V,E)$, де V – множина вершин, що представляють пункти доставки, а E – множина ребер із вагою $w(u,v)$, яка позначає відстань або вартість між вершинами u і v. Метод починається з довільно обраної стартової вершини s та формує маршрут, поступово додаючи до нього вершини.

На кожному кроці алгоритм обирає вершину $v \in V$, яка ще не входить до маршруту, для якої виконується:

$$v = \arg\min_{v \in V} w(u, v), \tag{1.6}$$

де u – поточна вершина, а $w(u,v)$ – вага ребра між вершинами u та v. Це означає, що з усіх можливих варіантів обирається вершина v, яка є найближчою до поточної вершини u.

Після додавання вершини v маршрут оновлюється, а множина невідвіданих вершин $V_{\text{не відвідані}}$ зменшується:

$$V_{\text{не відвідані}} = V_{\text{не відвідані}} \backslash \{v\}\,. \tag{1.7}$$

Процес повторюється до тих пір, поки всі вершини не будуть включені до маршруту, після чого алгоритм повертається до стартової вершини s, завершуючи цикл. Загальний шлях R формується як впорядкована послідовність вершин:

$$R = (s, v_1, v_2, \dots, v_k, s), \tag{1.8}$$

де v_i – вершини, відвідані в порядку їх додавання до маршруту.

Метод найближчого сусіда є простим і швидким підходом до вирішення задач маршрутизації, що дозволяє отримувати наближені рішення з мінімальними обчислювальними витратами. Завдяки своїй жадібній природі алгоритм може

використовуватися для початкової оцінки маршрутів або в ситуаціях, коли обчислювальні ресурси обмежені. Хоча він не завжди гарантує глобально оптимальний маршрут, його простота і швидкість роблять його корисним у практичних застосуваннях, особливо для невеликих графів або як етап попередньої оптимізації у складніших алгоритмах [11].

1.3 Огляд існуючих програмних рішень

Сучасні логістичні компанії широко застосовують програмні рішення для автоматизації процесів планування маршрутів, управління доставкою та оптимізації транспортних витрат. Такі системи спрямовані на забезпечення ефективного використання ресурсів, скорочення часу перевезень та підвищення якості обслуговування клієнтів. Ринок програмного забезпечення для маршрутизації пропонує як універсальні рішення, так і вузькоспеціалізовані продукти, які орієнтовані на специфічні задачі.

LogiNext Mile – це хмарне програмне забезпечення для управління транспортом, яке дозволяє користувачам планувати маршрути доставки та відстежувати транспортні засоби (рис. 1.3). Рішення підходить для таких галузей, як виробництво, фармацевтика, роздрібна торгівля, банківська справа та страхування.

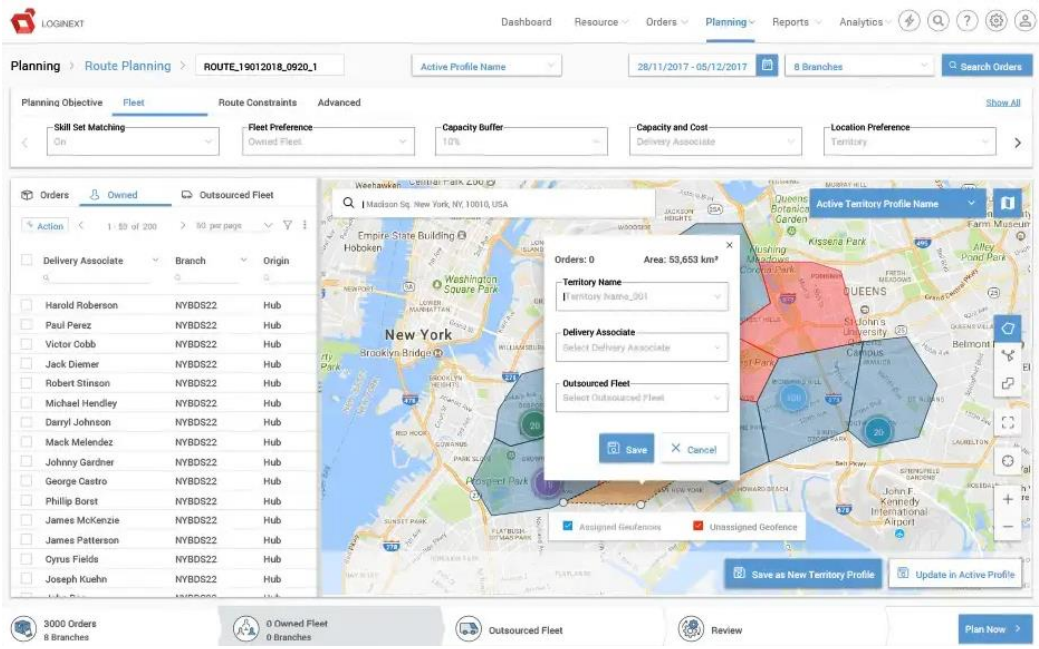


Рис. 1.3 Приклад інтерфейсу системи «LogiNext Mile» [12]

Архітектура LogiNext Mile базується на хмарній платформі, що забезпечує масштабованість та доступність для користувачів з різних галузей. Система інтегрується з мобільними додатками, які дозволяють водіям отримувати оновлення маршрутів у реальному часі та надавати зворотний зв’язок. Це забезпечує ефективне планування, моніторинг та оптимізацію маршрутів доставки, сприяючи підвищенню продуктивності та зниженню витрат [13].

Переваги LogiNext Mile:

– хмарна архітектура забезпечує масштабованість, доступність з будь-якої точки та інтеграцію з іншими системами;

– моніторинг у реальному часі дозволяє відстежувати транспортні засоби та оновлювати маршрути для підвищення ефективності доставки;

– інтуїтивно зрозумілий інтерфейс спрощує роботу як для менеджерів, так і для водіїв.

– можливість інтеграції з мобільними додатками забезпечує автоматизацію та ефективну комунікацію між логістичними командами.

Недоліки LogiNext Mile:

– залежність від стабільного інтернет-з’єднання, що може створювати проблеми в регіонах із поганим покриттям;

– висока вартість впровадження, що може бути недоступною для малих підприємств;

– вимоги до навчання персоналу через специфічний функціонал та особливості налаштування;

– обмежена адаптація до локальних умов у деяких регіонах через загальну орієнтацію на глобальні логістичні стандарти [14].

Отже, LogiNext Mile є потужним інструментом для управління логістичними операціями, який забезпечує високий рівень автоматизації та зручності в роботі з маршрутами доставки. Завдяки хмарній архітектурі та інтеграції з мобільними додатками, цей застосунок може значно покращити продуктивність логістичних компаній. Однак його використання може бути обмежене для компаній із невеликим бюджетом або у регіонах із поганою інфраструктурою інтернету.

Paragon Routing and Scheduling – це програмне забезпечення, призначене для автоматизації планування маршрутів та розкладів транспортних засобів, що дозволяє створювати оптимізовані плани перевезень за лічені хвилини, зменшуючи експлуатаційні витрати автопарку (рис. 1.4). Система розроблена для задоволення щоденних викликів транспортних операцій у різних галузях, забезпечуючи ефективне управління доставками до магазинів, офісів, домівок та інших локацій.

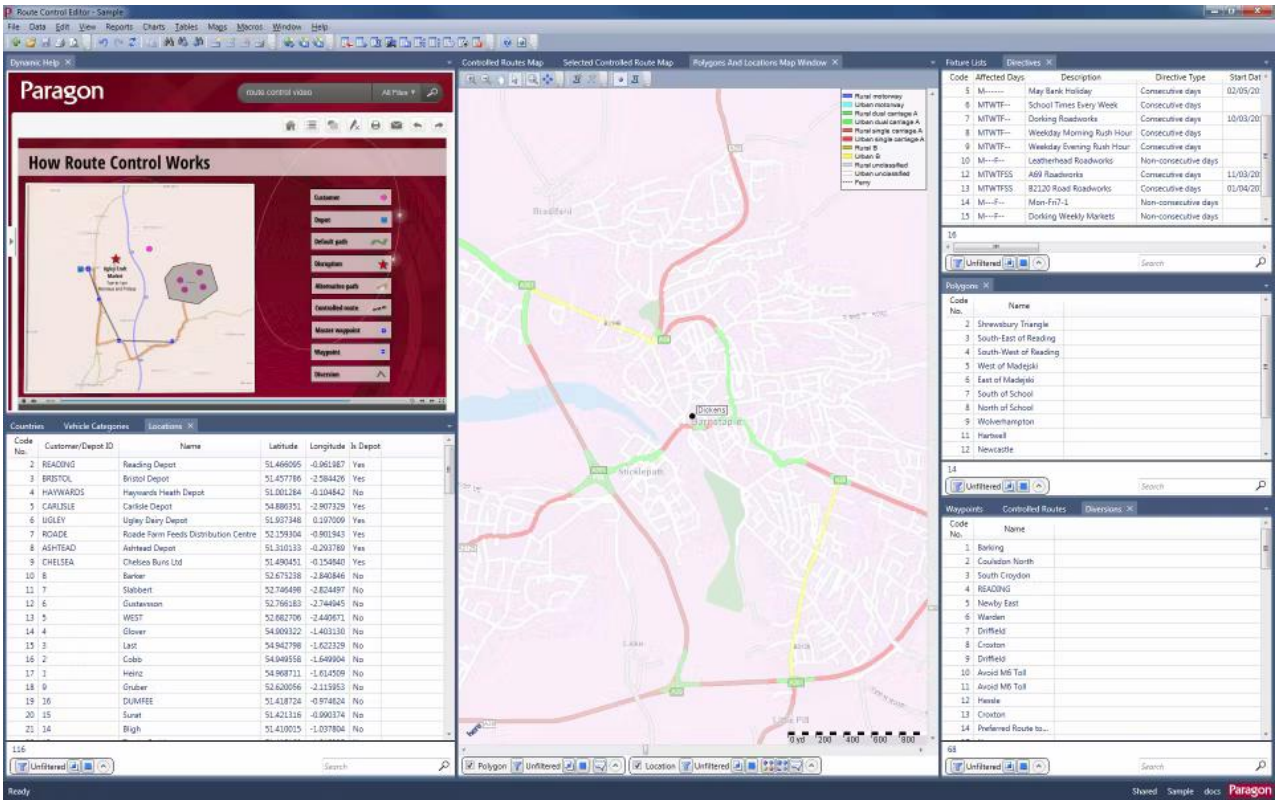


Рис. 1.4 Приклад інтерфейсу системи «Paragon Routing and Scheduling» [15]

Архітектура Paragon Routing and Scheduling базується на потужних алгоритмах оптимізації маршрутів, які враховують тисячі можливих варіантів для створення реалістичних та економічно ефективних планів перевезень. Користувачі можуть вводити або імпортувати дані про доставку, включаючи адреси, обсяги замовлень, часові вікна, розміри транспортних засобів та графіки водіїв. Система обробляє цю інформацію, генеруючи транспортний план, що максимально використовує наявні ресурси та відповідає бізнес-показникам і вимогам клієнтів.

Дане ПЗ також інтегрується з телематичними рішеннями для транспортних засобів та власним програмним забезпеченням для підтвердження доставки, що дозволяє в реальному часі аналізувати заплановані та фактичні маршрути, виявляти розбіжності та постійно покращувати плани маршрутів. Крім того, система надає можливість моделювання сценаріїв "що, якщо" для стратегічного планування та визначення покращень в операціях, територіальному покритті або транспортній інфраструктурі [16].

Переваги Paragon Routing and Scheduling:

– потужні алгоритми оптимізації маршрутів, які забезпечують ефективне планування навіть для складних операцій;

– інтеграція з телематичними рішеннями дозволяє здійснювати моніторинг та аналіз маршрутів у реальному часі;

– гнучкість у налаштуванні параметрів для різних типів транспорту, обмежень та часових вікон;

– функція моделювання сценаріїв "що, якщо" дає змогу стратегічно планувати та оптимізувати майбутні операції.

Недоліки Paragon Routing and Scheduling:

– висока вартість впровадження та ліцензії, що робить продукт менш доступним для малих і середніх компаній;

– складність налаштування для користувачів без технічного досвіду, що може вимагати додаткового навчання персоналу;

– залежність від точності вхідних даних, через що помилки або неточності можуть впливати на якість маршрутів.

– вимоги до інтеграції з іншими системами, що може потребувати додаткових ресурсів і часу [17].

Отже, Paragon Routing and Scheduling є високоефективним інструментом для автоматизації логістичних операцій, який підходить для компаній із великими обсягами перевезень і складними транспортними вимогами. Завдяки потужним алгоритмам, інтеграції з телематичними системами та функціоналу моделювання, цей застосунок сприяє зниженню витрат і підвищенню точності планування. Однак висока вартість та складність у налаштуванні обмежують його доступність для менших підприємств.

Onfleet – це програмне забезпечення для управління доставками, яке допомагає компаніям ефективно керувати процесами доставки на «останній милі» (рис. 1.5). Воно дозволяє підприємствам призначати та відстежувати місцеві доставки, спілкуватися з водіями та клієнтами, а також автоматично оптимізувати маршрути.

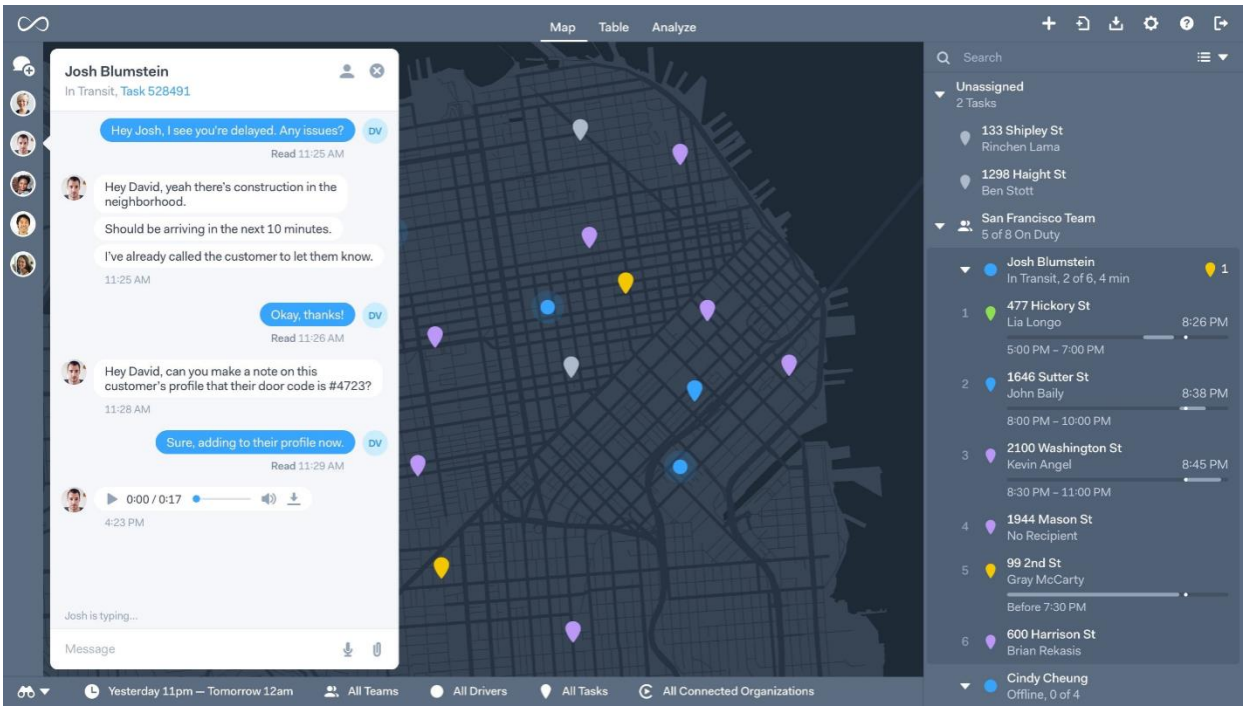


Рис. 1.5 Приклад інтерфейсу системи «Onfleet» [18]

Архітектура Onfleet базується на RESTful API, що дозволяє розробникам програмно взаємодіяти з даними та функціоналом управління доставками в реальному часі. Обмін даними між клієнтами та API здійснюється через JSON по HTTPS, забезпечуючи безпечну та ефективну інтеграцію з іншими системами. Окрім того, Onfleet надає мобільний додаток для водіїв, який дозволяє їм отримувати завдання, оновлювати статуси доставок та спілкуватися з диспетчерами, що сприяє покращенню координації та підвищенню продуктивності [19].

Переваги Onfleet:

– зручний інтерфейс для диспетчерів, водіїв і клієнтів, який забезпечує простоту використання та ефективну комунікацію;

– мобільний додаток для водіїв дозволяє в реальному часі оновлювати статуси доставок та отримувати оптимізовані маршрути;

– інтеграція через RESTful API забезпечує гнучкість і можливість легкої інтеграції з іншими системами компанії;

– функції оптимізації маршрутів автоматично розподіляють завдання, знижуючи витрати часу та ресурсів.

Недоліки Onfleet:

– висока вартість ліцензії, що може бути обмеженням для невеликих компаній;

– обмежений функціонал для багатонаціональних операцій, орієнтованих на масштабовані перевезення;

– залежність від стабільного інтернет-з’єднання, без якого основні функції програми стають недоступними;

– відсутність детальних звітів для глибокого аналізу ефективності логістичних операцій, що може вимагати додаткових інструментів [20].

Отже, Onfleet є потужним рішенням для управління доставками на «останній милі», орієнтованим на підприємства, які прагнуть автоматизувати свої логістичні операції. Інтуїтивно зрозумілий інтерфейс, мобільний додаток для водіїв і можливість інтеграції через API роблять його зручним інструментом для компаній, які потребують гнучкого рішення. Проте висока вартість і обмеження у функціоналі для великих чи багатонаціональних операцій можуть обмежувати його застосування в деяких сегментах бізнесу.

Аналіз існуючих програмних рішень показав, що більшість із них орієнтовані на великі компанії, мають високу вартість впровадження та складність у налаштуванні, що робить їх недоступними для малого та середнього бізнесу. Крім того, залежність від постійного інтернет-з’єднання обмежує їх використання в регіонах із нестабільною мережею. Це обґрунтовує необхідність розробки нового рішення з простішим функціоналом, який забезпечує легкість впровадження та використання, а також надає безкоштовну базову версію для тестування і старту. Важливим аспектом такого рішення є можливість офлайнової роботи, що дозволить забезпечити стабільність операцій навіть у віддалених або важкодоступних регіонах. Така система стане ефективним інструментом для широкого кола користувачів, що підвищить її адаптивність до локальних умов.

1.4 Формулювання задачі дослідження

У рамках даного дослідження формулюється задача створення програмного забезпечення, яке стане інструментом для планування маршрутів, здатним вирішувати актуальні проблеми сучасних логістичних систем. Основними характеристиками розробленого рішення будуть простота використання, що забезпечить мінімальні вимоги до технічних навичок користувачів, а також можливість функціонування в офлайновому режимі, що дозволить застосовувати систему навіть у регіонах із нестабільним інтернет-з'єднанням. Крім того, система буде відзначатися легкою інтеграцією в існуючі бізнес-процеси компаній та доступністю, що робить її привабливою для малих і середніх підприємств із обмеженими ресурсами.

Функціонал розроблюваного програмного забезпечення включатиме ідентифікацію користувачів для забезпечення контролю доступу, управління базою даних міст і маршрутів, проведення симуляцій для тестування різних сценаріїв доставки, а також можливість збереження результатів і перегляду системних подій. Основою роботи системи стане реалізація алгоритмів маршрутизації, які забезпечать можливість автоматизованого визначення найкоротших маршрутів між заданими пунктами. У межах розроблюваної системи передбачається створення функціоналу, що дозволить користувачам обирати початковий пункт для пошуку оптимальних маршрутів до інших міст із подальшим відображенням результатів у зручному текстовому форматі. Також система має забезпечити генерацію бази даних маршрутів із зазначенням початкових і кінцевих точок та відповідних відстаней, що сприятиме зручності аналізу. Такий підхід дозволить створити інструмент для підвищення ефективності логістичних процесів за рахунок оптимального використання ресурсів і зниження витрат.

1.5 Висновок до розділу

У рамках даного розділу проведено аналіз сучасних методів і підходів до побудови маршрутів у логістичних системах, що дозволило визначити основні

проблеми та вимоги до маршрутної мережі. Розглянуто методи оптимізації маршрутів, зокрема метод гілок і кордонів, метод Дейкстри та метод найближчого сусіда, що забезпечило розуміння можливостей та обмежень кожного з них. Проведено огляд існуючих програмних рішень, таких як LogiNext Mile, Paragon Routing and Scheduling та Onfleet, із виділенням їх переваг та недоліків, що обґрунтовує необхідність розробки нового адаптованого рішення. На основі аналізу сформульовано задачу дослідження, яка полягає у створенні системи для планування оптимальних маршрутів, що буде відзначатися простотою використання, офлайновою доступністю, легкістю впровадження та можливістю задовольняти потреби малого і середнього бізнесу. Отримані результати дозволяють перейти до наступного розділу, де будуть обґрунтовані вибір технологій, архітектура системи та її реалізація.

РОЗДІЛ 2. ТЕХНОЛОГІЧНІ ОСНОВИ ТА АРХІТЕКТУРНІ РІШЕННЯ

2.1 Обґрунтування вибору технологій

При розробці систем підтримки прийняття рішень для вибору оптимальних маршрутів доставки в логістичних компаніях важливим аспектом є вибір програмного забезпечення, що базується на сучасних алгоритмах і враховує вимоги до масштабованості, продуктивності та гнучкості. Одним із ключових рішень на ранніх етапах розробки є вибір мови програмування, яка найкраще відповідатиме поставленим цілям. Залежно від обраної мови програмування, реалізація алгоритмів, інтеграція з базами даних і модульність системи можуть мати різний рівень складності.

Мова програмування Python є однією з найпопулярніших завдяки простоті використання, великій кількості бібліотек для обробки даних (наприклад, Pandas, NumPy) і алгоритмів оптимізації (Scikit-learn, NetworkX) [21]. Вона ідеально підходить для швидкого прототипування та аналізу даних, однак її продуктивність на великих обсягах даних може бути нижчою порівняно з іншими мовами. Крім того, Python має широке ком’юніті, що забезпечує підтримку для складних проектів.

Java вирізняється високою продуктивністю, платформною незалежністю та добре розвиненою екосистемою для розробки корпоративних рішень [22]. Завдяки потужній стандартній бібліотеці Java часто використовується для створення масштабованих систем, які потребують високого рівня безпеки та стійкості. Проте синтаксис Java є складнішим, що може збільшувати час розробки у порівнянні з Python.

Мова програмування C# є популярним вибором для розробки високопродуктивних і інтегрованих рішень, особливо в середовищі Microsoft [23]. Вона забезпечує простоту роботи з графічними інтерфейсами, інтеграцію з базами даних та підтримку сучасних хмарних технологій. C# відома своєю

продуктивністю та можливостями ефективної розробки візуально-орієнтованих систем, хоча її застосування може бути обмеженим поза межами екосистеми Microsoft.

Табл. 2.1 містить оцінки як якісних, так і кількісних показників, що дозволяє детальніше розглянути їхні переваги та обмеження.

Таблиця 2.1.

Порівняння мов програмування

Характеристика	Python	Java	C#
Простота синтаксису	Дуже висока	Середня	Висока
Поріг входження	Низький	Середній	Низький
Продуктивність	Середня	Висока	Висока
Інструменти для GUI	Обмежені	Середні	Широкі
Інтеграція з MS екосистемою	Обмежена	Обмежена	Повна
Час виконання, мс (простий алгоритм)	12-15	8-10	7-9
Вартість розробки	Низька	Середня	Середня
Багатоплатформність	Висока	Висока	Висока
Підтримка паралельності	Обмежена	Висока	Висока

Вибір мови програмування C# для розробки системи підтримки прийняття рішень у логістичних компаніях є обґрунтованим через її високі показники продуктивності, можливість ефективної інтеграції з екосистемою Microsoft та розвинуті інструменти для створення графічних інтерфейсів. Завдяки потужній підтримці паралельного програмування C# забезпечує оптимальне виконання алгоритмів маршрутизації навіть при великих обсягах даних. Окрім того, багатоплатформність цієї мови дозволяє створювати рішення, які можуть функціонувати на різних пристроях, а інтеграція з хмарними сервісами значно розширює можливості масштабування системи. Зважаючи на ці переваги, використання C# є раціональним вибором для побудови надійної, продуктивної та гнучкої системи.

Також, важливим аспектом є вибір середовища розробки, яке забезпечує зручність програмування, потужні інструменти налагодження та підтримку

сучасних технологій. Серед найпоширеніших середовищ для роботи з мовою програмування C# можна виділити Visual Studio 2022, Visual Studio Code та Rider, кожне з яких має свої особливості та переваги.

Visual Studio 2022 – це потужне інтегроване середовище розробки (IDE), що забезпечує розширені функції налагодження, автоматизацію збірки та інтеграцію з хмарними сервісами Microsoft [24]. Воно є оптимальним вибором для великих проектів, оскільки підтримує широкі можливості для командної роботи та роботу з великими кодовими базами.

Visual Studio Code – це легке, кросплатформенне середовище з відкритим кодом, яке підтримує широкий спектр мов програмування через розширення [25]. Воно підходить для швидкого прототипування та невеликих проектів завдяки простоті налаштування і високій швидкості роботи.

Rider – це інтегроване середовище розробки від JetBrains, яке поєднує багатий функціонал із оптимізованою продуктивністю [26]. Rider особливо цінують розробники, які працюють із .NET-проектами, завдяки ефективному аналізу коду та зручному інтерфейсу.

Табл. 2.2 демонструє ключові аспекти, які мають значення для розробки систем підтримки прийняття рішень, зокрема продуктивність, можливості інтеграції та підтримка інструментів.

Таблиця 2.2.

Порівняння середовищ розробки

Характеристика	Visual Studio 2022	Visual Studio Code	Rider
Продуктивність на великих проектах	Висока	Середня	Висока
Інструменти налагодження	Дуже розвинені	Середні	Розвинені
Підтримка .NET Framework	Повна	Через розширення	Повна
Інструменти аналізу коду	Вбудовані	Через розширення	Розвинені
Можливості для командної роботи	Широкі	Середні	Широкі
Підтримка великих кодових баз	Висока	Середня	Висока
Кількість готових розширень	Дуже велика	Дуже велика	Велика
Швидкість запуску середовища	Середня	Висока	Середня

Вибір Visual Studio 2022 для розробки системи підтримки прийняття рішень у логістичних компаніях є обґрунтованим завдяки її потужним інструментам для роботи з великими кодовими базами та високим рівнем інтеграції з платформою .NET. Це середовище пропонує розвинуті засоби налагодження, що дозволяють ефективно виявляти та виправляти помилки на різних етапах розробки. Повна інтеграція з хмарними сервісами Azure забезпечує можливість масштабування системи та підтримку сучасних технологій. Окрім того, Visual Studio 2022 підтримує інструменти для командної роботи, що є важливим для координації в умовах багатокористувацької розробки. Завдяки цим характеристикам середовище є оптимальним вибором для побудови комплексної, стабільної та продуктивної системи.

