Міністерство освіти і науки україни

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних наук та технологій

Кафедра комп'ютеризованих систем управління

ДОПУСТИТИ ДО ЗАХИСТУ
Завідувач кафедри

_____Олександр ЛИТВИНЕНКО

«_____»_____2023р.

КВАЛІФІКАЦІЙНА РОБОТА
(ПОЯСНЮВАЛЬНА ЗАПИСКА)

ЗДОБУВАЧА ВИЩОЇ ОСВІТИ
ОСВІТНЬОГО СТУПЕНЯ «МАГІСТР»

Тема: Програмний модуль побудови маршрутів авіаційних перевезень

Виконавець: _____Данило СКОВПЕНЬ

Керівник: _____Олександр ЛИТВИНЕНКО

Нормоконтролер: _____Євгеній ТУПОТА

Київ 2023

НАЦІОНАЛЬНИЙ АВІАЦІЙНИЙ УНІВЕРСИТЕТ

Факультет комп'ютерних наук та технологій
Кафедра комп'ютеризованих систем управління
Спеціальність 123 «Комп'ютерна інженерія»
(шифр, найменування)

Освітньо-професійна програма «Системне програмування»
Форма навчання денна

ЗАТВЕРДЖУЮ
Завідувач кафедри

Олександр ЛИТВИНЕНКО

«____»_____ 2023р.

ЗАВДАННЯ
на виконання кваліфікаційної роботи
Сковпня Данила Васильовича
(прізвище, ім'я, по батькові випускника в родовому відмінку)

1. **Тема кваліфікаційної роботи:** «Програмний модуль побудови маршрутів авіаційних перевезень» затверджена наказом ректора від «28»серпня 2023р.№1494/ст.

2. Термін виконання роботи (проєкту): з 02.10.2023 р. по 24.12.2023 р.

3. Вихідні дані до роботи (проєкту): технологія маршрутизації авіаційних перевезень.

4. Зміст пояснювальної записки:

1. Аналітичний огляд літературних джерел з тематики кваліфікаційної роботи.

2. Дослідження евристичного методу розв'язання задачі комівояжера.

3. Застосування методу комівояжера до розв'язання задач побудови маршрутів авіаційних перевезень.

4. Розробка програмного модуля побудови маршрутів авіаційних перевезень.

5. Перелік обов'язкового графічного (ілюстративного) матеріалу:

1. Лексикографічне впорядкування(схема алгоритму);

2. Інтерфейс користувача застосунку;

3. Діаграма класів застосунку.

6. Календарний план-графік

№ пор.	Завдання	Термін виконання	Відмітка про виконання
1	Дослідити технологію організації авіаційних перевезень	02.10.2023–05.10.2023	
2	Проаналізувати існуючі методи маршрутизації авіаційних перевезень	06.10.2023–13.10.2023	
3	Дослідити евристичні методи побудови маршрутів	14.10.2023–02.11.2023	
4	Адаптувати евристичний алгоритм комівояжера до розв’язання задачі маршрутизації авіаційних перевезень	02.11.2023–28.11.23	
5	Визначити технічне та інформаційне забезпечення програмного модуля маршрутизації авіаційних перевезень	29.11.2023–01.12.2023	
6	Здійснити програмну реалізацію евристичного алгоритму побудови маршрутів авіаційних перевезень	02.12.2023–10.12.2023	
7	Розробити інтерфейс програмного модуля маршрутизації авіаційних перевезень	11.12.2023–14.12.2023	
8	Скласти та надати керівнику пояснювальну записку кваліфікаційної роботи у повному обсязі, а також текст та презентацію доповіді	19.12.2023–20.12.2023	

7. Дата видачі завдання: «02» жовтня 2023 р.

Керівник кваліфікаційної роботи _____ Олександр ЛИТВИНЕНКО
(підпис керівника) (П.І.Б.)

Завдання прийняв до виконання _____ Данило СКОВПЕНЬ
(підпис студента) (П.І.Б.)