2.2 Визначення вимог до системи та варіанти використання

Для успішної розробки системи підтримки прийняття рішень для вибору оптимальних маршрутів доставки необхідно чітко визначити вимоги до її функціональності, які забезпечуватимуть ефективне виконання всіх необхідних задач. Система має бути здатною надавати широкий спектр можливостей для різних типів користувачів, включаючи системних адміністраторів, які керують основними процесами, та звичайних користувачів, що працюють з функціями аналізу та моделювання [27]. Водночас система повинна гарантувати інтеграцію з сучасними інструментами обробки даних, моделювання маршрутів та забезпечувати легкий доступ до управління всіма ключовими ресурсами. Основні вимоги визначаються на основі ключових задач, які виконує система, зокрема управління користувачами, моделювання маршрутів, обробка даних про міста, а також ведення журналу подій, що є важливими для прозорості та контролю.

Функціональні вимоги системи є основою її реалізації, адже саме вони визначають її здатність відповідати реальним потребам користувачів і вимогам організації. Їх формалізоване представлення не лише спрощує процес розробки, а

й забезпечує гарантію, що створене рішення буде відповідати запитам та очікуванням. Основні функціональні вимоги до системи наведені в таблиці 2.3.

Таблиця 2.3.

Функціональні вимоги до системи

№	Опис вимоги
1	Система повинна забезпечувати ідентифікацію користувачів через авторизацію.
2	Система повинна дозволяти системним адміністраторам переглядати каталог усіх користувачів.
3	Система повинна дозволяти створення, редагування та видалення інформації про користувачів.
4	Система повинна дозволяти створення, редагування та видалення інформації про міста.
5	Система повинна забезпечувати проведення симуляції маршрутів із заданою кількістю міст.
6	Система повинна надавати можливість зберігати та завантажувати результати симуляцій.
7	Система повинна вести журнал подій із можливістю перегляду всіма користувачами.

Розробка системи передбачає підтримку як базових функцій управління даними, так і спеціалізованих задач, таких як симуляція маршрутів і обробка результатів. Функціональні вимоги забезпечують комплексний підхід до реалізації системи, де кожен компонент відповідає за виконання конкретної задачі, гарантуючи високу продуктивність і зручність використання.

Окрім функціональних вимог, важливим аспектом успішної розробки є визначення нефункціональних вимог, які визначають якість роботи системи, її надійність, продуктивність і зручність використання. Ці вимоги задають рамки для реалізації функціональних можливостей та забезпечують відповідність системи очікуванням користувачів і потребам організації. Вони гарантують, що система працюватиме стабільно в реальних умовах експлуатації та забезпечуватиме необхідний рівень інтеграції.

Нефункціональні вимоги до системи сформульовані з урахуванням характеристик, важливих для логістичних компаній, таких як масштабованість, продуктивність і безпека. Основні нефункціональні вимоги наведені в табл. 2.4.

Таблиця 2.4.

Нефункціональні вимоги до системи

№	Опис вимоги
1	Система повинна обробляти запити користувачів із затримкою не більше 1 секунди.
2	Система повинна підтримувати одночасну роботу не менше 100 користувачів.
3	Система повинна бути сумісною з операційними системами Windows, Linux та macOS.
4	Система повинна забезпечувати безпечне зберігання даних із використанням механізмів шифрування.
5	Система повинна забезпечувати автоматичне резервне копіювання даних не рідше одного разу на добу.
6	Інтерфейс системи повинен бути інтуїтивно зрозумілим і відповідати сучасним стандартам UX/UI.
7	Система повинна бути адаптована для роботи на пристроях із різними розмірами екрану.

Нефункціональні вимоги забезпечують стабільність роботи системи, її відповідність технічним умовам та потребам кінцевих користувачів. Наприклад, підтримка високої продуктивності та сумісності гарантує ефективну роботу системи навіть при значних навантаженнях, а сучасні підходи до безпеки та резервного копіювання даних мінімізують ризики втрати інформації. Це створює основу для розробки системи, яка відповідатиме високим стандартам якості.

Актори – це особи або сутності, які взаємодіють із системою з метою виконання певних завдань, що визначаються їхніми конкретними ролями, функціональними обов’язками та цілями у межах роботи системи. У даній системі основними акторами є адміністратор, користувач і база даних, кожен із яких виконує унікальні функції, які безпосередньо впливають на працездатність і ефективність роботи застосунку. Адміністратор контролює безпеку та стабільність, користувачі взаємодіють із функціональними можливостями системи, а база даних забезпечує надійне збереження й доступ до інформації. Мета кожного актора полягає у виконанні своїх завдань із максимальною

ефективністю, що дозволяє оптимізувати ключові процеси управління маршрутами, підтримувати безпеку й доступність даних.

Таблиця 2.5.

Актори та їхні цілі

Актори	Цілі
Адміністратор	Адміністратор відповідає за управління доступом користувачів, забезпечення безпеки системи, цілісності та конфіденційності даних, а також контроль за стабільною роботою системи.
Користувач	Користувачі взаємодіють із системою для симуляції маршрутів, управління даними про міста та збереження результатів моделювання, спрощуючи процеси прийняття логістичних рішень.
База даних	База даних забезпечує збереження, організацію та швидкий доступ до даних системи, підтримуючи інтеграцію та узгодженість інформації для всіх акторів.

Варіанти використання визначають конкретні сценарії взаємодії користувачів із системою для досягнення визначених цілей. Вони описують дії, які можуть виконувати користувачі, адміністратори чи система в цілому, а також результати цих дій. У межах цієї системи варіанти використання включають ідентифікацію користувачів, управління даними про користувачів та міста, проведення симуляцій маршрутів, а також обробку та збереження даних (табл. 2.4). Кожен варіант використання має унікальне ім’я, яке відображає його функціональне призначення, і короткий опис, що пояснює його мету.

Таблиця 2.4.

Варіанти використання системи

Варіант	Ім’я	Опис
1	2	3
UC1	Авторизація в системі	Забезпечує процес входу користувачів до системи через перевірку введених облікових даних.
UC2	Реєстрація користувача	Дозволяє новому користувачу створити обліковий запис, ввівши необхідні дані для подальшої роботи із системою.
UC3	Перегляд списку користувачів	Дозволяє адміністратору отримати доступ до повного списку зареєстрованих користувачів.

UC4	Додавання нового користувача	Надає можливість додати до системи нового користувача із зазначенням його персональних даних.

Продовження таблиці 2.4.

1	2	3
UC5	Оновлення даних користувача	Дозволяє адміністратору редагувати інформацію про існуючого користувача системи.
UC6	Видалення облікового запису	Забезпечує видалення облікового запису вибраного користувача із бази даних системи.
UC7	Перегляд списку міст	Дозволяє користувачам переглядати базу даних усіх доступних міст із їхніми характеристиками.
UC8	Додавання нового міста	Дозволяє користувачеві додати новий запис про місто до бази даних.
UC9	Оновлення інформації про місто	Забезпечує редагування існуючої інформації про вибране місто.
UC10	Видалення міста	Надає можливість видалення даних про місто із бази системи.
UC11	Симуляція маршрутів	Дозволяє змоделювати оптимальні маршрути доставки, враховуючи вибрану кількість міст.
UC12	Збереження результатів моделювання	Забезпечує можливість збереження результатів моделювання маршрутів для подальшого аналізу.
UC13	Завантаження даних моделювання	Дозволяє відновити раніше збережені результати симуляції для подальшої роботи.
UC14	Журнал подій	Надає можливість переглянути хронологію основних подій, що відбувалися у системі.

Діаграма використання для ролі адміністратора демонструє всі можливості взаємодії з системою, доступні для цього типу користувача (рис. 2.1). Адміністратор має розширені повноваження, включаючи управління обліковими записами користувачів, модифікацію та видалення інформації про міста, а також перегляд журналу подій. Крім того, адміністратор може виконувати симуляцію маршрутів, зберігати результати моделювання та завантажувати попередньо збережені дані.

У свою чергу діаграма використання для ролі користувача ілюструє функціонал, доступний звичайному користувачеві системи (рис. 2.2). Основні задачі включають перегляд списку міст, додавання, редагування та видалення інформації про міста, а також проведення симуляцій маршрутів. Користувач може зберігати результати моделювання та завантажувати збережені дані для подальшого використання.

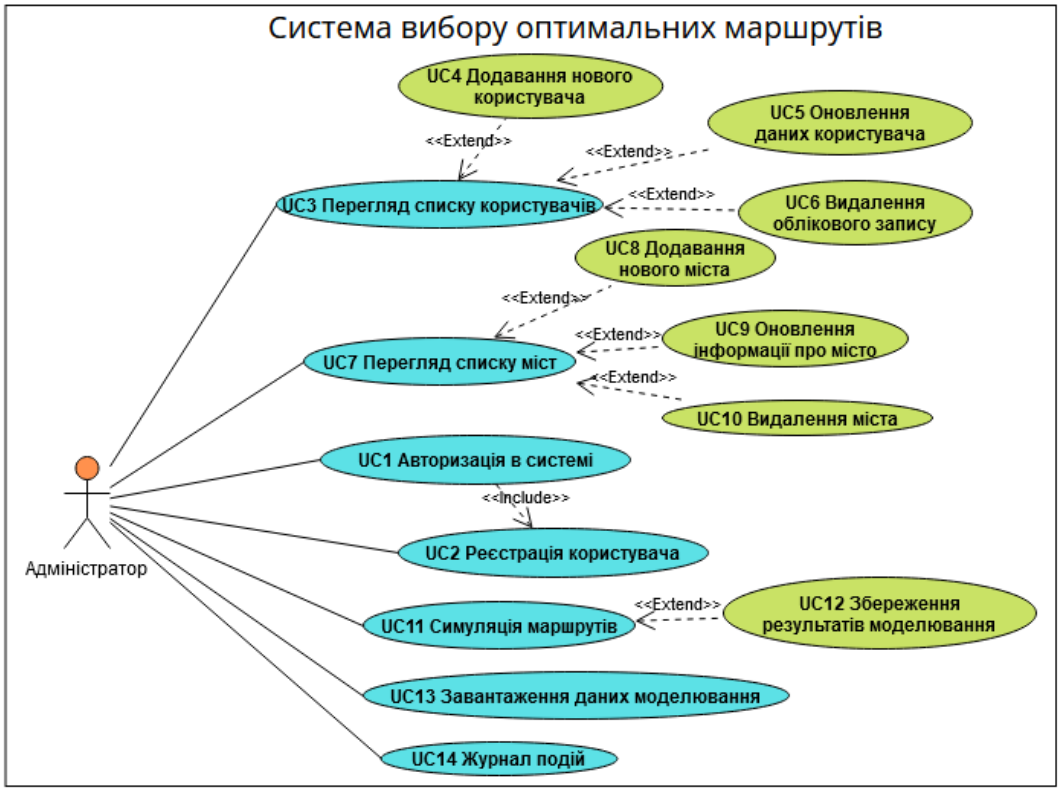


Рис. 2.1 Діаграма використання системи для ролі «адміністратор»

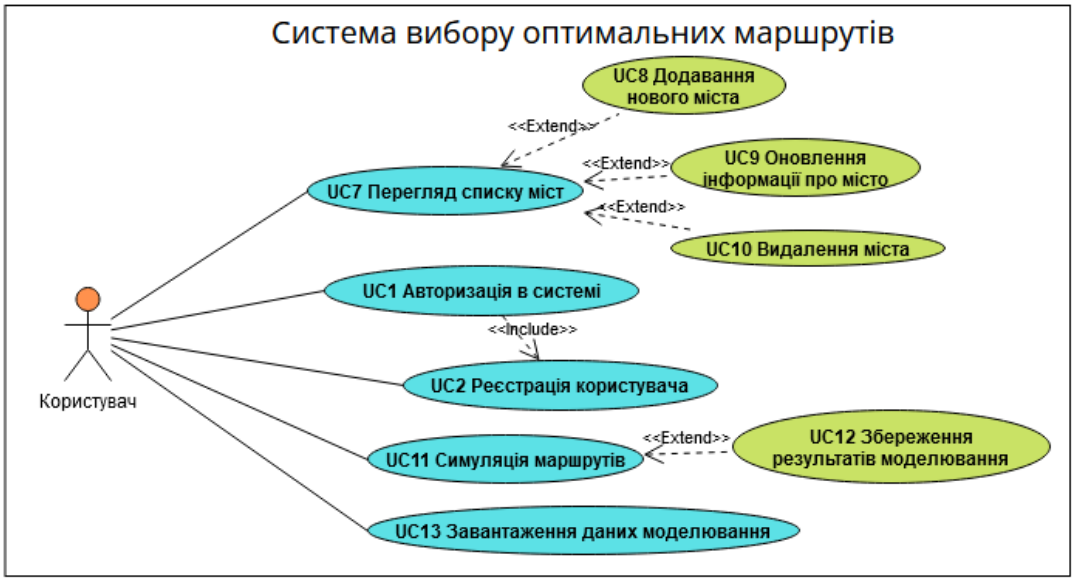


Рис. 2.2 Діаграма використання системи для ролі «користувач»

Таким чином, чітке визначення функціональних і нефункціональних вимог, а також варіантів використання системи дозволяє створити основу для її ефективного проєктування. Діаграми використання для кожної ролі актора допомагають візуалізувати взаємодію користувачів із системою, що полегшує розробку і впровадження програмного забезпечення.

2.3 Моделювання основних процесів

Для розробки системи підтримки прийняття рішень для вибору оптимальних маршрутів доставки важливо детально змоделювати основні процеси її функціонування. Це дозволяє виявити можливі проблеми, визначити логіку виконання основних задач і забезпечити відповідність системи її функціональним вимогам. Моделювання процесів відображає ключові кроки взаємодії користувачів із системою, зокрема процеси авторизації, управління даними та проведення симуляцій.

Одним із основних процесів системи є авторизація користувачів, яка забезпечує безпечний доступ до функцій застосунку. Діаграма діяльності на рис. 2.3 демонструє етапи цього процесу.

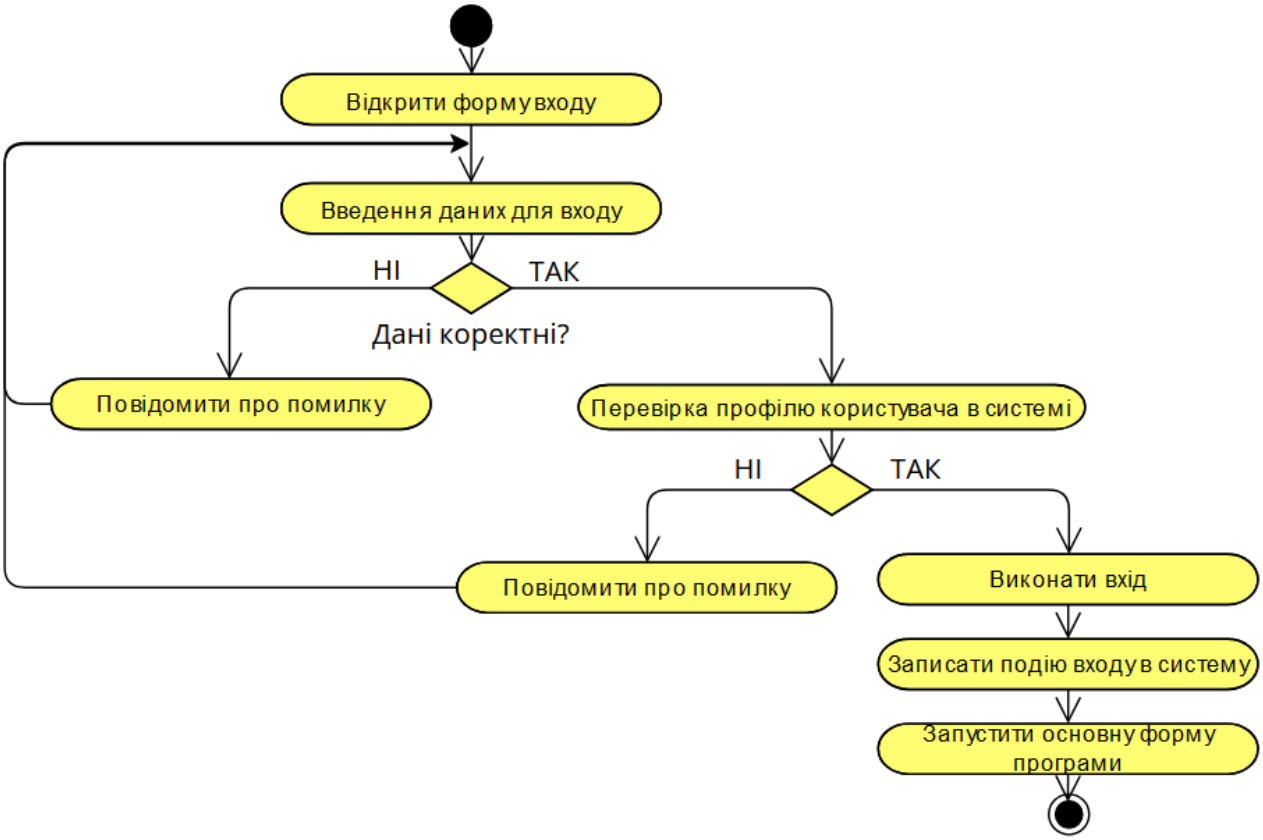


Рис. 2.3 Процес авторизації користувачів

Процес авторизації починається з відкриття форми входу до системи, після чого користувач вводить свої дані для входу (логін та пароль). Система перевіряє коректність введених даних: якщо вони неправильні, користувач отримує повідомлення про помилку та може повторити спробу. У разі успішної перевірки введених даних система виконує додаткову перевірку профілю користувача в базі даних. Якщо обліковий запис не знайдено або є інші проблеми, система повідомляє про помилку. У разі успішного підтвердження профілю здійснюється вхід до системи, запис події входу до журналу та запуск основної форми програми.

На рис. 2.4 наведено діаграму діяльності, яка описує процес додавання нового міста або населеного пункту до системи.

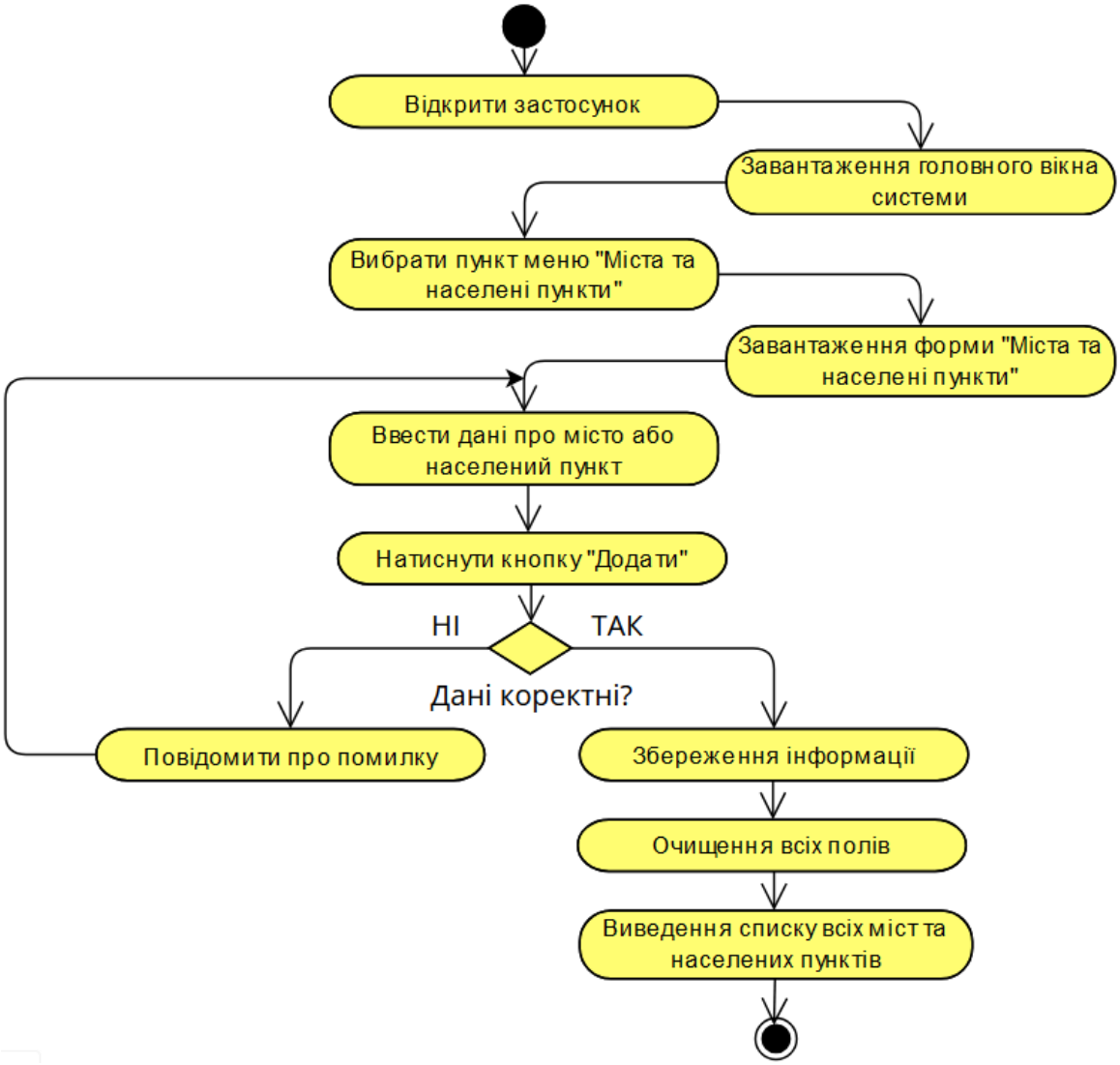


Рис. 2.4 Процес додавання інформації про населений пунктабо місто

Процес починається з відкриття застосунку, після чого завантажується головне вікно системи. Користувач вибирає пункт меню "Міста та населені пункти", що викликає завантаження відповідної форми для роботи з даними про міста.

Користувач вводить дані про нове місто або населений пункт у зазначені поля форми, а потім натискає кнопку «Додати». Система виконує перевірку коректності введених даних. Якщо введені дані містять помилки або не відповідають вимогам, система повідомляє користувача про помилку, дозволяючи виправити дані та повторити спробу. У разі успішної перевірки система зберігає введену інформацію, очищає всі поля форми для введення нових даних та оновлює список міст і населених пунктів, який відображається користувачеві.

У разі успішної перевірки даних система зберігає інформацію про продукцію, очищує всі поля для можливості наступного введення і виводить оновлений список усіх товарів. Цей процес забезпечує швидке та зручне додавання нових товарів до бази даних із мінімальним ризиком помилок.

Такий підхід дозволяє забезпечити зручність та надійність введення даних, виключаючи можливість збереження некоректної інформації. Процес автоматизує управління даними про міста та гарантує їхню актуальність у базі системи.

На рис. 2.5 зображено діаграму діяльності, яка описує процес редагування даних обраного населеного пункту або міста в системі. Процес починається з відкриття застосунку та завантаження головного вікна системи. Користувач обирає пункт меню "Міста та населені пункти", після чого завантажується форма для роботи з відповідними даними. Далі користувач зі списку доступних записів обирає необхідний запис (населений пункт або місто), який потребує змін. Після цього завантажується форма для редагування даних обраного запису, де користувач може внести необхідні зміни. Після завершення редагування користувач натискає кнопку "Зберегти". Система перевіряє введені дані на коректність.

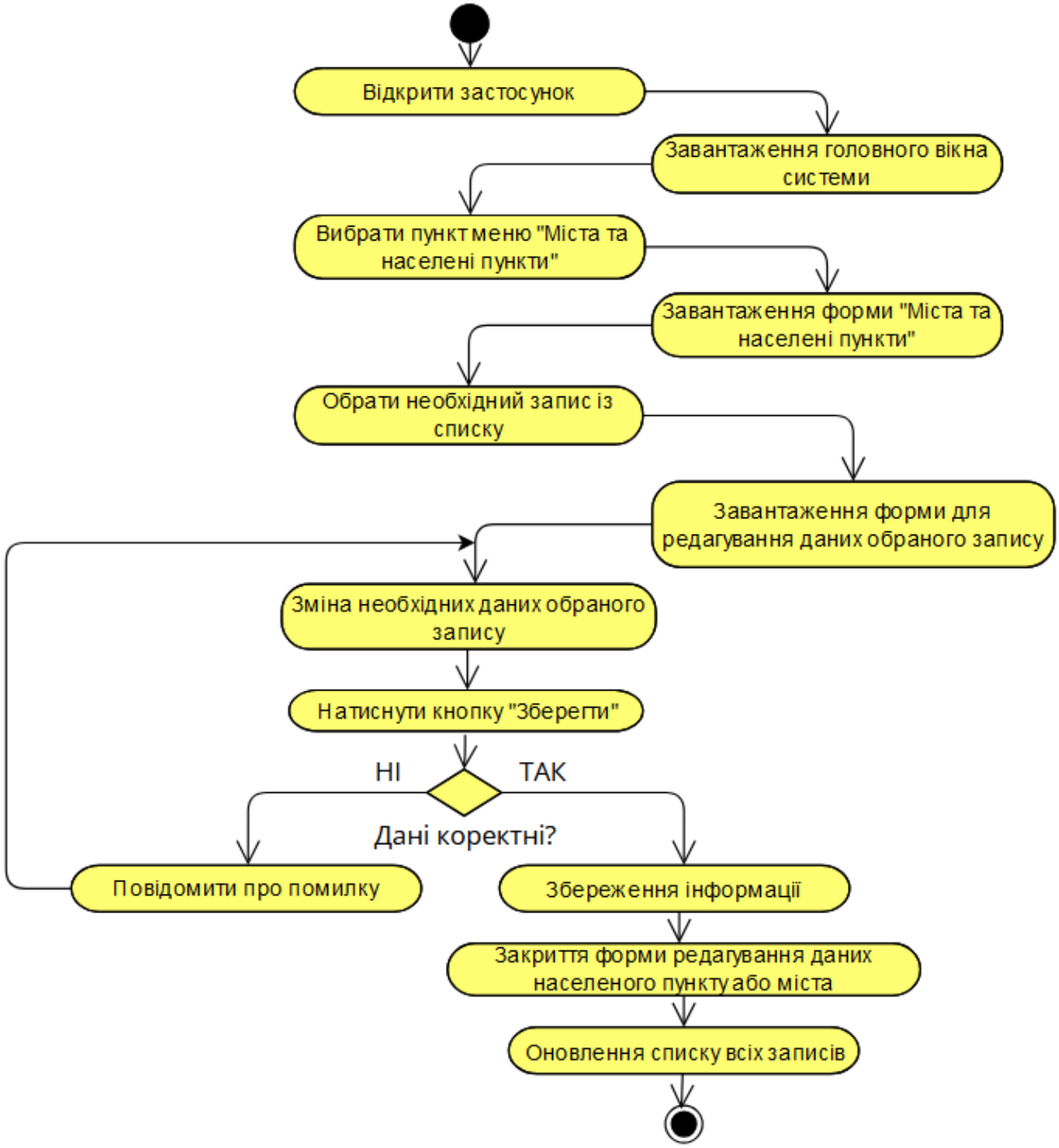


Рис. 2.5 Процес редагування даних обраного населеного пункту або міста

Якщо введені дані некоректні або містять помилки, система повідомляє користувача про помилку і дозволяє виправити дані. У разі успішної перевірки дані зберігаються в базі системи, форма редагування автоматично закривається, і список записів оновлюється для відображення актуальної інформації.

Діаграми послідовності є ключовим інструментом для моделювання динамічної поведінки системи [28]. Вони дозволяють візуалізувати порядок

взаємодій між об'єктами та акторами системи в рамках конкретного процесу. Такі діаграми показують, як інформація чи команди передаються між компонентами, і забезпечують краще розуміння логіки роботи системи. У контексті системи вибору оптимальних маршрутів діаграми послідовності відображають порядок дій користувача, системи та внутрішніх модулів у різних сценаріях роботи, таких як авторизація, управління даними або моделювання маршрутів.