РЕФЕРАТ

Пояснювальна записка до кваліфікаційної роботи “Програмний модуль побудови маршрутів авіаційних перевезень”: 80 сторінок, 7 рисунків, 2 таблиці, 28 використаних джерел.

ЕВРЕСТИЧНИЙ МЕТОД, ЛЕКСИКОГРАФІЧНЕ ВПОРЯДКУВАННЯ, АЛГОРИТМ ГЕНЕРАЦІЇ ПЕРЕСТАНОВОК, ЗАМІНА, АВІАЦІЙНІ ПЕРЕВЕЗЕННЯ.

Мета дослідження: зменшення витрат на авіаційні перевезення.

Об’єкт дослідження: процес планування авіаційних перевезень.

Предмет дослідження: програмний модуль маршрутизації авіаційних перевезень на основі алгоритму генерації перестановок за допомогою лексикографічного впорядкування.

Результатом виконання кваліфікаційної роботи є розроблена веб-сторінка для знаходження найоптимальнішого шляху, використовуючи лексикографічне впорядкування.

Результати кваліфікаційної роботи рекомендується застосувати пасажирами чи авіакомпаніями для економії часу та коштів на подорожі чи переміщення багажу.

Галузь впровадження матеріалів кваліфікаційної роботи є сфера організаціїї авіаційних перевезень. Ступінь впровадження матеріалів кваліфікаційної роботи залишається на дослідниькому етапі, з врахуванням специфікації областей використання. Для подальшого розвитку кваліфікаційної роботи наступним етапом має бути тестування та валідація алгоритму на моделях та симуляторах для визначення його коректності.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СКОРОЧЕНЬ, ТЕРМІНИ

Fullstack веб-додаток – *Fullstack.*

JS – JavaScript.

ЛВ – Лексикографічне впорядкування.

VS Code – Visual Studio Code.

IDE – Integrated Development Environment

ВСТУП

Актуальність теми. На даний момент, побудова маршрутів авіаційних перевезень залишається актуальною і важливою частиною глобальної транспортної системи. Декілька аспектів свідчать про актуальність цього питання:

Швидкість і ефективність: авіаперевезення залишаються найшвидшим і найефективнішим засобом транспорту для довгих відстаней. Це особливо важливо для швидкого переміщення пасажирів та вантажів між континентами.

Глобалізація та світова економіка: зростання світової торгівлі і глобалізація економіки призводять до збільшення потреби в ефективних системах авіаперевезень для забезпечення швидкої та надійної поставки товарів та послуг.

Конкуренція між авіакомпаніями: зростаюча конкуренція в галузі авіації стимулює авіакомпанії до вдосконалення маршрутів та надання більш ефективних послуг, щоб залучити пасажирів.

Екологічні аспекти: природні обмеження в області забруднення та збільшення усвідомленості про зміни клімату підштовхують авіаційну індустрію до розробки більш екологічно ефективних маршрутів і використання більш сучасних літаків.

Технологічні інновації: завдяки розвитку технологій, таких як штучний інтелект, аналіз даних та системи управління маршрутами, авіакомпанії можуть оптимізувати свої маршрути, зменшуючи витрати та підвищуючи ефективність.

У зв'язку з цим, важливо продовжувати дослідження та розвиток у галузі планування маршрутів, щоб адаптувати їх до змінних умов і вимог ринку, а також для забезпечення ефективної та стійкої авіаційної інфраструктури.

Мета і завдання виконання кваліфікаційної роботи (проекту). Зменшення витрат на авіаційні перевезення.

Об’єкт дослідження – процес планування авіаційних перевезень.

Предмет дослідження – програмний модуль маршрутизації авіаційних перевезень на основі алгоритму генерації перестановок за допомогою лексикографічного впорядкування.

Методи дослідження. Дослідження проводилося математичним методом алгоритму генерації перестановок за допомогою лексикографічного впорядкування.