На рис. 2.6 представлено діаграму послідовності процесу симуляції маршрутів, яка демонструє кроки взаємодії між користувачем, формою симуляції, модулями генерації графів, пошуку найкоротших шляхів та базою даних.

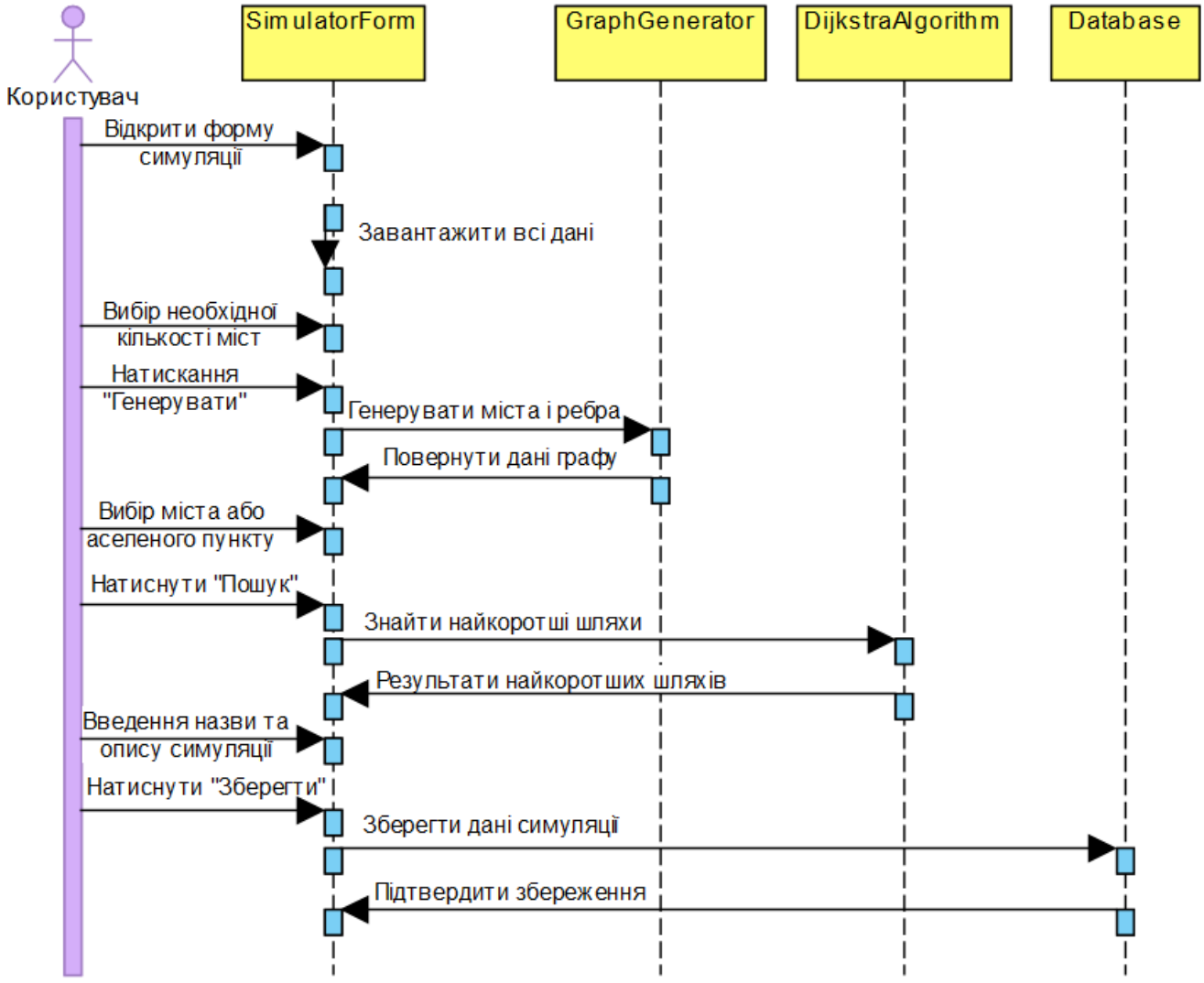


Рис. 2.6 Процес симуляції маршрутів

Процес починається з того, що користувач відкриває форму симуляції у системі. Після цього форма симуляції (SimulatorForm) завантажує всі необхідні

дані, які зберігаються в базі даних. Користувач обирає необхідну кількість міст для моделювання і натискає кнопку "Генерувати". Модуль генерації графів (GraphGenerator) створює дані про міста та зв’язки між ними (ребра графу) і повертає ці дані формі симуляції.

Наступним етапом користувач обирає місто або населений пункт, задає параметри пошуку маршруту та натискає кнопку «Пошук». Система передає дані до модуля пошуку найкоротших шляхів (DijkstraAlgorithm), який обчислює оптимальні маршрути. Результати цих обчислень повертаються користувачеві через форму симуляції.

На завершальному етапі користувач вводить назву та опис симуляції, після чого натискає кнопку "Зберегти". Дані про результати симуляції передаються в базу даних для збереження, а система підтверджує успішне виконання операції.

На рис. 2.7 наведено діаграму послідовності, яка ілюструє процес роботи з раніше збереженими симуляціями в системі. Цей процес дає користувачам можливість переглядати, завантажувати та аналізувати дані попередніх симуляцій.

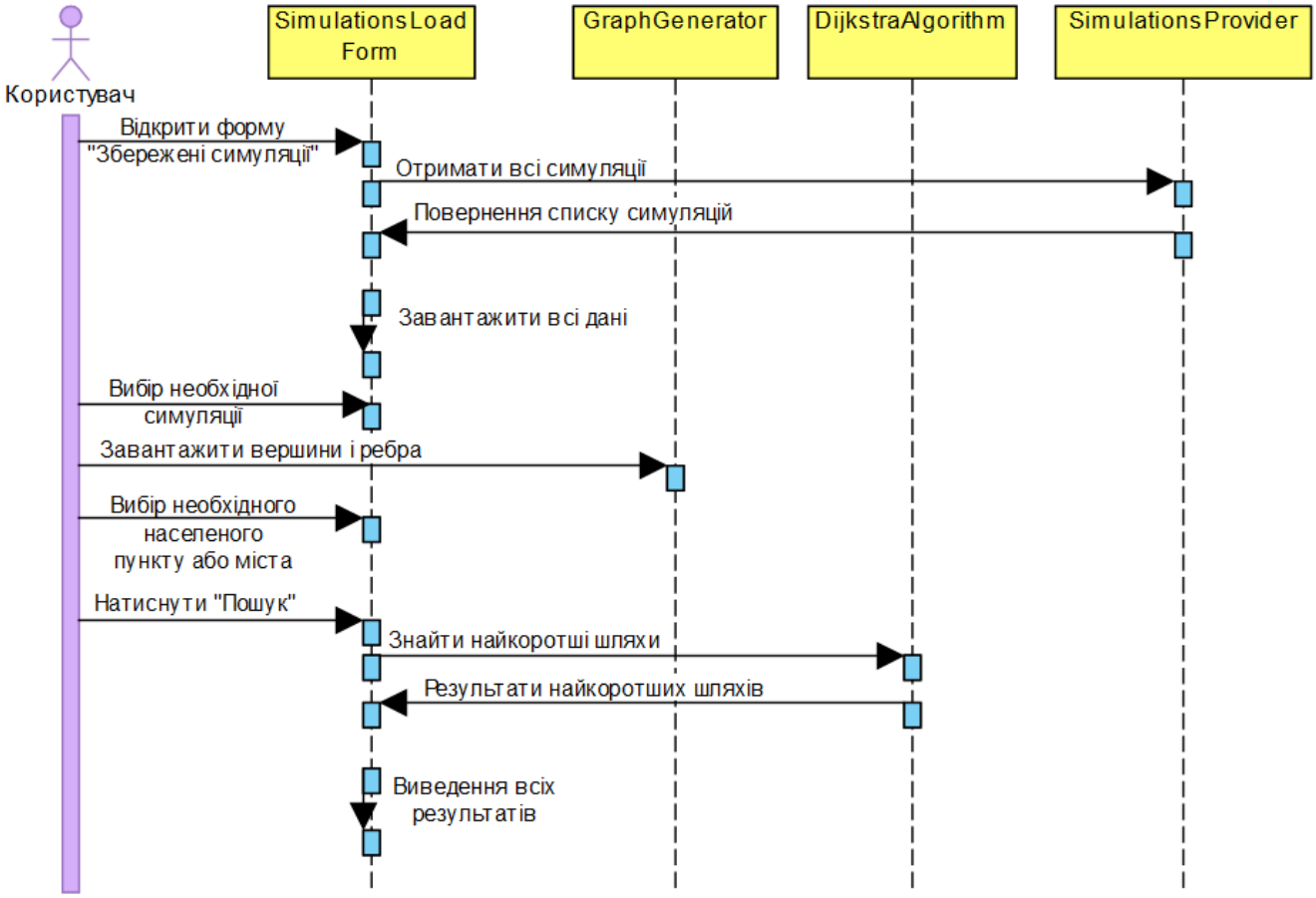


Рис. 2.7 Процес керування завантаженою симуляцією

Процес починається з того, що користувач відкриває форму "Збережені симуляції" (SimulationsLoadForm). Форма надсилає запит до модуля SimulationsProvider, щоб отримати список усіх збережених симуляцій. Після цього список симуляцій повертається та відображається користувачеві. Користувач обирає потрібну симуляцію зі списку, і система завантажує всі дані, пов'язані з вибраною симуляцією, включаючи вершини (міста) та ребра (зв’язки між містами). Далі користувач задає параметри для аналізу, обираючи конкретний населений пункт або місто, та натискає кнопку "Пошук". Дані передаються модулю GraphGenerator для побудови графа, після чого модуль DijkstraAlgorithm розраховує найкоротші маршрути на основі завантаженої симуляції. Результати обчислень повертаються у форму, де користувач може переглянути їх у зручному форматі.

На фінальному етапі система відображає всі результати симуляції, що дозволяє користувачу проаналізувати їх або використовувати для подальших дій.

Цей процес забезпечує швидкий доступ до раніше збережених симуляцій і спрощує аналіз уже згенерованих даних, надаючи зручний функціонал для підтримки прийняття рішень.

2.4 Архітектура програмного продукту

Архітектура програмного продукту є ключовим аспектом його розробки, що визначає загальну структуру системи, взаємодію між її компонентами та ефективність виконання поставлених задач [29]. Для системи підтримки прийняття рішень у сфері логістики було обрано трьохрівневу архітектуру, яка забезпечує оптимальний розподіл функціональності між рівнями. Цей підхід дозволяє відокремити логіку користувацького інтерфейсу, бізнес-логіку та роботу з даними, що полегшує масштабування системи та спрощує її підтримку.

Трьохрівнева архітектура є раціональним вибором для даного програмного продукту, оскільки вона сприяє підвищенню продуктивності та надійності завдяки чіткому поділу обов'язків між рівнями. Логіка бізнес-процесів реалізується на окремому рівні, що дозволяє спростити зміни в бізнес-функціях без впливу на інші компоненти. Робота з базами даних виділена в окремий рівень, що забезпечує безпеку та узгодженість даних, а також спрощує інтеграцію з різними джерелами інформації. Крім того, така архітектура є стандартною для корпоративних систем, що робить її легкою для розуміння і впровадження, враховуючи наявність перевірених підходів і технологій.

На рис. 2.8 представлено діаграму класів рівня даних системи, яка моделює структуру та взаємодію класів, відповідальних за управління даними в програмному продукті. Діаграма демонструє основні класи, їхні атрибути, методи та взаємозв'язки, що забезпечують ефективну організацію та обробку інформації у системі. Кожен клас має свою функціональність і спеціалізацію, що дозволяє розподілити обов’язки між компонентами.

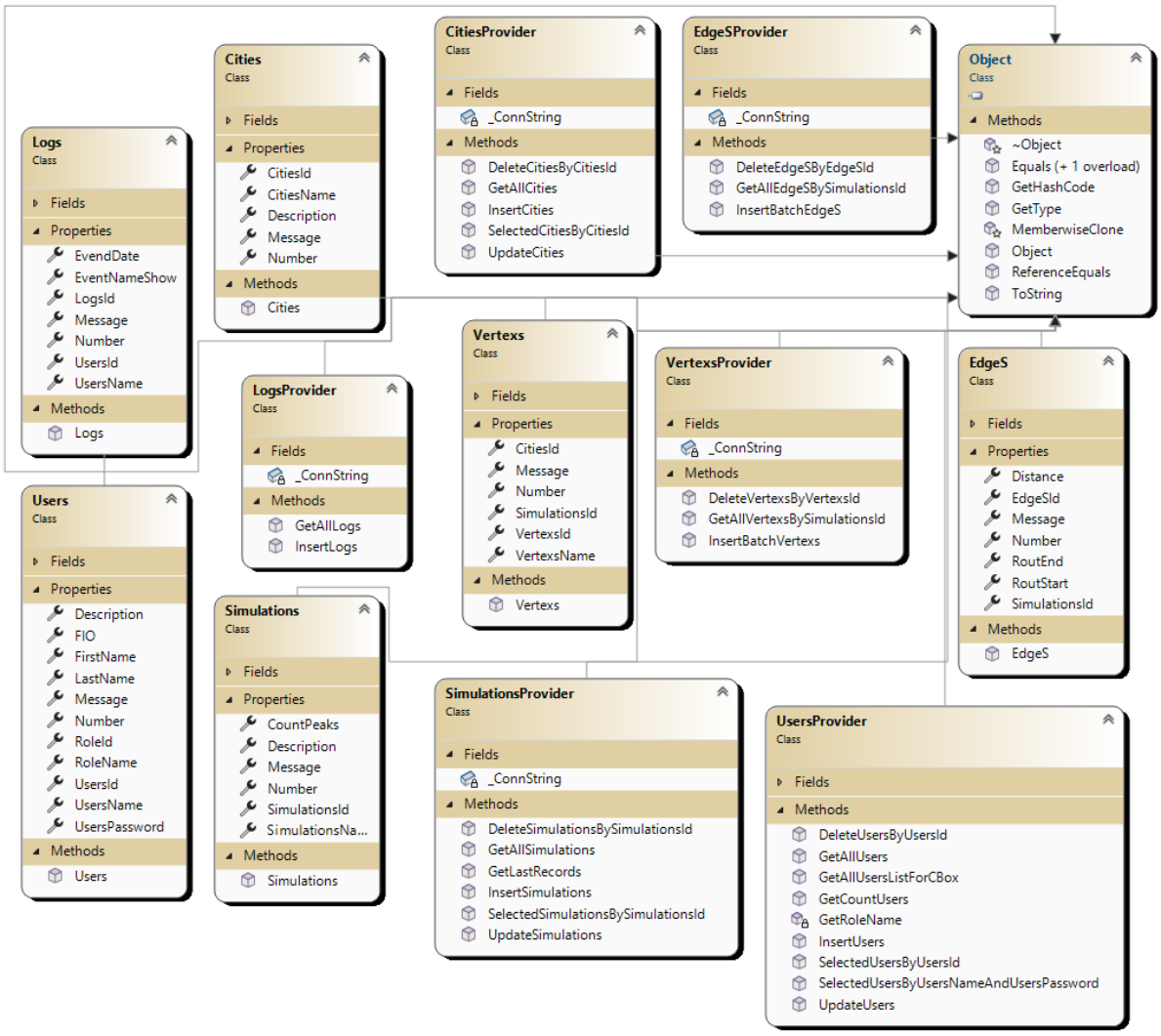


Рис. 2.8 Діаграма класів рівня даних системи

Основу даного рівня складають наступні класи:

– клас CitiesProvider, який відповідає за управління даними про міста. Він містить методи для отримання, оновлення, видалення та вибору даних про міста, забезпечуючи централізований доступ до цієї інформації;

– клас EdgesProvider, призначений для роботи із зв’язками між містами (ребрами графу). Він дозволяє додавати, видаляти та отримувати зв’язки, що використовуються для моделювання маршрутів у системі;

– клас LogsProvider, який забезпечує запис і отримання даних про події у системі. Це дозволяє вести журнал дій користувачів та адміністраторів, підвищуючи прозорість і контроль;

– клас SimulationsProvider, що відповідає за управління даними про проведені симуляції маршрутів. Він містить методи для додавання, оновлення, видалення та отримання результатів моделювання, що зберігаються у базі даних;

– клас UsersProvider, який займається управлінням обліковими записами користувачів. Він дозволяє додавати нових користувачів, редагувати їхні дані, видаляти записи та отримувати списки користувачів;

– клас VertexProvider, призначений для роботи з вершинами графу (містами). Він підтримує методи для додавання, оновлення та отримання даних про вершини, необхідні для коректної роботи графових алгоритмів.

Кожен із перелічених класів реалізує доступ до даних через окремий набір методів, що забезпечує модульність, легкість у підтримці та масштабуванні системи. Завдяки чітко визначеній структурі взаємодії між класами досягається узгодженість даних і висока продуктивність роботи рівня даних.

У системах підтримки прийняття рішень ефективна організація користувацького інтерфейсу є одним із найважливіших аспектів розробки. На рис. 2.9 представлено діаграму класів рівня користувацького інтерфейсу, яка відображає структуру і взаємодію форм, що забезпечують взаємодію користувачів із системою. Ці класи відповідають за візуалізацію даних, обробку дій користувачів і передачу інформації до бізнес-логіки та рівня даних.

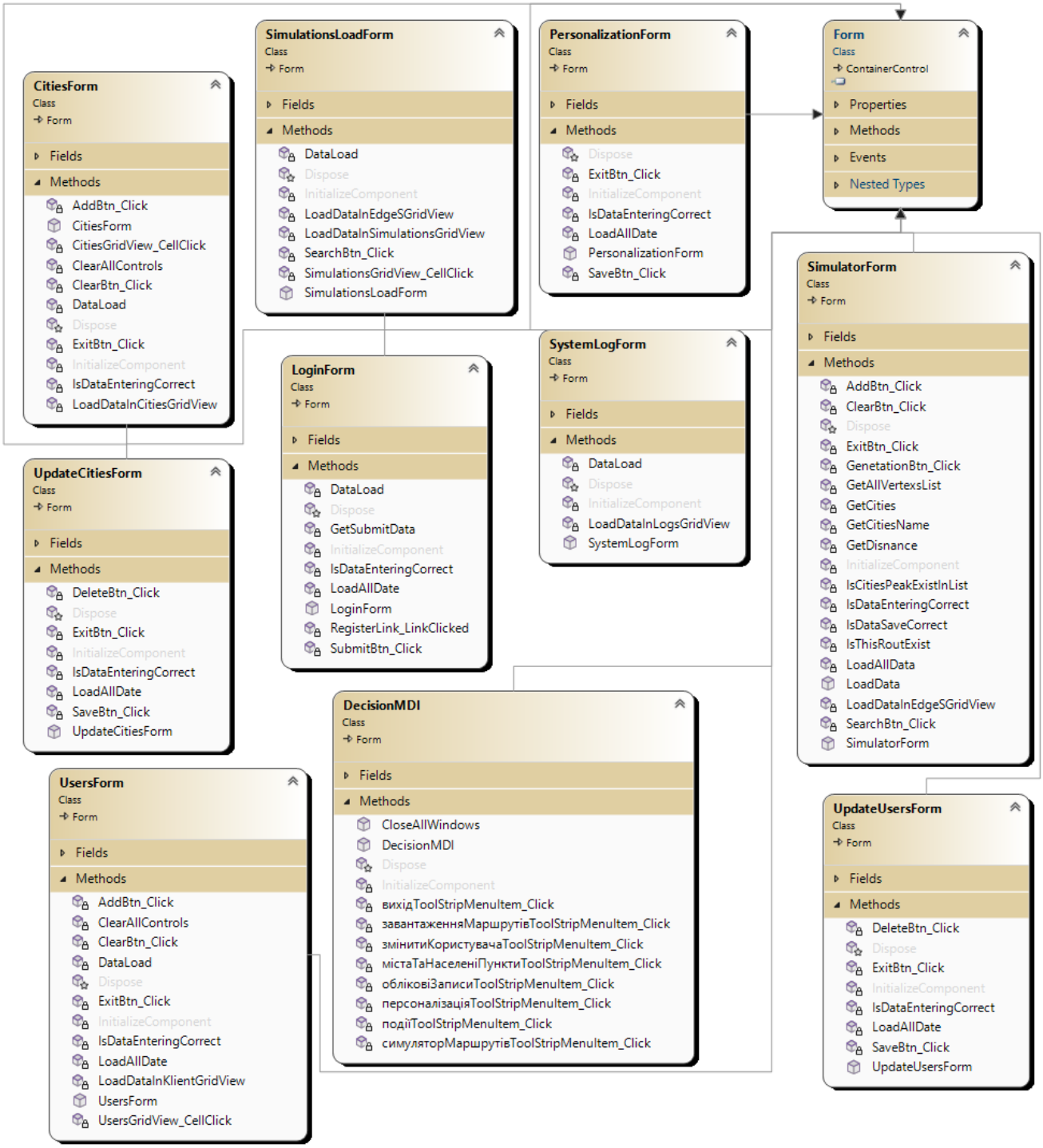


Рис. 2.9 Діаграма класів рівня користувацького інтерфейсу

Діаграма складається з таких ключових класів:

– клас CitiesForm – відповідає за відображення та управління даними про міста. Ця форма надає користувачам можливість переглядати список міст, додавати нові записи, очищувати поля введення та виконувати інші дії;

– клас UpdateCitiesForm – забезпечує редагування даних про міста. Він включає методи для збереження змін, видалення записів та перевірки введених даних;

– клас UsersForm – відповідає за управління обліковими записами користувачів. Форма дозволяє переглядати список користувачів, додавати нові записи та редагувати наявні дані;

– клас LoginForm – реалізує процес авторизації користувачів. Він забезпечує перевірку коректності введених даних, передачу їх до бізнес-логіки для верифікації та обробку можливих помилок;

– клас SimulatorForm – підтримує моделювання маршрутів. Цей клас дозволяє користувачеві вибирати міста, генерувати графи, запускати алгоритм пошуку оптимальних маршрутів і зберігати результати;

– клас SystemLogForm – відображає журнал системних подій, дозволяючи користувачеві переглядати записи про дії в системі для моніторингу та аналізу;

– клас SimulationsLoadForm – забезпечує роботу із завантаженими симуляціями. Він дозволяє користувачам переглядати збережені симуляції та обирати їх для подальшого аналізу;

– клас DecisionMDI – реалізує контейнер для організації головного меню та управління іншими формами. Це дозволяє забезпечити зручну навігацію між функціональними модулями системи;

– клас UpdateUsersForm – використовується для редагування даних користувачів. Він надає можливість зберігати зміни, видаляти записи та перевіряти коректність введеної інформації;

– клас PersonalizationForm – відповідає за налаштування персоналізованих параметрів інтерфейсу користувача, забезпечуючи зручність роботи із системою.

Діаграма класів рівня користувацького інтерфейсу демонструє чітке розділення функціональних модулів, кожен із яких відповідає за виконання певного набору задач. Така структура забезпечує зручність використання, полегшує підтримку системи та сприяє розширюваності її функціоналу.

У системах підтримки прийняття рішень рівень бізнес-логіки відіграє ключову роль, забезпечуючи виконання основних операцій і реалізацію бізнес-процесів. На рис. 2.10 представлено діаграму класів рівня бізнес-логіки, яка моделює структуру класів, відповідальних за обробку даних, алгоритми та взаємодію між різними компонентами системи. Ця діаграма відображає логіку системи, яка забезпечує її функціонування відповідно до визначених вимог.

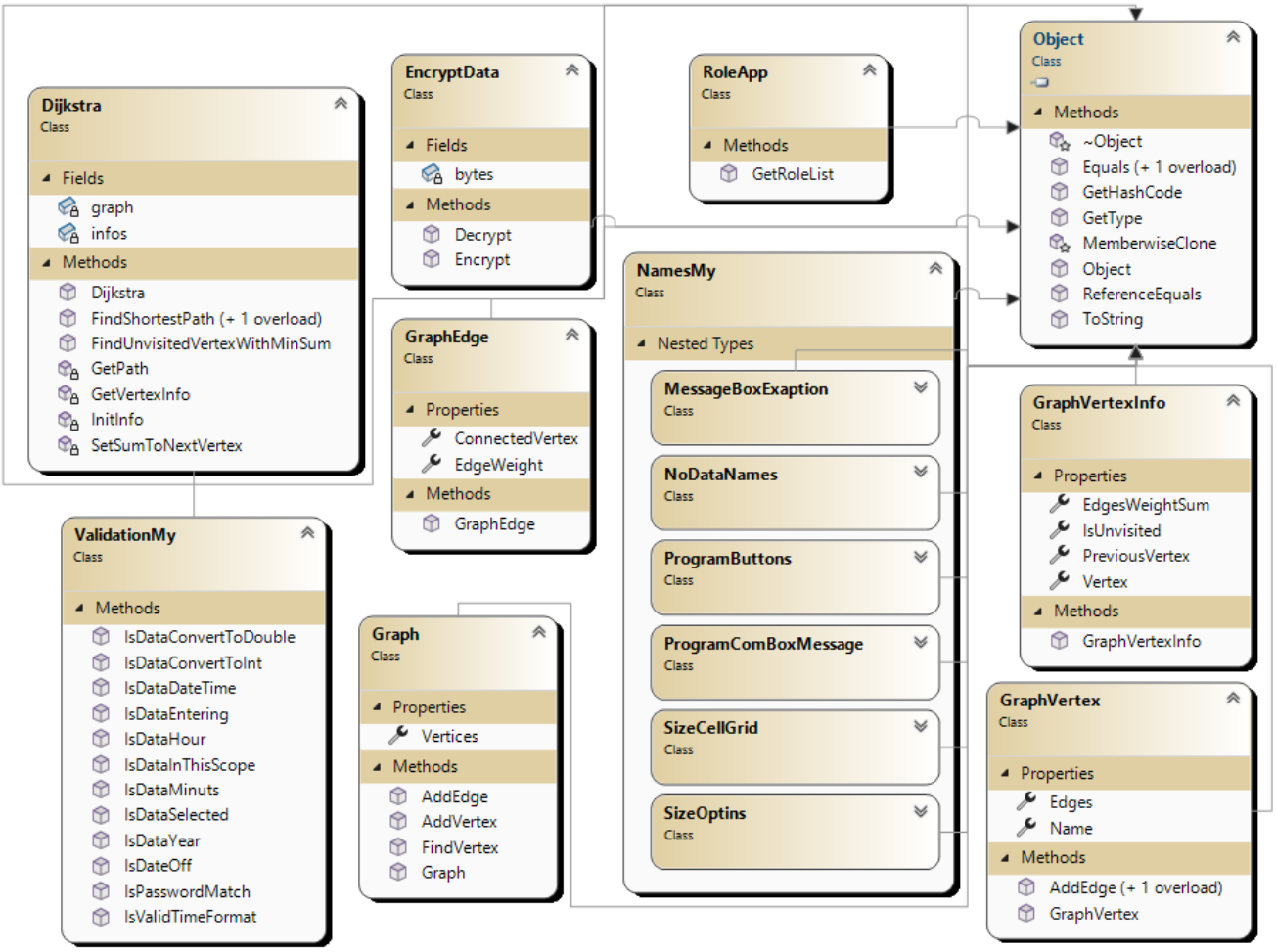


Рис. 2.10 Діаграма класів рівня бізнес-логіки

Діаграма включає такі основні класи:

– клас Dijkstra – реалізує алгоритм пошуку найкоротших шляхів. Він містить методи для знаходження оптимальних маршрутів між вершинами графу, отримання інформації про вершини та підготовки даних для моделювання;

– клас ValidationMy – відповідає за перевірку введених даних. Він містить методи для перевірки коректності числових, часових і текстових значень, що гарантує точність і узгодженість даних, які обробляються системою;

– клас Graph – забезпечує роботу з графами, які використовуються для моделювання маршрутів. Він містить властивості для зберігання вершин і методи для додавання зв’язків, пошуку вершин та обробки графів;

– кас GraphEdge – представляє зв’язки (ребра) між вершинами графу. Він містить властивості для зберігання ваги ребра та пов’язаної вершини, що використовується при обчисленні маршрутів;

– клас GraphVertex – відповідає за вершини графу (міста). Він включає властивості для зберігання даних про вершину та методи для додавання зв’язків між вершинами;

– клас GraphVertexInfo – містить інформацію про стан вершини під час виконання алгоритму, включаючи суму ваг ребер, статус відвідування та попередню вершину;

– клас EncryptData – забезпечує шифрування та розшифрування даних, що гарантує безпеку інформації під час її зберігання та передачі;

– клас RoleApp – містить методи для управління ролями користувачів у системі, що дозволяє визначати доступ до різних функцій;

– клас NamesMy – містить підкласи для роботи з назвами, повідомленнями, кнопками та іншими елементами інтерфейсу, що забезпечує узгодженість між бізнес-логікою та інтерфейсом.