Наукова новизна отриманих результатів. Використання алгоритмів генерації перестановок за допомогою лексикографічного впорядкування може мати наукову новизну в побудові маршрутів авіаційних перевезень через оптимізацію порядку відвідування різних локацій або аеропортів. Наукова новизна може бути виявлена в низці таких аспектах:

Оптимізація маршрутів: використання алгоритмів генерації перестановок дозволяє оптимізувати порядок відвідування місць, аеропортів або проміжних пунктів на маршруті. Це може призвести до зменшення часу подорожі, вартості пального або інших оптимізаційних критеріїв.

Підтримка обмежень та умов: системи, які використовують алгоритми генерації перестановок, можуть ефективно обробляти різноманітні обмеження та умови, такі як графік рейсів, доступні ресурси, технічні обмеження літаків і т.д.

Реалізація змінних критеріїв: новизна може полягати в здатності впорядковувати маршрути залежно від змінних критеріїв, таких як погода, політична ситуація, транспортні та інфраструктурні зміни, що дозволяє системі адаптуватися до різних умов.

Урахування попереджень та попередження відхилень: система може використовувати алгоритми генерації перестановок для автоматичної генерації альтернативних маршрутів у випадку виникнення непередбачуваних ситуацій або відхилень від плану.

Скасування та відновлення маршрутів: алгоритми генерації перестановок можуть бути використані для швидкого перерахування та відновлення маршрутів у випадку скасування або змін в графіку рейсів.

Загалом, застосування алгоритмів генерації перестановок у побудові маршрутів авіаційних перевезень може призвести до створення більш ефективних та гнучких систем управління маршрутами, що може бути корисним в умовах швидкозмінюючого середовища авіаційної індустрії

Практичне значення отриманих результатів. Цей інструмент безумовно

корисний для компаній, які спеціалізуються на авіатранспорті та бажають ефективно планувати свої рейси. Завдяки даному модулю можна створити оптимальні маршрути з урахуванням часових обмежень та інших факторів, якими необхідно керуватися при перевезеннях пасажирів або вантажів літаками. Крім того, цей продукт може бути корисний і подорожуючим людям, в яких є потреба швидко та комфортно запланувати маршрут до конкретного пункту призначення. Загалом програмний модуль дає змогу скласти найоптимальніший шлях для подорожі за допомогою машини чи автобуса по суші і/або літака у повітряному просторі.

РОЗДІЛ 1
АНАЛІЗ СТАНУ ПРОБЛЕМИ ТА ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

1.1. Організація авіаційних перевезень

Організація авіаційних перевезень – це процес величезний процес планування, координації та забезпечення перевезення пасажирів і вантажів повітряним транспортом. Ця діяльність включає в себе ряд важливих аспектів, які мають бути враховані для забезпечення ефективного та безпечного авіаперевезення. Ключові етапів організації авіаційних перевезень:

1. Планування маршруту. Визначення оптимального маршруту для пасажирів або вантажу, враховуючи місця призначення, доступність аеропортів та інші фактори;
2. Бронювання і квитки. Організація процесу бронювання місць на польотах і видача авіаквитків пасажирам;
3. Забезпечення вантажу. Для вантажних перевезень, це включає в себе пакування, маркування та обробку вантажу перед відправленням;
4. Відправлення і прийом вантажу. Організація процесу завантаження вантажу на борт літака та його вивантаження після приземлення;
5. Пасажирське обслуговування. Забезпечення пасажирів комфортним перельотом, включаючи реєстрацію, обслуговування на борту і послуги в аеропорту;
6. Безпека. Забезпечення безпеки пасажирів і вантажу під час авіаперельотів. Це включає в себе безпеку бортового обладнання і персоналу;
7. Логістика. Координація руху літаків, обслуговуючого персоналу і вантажів для забезпечення пунктуальності і ефективності авіаперевезень;
8. Документація. Збір та обробка всіх необхідних документів, таких як літні дані, митні декларації тощо.

Далі приведено декілька прикладів систем організації авіаційних перевезень:

1. Інтернаціональна асоціація повітряного транспорту (*IATA*).