Діаграма рівня бізнес-логіки демонструє чіткий розподіл відповідальностей між класами, що сприяє легкості підтримки, модифікації та розширення системи. Цей рівень забезпечує обробку даних і реалізацію алгоритмів, необхідних для ефективної роботи всієї системи.

2.5 Проєктування бази даних

Проєктування бази даних є одним із ключових етапів розробки програмного забезпечення, який забезпечує структуроване зберігання та обробку даних [30]. Для системи підтримки прийняття рішень у сфері логістики важливою є організація даних, пов’язаних із містами, маршрутами, користувачами та

симуляціями. База даних повинна забезпечувати цілісність, продуктивність і доступність даних, необхідних для роботи всіх компонентів системи.

Однією з основних таблиць у базі даних є таблиця «Cities», яка призначена для зберігання інформації про міста та населені пункти, включаючи їхні назви, описи та унікальні ідентифікатори (табл. 2.5). Ця таблиця забезпечує централізоване управління даними про географічні об’єкти, які використовуються під час моделювання маршрутів.

Таблиця 2.5

Атрибути сутності «Cities»

№	Найменування	Тип даних	Призначення
1	CitiesId	INT	Ідентифікатор міста або населеного пункту.
2	CitiesName	NVARCHAR(250)	Назва міста або населеного пункту.
3	Description	NVARCHAR(MAX)	Додаткова інформація про місто.

Таблиця «EdgeS» призначена для зберігання даних про зв’язки між містами, включаючи початкову і кінцеву точки маршруту, відстань між ними та приналежність до конкретної симуляції (табл. 2.6). Вона використовується для побудови графів, що є основою моделювання маршрутів у системі.

Таблиця 2.6

Атрибути сутності «EdgeS»

№	Найменування	Тип даних	Призначення
1	EdgeSId	INT	Унікальний ідентифікатор зв’язку між двома містами (ребро графу).
2	RoutStart	NVARCHAR(250)	Назва початкового пункту маршруту.
3	RoutEnd	NVARCHAR(250)	Назва кінцевого пункту маршруту.
4	Distance	FLOAT	Відстань між початковим і кінцевим пунктами.
5	SimulationsId	INT	Ідентифікатор симуляції, до якої належить цей маршрут.

Таблиця «Logs» призначена для ведення журналу подій у системі. Вона містить інформацію про користувачів, які виконували дії, назви цих дій та час їх

виконання(табл. 2.7). Ця таблиця використовується для моніторингу активності користувачів і забезпечення прозорості роботи системи.

Таблиця 2.7

Атрибути сутності «Logs»

№	Найменування	Тип даних	Призначення
1	LogsId	INT	Унікальний ідентифікатор запису журналу подій.
2	UsersId	INT	Ідентифікатор користувача, що виконав подію.
3	EventNameShow	NVARCHAR(250)	Назва події, що відображає її зміст.
4	EventDate	DATETIME	Дата та час, коли відбулася подія.

Таблиця «Users» зберігає інформацію про користувачів системи, включаючи особисті дані, облікову інформацію (логін і пароль) та роль, що визначає рівень доступу(табл. 2.8). Ця таблиця забезпечує управління доступом до системи та персоналізацію функціоналу.

Таблиця 2.8

Атрибути сутності «Users»

№	Найменування	Тип даних	Призначення
1	UsersId	INT	Унікальний ідентифікатор користувача.
2	FirstName	NVARCHAR(250)	Ім’я користувача.
3	LastName	NVARCHAR(250)	Прізвище користувача.
4	UsersName	NVARCHAR(250)	Унікальне ім’я користувача для входу в систему.
5	UsersPassword	NVARCHAR(250)	Пароль користувача для доступу до системи.
6	RoleId	INT	Ідентифікатор ролі користувача, який визначає його права доступу.
7	Description	NVARCHAR(MAX)	Додаткова інформація або коментарі щодо користувача.

Таблиця «Vertexs» призначена для зберігання інформації про вершини графу, які використовуються в симуляціях маршрутів(табл. 2.9). Вона містить назви міст або населених пунктів, їх унікальні ідентифікатори та прив’язку до конкретних симуляцій. Ця таблиця є основою для побудови графів і розрахунку оптимальних маршрутів у системі.

Таблиця 2.9

Атрибути сутності «Vertexs»

№	Найменування	Тип даних	Призначення
1	VertexsId	INT	Унікальний ідентифікатор вершини графу.
2	VertexName	NVARCHAR(250)	Назва вершини (міста або населеного пункту).
3	SimulationsId	INT	Ідентифікатор симуляції, до якої належить вершина.

Таблиця «Simulations» призначена для зберігання даних про симуляції маршрутів(табл. 2.10). Вона включає назву, опис, кількість міст у моделюванні та унікальний ідентифікатор.

Таблиця 2.10

Атрибути сутності «Simulations»

№	Найменування	Тип даних	Призначення
1	SimulationsId	INT	Унікальний ідентифікатор симуляції.
2	SimulationsName	NVARCHAR(250)	Назва симуляції, яка визначає її зміст або тип.
3	Description	NVARCHAR(MAX)	Додаткова інформація чи опис симуляції.
4	CountPeaks	INT	Кількість вершин (міст), що використовуються у симуляції.

Під час створення бази даних для інформаційної системи було розроблено ER-діаграму, яка демонструє всі сутності та зв'язки між ними (рис. 2.11).

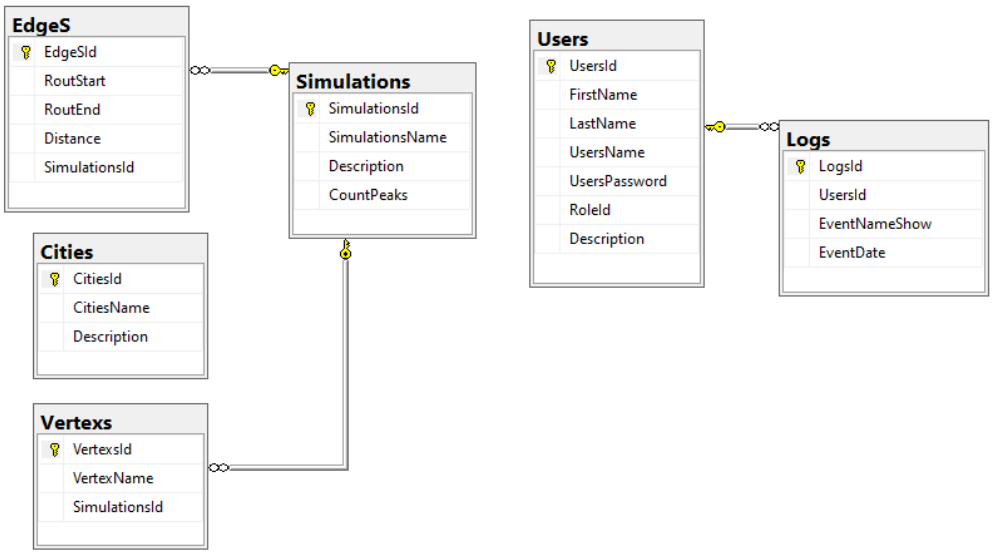


Рис. 2.11 Діаграма бази даних

ER-діаграма відіграє ключову роль у проектуванні бази даних, оскільки вона відображає логічну структуру даних і їхні взаємозв’язки. Цей інструмент дозволяє розробникам зрозуміти, як мають взаємодіяти сутності, що забезпечує правильне зберігання й обробку інформації.

2.6 Висновок до розділу

У цьому розділі проведено аналіз технологій і архітектурних рішень для реалізації системи вибору оптимальних маршрутів. Розглянуто мови Python, Java, C# та середовища Visual Studio 2022, Visual Studio Code і Rider. Обрано мову C# і середовище Visual Studio 2022, які найбільше відповідають вимогам системи та забезпечують ефективність розробки.

Визначено функціональні та нефункціональні вимоги до системи, описано акторів, їхні цілі, а також варіанти використання системи, включаючи побудову відповідних діаграм використання для ролей адміністратора та користувача. Для моделювання ключових процесів було створено діаграми активності, які деталізують сценарії авторизації користувачів, додавання та редагування інформації. Також побудовано діаграми послідовності, що описують процес симуляції маршрутів і управління завантаженими симуляціями.

У межах архітектурного проектування розроблено трьохрівневу архітектуру системи, яка забезпечує розподіл відповідальностей між рівнями, спрощує підтримку та масштабування. Окрім того, проведено проєктування бази даних, розроблено ER-діаграму та описано основні таблиці, що забезпечують зберігання та обробку даних системи.

Результати цього розділу створюють необхідну основу для реалізації системи, описаної у наступному розділі. Визначені вимоги, моделі даних, архітектурні рішення та детальні описи процесів дозволяють перейти до етапу реалізації функціоналу системи та проведення її тестування.

РОЗДІЛ 3. РЕАЛІЗАЦІЯ ТА ТЕСТУВАННЯ СИСТЕМИ

3.1 Розробка програмних компонентів

Розробка програмних компонентів для системи підтримки прийняття рішень у сфері логістики є ключовим етапом створення ефективного програмного забезпечення. Основна задача полягає у забезпеченні можливості обчислення оптимальних маршрутів доставки на основі введених даних про точки доставки, транспортні засоби та дорожню мережу. Для досягнення цієї мети було реалізовано алгоритм пошуку найкоротших маршрутів, який використовує графову модель транспортної мережі.

Одним із базових компонентів системи є клас Dijkstra, що реалізує алгоритм Дейкстри (рис. 3.1). Цей клас використовується для роботи з графами, де вершини представляють точки маршруту, а ребра – шляхи між ними із зазначеними вагами (відстанню між населеними пунктами). У конструкції класу передбачено обробку графа, ініціалізацію даних та виявлення можливих некоректностей.

```
public class Dijkstra {
  Graph graph;
  List<GraphVertexInfo> infos;
  /// <summary>
  /// Конструктор
  /// </summary>
  /// <param name="graph">Граф</param>
  2 references
  public Dijkstra(Graph graph) {
    this.graph = graph;
  }
  /// <summary>
  /// Ініціалізація інформації
  /// </summary>
  1 reference
  void InitInfo() {
    infos = new List<GraphVertexInfo>();
    foreach (var v in graph.Vertices) {
      if (v == null) {
        throw new InvalidOperationException("Граф містить null-вершини.");
      }
      infos.Add(new GraphVertexInfo(v));
    }
  }
```

Рис. 3.1 Діаграма класів рівня бізнес-логіки

Клас має поле graph, яке відповідає за збереження графа з вершинами та ребрами, а також поле infos – список об'єктів типу GraphVertexInfo, які містять додаткову інформацію про вершини. Конструктор класу приймає граф як параметр і зберігає його для подальшого використання. Основним методом у цьому прикладі є InitInfo, який відповідає за підготовку списку інформаційних об'єктів для кожної вершини графа. У процесі ініціалізації перевіряється наявність некоректних (null) вершин, і в разі їх виявлення метод викидає виняток InvalidOperationException із відповідним повідомленням. Це дозволяє забезпечити надійну роботу алгоритму навіть за умови потенційно некоректних вхідних даних.

Запропонована реалізація є основою для моделювання транспортної мережі в системі підтримки прийняття рішень. Завдяки гнучкості та структурованому підходу, цей компонент може бути легко інтегрований у складніші програмні рішення, що враховують додаткові фактори, такі як обмеження за часом, вагою вантажу чи витратами на доставку.

Код на рис. 3.2 реалізує метод для отримання інформації про конкретний населений пункт (місто), представлене у графі транспортної мережі. Метод «GetVertexInfo» приймає як вхідний параметр об’єкт типу GraphVertex, що відповідає одному з населених пунктів. Усередині методу виконується пошук об'єкта типу GraphVertexInfo, який зберігає детальну інформацію про заданий населений пункт.

```
GraphVertexInfo GetVertexInfo(GraphVertex v) {
    var info = infos.FirstOrDefault(i => i.Vertex == v);
    if (info == null) {
        throw new InvalidOperationException($"Інформація для вершини {v.Name} не знайдена.");
    }
    return info;
}
```

Рис. 3.2 Код методу «GetVertexInfo»

Пошук здійснюється у списку infos, що містить інформацію про всі населені пункти графа. Для пошуку використовується функція FirstOrDefault, яка знаходить перший елемент у списку, що відповідає умові, або повертає null, якщо відповідного елемента не знайдено. Умова полягає в порівнянні переданого об'єкта GraphVertex із властивістю Vertex кожного елемента списку.

Якщо відповідна інформація знайдена, вона повертається у вигляді об’єкта GraphVertexInfo. Якщо ж інформацію про населений пункт знайти не вдалося, метод викидає виняток InvalidOperationException. У повідомленні винятку зазначається ім’я населеного пункту (v.Name), для якого не вдалося знайти дані. Такий підхід забезпечує надійну перевірку наявності необхідної інформації та допомагає ідентифікувати потенційні проблеми з графом або його даними.

На рис. 3.3 наведено метод, який реалізує функціонал для пошуку найближчого до обробки населеного пункту серед тих, які ще не відвідані в процесі роботи алгоритму. Його мета – знайти та повернути об'єкт типу GraphVertexInfo, що містить інформацію про населений пункт із мінімальним значенням ваги маршрутів (сума ваг від початкового пункту до поточного).

```
public GraphVertexInfo FindUnvisitedVertexWithMinSum() {
    var minValue = int.MaxValue;
    GraphVertexInfo minVertexInfo = null;
    foreach (var i in infos) {
        if (i.IsUnvisited && i.EdgesWeightSum < minValue) {
            minVertexInfo = i;
            minValue = i.EdgesWeightSum;
        }
    }

    return minVertexInfo;
}
```

Рис. 3.3 Код методу «FindUnvisitedVertexWithMinSum»

На початку методу ініціалізуються змінні для зберігання мінімального значення ваги (minValue) та посилання на відповідний населений пункт (minVertexInfo). Змінній minValue задається початкове значення int.MaxValue, щоб порівняння на першому кроці гарантовано оновило її. Далі проходить ітерація по всіх об'єктах у списку infos, який містить інформацію про всі міста графа. Для кожного пункту перевіряється, чи він ще не був відвіданий (властивість IsUnvisited) і чи його сумарна вага маршруту (EdgesWeightSum) є меншою за поточне мінімальне значення. Якщо ці умови виконуються, змінна minVertexInfo оновлюється, отримуючи посилання на цей пункт, а minValue приймає його значення ваги.

Після завершення ітерації метод повертає об'єкт minVertexInfo, який містить інформацію про населений пункт із найменшою вагою серед усіх невідвіданих. Якщо всі пункти вже оброблені, метод повертає null. Такий підхід дозволяє алгоритму Дейкстри коректно обирати наступний населений пункт для подальшого обчислення найкоротшого маршруту.

На рис. 3.4 наведено код методу, що виконує пошук найкоротшого шляху між двома населеними пунктами, представленими об'єктами графа. На першому етапі здійснюється ініціалізація всіх даних за допомогою методу InitInfo, який підготовлює інформацію про кожне місто в графі. Потім визначається початковий населений пункт за допомогою функції GetVertexInfo. Якщо стартовий пункт відсутній у графі, викидається виняток із повідомленням, що пояснює проблему.

```
public string FindShortestPath(GraphVertex startVertex, GraphVertex finishVertex) {
    InitInfo();
    var first = GetVertexInfo(startVertex);
    if (first == null) {
        throw new ArgumentException($"Вершина {startVertex.Name}" +
            $" не знайдена у графі.");
    }
    first.EdgesWeightSum = 0;
    while (true) {
        var current = FindUnvisitedVertexWithMinSum();
        if (current == null) {
            break;
        }
        SetSumToNextVertex(current);
    }
    // Перевірка доступності шляху
    var finishInfo = GetVertexInfo(finishVertex);
    if (finishInfo == null || finishInfo.EdgesWeightSum == int.MaxValue) {
        throw new InvalidOperationException($"Неможливо знайти шлях між '" +
            $"{startVertex.Name}' та '{finishVertex.Name}'.");
    }
    return GetPath(startVertex, finishVertex);
}
```

Рис. 3.4 Пошук найкоротшого шляху між двома населеними пунктами

Після успішної ініціалізації початковий пункт отримує початкове значення суми ваг – нуль, оскільки відстань до самого себе дорівнює нулю. Далі метод входить у цикл, у якому шукається наступне місто, що ще не відвідано, але має мінімальну суму ваги маршрутів. Це здійснюється за допомогою методу

FindUnvisitedVertexWithMinSum. Якщо доступних для обробки пунктів не залишилося, цикл завершується.

Для кожного знайденого міста обчислюється нова сума ваг для його сусідів за допомогою функції SetSumToNextVertex. Це оновлює інформацію про маршрути та забезпечує коректну роботу алгоритму пошуку найкоротшого шляху. Після завершення циклу перевіряється, чи можливо досягти кінцевий населений пункт. Якщо інформація про нього недоступна або сума ваг залишилася максимальною (тобто недосяжною), метод викидає виняток із повідомленням, що шлях між вказаними пунктами знайти неможливо. Якщо ж маршрут існує, результат обчислюється за допомогою методу GetPath, який повертає послідовність міст, що утворюють найкоротший шлях.

Метод на рис. 3.5 виконує ключову операцію алгоритму пошуку найкоротшого шляху – оновлення ваг для сусідніх населених пунктів, які з'єднані з поточним. Передусім відзначається, що поточне місто вважається відвіданим, змінюючи відповідне поле IsUnvisited на значення false.

```
void SetSumToNextVertex(GraphVertexInfo info) {
  info.IsUnvisited = false;
  foreach (var e in info.Vertex.Edges) {
    var nextInfo = GetVertexInfo(e.ConnectedVertex);
    var sum = info.EdgesWeightSum + e.EdgeWeight;
    if (sum < nextInfo.EdgesWeightSum) {
      nextInfo.EdgesWeightSum = sum;
      nextInfo.PreviousVertex = info.Vertex;
    }
  }
}
```

Рис. 3.5 Код методу «SetSumToNextVertex»

Даний метод проходить по всіх маршрутах (ребрах), які виходять із поточного міста. Для кожного маршруту визначається сусіднє місто через об'єкт ConnectedVertex. Потім отримується інформація про це місто за допомогою методу GetVertexInfo. На основі поточної суми ваг маршруту (EdgesWeightSum) та ваги цього маршруту (EdgeWeight) обчислюється нова сума ваг. Якщо нова сума ваг виявляється меншою за поточну відому суму для сусіднього міста (EdgesWeightSum), відбувається її оновлення. Окрім того, у властивість

PreviousVertex записується посилання на поточне місто, що позначає попереднє місто в оптимальному маршруті. Таким чином, метод поступово "протягує" найкоротші шляхи до всіх доступних населених пунктів графа.

Процес оновлення гарантує, що для кожного міста зберігається найоптимальніший маршрут, а також створюється ланцюг попередніх міст, який згодом використовується для відтворення найкоротшого шляху. Такий підхід дозволяє ефективно враховувати всі можливі варіанти шляхів у графі та обирати найкращі з них.

На рис. 3.6 наведено метод, який відповідає за побудову маршруту між двома населеними пунктами, використовуючи дані про попередні міста, які зберігаються в результаті роботи алгоритму пошуку найкоротшого шляху. Початкова перевірка гарантує, що і стартове, і кінцеве місто задані. У разі їхньої відсутності викидається виняток ArgumentNullException, оскільки без них побудова маршруту неможлива.

```
string GetPath(GraphVertex startVertex, GraphVertex endVertex) {
  if (startVertex == null || endVertex == null) {
    throw new ArgumentNullException("Стартова або кінцева вершина не задана.");
  }
  var path = new List<string>();
  var currentVertex = endVertex;
  while (currentVertex != null) {
    path.Add(currentVertex.Name);
    var vertexInfo = GetVertexInfo(currentVertex);
    if (vertexInfo == null || vertexInfo.PreviousVertex == null) {
      break; // Відсутній шлях до стартової вершини
    }
    currentVertex = vertexInfo.PreviousVertex;
  }
  if (!path.Contains(startVertex.Name)) {
    throw new InvalidOperationException("Не вдалося побудувати шлях:" +
      " вершини не з'єднані.");
  }
  path.Reverse();
  return string.Join(" -> ", path);
}
```

Рис. 3.6 Код методу «GetPath»

Для зберігання маршруту створюється список path, куди послідовно додаються назви міст. Ініціалізація починається з кінцевого міста, після чого алгоритм поступово відстежує попередні міста через поле PreviousVertex,

зберігаючи їх у списку. На кожному кроці назва поточного міста додається до маршруту. Якщо в процесі побудови маршруту виявляється, що для якогось міста немає інформації про попереднє, це означає, що шлях від стартового до кінцевого пункту не існує, і цикл завершується. На фінальному етапі виконується перевірка, чи містить маршрут стартовий пункт. Якщо його немає, це свідчить про те, що населені пункти не пов'язані, і метод викидає виняток InvalidOperationException.

У разі успішного завершення список із назвами міст перевертається, щоб відобразити послідовність маршруту від стартового до кінцевого пункту. Метод повертає маршрут у вигляді строки, де назви міст з’єднані символом "->", що візуально відображає послідовність руху. Такий підхід забезпечує зручну інтерпретацію маршруту, яку легко сприйняти як для користувача, так і для інтеграції з іншими компонентами системи.

Після цього проведено реалізацію методів для роботи з базою даних, це забезпечує інтеграцію програмного коду з джерелом даних, необхідних для аналізу та прийняття рішень. У даній системі методи для роботи з базою даних зосереджені на збереженні, оновленні та отриманні інформації про маршрути, населені пункти та інші об'єкти транспортної мережі. Особливий акцент зроблено на ефективності виконання операцій, що є критично важливим у випадках, коли система обробляє великі обсяги даних.

Метод, який наведено на рис. 3.7, призначений для пакетного додавання даних про маршрути між населеними пунктами до бази даних. Він дозволяє одночасно вставляти кілька записів, що представляють маршрути з такими параметрами, як початковий та кінцевий пункти, відстань між ними та ідентифікатор моделювання.

```
public void InsertBatchEdgeS(List<EdgeS> EdgeS) {
  string SqlString = "INSERT INTO EdgeS (RoutStart, RoutEnd, Distance, SimulationsId) Values(?, ?, ?, ?)";
  using (OleDbConnection conn = new OleDbConnection(_ConnString)) {
    using (OleDbCommand cmd = new OleDbCommand(SqlString, conn)) {
      cmd.CommandType = CommandType.Text;
      conn.Open();
      for (int i = 0; i < EdgeS.Count; i++) {
        cmd.Parameters.AddWithValue("RoutStart", EdgeS[i].RoutStart);
        cmd.Parameters.AddWithValue("RoutEnd", EdgeS[i].RoutEnd);
        cmd.Parameters.AddWithValue("Distance", EdgeS[i].Distance);
        cmd.Parameters.AddWithValue("SimulationsId", EdgeS[i].SimulationsId);
        cmd.ExecuteNonQuery();
        while (cmd.Parameters.Count > 0) {
          cmd.Parameters.RemoveAt(0);
        }
      }
      conn.Close();
    }
  }
}
```

Рис. 3.7 Код методу «InsertBatchEdgeS»

Реалізація починається з підготовки SQL-запиту, який визначає структуру вставки даних у таблицю EdgeS. У запиті використовуються параметри для кожного з полів: початкового пункту (RoutStart), кінцевого пункту (RoutEnd), відстані (Distance) та ідентифікатора моделювання (SimulationsId). Це забезпечує динамічну передачу даних під час виконання запиту.

Метод обробляє список маршрутів, переданий як параметр EdgeS. Для кожного маршруту з цього списку додаються параметри до команди SQL, які відповідають значенням початкового пункту, кінцевого пункту, відстані та ідентифікатора моделювання. Після встановлення параметрів команда виконується за допомогою ExecuteNonQuery, яка додає запис до бази даних.

Метод на рис. 3.8 забезпечує отримання списку населених пунктів, які належать до певної симуляції, визначеної за унікальним ідентифікатором. Він дозволяє динамічно завантажувати дані про всі міста, пов’язані з конкретним моделюванням, що є важливим для аналізу маршрутів у межах заданого сценарію.

```
public List<Vertexs> GetAllVertexsBySimulationId(int SimulationId) {
  int i = 0;
  string SqlString = "SELECT * FROM Vertexs WHERE SimulationsId=" + SimulationId;
  List<Vertexs> Vertexs = new List<Vertexs>();
  using (OleDbConnection conn = new OleDbConnection(_ConnString)) {
    using (OleDbCommand cmd = new OleDbCommand(SqlString, conn)) {
      conn.Open();
      using (OleDbDataReader reader = cmd.ExecuteReader()) {
        while (reader.Read()) {
          Vertexs oneVertexs = new Vertexs();
          oneVertexs.Number = ++i;
          oneVertexs.VertexsId = Convert.ToInt32(reader["VertexsId"]);
          oneVertexs.VertexsName = reader["VertexName"].ToString();
          oneVertexs.SimulationsId = Convert.ToInt32(reader["SimulationsId"]);
          Vertexs.Add(oneVertexs);
        }
      }
      conn.Close();
    }
  }
  return Vertexs;
}
```

Рис. 3.8 Код методу «GetAllVertexsBySimulationId»

Метод починається з формування SQL-запиту, який вибирає всі записи з таблиці Vertexs, де ідентифікатор симуляції збігається з переданим параметром SimulationsId. Для зберігання результатів створюється список об'єктів Vertexs, у який будуть додаватися всі знайдені записи. Кожен запис у таблиці перетворюється на об'єкт типу Vertexs. Для цього зчитуються поля з результату запиту: ідентифікатор населеного пункту (VertexsId), його назва (VertexName) і відповідний ідентифікатор симуляції (SimulationsId). Крім того, кожному пункту присвоюється порядковий номер, який інкрементується під час додавання нового елемента до списку.

Отримані дані додаються до списку Vertexs, який наприкінці методу повертається як результат. Закриття з’єднання після завершення операції гарантує звільнення ресурсів та коректну роботу з базою даних. Цей метод надає зручний спосіб динамічного завантаження населених пунктів, що входять до складу симуляції, забезпечуючи інтеграцію із системою обробки маршрутів.