   IATA – це торгова асоціація, що об'єднує світові авіакомпанії. Основана в 1945 році, *IATA* виступає в ролі глобального представника і координатора для авіаційної промисловості. Організація базується в Женеві, Швейцарія.

   Основні функції *IATA* включають:

   1) Стандартизація.

      IATA розробляє та впроваджує стандарти для спрощення операцій та забезпечення єднання в авіаційній галузі. Наприклад, стандарти багажної етикетки, системи резервування квитків тощо;

   2) Безпека.

      IATA співпрацює з іншими міжнародними організаціями та регулюючими органами для забезпечення високого рівня безпеки в авіаційній промисловості;

   3) Лобіювання.

      Асоціація представляє інтереси авіакомпаній перед різними урядовими органами та міжнародними організаціями;

   4) Сервіси для авіакомпаній.

      IATA надає різноманітні послуги для авіакомпаній, такі як навчання, технічна підтримка, послуги у сфері пасажирського обслуговування тощо;

   5) Розробка та впровадження технологій.

      Асоціація активно займається впровадженням новітніх технологій в авіаційній промисловості;

   6) Розвиток інфраструктури.

      IATA сприяє розвитку та модернізації авіаційної інфраструктури у всьому світі.

2. Європейська організація з безпеки повітряного транспорту (*Eurocontrol*):

   Eurocontrol – це міжурядова організація, яка була створена з метою поліпшення безпеки та ефективності повітряного транспорту в Європі. Організація базується у Брюсселі, Бельгія.

Основні завдання *Eurocontrol* включають:

1) Управління повітряним простором.

   Eurocontrol відповідає за управління та координацію повітряним простором над Європою з метою забезпечення ефективного та безпечного руху повітря;

2) Оптимізація траєкторій польоту.

   Організація розробляє та впроваджує технології та процедури, спрямовані на оптимізацію траєкторій польоту для зменшення витрат пального та покращення ефективності руху повітря;

3) Розвиток системи *ATM* (управління повітряним рухом).

   Eurocontrol працює над розвитком та вдосконаленням систем управління повітряним рухом для забезпечення високого рівня безпеки та ефективності;

4) Навчання та дослідження:

   Організація забезпечує навчання та дослідження у сфері управління повітряним рухом та безпеки авіації.

3. Федеральна авіаційна адміністрація (*FAA*): Федеральна авіаційна адміністрація (*Federal Aviation Administration* або *FAA*) – це агентство Сполучених Штатів Америки, відповідальне за регулювання та нагляд за цивільною авіацією в США.

   Основні завдання та функції *FAA* включають:

   1) Безпека авіації.

      Встановлення та забезпечення дотримання стандартів та правил безпеки в цивільній авіації;

   2) Сертифікація повітряних засобів.

      Видача сертифікатів на повітряні засоби, включаючи літаки, вертольоти, літакові компоненти та обладнання;

   3) Управління повітряним простором.

      Координація та управління повітряним простором в США для забезпечення безпеки та ефективності повітряного руху;

4) Ліцензування та навчання.

   Видача ліцензій для пілотів, авіатехніків та інших фахівців цивільної авіації. Здійснення навчання та тестування;

5) Дослідження та розвиток.

   Проведення досліджень та розвиток технологій, спрямованих на покращення безпеки та ефективності авіаційної системи;

6) Регулювання та законодавство.

   Розробка та впровадження правил, законів та стандартів, спрямованих на регулювання цивільної авіації відповідно до федеральних законів;

7) Міжнародне співробітництво.

   Співпраця з міжнародними авіаційними організаціями та іншими країнами для підтримки міжнародних стандартів та безпеки.

   FAA відіграє ключову роль у забезпеченні безпеки та ефективності цивільної авіації в Сполучених Штатах та грає важливу роль у глобальній авіаційній системі;

4. *Air Navigation Service Providers* (*ANSPs*): *Air Navigation Service Providers* (*ANSPs*) – це організації, що забезпечують навігаційні послуги для повітряного транспорту. Їхні основні завдання полягають в управлінні та контролюванні повітряного простору, забезпеченні безпеки руху повітря, наданні послуг з контролю повітряного руху та підтримці навігаційних систем та послуг. Нижче подано деякі ключові аспекти організацій *ANSP*:

   1) Управління повітряним простором.