У процесі розробки системи значну увагу було приділено створенню користувацького інтерфейсу, який забезпечує зручну взаємодію користувача із програмним забезпеченням. Для цього були створені форми та методи, які

дозволяють виконувати ключові операції, пов'язані з побудовою графів, управлінням маршрутами та аналізом транспортної мережі. Особливу увагу приділено подієвому програмуванню, яке відповідає за обробку взаємодій користувача з елементами інтерфейсу, такими як кнопки, випадаючі списки та текстові поля.

На рис. 3.9 наведено код методу, який обробляє подію натискання кнопки, яка відповідає за ініціалізацію процесу генерації даних для графа, що моделює транспортну мережу. Головним завданням є підготовка списків міст і маршрутів для подальшої роботи в системі.

```
private void GenetationBtn_Click(object sender, EventArgs e) {
  if (IsDataEnteringCorrect()) {
    _Graph = new Graph();
    _VertexsList.Clear();
    _EdgeSList.Clear();
    VertexCBox.Items.Clear();
    //Генеруємо міста
    _VertexsList = GetAllVertexsList(Convert.ToInt32(CountPeaksTBox.Text));
    LoadData();
  }
}
```

Рис. 3.9 Код методу «GenetationBtn_Click»

На початку роботи методу перевіряється правильність введених користувачем даних за допомогою функції IsDataEnteringCorrect. Якщо перевірка проходить успішно, створюється новий екземпляр об’єкта графа _Graph, що представляє транспортну мережу. Після цього відбувається очищення пов’язаних з графом даних: список міст _VertexsList, список маршрутів _EdgeSList та випадаючий список міст у графічному інтерфейсі VertexCBox. Це дозволяє почати генерацію з чистого стану, уникнувши потенційних конфліктів із попередніми даними.

Для генерації списку населених пунктів викликається метод GetAllVertexsList, який приймає кількість міст, зазначену користувачем у текстовому полі CountPeaksTBox. Результати генерації записуються в _VertexsList, після чого викликається метод LoadData, який завантажує ці дані в граф і готує їх для подальшої обробки.

Цей метод дозволяє динамічно ініціалізувати нову транспортну мережу на основі заданих параметрів, гарантуючи очищення попередніх даних та підготовку до роботи з оновленими маршрутами й містами. Він є зручним механізмом для інтерактивного налаштування сценаріїв моделювання.

На рис. 3.10 наведено метод, який відповідає за завантаження даних у граф, що використовується для моделювання транспортної мережі. Він працює з містами та маршрутами між ними, додаючи відповідну інформацію до графа, а також оновлює елементи користувацького інтерфейсу.

На початку створюється тимчасовий список маршрутів edgeSList, який буде використовуватися для збереження даних про зв'язки між містами. Паралельно очищується основний список _EdgeSList, щоб уникнути конфліктів із попередньо доданими маршрутами. Метод проходить через список міст _VertexsList, додаючи кожне з них до графа за допомогою методу AddVertex. Кожна назва міста також додається до випадаючого списку VertexCBox у графічному інтерфейсі, що дозволяє користувачу взаємодіяти з даними. Після додавання всіх міст перший пункт у списку встановлюється як вибраний.

Такожреалізовано проходження до створення маршрутів між містами. Для кожного міста випадково генерується кількість маршрутів, які воно буде мати, використовуючи генератор випадкових чисел _GetRandom. Кожен маршрут визначає, до якого іншого міста буде створений зв’язок, а також встановлює випадкову відстань між ними в діапазоні від 10 до 200 одиниць.

```
public void LoadData() {
  List<EdgeS> edgeSList = new List<EdgeS>();
  _EdgeSList.Clear();
  //Додавання вершин
  for (int i = 0; i < _VertexsList.Count; i++) {
    _Graph.AddVertex(_VertexsList[i].VertexsName);
    VertexCBox.Items.Add(_VertexsList[i].VertexsName);
  }
  VertexCBox.SelectedIndex = 0;
  //Додавання ребер
  for (int i = 0; i < _VertexsList.Count; i++) {
    int genRoutCount = _GetRandom.Next(0, _VertexsList.Count - 1) + 1;
    int count = 0;
    while (count < genRoutCount) {
      EdgeS oneEdgeS = new EdgeS();
      int genCitiesId = _GetRandom.Next(0, _VertexsList.Count);
      if (i != genCitiesId) {
        oneEdgeS.RoutStart = _VertexsList[i].VertexsName;
        oneEdgeS.RoutEnd = GetCitiesName(genCitiesId, _VertexsList);
        oneEdgeS.Distance = _GetRandom.Next(1, 20) * 10;
        if (!IsThisRoutExist(oneEdgeS, edgeSList)) {
          edgeSList.Add(oneEdgeS);
          count++;
        }
      }
    }
  }
```

Рис. 3.10 Завантаження даних у граф

Для уникнення дублювання маршрутів використовується перевірка за допомогою методу IsThisRoutExist. Якщо маршрут між двома містами ще не існує, він додається до тимчасового списку edgeSList. У результаті кожне місто отримує унікальні зв’язки з іншими, формуючи транспортну мережу із заданою випадковістю.

Метод забезпечує динамічну генерацію даних для графа, інтегруючи їх із інтерфейсом користувача, та створює умови для подальшого аналізу маршрутів і розрахунків у системі. Завдяки такому підходу кожен запуск може моделювати нову конфігурацію транспортної мережі.

Рис. 3.11 висвітлює код методу, який реалізує обробку події натискання кнопки пошуку, яка запускає алгоритм Дейкстри для визначення найкоротших маршрутів від заданого міста до всіх інших у транспортній мережі. Його головна мета – обчислити та вивести на екран результати розрахунків для подальшого аналізу.

```
private void SearchBtn_Click(object sender, EventArgs e) {
  if (VertexCBox.Text != "") {
    _Dijkstra = new Dijkstra(_Graph);
    ResultTBox.Text = "Найкоротші маршрути від заданого місця, до всіх інших місць: \r\n";
    for (int i = 0; i < Convert.ToInt32(CountPeaksTBox.Text); i++) {
      if (VertexCBox.Text != i.ToString()) {
        var path = _Dijkstra.FindShortestPath(VertexCBox.Text, _VertexsList[i].VertexsName);
        if (path != VertexCBox.Text) {
          ResultTBox.Text += path + "\r\n";
          ResultTBox.Text += "------------------------------\r\n";
        }
      }
    }
  }
}
```

Рис. 3.11 Код методу «SearchBtn_Click»

Метод починає виконання з перевірки, чи вибрано користувачем місто у випадаючому списку VertexCBox. Якщо вибір здійснено, створюється новий екземпляр об'єкта _Dijkstra, який отримує граф _Graph для проведення розрахунків. У текстове поле ResultTBox записується вступний текст, що пояснює, які результати буде відображено: найкоротші маршрути від вибраного міста до всіх інших. Далі метод виконує цикл, у якому проходить через усі міста, кількість яких зазначена у полі CountPeaksTBox. Для кожного міста, відмінного від вибраного, викликається метод FindShortestPath алгоритму Дейкстри, який розраховує маршрут від поточного міста до цільового. Результат у вигляді послідовності населених пунктів додається до текстового поля ResultTBox.

Метод на рис. 3.12 обробляє подію натискання кнопки для збереження симуляції транспортної мережі, забезпечуючи збереження даних про міста, маршрути та відповідну симуляцію в базі даних. Перед початком збереження виконується перевірка коректності введених користувачем даних за допомогою методу IsDataSaveCorrect. У разі успішної перевірки здійснюється послідовне внесення інформації.

```
private void AddBtn_Click(object sender, EventArgs e) {
  if (IsDataSaveCorrect()) {
    LogsProvider logsProvider = new LogsProvider();
    SimulationsProvider simulationsProvider = new SimulationsProvider();
    simulationsProvider.InsertSimulations(SimulationsNameTBox.Text, DescriptionTBox.Text,
      Convert.ToInt32(CountPeaksTBox.Text));
    int lastSimulationsId = simulationsProvider.GetLastRecords();
    for (int i = 0; i < _VertexsList.Count; i++) {
      _VertexsList[i].SimulationsId = lastSimulationsId;
    }
    for (int i = 0; i < _EdgeSList.Count; i++) {
      _EdgeSList[i].SimulationsId = lastSimulationsId;
    }
    _VertexsProvider.InsertBatchVertexs(_VertexsList);
    _EdgeSProvider.InsertBatchEdgeS(_EdgeSList);
    MessageBox.Show("Симуляцію успішно збережено!");
    logsProvider.InsertLogs(LoginForm.CurrentUser.UsersId, "Було збережено симуляцію під назвою: " +
     SimulationsNameTBox.Text, DateTime.Now);
    SimulationsNameTBox.Text = String.Empty;
    DescriptionTBox.Text = String.Empty;
  }
}
```

Рис. 3.12Збереження симуляції транспортної мережі

Створюється екземпляр класу SimulationsProvider, який відповідає за роботу з даними симуляцій. З нього викликається метод InsertSimulations, який додає новий запис про симуляцію, використовуючи дані, введені користувачем у текстові поля SimulationsNameTBox і DescriptionTBox, а також кількість міст із поля CountPeaksTBox. Після цього за допомогою GetLastRecords отримується ідентифікатор щойно доданої симуляції. Цей ідентифікатор призначається всім містам у списку _VertexsList та маршрутам у списку _EdgeSList, щоб вони були пов’язані з новою симуляцією. Потім викликаються методи для пакетного збереження даних: InsertBatchVertexs зберігає всі міста, а InsertBatchEdgeS – всі маршрути, використовуючи відповідні провайдери.

Коли всі дані успішно збережені, користувачу відображається повідомлення про успішне збереження симуляції. Для забезпечення прозорості дій також записується журнал подій за допомогою класу LogsProvider. У журналі фіксується, хто і коли зберіг симуляцію, а також її назва.

Наприкінці методу очищуються текстові поля SimulationsNameTBox і DescriptionTBox, щоб підготувати інтерфейс до введення нових даних. Метод гарантує надійне збереження даних і створює зв’язок між симуляцією, містами та маршрутами, забезпечуючи їхнє подальше використання в системі.

На рис. 3.13 зображено код методу, що забезпечує завантаження даних про симуляції з бази даних і відображення їх у таблиці користувацького інтерфейсу. Його головна мета – синхронізувати дані системи з візуальним відображенням, зберігаючи при цьому контекст попереднього вибору користувача.

```
private void DataLoad() {
  int firstRowIndex = 0;
  if (SimulationsGridView.FirstDisplayedScrollingRowIndex > 0) {
    firstRowIndex = SimulationsGridView.FirstDisplayedScrollingRowIndex;
  }
  try {
    _SimulationsList = _SimulationsProvider.GetAllSimulations();
    LoadDataInSimulationsGridView(_SimulationsList);
    if (_selectedRowIndex == SimulationsGridView.Rows.Count) {
      _selectedRowIndex = SimulationsGridView.Rows.Count - 1;
    }
    if (_selectedRowIndex >= 0) {
      SimulationsGridView.FirstDisplayedScrollingRowIndex = firstRowIndex;
      SimulationsGridView.Rows[_selectedRowIndex].Selected = true;
    }
  } catch { }
}
```

Рис. 3.13 Збереження симуляції транспортної мережі

На початку метод зберігає індекс першого видимого рядка таблиці SimulationsGridView, щоб після оновлення даних відновити попереднє положення прокрутки. Це дозволяє уникнути незручностей для користувача, пов’язаних із перескакуванням таблиці на початок при кожному завантаженні.

Потім викликається метод із провайдера _SimulationsProvider, який отримує список усіх симуляцій із бази даних. Цей список зберігається у змінній _SimulationsList, яка виступає як проміжний контейнер для зручної роботи з даними. Далі список передається методу LoadDataInSimulationsGridView, що відповідає за оновлення вмісту таблиці SimulationsGridView.

Після завантаження даних виконується перевірка поточного індексу вибраного рядка _selectedRowIndex. Якщо індекс виходить за межі кількості рядків у таблиці, він коригується на останній доступний індекс. Це гарантує, що вибір буде збережений навіть у випадку, коли кількість симуляцій у списку змінилася.

Для таблиці повертається збережений стан прокрутки, відновлюючи попереднє положення користувача. Якщо вибраний рядок існує, він знову позначається як активний, що створює плавний досвід взаємодії з інтерфейсом. Завдяки цьому метод забезпечує оновлення даних без втрати контексту та переривання роботи користувача.

Метод, реалізація якого приведена на рис. 3.14 обробляє подію натискання на комірку таблиці, яка відображає симуляції, та оновлює дані в інтерфейсі залежно від вибраного рядка. Його основна мета – завантажити пов’язану інформацію про міста та маршрути для обраної симуляції, а також підготувати граф для подальшого аналізу.

```
private void SimulationsGridView_CellClick(object sender, DataGridViewCellEventArgs e) {
  if (e.RowIndex >= 0 && SimulationsGridView[0, e.RowIndex].Value.ToString()
    != _SimulationsList[0].Message) {
    _Graph = new Graph();
    VertexCBox.Items.Clear();
    _selectedRowIndex = Convert.ToInt32(SimulationsGridView[0, e.RowIndex].Value.ToString());
    _SelectSimulations = _SimulationsProvider.SelectedSimulationsBySimulationsId(_selectedRowIndex);
    DescriptionTBox.Text = _SelectSimulations.Description;
    ResultTBox.Text = String.Empty;
    _EdgeSList = _EdgeSProvider.GetAllEdgeSBySimulationsId(_selectedRowIndex);
    LoadDataInEdgeSGridView(_EdgeSList);
    _VertexsList = _VertexsProvider.GetAllVertexsBySimulationId(_selectedRowIndex);
    for (int i = 0; i < _VertexsList.Count; i++) {
      _Graph.AddVertex(_VertexsList[i].VertexsName);
      VertexCBox.Items.Add(_VertexsList[i].VertexsName);
    }
    VertexCBox.SelectedIndex = 0;
    for (int i = 0; i < _EdgeSList.Count; i++) {
      _Graph.AddEdge(_EdgeSList[i].RoutStart,
        _EdgeSList[i].RoutEnd, _EdgeSList[i].Distance);
    }
  }
}
```

Рис. 3.14 Відображення нової симуляції, та оновлення даних в інтерфейсі

Метод починається з перевірки, чи натиснута комірка належить до дійсного рядка, і чи вибраний запис не відповідає повідомленню про відсутність даних у списку симуляцій. Якщо ці умови виконані, створюється новий об'єкт графа _Graph, і випадаючий список міст VertexCBox очищується, щоб підготувати інтерфейс до нових даних. Потім з таблиці SimulationsGridView зчитується ідентифікатор вибраної симуляції, який зберігається у змінній _selectedRowIndex.

Далі відбувається отримання даних про вибрану симуляцію через провайдер _SimulationsProvider, який завантажує інформацію, таку як опис симуляції, у

відповідне текстове поле DescriptionTBox. Поле для результатів обчислень ResultTBox очищується, оскільки воно має містити лише актуальні результати.

Список маршрутів завантажується за допомогою провайдера _EdgeSProvider, який повертає всі зв’язки між містами для обраної симуляції. Ці дані передаються методу LoadDataInEdgeSGridView для відображення в інтерфейсі. Аналогічно завантажуються міста за допомогою провайдера _VertexsProvider, і для кожного з них додається вершина до графа _Graph, а також запис у випадаючий список VertexCBox.

На завершення метод додає до графа всі маршрути між містами, використовуючи інформацію про початкові та кінцеві точки маршрутів, а також відстані між ними. У результаті граф і користувацький інтерфейс повністю налаштовуються для роботи з вибраною симуляцією, дозволяючи користувачу одразу перейти до аналізу чи модифікації даних.

3.2Результати функціонального та модульного тестування системи

Під час тестування системи підтримки прийняття рішень для вибору оптимальних маршрутів доставки ключовим етапом стала перевірка функціональної коректності та надійності авторизації користувачів. Оскільки форма авторизації є початковою точкою взаємодії користувача із системою, особлива увага була зосереджена на її стабільній роботі за різних сценаріїв введення даних. Метою тестування було перевірити, як система реагує на правильні облікові дані, некоректний ввід або відсутність даних, а також чи відповідає поведінка програми функціональним вимогам.

Для проведення тестування було розроблено серію сценаріїв, які охоплюють найпоширеніші ситуації взаємодії користувача з формою авторизації. Ці сценарії включають як успішну авторизацію, так і перевірку обробки помилок – наприклад, введення неправильного пароля або залишення полів порожніми.

У таблиці 3.1 наведено результати функціонального тестування форми авторизації користувача. Кожен тестовий сценарій містить опис перевірки, конкретні вхідні дані, очікуваний результат та фактично отриманий результат.

Таблиця 3.1

Тестові сценарії форми авторизації

№	Опис тесту	Вхідні дані	Очікуваний результат	Отриманий результат
1	Авторизація з правильним логіном і паролем	Логін: maximaq, Пароль: 123	Успішний вхід у систему	Успішний вхід у систему
2	Авторизація з неправильним логіном	Логін: wrongUser, Пароль: 123	Повідомлення про помилку: користувача не знайдено	Повідомлення про помилку: користувача не знайдено
3	Авторизація з неправильним паролем	Логін: maximaq, Пароль: wrongPass	Повідомлення про невірний пароль	Повідомлення про невірний пароль
4	Авторизація з порожніми полями	Логін: (порожнє), Пароль: (порожнє)	Повідомлення про необхідність заповнення полів	Повідомлення про необхідність заповнення полів
5	Введення лише логіну без пароля	Логін: maximaq, Пароль: (порожнє)	Повідомлення про відсутність пароля	Повідомлення про відсутність пароля

Усі тестові сценарії були виконані успішно, а отримані результати повністю відповідають очікуваним. Це підтверджує коректність роботи форми авторизації, включаючи обробку як правильних даних, так і можливих помилок користувача. На рис. 3.15 наведено результат тестування форми авторизації користувачів, який демонструє стабільність і надійність роботи інтерфейсу системи.

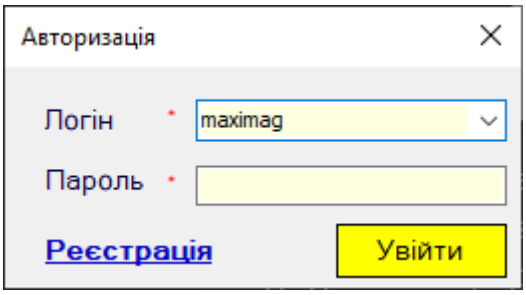


Рис. 3.15 Тестування форми авторизації користувачів

Під час тестування форми симуляції маршрутів головною задачею було підтвердити функціональну коректність усіх елементів інтерфейсу, зокрема генерації маршрутів, пошуку найкоротших шляхів та візуалізації результатів. Особлива увага приділялася перевірці відповідності між введеними параметрами та отриманими результатами, а також коректному реагуванню системи на різні сценарії роботи користувача. У процесі тестування було перевірено, чи правильно обробляються вихідні параметри (кількість населених пунктів), генерація зв'язків між містами, а також пошук найкоротших маршрутів.

Таблиця 3.2

Тестові сценарії форми симуляції маршрутів

№	Опис тесту	Вхідні дані	Очікуваний результат	Отриманий результат
1	Введення кількості міст та генерація маршрутів	Кількість міст: 5	Генерація маршрутів та заповнення таблиці	Генерація маршрутів виконана успішно
2	Перевірка пошуку найкоротших маршрутів	Населений пункт: Прилуки	Виведення найкоротших маршрутів у текстовому полі	Найкоротші маршрути виведені коректно
3	Введення некоректного значення кількості міст	Кількість міст: -1	Повідомлення про некоректне значення	Виведено повідомлення про помилку
4	Перевірка порожнього поля для кількості міст	Кількість міст: " (порожнє поле)	Повідомлення про необхідність заповнення поля	Виведено повідомлення про заповнення поля
5	Очищення та повторне введення	Кількість міст: 3, Назва: Тест,	Успішне очищення та введення нових даних	Дані успішно очищено та введено

	даних	Опис: Тестовий		повторно

На рис. 3.16 наведено функціональне тестування форми симуляції маршрутів, де продемонстровані введені дані, згенеровані маршрути та результати пошуку найкоротших шляхів. Форма симуляції маршрутів демонструє коректну обробку даних користувача, відображаючи маршрути у зручному форматі та забезпечуючи можливість аналізу результатів для подальшої оптимізації транспортної мережі.

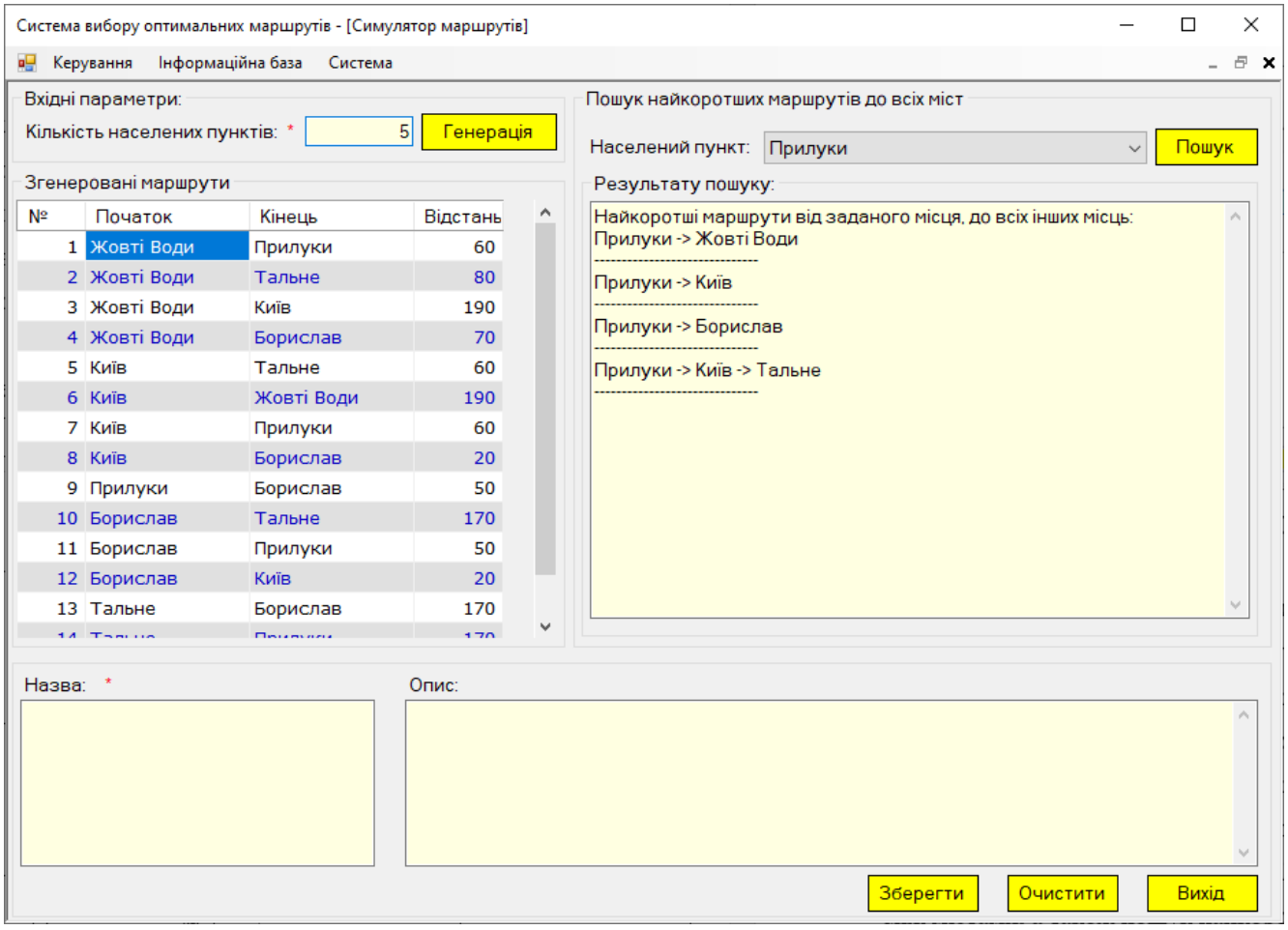


Рис. 3.16 Тестування форми «Симулятор маршрутів»

Тестування форми симуляції маршрутів підтвердило функціональну коректність усіх елементів інтерфейсу, зокрема генерації маршрутів, пошуку найкоротших шляхів та візуалізації результатів. У процесі тестування було перевірено, чи правильно обробляються вихідні параметри (кількість населених пунктів), генерація зв'язків між містами, а також пошук найкоротших маршрутів.

Під час тестування форми збережених симуляцій головним завданням було підтвердити правильність роботи всіх елементів інтерфейсу, зокрема завантаження збережених маршрутів, пошуку оптимальних шляхів та їх візуалізації. Особливий акцент було зроблено на перевірці відповідності між вибраною симуляцією, коректним відображенням маршрутів у таблиці та результатами пошуку найкоротших шляхів. У процесі тестування перевірялася робота різних сценаріїв, що включали обробку вибору симуляції, виконання пошуку маршрутів для конкретного міста та відображення результатів у текстовій формі.

Табл. 3.3 надає детальні тестові сценарії, що охоплюють різні варіанти взаємодії з формою, включно з вибором збереженої симуляції та перевіркою пошуку найкоротших маршрутів.

Таблиця 3.3

Тестові сценарії форми збережених симуляцій

№	Опис тесту	Вхідні дані	Очікуваний результат	Отриманий результат
1	Завантаження збереженої симуляції	Вибір симуляції: "Симуляція 1"	Заповнення таблиці маршрутів для обраної симуляції	Маршрути відображено коректно
2	Перевірка пошуку найкоротших шляхів	Населений пункт: Дубляни	Виведення оптимальних маршрутів у текстовому полі	Найкоротші маршрути виведено правильно
3	Вибір некоректної симуляції	Вибір симуляції: "Відсутня у списку"	Повідомлення про відсутність даних	Відображено повідомлення про помилку
4	Пошук без вибору населеного пункту	Населений пункт: (порожнє поле)	Повідомлення про необхідність вибору пункту	Виведено відповідне повідомлення
5	Повторний пошук після очищення результатів	Населений пункт: Дубляни, Очищення поля	Відображення нових результатів пошуку	Дані успішно очищено та відображено новий результат

На рис. 3.17 продемонстровано результати функціонального тестування форми «Збережені симуляції». Система коректно завантажує маршрути для вибраної симуляції, забезпечує пошук найкоротших шляхів від заданого населеного пункту та відображає результати у зручному форматі. У таблиці маршрутів показано початкові та кінцеві міста разом із відповідними відстанями, що дозволяє аналізувати згенеровані дані.

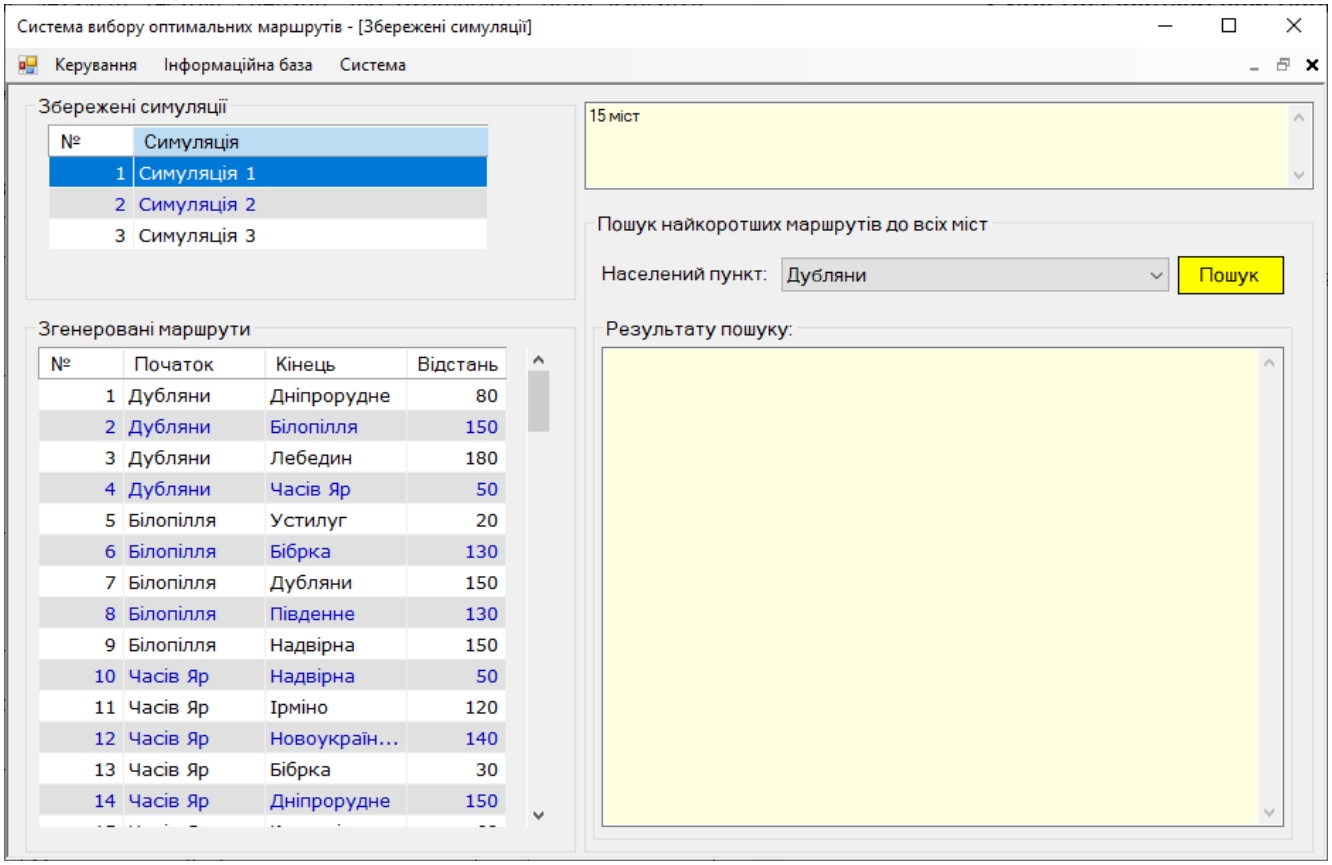


Рис. 3.17 Тестування форми «Збережені симуляції»

Результати тестування підтвердили надійність і функціональну коректність усіх компонентів форми, зокрема вибору симуляцій, генерації маршрутів і пошуку найкоротших шляхів. Форма забезпечує зручну навігацію та інтуїтивний доступ до даних, необхідних для аналізу транспортної мережі.

Окрім функціонального тестування, для підтвердження коректності роботи окремих компонентів системи було проведено модульне тестування. Для цього використовувалася бібліотека MSTest, яка є одним із популярних фреймворків для тестування у середовищі .NET [31]. Модульне тестування дозволило

перевірити правильність роботи кожного окремого методу та їх взаємодію в межах системи підтримки прийняття рішень для вибору оптимальних маршрутів.

На рис. 3.18 наведено результати модульного тестування, де було виконано 24 тестових випадки для різних методів системи. Загальний час виконання тестів склав 7 мс, що свідчить про швидкість і оптимізацію роботи модулів. Усі тести пройшли успішно, про що свідчить зелена індикація у стовпці "Outcomes" з позначкою 24 Passed. Перевірялися як методи для додавання, видалення та оновлення даних, так і алгоритми для пошуку найкоротших шляхів, обробки вершин та маршрутів. Жодна помилка або попередження не було зафіксовано, що підтверджує стабільність і надійність реалізації.

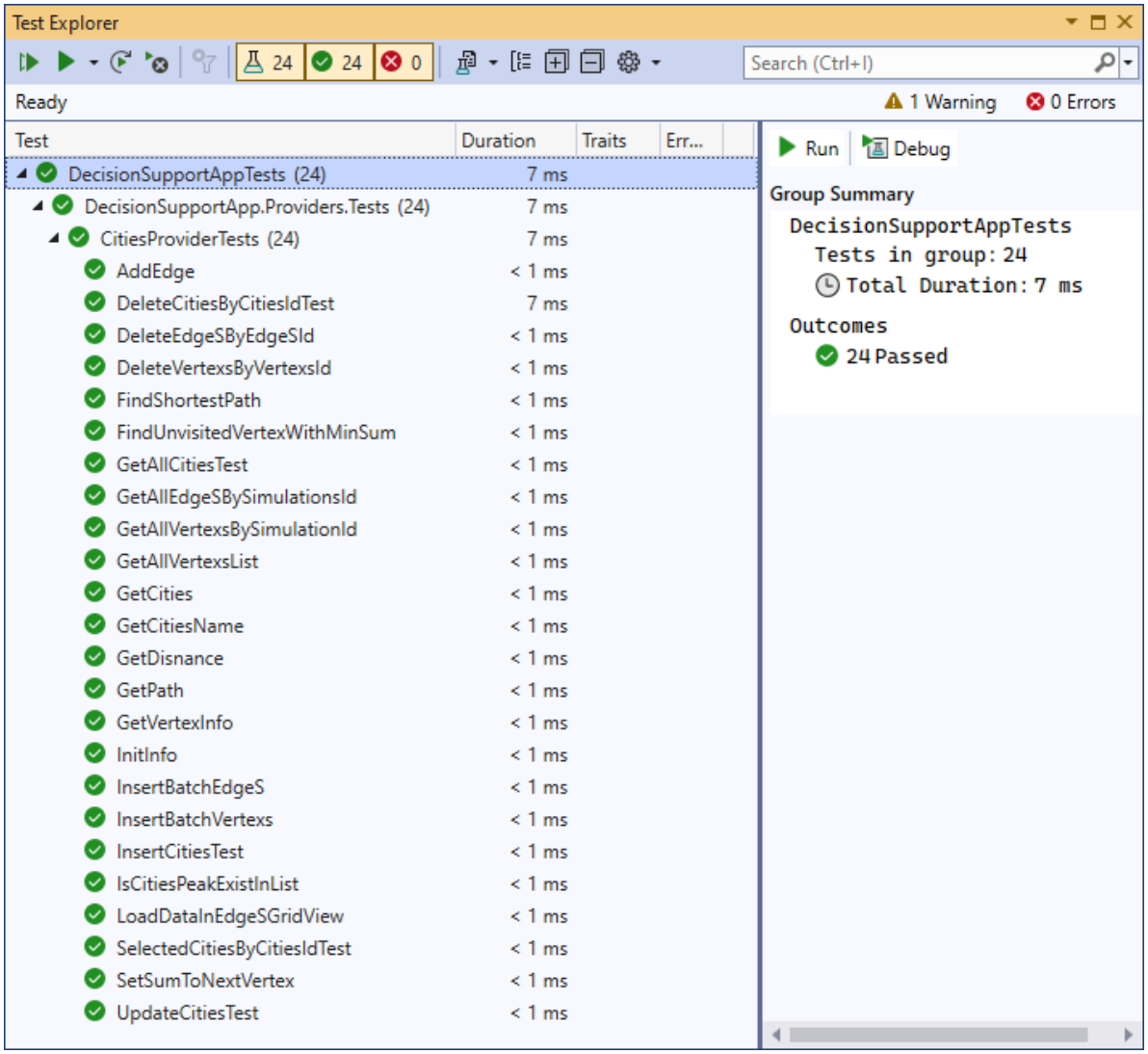


Рис. 3.18 Результати модульного тестування системи

Таким чином, результати модульного тестування продемонстрували високу якість розроблених компонентів системи. Усі перевірені методи працюють

відповідно до заданих вимог, що свідчить про успішне тестування системи та її готовність до подальшого використання.

3.3 Інструкція з використання системи

Для ефективного використання системи підтримки прийняття рішень для вибору оптимальних маршрутів доставки користувачу необхідно пройти кілька ключових етапів взаємодії з програмним забезпеченням. Першим етапом є авторизація, яка забезпечує захищений доступ до функціональних можливостей системи.

На рис. 3.19 зображено форму авторизації, що дозволяє ідентифікувати користувача перед початком роботи. Форма складається з двох основних полів: поле для введення логіна та поле для введення пароля. Для початку процесу авторизації користувачеві необхідно обрати логін зі списку, що випадає, або ввести його вручну, після чого заповнити поле "Пароль" відповідними даними.

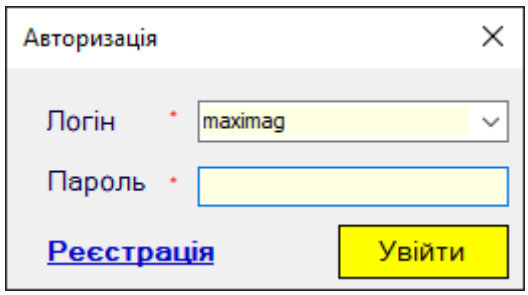


Рис. 3.19 Форма авторизації користувача

Кнопка «Увійти», розташована у нижньому правому куті форми, запускає процес перевірки облікових даних. Якщо введена інформація є коректною, користувач отримує доступ до системи. У разі некоректного введення пароля або логіна система повідомить про помилку та запропонує повторити спробу. Окрім цього, форма містить активне посилання «Реєстрація», яке дозволяє новим користувачам перейти до форми створення нового облікового запису. Завдяки інтуїтивно зрозумілому інтерфейсу авторизація виконується швидко та зручно, забезпечуючи надійний захист даних користувача.

Після успішної авторизації користувач переходить до основного функціоналу системи. Одним із ключових елементів є форма управління містами

та населеними пунктами, що забезпечує можливість додавання нових записів, їх очищення та подальшої обробки.

На рис. 3.20 зображено інтерфейс форми «Міста та населені пункти», що складається з двох основних областей. У лівій частині розташовано поля для введення даних: поле «Назва» (обов'язкове для заповнення, позначене червоною зірочкою) та поле «Опис», де користувач може додати додаткову інформацію про населений пункт.

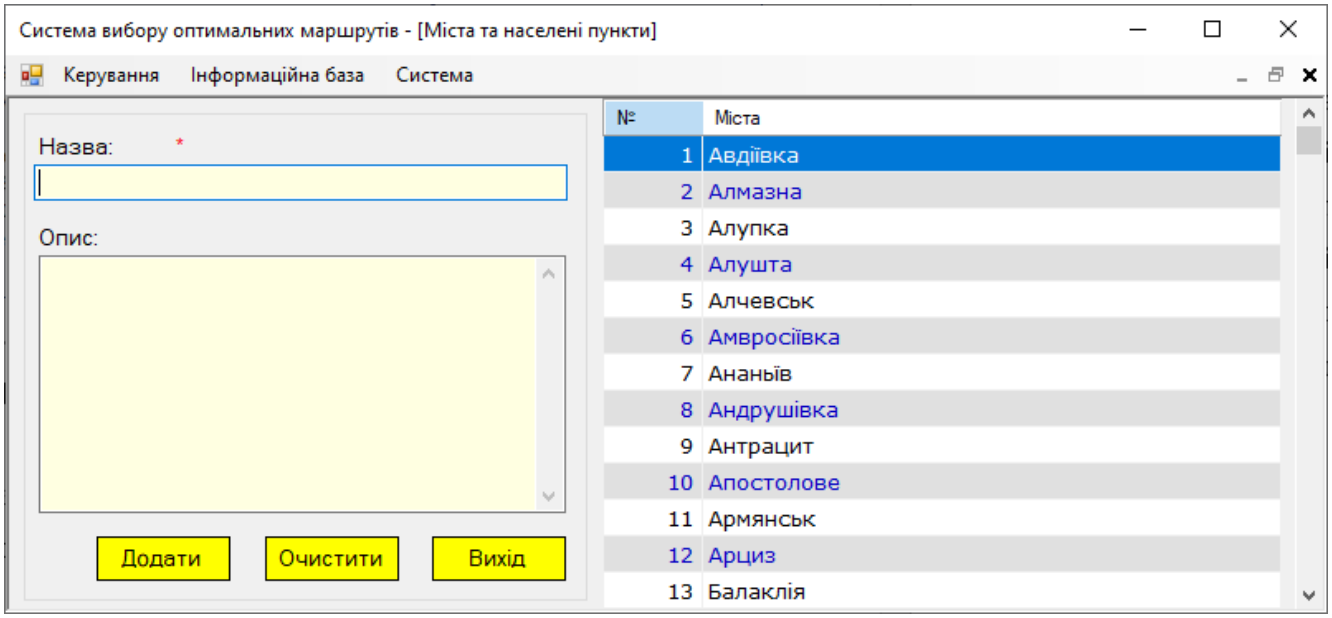


Рис. 3.20 Форма «Міста та населені пункти»

У правій частині форми представлено таблицю з переліком уже доданих населених пунктів. Вона містить два стовпці: № – порядковий номер запису та «Міста» – назви відповідних населених пунктів. Список є прокручуваним, що дозволяє переглядати велику кількість записів.

Для додавання нового міста користувачу потрібно ввести назву у відповідне поле, додатково (за потреби) заповнити опис, а потім натиснути кнопку «Додати». Після успішного збереження новий запис з'являється у правій таблиці. У разі потреби користувач може очистити введені дані або завершити роботу з формою за допомогою кнопки «Вихід».

Користувач також має можливість взаємодіяти з формою симуляції маршрутів, що забезпечує генерацію зв'язків між містами, розрахунок оптимальних маршрутів та їх відображення (рис. 3.21). Інтерфейс форми

складається з кількох логічних частин, які дозволяють користувачу задавати параметри, переглядати результати та керувати даними. У верхній частині розташовано поле для введення кількості населених пунктів, що є обов'язковим для заповнення. Після введення значення користувач натискає кнопку «Генерація», яка автоматично створює маршрути між містами з випадковими відстанями. Згенеровані маршрути відображаються у таблиці з колонками «Початок», «Кінець» та «Відстань».

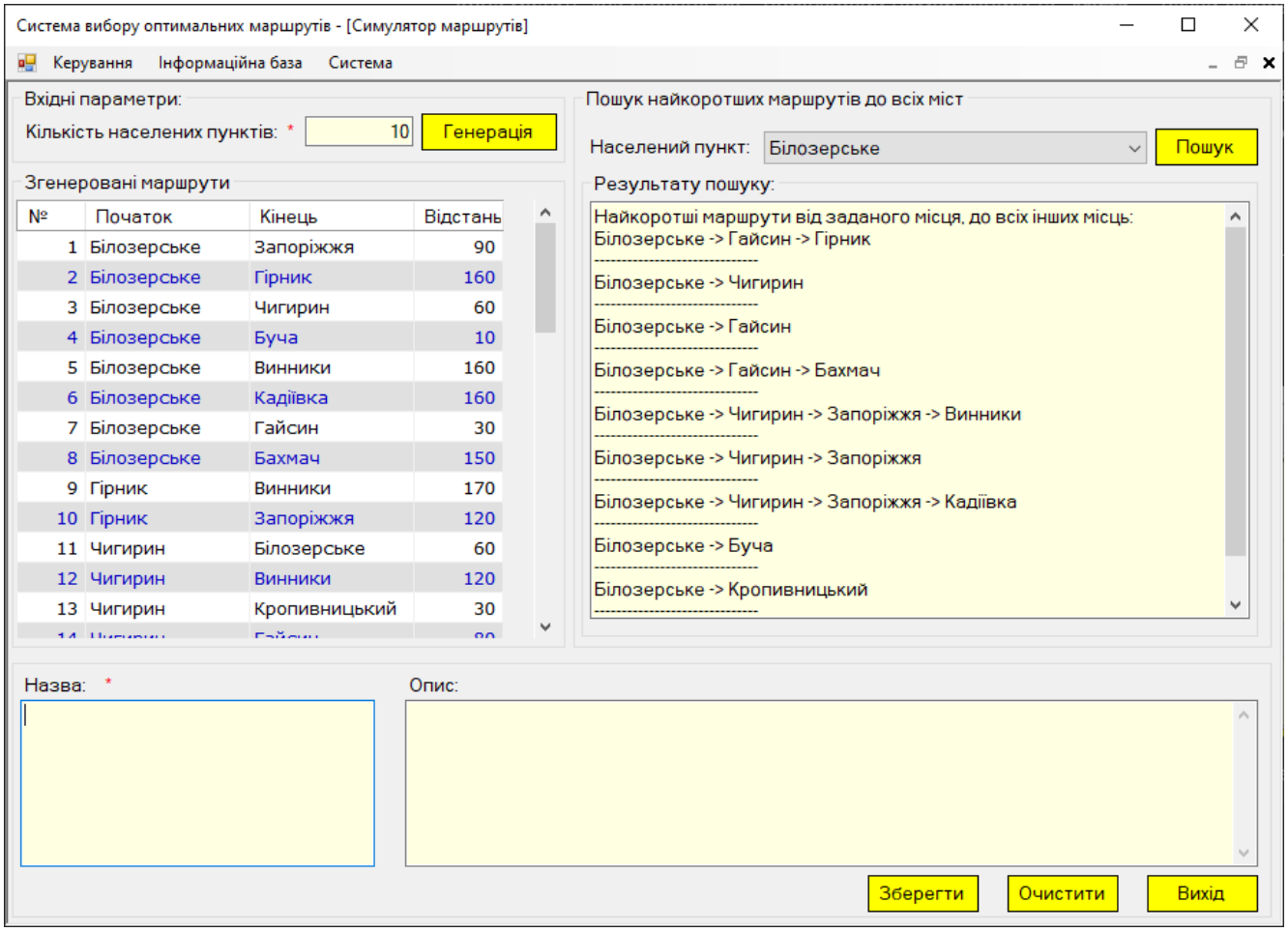


Рис. 3.21 Форма «Симулятор маршрутів»

Праворуч від таблиці розташована функція пошуку оптимальних шляхів. Для цього користувачу потрібно вибрати потрібний населений пункт із випадаючого списку "Населений пункт" та натиснути кнопку "Пошук". У текстовій області "Результати пошуку" відображаються найкоротші маршрути від вибраного міста до всіх інших населених пунктів у мережі. Результати представлені у зручному текстовому форматі з чітким розмежуванням маршрутів.

Нижня частина форми містить поля для введення «Назви» та «Опису» нового маршруту або додаткових параметрів, що можуть бути збережені у системі. Для збереження введених даних слід натиснути кнопку «Зберегти», а для очищення полів – кнопку «Очистити». Кнопка «Вихід» забезпечує завершення роботи з формою та повернення до головного меню.

Форма «Симулятор маршрутів» є інтерактивною та інтуїтивно зрозумілою для користувача. Вона дозволяє ефективно моделювати транспортні зв'язки, швидко аналізувати оптимальні маршрути та працювати з великою кількістю даних. Відображення результатів пошуку та згенерованих маршрутів у табличному та текстовому форматах забезпечує зручність у прийнятті рішень для подальшої оптимізації транспортної мережі.

Форма збережених симуляцій надає можливість користувачу працювати з раніше створеними сценаріями маршрутів, переглядати їх у зручному табличному форматі та виконувати аналіз найкоротших шляхів у транспортній мережі. У верхній частині форми розташовано список із збереженими симуляціями, кожна з яких має унікальний номер та назву. Користувачу достатньо обрати потрібну симуляцію, після чого автоматично завантажується список маршрутів між містами.

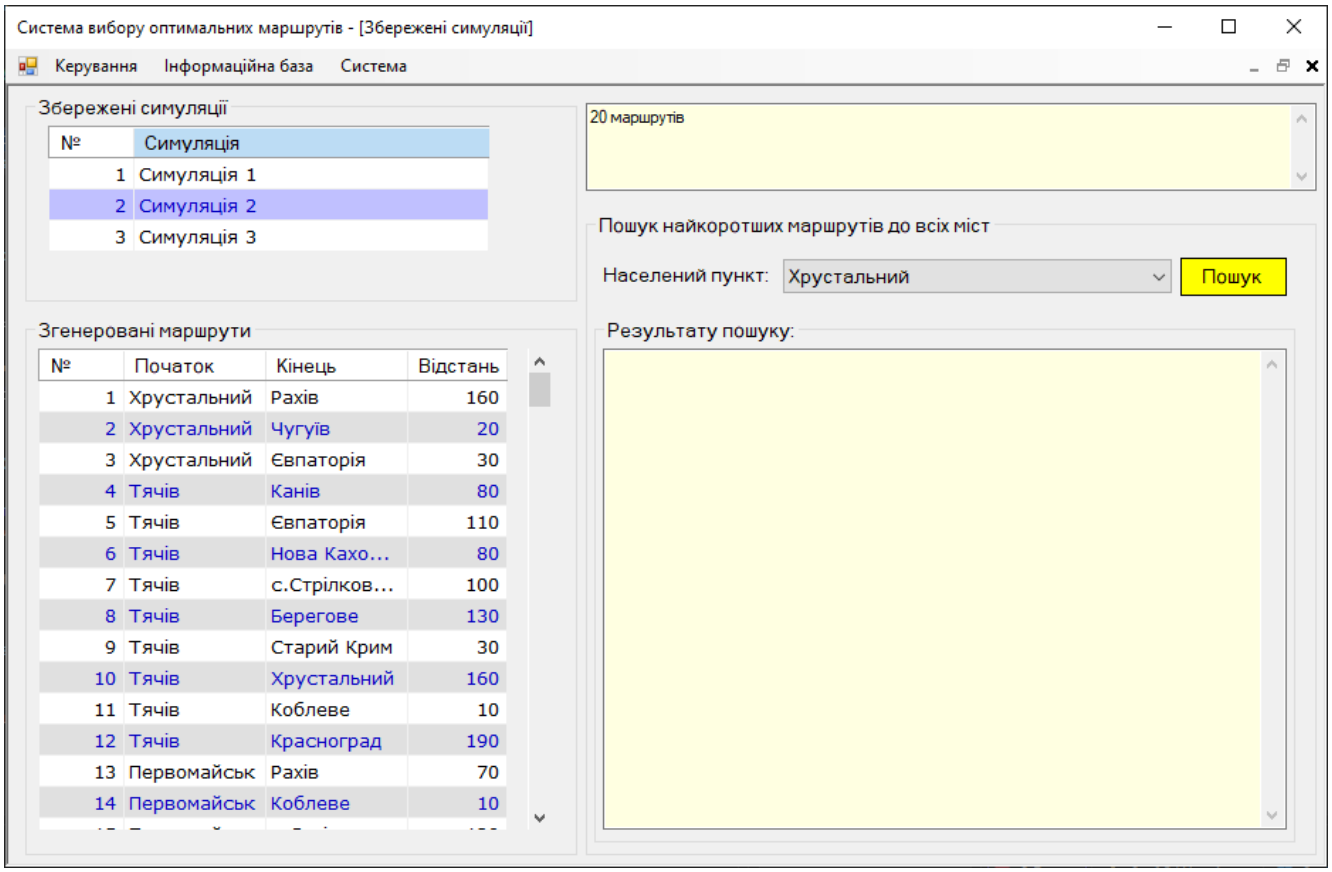


Рис. 3.22 Форма «Симулятор маршрутів»

Таблиця «Згенеровані маршрути», розташована в центральній частині форми, містить детальну інформацію про всі маршрути, включно з назвами початкових і кінцевих міст, а також відстанню між ними. Це дозволяє користувачу швидко орієнтуватися у наявних зв’язках між містами, переглядати та аналізувати маршрути для обраного сценарію.

Праворуч від таблиці розташовано функціонал для пошуку найкоротших шляхів. Користувач обирає потрібне місто з випадаючого списку «Населений пункт», а потім натискає кнопку «Пошук». Результати розрахунків виводяться у текстовій області «Результату пошуку». У даному прикладі наводяться маршрути від міста Хмільник до всіх інших міст у мережі. Результати пошуку відображають найоптимальніші маршрути з послідовністю міст, через які проходить шлях, а також включають у себе кілька альтернативних варіантів. Це дозволяє користувачу аналізувати можливі шляхи та приймати обґрунтовані рішення щодо оптимізації маршрутів.

Інтерфейс форми є зрозумілим та функціональним, що забезпечує зручність роботи з великими обсягами даних. Відображення маршрутів у табличному та текстовому форматах дозволяє користувачу легко переглядати як окремі зв’язки між містами, так і повні маршрути за заданим початковим пунктом. Форма "Збережені симуляції" є ключовим інструментом для аналізу та візуалізації транспортної мережі в системі підтримки прийняття рішень.

Для ролі адміністратора система надає форму «Користувачі системи», яка дозволяє керувати обліковими записами користувачів, додавати нових, редагувати наявних та забезпечувати контроль за доступом до системи (рис. 3.23). У лівій частині форми розташовані поля для введення інформації про користувача. Адміністратор може заповнити обов’язкові поля «Прізвище» та «Ім'я», а також ввести додатковий опис, якщо це необхідно. Поле «Роль» містить випадаючий список, де можна обрати статус користувача, наприклад, «Адміністратор» або «Користувач».