      ANSPs визначають структуру та межі повітряного простору, розробляють маршрути та забезпечують ефективне використання цього простору;

   2) Контроль повітряного руху.

      ANSPs забезпечують послуги з контролю повітряного руху, надаючи інструкції пілотам для забезпечення безпеки та відсутності конфліктів між літаками;

   3) Навігаційні послуги.

Вони управляють системами навігації, такими як радіонавігаційні точки, системи орієнтації та інші засоби, які сприяють безпеці та ефективності руху повітря;

4) Сервіси даних та інформації.

   ANSPs надають пілотам та авіаційним компаніям важливі дані, такі як погодні умови, аеронавігаційні картографічні матеріали та інші інформаційні послуги;

5) Безпека.

   ANSPs вносять величезний внесок у забезпечення безпеки в цивільній авіації, розробляючи та впроваджуючи найсучасніші стандарти та процедури;

6) Співпраця з іншими *ANSPs*.

   Вони можуть співпрацювати та обмінюватися інформацією для оптимізації роботи повітряного простору на різних територіях;

7) Впровадження новітніх технологій.

   ANSPs працюють над впровадженням та модернізацією технологій, таких як системи управління повітряним рухом через супутник (*ADS-B*), для покращення точності та ефективності.

Існують різні *ANSPs* по всьому світу, і багато з них співпрацюють на міжнародному рівні, щоб забезпечити безпеку та ефективність глобального повітряного транспорту. Системи управління аеропортом. Багато великих аеропортів використовують власні системи для координації руху літаків, миттєвої обробки багажу та забезпечення зручностей для пасажирів. Наприклад система АБС (*Airport Operational Control Centre)* в аеропорту Хітроу у Лондоні.

Цей центр зазвичай є централізованим пунктом управління для координації та моніторингу різних аспектів операцій в аеропорту. Система АБС може включати в себе різні технології та підсистеми для ефективного управління різними аспектами аеропортових операцій.

Основні функції системи АБС можуть включати:

1) Моніторинг авіаційної безпеки.

   Система АБС може відстежувати рух повітряних суден на площі аеропорту та навколишньому повітряному просторі для забезпечення безпеки;

2) Керування операціями на землі.

   Координація руху повітряних суден, технічного обслуговування літаків, руху автотранспорту на землі, а також інших пов'язаних операцій;

3) Моніторинг інфраструктури.

   Відстеження стану аеропортової інфраструктури, такої як стани злітно-посадкових смуг, таксівей, вантажних зон тощо;

4) Керування багажними операціями.

   Контроль за операціями з багажем та його перевезенням від терміналу до літака та навпаки;

5) Планування розкладу.

   Розробка та моніторинг розкладу руху літаків, враховуючи різні фактори, такі як погода, попит та інші обставини;

6) Керування кризовими ситуаціями.

   Взаємодія з екстреними службами та координація дій в разі аварій чи інших надзвичайних ситуацій.

Система АБС є ключовим елементом для забезпечення безперебійної та безпечної роботи аеропорту. Технології, які використовуються в системах АБС, можуть значно відрізнятися в залежності від розміру та потреб конкретного аеропорту;

5. *SITA* (*Société Internationale de Télécommunications Aéronautiques*):

   SITA (*Société Internationale de Télécommunications Aéronautiques*) – це міжнародна компанія, яка спеціалізується на наданні телекомунікаційних та інформаційних послуг для галузі авіації та пов'язаних індустрій. *SITA* виникла у 1949 році та швидко стала важливим гравцем у галузі технологій для авіаційної індустрії.

Ключові аспекти *SITA* включають:

1) Телекомунікаційні послуги.