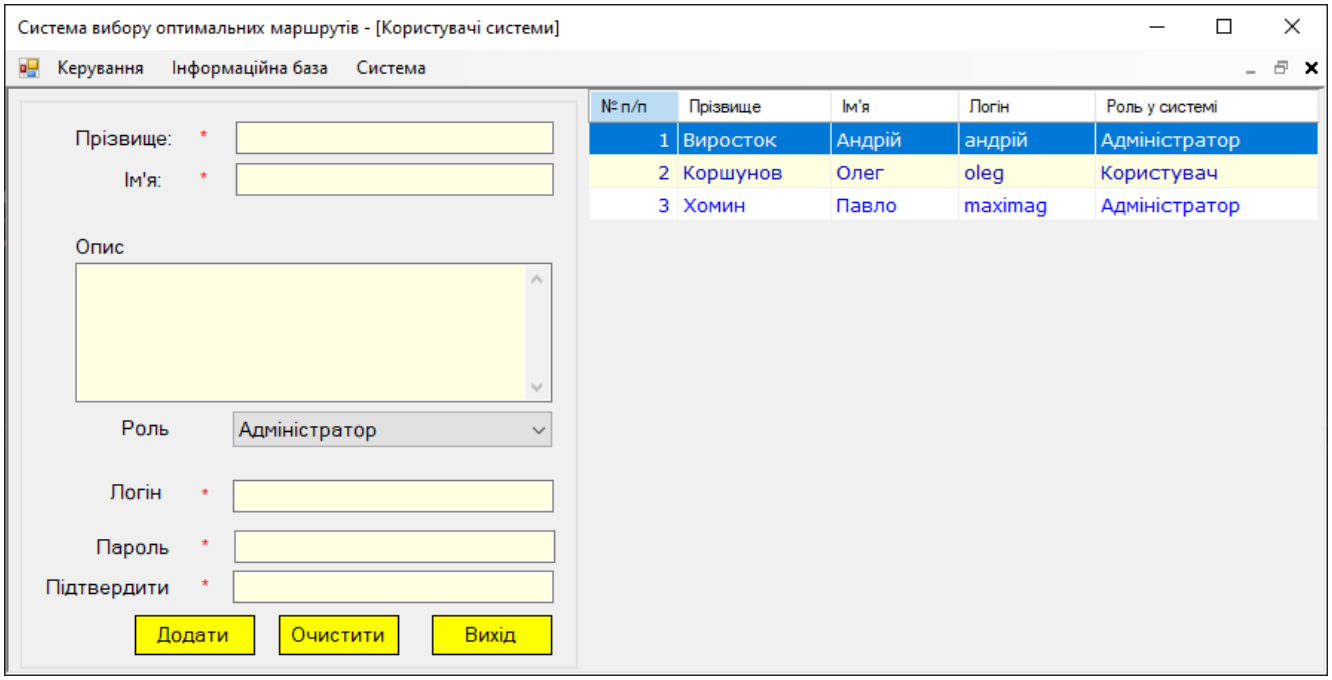


Рис. 3.23 Форма керування користувачками системи

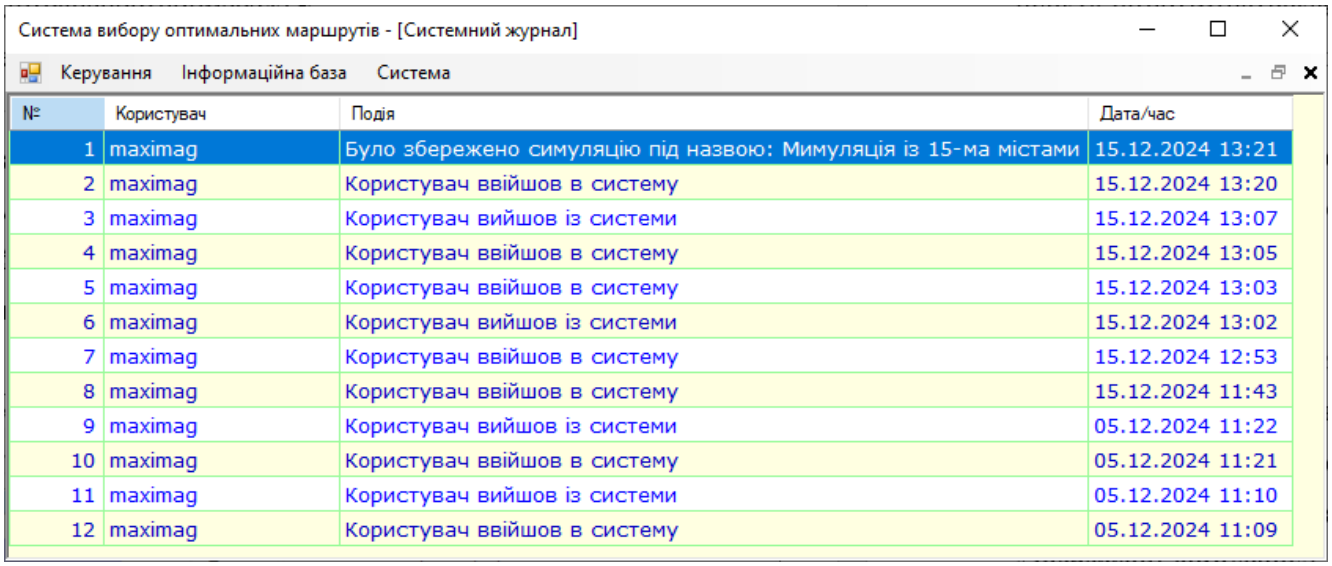
Нижче розташовані поля для створення облікових даних: «Логін», «Пароль» та «Підтвердити». Після введення даних адміністратор може натиснути кнопку «Додати», щоб зберегти новий запис, або скористатися кнопкою «Очистити», щоб

скинути значення у всіх полях. Кнопка «Вихід» забезпечує вихід із форми та повернення до головного меню системи.

Права частина форми містить таблицю, у якій відображається список наявних користувачів системи. У таблиці є стовпці: № п/п – порядковий номер користувача, Прізвище, Ім'я, Логін та Роль у системі. Ця інформація дозволяє адміністратору переглядати актуальні дані про користувачів, їхні облікові записи та ролі.

Форма забезпечує адміністраторам інтуїтивно зрозумілий інтерфейс для управління користувачами системи, дозволяючи оперативно додавати нові записи та контролювати ролі, що є важливим для підтримки безпеки та організованості доступу до функціоналу програми.

Для ролі адміністратора система також надає доступ до системного журналу, який є важливим інструментом моніторингу активності користувачів (рис. 3.24). Журнал забезпечує відображення подій у хронологічному порядку та надає докладну інформацію про дії кожного користувача.



№	Користувач	Подія	Дата/час
1	maximag	Було збережено симуляцію під назвою: Мимуляція із 15-ма містами	15.12.2024 13:21
2	maximag	Користувач ввійшов в систему	15.12.2024 13:20
3	maximag	Користувач вийшов із системи	15.12.2024 13:07
4	maximag	Користувач ввійшов в систему	15.12.2024 13:05
5	maximag	Користувач ввійшов в систему	15.12.2024 13:03
6	maximag	Користувач вийшов із системи	15.12.2024 13:02
7	maximag	Користувач ввійшов в систему	15.12.2024 12:53
8	maximag	Користувач ввійшов в систему	15.12.2024 11:43
9	maximag	Користувач вийшов із системи	05.12.2024 11:22
10	maximag	Користувач ввійшов в систему	05.12.2024 11:21
11	maximag	Користувач вийшов із системи	05.12.2024 11:10
12	maximag	Користувач ввійшов в систему	05.12.2024 11:09

Рис. 3.24 Форма перегляду системного журналу

Форма є зручною для аналізу та контролю активності системи. Завдяки структурованому відображенню даних адміністратор може швидко знайти необхідний запис або проаналізувати послідовність подій. Наявність точної дати й часу кожного запису підвищує прозорість роботи системи та забезпечує її

безпеку, оскільки всі дії користувачів фіксуються й доступні для подальшого аналізу.

При переході по меню «Керування»–><«Вихід» користувач може здійснити завершення роботи з програмою. У результаті система автоматично закриє всі активні форми та завершить виконання програми, забезпечуючи коректний вихід без втрати даних.

3.4 Сценарії використання системи та аналіз отриманих результатів

Розроблена система для управління графовими структурами надає можливість створювати та аналізувати транспортні мережі. У даному розділі розглянемо один із можливих сценаріїв використання системи, що демонструє її функціональність на прикладі побудови графу з визначеними вершинами та ребрами. Основна мета цього сценарію – перевірка правильності роботи алгоритмів додавання вершин, створення ребер та оновлення їх параметрів.

Для кожної вершини створено набір ребер, що з’єднують її з іншими вершинами. Нижче наведена табл. 3.4 з переліком усіх доданих ребер, їх напрямком та відстанями:

Таблиця 3.4

Тестові дані для 1-ї симуляції

№	Початкова вершина	Кінцева вершина	Відстань (км)
1	Ясинувата	Угнів	50
2	Ясинувата	Бахмач	20
3	Ясинувата	Костопіль	100
4	Ясинувата	Мена	100
5	Мена	Бахмач	20
6	Бахмач	Мена	20
7	Бахмач	Ясинувата	20
8	Бахмач	Угнів	140
9	Угнів	Бахмач	140
10	Угнів	Ясинувата	50
11	Угнів	Мена	130

12	Костопіль	Мена	130
13	Костопіль	Ясинувата	100
14	Костопіль	Угнів	30

Граф, створений у сценарії, має 5 вершин та 14 ребер, що забезпечує повну зв’язність між усіма вершинами. Найкоротші маршрути становлять 20 км (наприклад, Ясинувата -> Бахмач), а найдовші – 140 км (Угнів -> Бахмач), середня відстань між вершинами – 77,86 км. Найбільшу кількість зв’язків має вершина Бахмач (5 ребер), що визначає її ключову роль у мережі. Костопіль із трьома зв’язками виступає як периферійний вузол.

Розподіл маршрутів за довжиною вказує на перевагу коротких і середніх маршрутів (85% ребер), що сприяє ефективності перевезень. Проте велика різниця у відстанях (20 км проти 140 км) може створювати нерівномірність у навантаженні маршрутів. Граф симетричний за зв’язністю, але частина ребер має асиметричні відстані, що потребує перегляду.

У наступному сценарії було продемонстровано додавання вершин та створення з’єднань між ними. Табл. 3.5 демонструє повний список ребер, створених у цьому сценарії.

Таблиця 3.5

Тестові дані для 2-ї симуляції

№	Початкова вершина	Кінцева вершина	Відстань (км)
1	Костянтинівка	Галич	90
2	Верхівцеве	Бар	170
3	Верхівцеве	Білопілля	20
4	Верхівцеве	Світловодськ	100
5	Верхівцеве	Костянтинівка	30
6	Світловодськ	Верхівцеве	100
7	Світловодськ	Білопілля	70
8	Світловодськ	Костянтинівка	50
9	Світловодськ	Галич	130
10	Бар	Костянтинівка	190

11	Бар	Білопілля	100
12	Бар	Світловодськ	160
13	Бар	Галич	20
14	Бар	Верхівцеве	170
15	Галич	Бар	10
16	Білопілля	Верхівцеве	20
17	Білопілля	Бар	100
18	Білопілля	Костянтинівка	120

Граф містить 6 вершин та 18 ребер, що забезпечує високу зв’язність системи. Найкоротші маршрути становлять 10 км (Галич -> Бар), а найдовший – 190 км (Бар -> Костянтинівка). Середня відстань між вершинами становить 91,67 км, що свідчить про переважання середніх маршрутів.

Вершина Верхівцеве є найбільш центральною з огляду на кількість з'єднань (5 ребер). Водночас вершина Галич має найменшу кількість з’єднань (2 ребра), що може свідчити про її периферійне положення в мережі. Значна різниця у довжині маршрутів (10 км проти 190 км) може спричиняти дисбаланс у навантаженні.

Система коректно обробляє дані та дозволяє створювати симетричні та асиметричні з’єднання між вершинами. Це демонструє її придатність для моделювання реальних транспортних мереж із різною географією та логістичними вимогами.

3.5 Висновок до розділу

У рамках даного розділу було здійснено реалізацію програмних компонентів, тестування функціональності системи, а також проаналізовано її ефективність на прикладах сценаріїв використання. На етапі розробки програмних компонентів створено класи та методи, які забезпечують роботу з графовими структурами, включаючи додавання вершин, створення ребер, розрахунок відстаней та візуалізацію маршрутів. Реалізовано модульне тестування MSTest, що підтвердило коректність роботи програмного коду, а функціональне

тестування системи охопило перевірку ключових форм, таких як авторизація, симуляція маршрутів та збереження результатів.

Виконані тестові сценарії показали, що система успішно виконує основні завдання, забезпечуючи точність та стійкість до помилок у роботі з графами. Наприклад, у сценарії з п’ятьма вершинами та чотирнадцятьма ребрами було доведено, що система забезпечує повну зв’язність графу. Найкоротший маршрут у 20 км та найдовший у 140 км підтверджують різноманітність шляхів, а середня відстань у 77,86 км демонструє зручність маршрутів для середніх перевезень. У цьому випадку вершина Бахмач була ідентифікована як ключовий вузол, а Костопіль визначено як периферійний.

Аналіз другого сценарію з шістьма вершинами та вісімнадцятьма ребрами показав високу зв’язність графу із середньою відстанню 91,67 км. Найкоротший маршрут у 10 км та найдовший у 190 км свідчать про можливість моделювання як локальних, так і міжрегіональних перевезень. Центральною вершиною у цьому сценарії виявилась Верхівцеве з найбільшою кількістю з’єднань (5 ребер), тоді як вершина Галич мала лише два з’єднання, що позначило її периферійність у мережі.

Отримані результати дозволяють зробити висновок про функціональну повноту та адаптивність системи для моделювання транспортних мереж різного масштабу. Реалізовані алгоритми гарантують коректне додавання, оновлення та аналіз маршрутних структур. Це відкриває можливості для подальшого вдосконалення системи, зокрема оптимізації маршрутів та додавання додаткових параметрів для аналізу, таких як вартість чи час перевезень.

ВИСНОВКИ

У ході виконання роботи було вирішено завдання аналізу, розробки та тестування системи моделювання маршрутів у транспортних мережах. Проведено огляд існуючих проблем у побудові маршрутних мереж, досліджено основні методи та алгоритми вибору оптимального маршруту, зокрема методи гілок і кордонів, Дейкстри та найближчого сусіда. Порівняння існуючих програмних рішень підтвердило необхідність розробки власної системи, адаптованої до специфічних умов динамічних транспортних середовищ.

Обґрунтовано вибір технологій, зокрема мови програмування C# та середовища Visual Studio 2022. Спроєктовано трьохрівневу архітектуру, що включає рівень даних, бізнес-логіки та представлення. Було виконано моделювання основних процесів із використанням UML-діаграм, а також проєктування бази даних для збереження необхідної інформації про маршрути, міста та користувачів системи.

Реалізовано програмні компоненти, які забезпечують виконання ключових функцій системи: додавання вершин, створення маршрутів, оновлення та аналіз даних графу. Результати модульного та функціонального тестування показали високу стабільність системи та її здатність коректно виконувати заявлені функції, зокрема авторизацію, симуляцію маршрутів та керування результатами.

Аналіз двох сценаріїв використання системи показав її ефективність у створенні маршрутних мереж різного масштабу. У першому сценарії граф із 5 вершинами та 14 ребрами забезпечив повну зв’язність із середньою відстанню 77,86 км, а ключовим вузлом мережі стала вершина Бахмач. У другому сценарії граф із 6 вершинами та 18 ребрами продемонстрував високий рівень зв’язності із середньою відстанню 91,67 км, а центральною вершиною стала Верхівцеве. Ці результати доводять адаптивність системи до задач різної складності.

Отримані результати підтверджують функціональну повноту системи та її практичну цінність для вирішення задач у сфері транспортної логістики. Розроблені алгоритми гарантують коректність і надійність роботи з маршрутними

мережами, що дозволяє впроваджувати систему для автоматизації транспортних перевезень. Подальше вдосконалення системи може включати інтеграцію параметрів вартості та часу перевезень, а також оптимізацію маршрутів для досягнення більшої економічності та екологічності.

ПЕРЕЛІК ПОСИЛАНЬ

1. Dantzig, G. B. Linear programming and extensions. In Linear programming and extensions. Princeton university press. 2016. 518 p.

2. Гірна О.; Петляківський О. Транспортні перевезення: виклики сьогодення. Академічні візії, 2024, 27с.

3. Дроздова В. А. Транспортні перевезення в логістичних системах. Економічні та соціальні аспекти розвитку України на початку XXI століття. Матеріали V Міжнародної науково-практичної конференції 12-13 жовтня 2017р. Одеса: Одеська національна академія харчових технологій, 2017. – 291 с.

4. УСТЕНКО М. О. Основні проблеми транспортної логістики. Вісник економіки транспорту і промисловості, 2010, 29: 236-238с.

5. Соловйова О. О.Загальний курс транспорту : навч. посібник – К. : НАУ, 2019. – 244 с.

6. Литвинов В. В.; Дехтярук М. Т. Імітаційне моделювання та оптимізація перевезень у транспортних логістичних системах. Сьома міжнародна науково-практична конференція «Математичне та імітаційне моделювання систем». Тези доповідей.–Чернігів-Жукин.–2012.–25-28 червня, 2012, 94-98.

7. Boyd Stephen; Mattingley Jacob. Branch and bound methods. Notes for EE364b, Stanford University, 2007, 2006: 07.

8. Парфьонова Т. О. Застосування методу гілок та меж до комбінаторних транспортних задач. 2008. 29с.

9. Погорілий С. Д.; Бойко Ю. В.; Білоус Р. В. Формування та аналіз паралельних схем алгоритму Дейкстри. 2008, 1.4: 61-72.

10. YU Kai; JI Liang; ZHANG Xuegong. Kernel nearest-neighbor algorithm. Neural Processing Letters, 2002, 15: 147-156.

11. Kizilates-Evin Gozde & Nuriyeva, Fidan. (2013). On the Nearest Neighbor Algorithms for the Traveling Salesman Problem. Advances in Intelligent Systems and Computing. 225. 111-118. URL: http://doi.org/10.1007/978-3-319-00951-3_11

12. Route Planning Software for Seamless Optimization. URL: https://www.loginextsolutions.com/products/mile/route-planning-software (дата звернення 06.12.2024).

13. LogiNext Mile URL: https://www.loginextsolutions.com/products/mile (дата звернення 06.12.2024).

14. LogiNext Mile Software Reviews, Pros and Cons URL: https://www.softwareadvice.com/fleet-management/loginext-mile-profile/reviews/ (дата звернення 06.12.2024).

15. Paragon Single Depot. URL: https://www.capterra.co.uk/software/91607/paragon-single-depot (дата звернення 06.12.2024).

16. Advanced Routing and Scheduling Software - Paragon routing URL: https://www.paragonrouting.com/en-us/ (дата звернення 06.12.2024).

17. Paragon Pros and Cons URL:https://www.g2.com/products/paragon/reviews?qs=pros-and-cons (дата звернення 06.12.2024).

18. Onfleet URL: https://onfleet.com/ (дата звернення 06.12.2024).

19. Learn About The Onfleet Last Mile Delivery App | Onfleet. URL: https://onfleet.com/learn-more (дата звернення 06.12.2024).

20. Onfleet Reviews: Features, Pricing, Pros & Cons - 2024 URL: https://getcircuit.com/teams/blog/onfleet-reviews (дата звернення 06.12.2024).

21. Бондаренко В.В., Коваленко В.В. Розробка програмного забезпечення з використанням Python, Java та C#: Збірник наукових праць. - Одеса: Видавництво "Сонячна Україна", 2018. - 250 с.

22. Kishori Sharan. Learn JavaFX 8 (The Expert’s Voice in Java): Building User Experience and Interfaces with Java 8. – New York, 2015. – 1173 p.

23. Безменов М.І., Безменова О.М., Калінін Д.В. Основи візуального програмування мовою C#: навч. посіб. для студентів навчально-наукового інституту комп’ютерних наук та інформаційних технологій. – Харків: ФОП Панов А. М., 2023. 648 с.

24. Microsoft Visual Studio Reviews. URL: https://www.softwareadvice.com/game-development/microsoft-visual-studio-profile/reviews/ (дата звернення 09.12.2024).

25. Kahlert T., Giza K. Visual Studio Code Tips & Tricks. Microsoft Deutschland GmbH. 2016. Vol. 1. 26 p.

26. Most Helpful Rider Reviews. URL: https://www.gartner.com/reviews/market/integrated-development-environment-ide-software/vendor/jetbrains/product/rider (дата звернення 09.12.2024).

27. Томашевський О. М. Інформаційні технології та моделювання бізнес-процесів : навч. посіб. / О. М. Томашевський, Г. Г. Цигелик, М. Б. Вітер, В. І. Дудук. – К. : Центр учбової літератури, 2012. – 296 с.

28. Томашевський О. М. Інформаційні технології та моделювання бізнес-процесів : навч. посіб. / О. М. Томашевський, Г. Г. Цигелик, М. Б. Вітер, В. І. Дудук. – К. : Центр учбової літератури, 2012. – 296 с.

29. Гладун А.В. Трьохрівнева архітектура побудови веб-систем. Вісник Національного університету "Львівська політехніка". 2018. – 315 с.

30. омашевський О. М. Інформаційні технології та моделювання бізнес-процесів : навч. посіб. / О. М. Томашевський, Г. Г. Цигелик, М. Б. Вітер, В. І. Дудук. – К. : Центр учбової літератури, 2012. – 296 с.

31. Fauzan, R., Siahaan, D., Rochimah, S., & Triandini, E. A different approach on automated use case diagram semantic assessment. International Journal of Intelligent Engineering and Systems, 14(1), 2021. – pp. 496-505.

ДОДАТКИ

Додаток А. Діаграма бази даних системи

```
-- Таблиця Simulations
CREATE TABLE Simulations (
    SimulationsId INT PRIMARY KEY IDENTITY(1,1), -- Унікальний ідентифікатор симуляції
    SimulationsName NVARCHAR(250) NOT NULL,      -- Назва симуляції
    Description NVARCHAR(MAX) NULL,              -- Опис
    CountPeaks INT NOT NULL                      -- Кількість вершин
);

-- Таблиця Users
CREATE TABLE Users (
    UsersId INT PRIMARY KEY IDENTITY(1,1), -- Унікальний ідентифікатор користувача
    FirstName NVARCHAR(250) NOT NULL,      -- Ім’я
    LastName NVARCHAR(250) NOT NULL,       -- Прізвище
    UsersName NVARCHAR(250) NOT NULL UNIQUE, -- Унікальне ім’я користувача
    UsersPassword NVARCHAR(250) NOT NULL,  -- Пароль
    RoleId INT NOT NULL,                   -- Ідентифікатор ролі
    Description NVARCHAR(MAX) NULL         -- Додаткова інформація
);

-- Таблиця Cities
CREATE TABLE Cities (
    CitiesId INT PRIMARY KEY IDENTITY(1,1), -- Ідентифікатор міста або населеного пункту
```

```
    CitiesName NVARCHAR(250) NOT NULL,      -- Назва міста
    Description NVARCHAR(MAX) NULL          -- Додаткова інформація
);

-- Таблиця EdgeS
CREATE TABLE EdgeS (
    EdgeSId INT PRIMARY KEY IDENTITY(1,1),      -- Унікальний ідентифікатор зв’язку
    RoutStart NVARCHAR(250) NOT NULL,           -- Початковий пункт маршруту
    RoutEnd NVARCHAR(250) NOT NULL,         -- Кінцевий пункт маршруту
    Distance FLOAT NOT NULL,                -- Відстань між пунктами
    SimulationsId INT NOT NULL,             -- Ідентифікатор симуляції
    FOREIGN KEY (SimulationsId) REFERENCES Simulations(SimulationsId)
);

-- Таблиця Logs
CREATE TABLE Logs (
    LogsId INT PRIMARY KEY IDENTITY(1,1),  -- Унікальний ідентифікатор запису
    UsersId INT NOT NULL,                   -- Ідентифікатор користувача
    EventNameShow NVARCHAR(250) NOT NULL,  -- Назва події
    EventDate DATETIME NOT NULL,            -- Дата і час події
    FOREIGN KEY (UsersId) REFERENCES Users(UsersId)
);

-- Таблиця Vertexs
CREATE TABLE Vertexs (
    VertexsId INT PRIMARY KEY IDENTITY(1,1), -- Унікальний ідентифікатор вершини
```

```
    VertexName NVARCHAR(250) NOT NULL,      -- Назва вершини
    SimulationsId INT NOT NULL,             -- Ідентифікатор симуляції
    FOREIGN KEY (SimulationsId) REFERENCES Simulations(SimulationsId)
);
```

Додаток Б. Код програмної системи

Лістинг 1. Код класу «DijkstraBLL»

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace DecisionSupportApp.AppCode {
  internal class DijkstraBLL {
  }
}

/// <summary>
/// Інформація про вершину
/// </summary>
public class GraphVertexInfo {
  /// <summary>
  /// Вершина
  /// </summary>
  public GraphVertex Vertex { get; set; }

  /// <summary>
  /// Не відвідана вершина
  /// </summary>
  public bool IsUnvisited { get; set; }

  /// <summary>
  /// Сума ваг ребер
  /// </summary>
  public int EdgesWeightSum { get; set; }

  /// <summary>
  /// Попередня вершина
  /// </summary>
  public GraphVertex PreviousVertex { get; set; }

  /// <summary>
  /// Конструктор
  /// </summary>
  /// <param name="vertex">Вершина</param>
  public GraphVertexInfo(GraphVertex vertex) {
    Vertex = vertex;
    IsUnvisited = true;
    EdgesWeightSum = int.MaxValue;
    PreviousVertex = null;
  }
}
```

```
/// <summary>
/// Граф
/// </summary>
public class Graph {
  /// <summary>
  /// Список вершин графа
  /// </summary>
  public List<GraphVertex> Vertices { get; set; }

  /// <summary>
  /// Конструктор
  /// </summary>
  public Graph() {
    Vertices = new List<GraphVertex>();
  }

  /// <summary>
  /// Додавання вершини
  /// </summary>
  /// <param name="vertexName">Імя вершини</param>
  public void AddVertex(string vertexName) {
    Vertices.Add(new GraphVertex(vertexName));
  }

  /// <summary>
  /// Поиск вершины
  /// </summary>
  /// <param name="vertexName">Назва вершини</param>
  /// <returns>Знайдена вершина</returns>
  public GraphVertex FindVertex(string vertexName) {
    foreach (var v in Vertices) {
      if (v.Name.Equals(vertexName)) {
        return v;
      }
    }

    return null;
  }

  /// <summary>
  /// Додавання ребра
  /// </summary>
  /// <param name="firstName">Імя першої вершини</param>
  /// <param name="secondName">Імя другої вершини</param>
  /// <param name="weight">Вага ребра зєдуючого вершини</param>
  public void AddEdge(string firstName, string secondName, int weight) {
    var v1 = FindVertex(firstName);
    var v2 = FindVertex(secondName);
    if (v2 != null && v1 != null) {
      v1.AddEdge(v2, weight);
      v2.AddEdge(v1, weight);
    }
```

```
  }
}

/// <summary>
/// Вершина графа
/// </summary>
public class GraphVertex {
  /// <summary>
  /// Назва вершини
  /// </summary>
  public string Name { get; set; }

  /// <summary>
  /// Список ребер
  /// </summary>
  public List<GraphEdge> Edges { get; set; }

  /// <summary>
  /// Конструктор
  /// </summary>
  /// <param name="vertexName">Назва вершини</param>
  public GraphVertex(string vertexName) {
    Name = vertexName;
    Edges = new List<GraphEdge>();
  }

  /// <summary>
  /// Додати ребро
  /// </summary>
  /// <param name="newEdge">Ребро</param>
  public void AddEdge(GraphEdge newEdge) {
    Edges.Add(newEdge);
  }

  /// <summary>
  /// Додати ребро
  /// </summary>
  /// <param name="vertex">Вершина</param>
  /// <param name="edgeWeight">Вага</param>
  public void AddEdge(GraphVertex vertex, int edgeWeight) {
    AddEdge(new GraphEdge(vertex, edgeWeight));
  }

  /// <summary>
  /// Преобразование в строку
  /// </summary>
  /// <returns>Имя вершины</returns>
}

/// <summary>
/// Ребро графа
/// </summary>
```

```
public class GraphEdge {
  /// <summary>
  /// Зв'язана вершина
  /// </summary>
  public GraphVertex ConnectedVertex { get; set; }

  /// <summary>
  /// Вага ребра
  /// </summary>
  public int EdgeWeight { get; set; }

  /// <summary>
  /// Конструктор
  /// </summary>
  /// <param name="connectedVertex">Зв'язана  вершина</param>
  /// <param name="weight">Вага ребра</param>
  public GraphEdge(GraphVertex connectedVertex, int weight) {
    ConnectedVertex = connectedVertex;
    EdgeWeight = weight;
  }
}

public class Dijkstra {
  Graph graph;
  List<GraphVertexInfo> infos;
  /// <summary>
  /// Конструктор
  /// </summary>
  /// <param name="graph">Граф</param>
  public Dijkstra(Graph graph) {
    this.graph = graph;
  }
  /// <summary>
  /// Ініціалізація інформації
  /// </summary>
  void InitInfo() {
    infos = new List<GraphVertexInfo>();
    foreach (var v in graph.Vertices) {
      if (v == null) {
        throw new InvalidOperationException("Граф містить null-вершини.");
      }
      infos.Add(new GraphVertexInfo(v));
    }
  }

  /// <summary>
  /// Дістати інформацію про вершину графа графа
  /// </summary>
  /// <param name="v">Вершина</param>
  /// <returns>Інформація про вершину</returns>
  GraphVertexInfo GetVertexInfo(GraphVertex v) {
    var info = infos.FirstOrDefault(i => i.Vertex == v);
```

```
    if (info == null) {
      throw new InvalidOperationException($"Інформація для вершини {v.Name} не знайдена.");
    }
    return info;
  }

  /// <summary>
  /// Пошук невідвіданої вершини з мінімальним значенням суми
  /// </summary>
  /// <returns>Інформація про вершину</returns>
  public GraphVertexInfo FindUnvisitedVertexWithMinSum() {
    var minValue = int.MaxValue;
    GraphVertexInfo minVertexInfo = null;
    foreach (var i in infos) {
      if (i.IsUnvisited && i.EdgesWeightSum < minValue) {
        minVertexInfo = i;
        minValue = i.EdgesWeightSum;
      }
    }

    return minVertexInfo;
  }

  /// <summary>
  /// Пошук найкоротшого шляху за назвами вершин
  /// </summary>
  /// <param name="startName">Назва стартової вершини/param>
  /// <param name="finishName">Назва фінішної вершини</param>
  /// <returns>Кратчайший путь</returns>
  public string FindShortestPath(string startName, string finishName) {
    // Перевірка вхідних даних
    var startVertex = graph.FindVertex(startName);
    if (startVertex == null) {
      throw new ArgumentException($"Стартова вершина з ім'ям '{startName}' не знайдена у
графі.");
    }
    var finishVertex = graph.FindVertex(finishName);
    if (finishVertex == null) {
      throw new ArgumentException($"Кінцева вершина з ім'ям '{finishName}' не знайдена у
графі.");
    }
    return FindShortestPath(startVertex, finishVertex);
  }


  /// <summary>
  /// Пошук найкоротшого шляху по вершинах
  /// </summary>
  /// <param name="startVertex">Стартова вершина</param>
  /// <param name="finishVertex">Фінішна вершина</param>
  /// <returns>Найкоротний шлях</returns>
  public string FindShortestPath(GraphVertex startVertex, GraphVertex finishVertex) {
```

```
  InitInfo();
  var first = GetVertexInfo(startVertex);
  if (first == null) {
    throw new ArgumentException($"Вершина {startVertex.Name}" +
      $" не знайдена у графі.");
  }
  first.EdgesWeightSum = 0;
  while (true) {
    var current = FindUnvisitedVertexWithMinSum();
    if (current == null) {
      break;
    }
    SetSumToNextVertex(current);
  }
  // Перевірка доступності шляху
  var finishInfo = GetVertexInfo(finishVertex);
  if (finishInfo == null || finishInfo.EdgesWeightSum == int.MaxValue) {
    throw new InvalidOperationException($"Неможливо знайти шлях між '" +
      $"{startVertex.Name}' та '{finishVertex.Name}'.");
  }
  return GetPath(startVertex, finishVertex);
}



/// <summary>
/// Обчислення суми ваг ребер для наступної вершини
/// </summary>
/// <param name="info">Інформація про поточну вершину</param>
void SetSumToNextVertex(GraphVertexInfo info) {
  info.IsUnvisited = false;
  foreach (var e in info.Vertex.Edges) {
    var nextInfo = GetVertexInfo(e.ConnectedVertex);
    var sum = info.EdgesWeightSum + e.EdgeWeight;
    if (sum < nextInfo.EdgesWeightSum) {
      nextInfo.EdgesWeightSum = sum;
      nextInfo.PreviousVertex = info.Vertex;
    }
  }
}

/// <summary>
///Формування шляху
/// </summary>
/// <param name="startVertex">Початкова вершина</param>
/// <param name="endVertex">Кінцева вершина</param>
/// <returns>Шлях</returns>
string GetPath(GraphVertex startVertex, GraphVertex endVertex) {
  if (startVertex == null || endVertex == null) {
    throw new ArgumentNullException("Стартова або кінцева вершина не задана.");
  }
  var path = new List<string>();
```

```
    var currentVertex = endVertex;
    while (currentVertex != null) {
      path.Add(currentVertex.Name);
      var vertexInfo = GetVertexInfo(currentVertex);
      if (vertexInfo == null || vertexInfo.PreviousVertex == null) {
        break; // Відсутній шлях до стартової вершини
      }
      currentVertex = vertexInfo.PreviousVertex;
    }
    if (!path.Contains(startVertex.Name)) {
      throw new InvalidOperationException("Не вдалося побудувати шлях: вершини не з'єднані.");
    }
    path.Reverse();
    return string.Join(" -> ", path);
  }

}
```

Лістинг 2. Код класу «SimulatorForm»

```
using DecisionSupportApp.AppCode;
using DecisionSupportApp.Forms.Systems;
using DecisionSupportApp.Provider;
using DecisionSupportApp.Providers;
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using System.Windows.Forms.VisualStyles;

namespace DecisionSupportApp.Forms.Controls {
  public partial class SimulatorForm : Form {
    Random _GetRandom = new Random();

    private Graph _Graph;
    private Dijkstra _Dijkstra;
    private CitiesProvider _CitiesProvider = new CitiesProvider();
    private List<Cities> _CitiesList = new List<Cities>();
    private VertexsProvider _VertexsProvider = new VertexsProvider();
    private List<Vertexs> _VertexsList = new List<Vertexs>();
    private EdgeSProvider _EdgeSProvider = new EdgeSProvider();
    private List<EdgeS> _EdgeSList = new List<EdgeS>();
    private ValidationMy _validation = new ValidationMy();

    public SimulatorForm() {
      InitializeComponent();
      LoadAllData();
    }
```

```
    private void GenetationBtn_Click(object sender, EventArgs e) {
      if (IsDataEnteringCorrect()) {
        _Graph = new Graph();
        _VertexsList.Clear();
        _EdgeSList.Clear();
        VertexCBox.Items.Clear();
        //Генеруємо міста
        _VertexsList = GetAllVertexsList(Convert.ToInt32(CountPeaksTBox.Text));
        LoadData();
      }
    }

    private void SearchBtn_Click(object sender, EventArgs e) {
      if (VertexCBox.Text != "") {
        _Dijkstra = new Dijkstra(_Graph);
        ResultTBox.Text = "Найкоротші маршрути від заданого місця, до всіх інших місць: \r\n";
        for (int i = 0; i < Convert.ToInt32(CountPeaksTBox.Text); i++) {
          if (VertexCBox.Text != i.ToString()) {
            var path = _Dijkstra.FindShortestPath(VertexCBox.Text, _VertexsList[i].VertexsName);
            if (path != VertexCBox.Text) {
              ResultTBox.Text += path + "\r\n";
              ResultTBox.Text += "-----------------------------\r\n";
            }
          }
        }
      }
    }

    private void AddBtn_Click(object sender, EventArgs e) {
      if (IsDataSaveCorrect()) {
        LogsProvider logsProvider = new LogsProvider();
        SimulationsProvider simulationsProvider = new SimulationsProvider();
        simulationsProvider.InsertSimulations(SimulationsNameTBox.Text, DescriptionTBox.Text,
          Convert.ToInt32(CountPeaksTBox.Text));
        int lastSimulationsId = simulationsProvider.GetLastRecords();
        for (int i = 0; i < _VertexsList.Count; i++) {
          _VertexsList[i].SimulationsId = lastSimulationsId;
        }
        for (int i = 0; i < _EdgeSList.Count; i++) {
          _EdgeSList[i].SimulationsId = lastSimulationsId;
        }
        _VertexsProvider.InsertBatchVertexs(_VertexsList);
        _EdgeSProvider.InsertBatchEdgeS(_EdgeSList);
        MessageBox.Show("Симуляцію успішно збережено!");
        logsProvider.InsertLogs(LoginForm.CurrentUser.UsersId, "Було збережено симуляцію під
назвою: " +
         SimulationsNameTBox.Text, DateTime.Now);
        SimulationsNameTBox.Text = String.Empty;
        DescriptionTBox.Text = String.Empty;
      }
    }
```

```
private void ClearBtn_Click(object sender, EventArgs e) {
  SimulationsNameTBox.Text = String.Empty;
  DescriptionTBox.Text = String.Empty;
}

private void ExitBtn_Click(object sender, EventArgs e) {
  this.Close();
}

private void LoadAllData() {
  _CitiesList = _CitiesProvider.GetAllCities();
}

private bool IsDataEnteringCorrect() {
  bool isCorrect = true;
  if (_validation.IsDataInThisScope(3, 30, Convert.ToInt32(CountPeaksTBox.Text))) {
    CountPeaksValiadtionLbl.Text = NamesMy.ProgramButtons.RequiredValidation;
  } else {
    CountPeaksValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
    isCorrect = false;
  }
  return isCorrect;
}


public void LoadData() {
  List<EdgeS> edgeSList = new List<EdgeS>();
  _EdgeSList.Clear();
  DescriptionTBox.Clear(); // Очистка текстового поля перед початком операцій.

  // Додавання вершин
  DescriptionTBox.AppendText("Додавання вершин:\r\n");
  for (int i = 0; i < _VertexsList.Count; i++) {
    _Graph.AddVertex(_VertexsList[i].VertexsName);
    VertexCBox.Items.Add(_VertexsList[i].VertexsName);
    DescriptionTBox.AppendText($"- Вершина '{_VertexsList[i].VertexsName}' додана.\r\n");
  }
  VertexCBox.SelectedIndex = 0;

  // Додавання ребер
  DescriptionTBox.AppendText("\nДодавання ребер:\r\n");
  for (int i = 0; i < _VertexsList.Count; i++) {
    int genRoutCount = _GetRandom.Next(0, _VertexsList.Count - 1) + 1;
    int count = 0;
    while (count < genRoutCount) {
      EdgeS oneEdgeS = new EdgeS();
      int genCitiesId = _GetRandom.Next(0, _VertexsList.Count);
      if (i != genCitiesId) {
        oneEdgeS.RoutStart = _VertexsList[i].VertexsName;
        oneEdgeS.RoutEnd = GetCitiesName(genCitiesId, _VertexsList);
        oneEdgeS.Distance = _GetRandom.Next(1, 20) * 10;
```

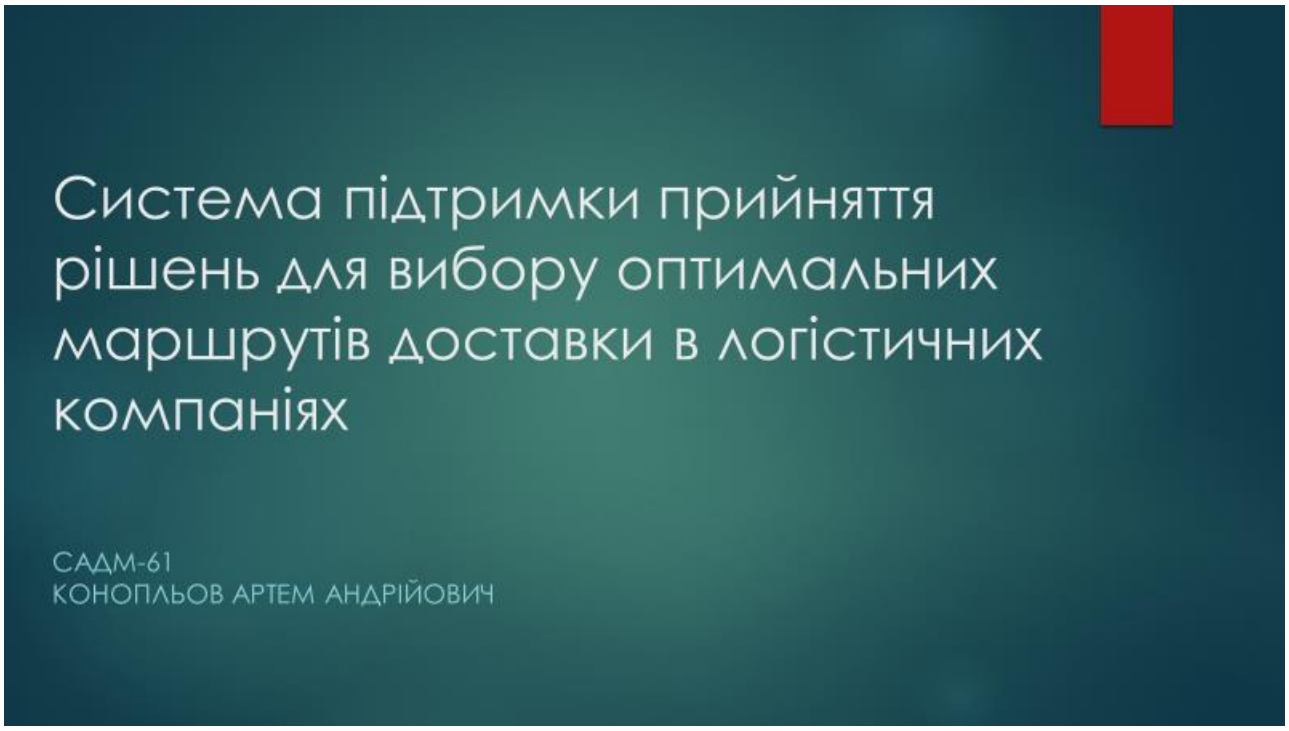
```
          if (!IsThisRoutExist(oneEdgeS, edgeSList)) {
            edgeSList.Add(oneEdgeS);
            count++;
            DescriptionTBox.AppendText($"- Ребро: {oneEdgeS.RoutStart} -> {oneEdgeS.RoutEnd},
відстань: {oneEdgeS.Distance}.\r\n");
          }
        }
      }
    }

    // Оновлення списку ребер
    DescriptionTBox.AppendText("\nОновлення списку ребер:\r\n");
    for (int i = 0; i < edgeSList.Count; i++) {
      EdgeS oneEdgeS = new EdgeS();
      oneEdgeS.Number = i + 1;
      oneEdgeS.RoutStart = edgeSList[i].RoutStart;
      oneEdgeS.RoutEnd = edgeSList[i].RoutEnd;
      int distanse = GetDisnance(oneEdgeS, _EdgeSList);
      if (distanse > -1) {
        oneEdgeS.Distance = distanse;
      } else {
        oneEdgeS.Distance = edgeSList[i].Distance;
      }
      _EdgeSList.Add(oneEdgeS);
      DescriptionTBox.AppendText($"- Оновлено ребро: {oneEdgeS.RoutStart} ->
{oneEdgeS.RoutEnd}, відстань: {oneEdgeS.Distance}.\r\n");
    }

    // Додавання ребер у граф
    DescriptionTBox.AppendText("\nДодавання ребер у граф:\r\n");
    for (int i = 0; i < _EdgeSList.Count; i++) {
      _Graph.AddEdge(_EdgeSList[i].RoutStart, _EdgeSList[i].RoutEnd, _EdgeSList[i].Distance);
      DescriptionTBox.AppendText($"- Ребро додано у граф: {_EdgeSList[i].RoutStart} ->
{_EdgeSList[i].RoutEnd}, відстань: {_EdgeSList[i].Distance}.\r\n");
    }

    // Завантаження даних у GridView
    DescriptionTBox.AppendText("\nЗавантаження даних у таблицю:\r\n");
    LoadDataInEdgeSGridView(_EdgeSList);
    DescriptionTBox.AppendText("Завантаження завершено.\r\n");
  }


  private void LoadDataInEdgeSGridView(List<EdgeS> EdgeSList) {
    EdgeSGridView.DataSource = null;
    EdgeSGridView.Columns.Clear();
    EdgeSGridView.AutoGenerateColumns = false;
    EdgeSGridView.RowHeadersVisible = false;

    EdgeSGridView.DataSource = EdgeSList;

    if (EdgeSList.Count > 0) {
```

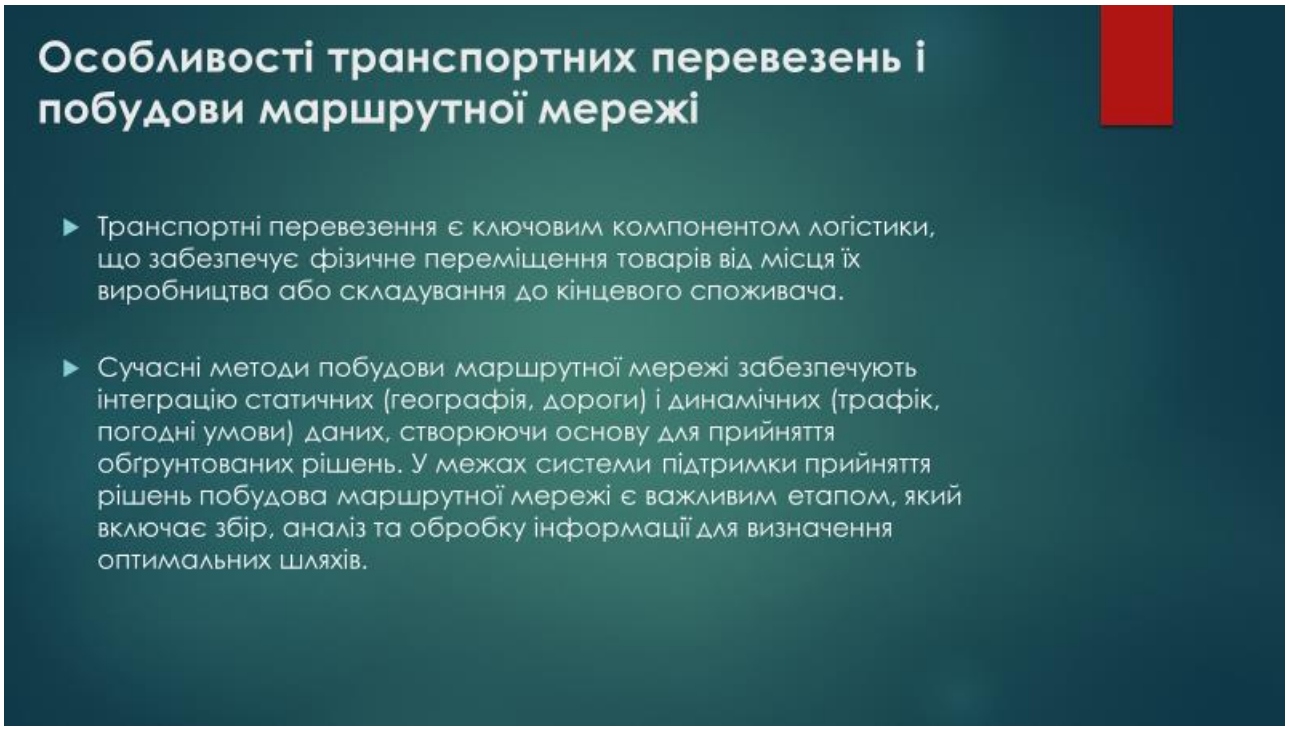
```
    if (EdgeSList[0].Message == NamesMy.NoDataNames.NoDataInEdgeS) {
      DataGridViewColumn messageColumn = new DataGridViewTextBoxColumn();
      messageColumn.DataPropertyName = "Message";
      messageColumn.Width = EdgeSGridView.Width - NamesMy.SizeOptins.MinusSizePanel;
      EdgeSGridView.Columns.Add(messageColumn);
    } else {
      DataGridViewColumn EdgeSIdColumn = new DataGridViewTextBoxColumn();
      EdgeSIdColumn.DataPropertyName = "EdgeSId";
      EdgeSGridView.Columns.Add(EdgeSIdColumn);
      EdgeSGridView.Columns[0].Visible = false;

      DataGridViewColumn numberColumn = new DataGridViewTextBoxColumn();
      numberColumn.HeaderText = "№ ";
      numberColumn.DataPropertyName = "Number";
      numberColumn.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
      numberColumn.Width = 50;
      numberColumn.ReadOnly = true;
      EdgeSGridView.Columns.Add(numberColumn);

      DataGridViewColumn RoutStartColumn = new DataGridViewTextBoxColumn();
      RoutStartColumn.HeaderText = "Початок";
      RoutStartColumn.DataPropertyName = "RoutStart";
      RoutStartColumn.Name = "RoutStart";
      RoutStartColumn.Width = 120;
      RoutStartColumn.ReadOnly = true;
      EdgeSGridView.Columns.Add(RoutStartColumn);

      DataGridViewColumn RoutEndColumn = new DataGridViewTextBoxColumn();
      RoutEndColumn.HeaderText = "Кінець";
      RoutEndColumn.DataPropertyName = "RoutEnd";
      RoutEndColumn.Name = "RoutEnd";
      RoutEndColumn.Width = 120;
      RoutEndColumn.ReadOnly = true;
      EdgeSGridView.Columns.Add(RoutEndColumn);

      DataGridViewColumn DistanceColumn = new DataGridViewTextBoxColumn();
      DistanceColumn.HeaderText = "Відстань";
      DistanceColumn.DataPropertyName = "Distance";
      DistanceColumn.Name = "Distance";
      DistanceColumn.DefaultCellStyle.Alignment = DataGridViewContentAlignment.MiddleRight;
      DistanceColumn.Width = 65;
      DistanceColumn.ReadOnly = false;
      EdgeSGridView.Columns.Add(DistanceColumn);

    }
    for (int i = 0; i < EdgeSGridView.Columns.Count; i++) {
      EdgeSGridView.Columns[i].HeaderCell.Style.BackColor = Color.LightGray;
    }
  }
}

private int GetDisnance(EdgeS CurEdgeS, List<EdgeS> EdgeSList) {
```

```
    for (int i = 0; i < EdgeSList.Count; i++) {
      if (CurEdgeS.RoutStart == EdgeSList[i].RoutEnd && CurEdgeS.RoutEnd ==
EdgeSList[i].RoutStart) {
        return EdgeSList[i].Distance;
      }
    }
    return -1;
  }

  private string GetCitiesName(int CitiesId, List<Vertexs> Vertexs) {
    return Vertexs[CitiesId].VertexsName;
  }

  private bool IsThisRoutExist(EdgeS genEdgex, List<EdgeS> EdgeList) {
    for (int i = 0; i < EdgeList.Count; i++) {
      if (genEdgex.RoutStart == EdgeList[i].RoutStart && genEdgex.RoutEnd ==
EdgeList[i].RoutEnd) {
        return true;
      }
    }
    return false;
  }

  private List<Vertexs> GetAllVertexsList(int PeaksCount) {
    List<Vertexs> selVertexsList = new List<Vertexs>();
    int count = 0;

    while (count < PeaksCount) {
      int randCitiesId = _GetRandom.Next(0, _CitiesList.Count - 1);
      if (!IsCitiesPeakExistInList(randCitiesId, selVertexsList)) {
        Cities oneCities = new Cities();
        oneCities = GetCities(randCitiesId, _CitiesList);
        count++;
        Vertexs oneVertexs = new Vertexs();
        oneVertexs.CitiesId = oneCities.CitiesId;
        oneVertexs.VertexsName = oneCities.CitiesName;
        selVertexsList.Add(oneVertexs);
      }
    }
    return selVertexsList;
  }

  private bool IsCitiesPeakExistInList(int CitiesId, List<Vertexs> Vertexs) {
    for (int i = 0; i < Vertexs.Count; i++) {
      if (CitiesId == Vertexs[i].CitiesId) {
        return true;
      }
    }
    return false;
  }

  private Cities GetCities(int CitiesId, List<Cities> CitiesList) {
```

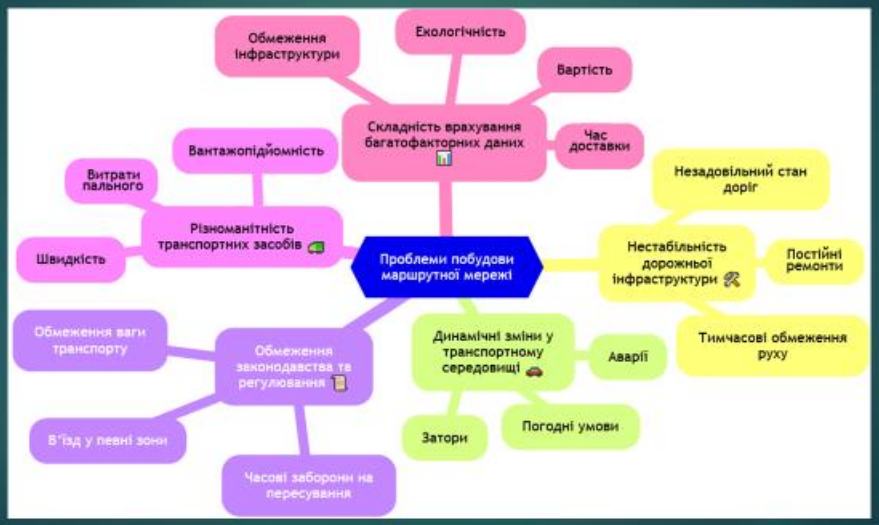
```
      for (int i = 0; i < CitiesList.Count; i++) {
        if (CitiesId == CitiesList[i].CitiesId) {
          return CitiesList[i];
        }
      }
      return null;
    }

    private bool IsDataSaveCorrect() {
      bool isCorrect = true;
      if (_validation.IsDataEntering(SimulationsNameTBox.Text)) {
        SimulationsNameValiadtionLbl.Text = NamesMy.ProgramButtons.RequiredValidation;
      } else {
        SimulationsNameValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
        isCorrect = false;
      }
      if (_validation.IsDataInThisScope(3, 30, Convert.ToInt32(CountPeaksTBox.Text))) {
        CountPeaksValiadtionLbl.Text = NamesMy.ProgramButtons.RequiredValidation;
      } else {
        CountPeaksValiadtionLbl.Text = NamesMy.ProgramButtons.ErrorValidation;
        isCorrect = false;
      }
      return isCorrect;
    }

  }
}
```

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ (Презентація)





Основні проблеми, які виникають у процесі побудови маршрутної мережі.



Проблеми, які виникають у процесі побудови маршрутної мережі

- Нестабільність дорожньої інфраструктури. Наявність незадовільного стану доріг, постійні ремонти та тимчасові обмеження руху;
- Динамічні зміни у транспортному середовищі. Затори, аварії та погодні умови, що змінюють доступність маршрутів у реальному часі;
- Обмеження законодавства та регулювання. Обмеження ваги транспорту, часові заборони на пересування або в'їзд у певні зони;
- Різноманітність транспортних засобів. Відмінності у вантажопідйомності, швидкості та витратах пального для різних типів транспорту;
- Складність врахування багатофакторних даних. Необхідність одночасного врахування вартості, часу доставки, екологічності та обмежень інфраструктури.

Ключові вимоги до побудови маршрутної мережі

Проблеми, які виникають у процесі побудови маршрутної мережі

- Нестабільність дорожньої інфраструктури. Наявність незадовільного стану доріг, постійні ремонти та тимчасові обмеження руху;
- Динамічні зміни у транспортному середовищі. Затори, аварії та погодні умови, що змінюють доступність маршрутів у реальному часі;
- Обмеження законодавства та регулювання. Обмеження ваги транспорту, часові заборони на пересування або в'їзд у певні зони;
- Різноманітність транспортних засобів. Відмінності у вантажопідйомності, швидкості та витратах пального для різних типів транспорту;
- Складність врахування багатофакторних даних. Необхідність одночасного врахування вартості, часу доставки, екологічності та обмежень інфраструктури.

Ключові вимоги до побудови маршрутної мережі

Аналіз методів та алгоритмів для вибору оптимального маршруту

У логістичних системах вибір оптимального маршруту є однією з найважливіших задач, оскільки саме від неї залежить ефективність транспортування вантажів