   SITA надає широкий спектр телекомунікаційних послуг для авіаційної галузі, включаючи послуги передачі даних, інтернет-підключення, мережі для авіакомпаній та аеропортів;

2) Інформаційні технології.

   Компанія розробляє та надає рішення в галузі інформаційних технологій для авіаційного сектору. Це може включати системи для керування багажем, бронювання та вирішення проблем управління пасажирськими операціями;

3) Безпека та ідентифікація.

   SITA веде розробку та впровадження технологій для забезпечення безпеки в авіації, таких як технології біометричної ідентифікації та інші рішення для авіаційної безпеки;

4) Автоматизація процесів.

   Компанія працює над впровадженням технологій для автоматизації різних процесів в авіаційній галузі, що сприяє покращенню ефективності та зменшенню ризиків помилок;

5) Взаємодія між учасниками ринку.

   SITA допомагає забезпечити ефективну взаємодію між різними учасниками авіаційного ринку, включаючи авіакомпанії, аеропорти, постачальників послуг та інші сторони.

   SITA грає важливу роль в розвитку та оптимізації технологічних рішень для авіаційної індустрії з метою покращення безпеки, ефективності та пасажирського досвіду.

Ці системи спільно визначають стандарти та забезпечують ефективність та безпеку в авіаційній індустрії, допомагаючи в організації та координації авіаційних перевезень по всьому світу.

1.2. Методи маршрутизації авіаційних перевезень

Маршрутизація авіаційних перевезень – це процес вибору оптимального маршруту для авіаперельоту, щоб забезпечити ефективність та безпеку перевезень. Існує кілька методів маршрутизації, які використовуються в авіаційній індустрії:

Метод великого кола. Цей метод базується на великому колі, яке об'єднує дві точки на глобусі. Планування маршруту відбувається вздовж цього кола, що дозволяє скоротити відстань і скоротити час польоту.

Великі авіаперехрестя (*GR*). Метод використовує географічні точки на глобусі, де літаки можуть змінити напрямок або здійснити посадку. Великі авіаперехрестя можуть бути використані для планування маршрутів і визначення найкоротшого шляху.

Метод оптимального рівня. Цей метод полягає в виборі оптимального рівня польоту для літака в залежності від ряду факторів, таких як метеоумови, планована відстань та вага вантажу.

Відомі маршрути. Деякі авіакомпанії використовують відомі маршрути, які є сталими траєкторіями між двома пунктами і їх можна використовувати для планування маршрутів.

Метеорологічна інформація. Метеорологічні умови грають важливу роль в маршрутизації. Планування маршруту може включати урахування погоди, турбулентності і інших метеорологічних факторів для забезпечення безпеки і комфорту польоту.

Економічні фактори. Вибір маршруту також може бути заснований на: економічними факторами, такими як вартість пального, збитки часу і вартість авіаподорожі.

Навігаційні правила і обмеження. Планування маршруту повинно дотримуватися навігаційних правил і обмежень, які встановлені відповідними авіаційними органами.

Для прикладу, є крупний хаб, на якому накопичилось багато вантажів, які необхідно розвести по деяким містам.

Для цього компанія організує чартерний рейс щоб розвести весь багаж. Але ж

для початку треба мінімізувати витрати, визначивши в який спосіб, та яким маршрутом буде краще полетіти до того чи іншого міста, та повернутися назад.

Маршрутизація авіаційних перевезень – це складний і багатогранний процес, який вимагає урахування різних факторів для досягнення ефективних та безпечних перевезень. Точний метод маршрутизації може варіюватися в залежності від конкретних обставин і вимог авіакомпанії.

1.3. Постановка задачі дослідження

Постановка завдання дослідження полягає у створенні найоптимальніших шляхів перевезення пасажирів та вантажу з одного місця до іншого за допомогою повітряного транспорту.

Цей метод вивчення та аналізу має ряд значущих переваг:

1. Швидкість та ефективність. Повітряний транспорт є одним з найшвидших та найефективніших способів переміщення. Здатність подолати великі відстані за короткий час робить його ідеальним вибором для пасажирських та вантажних перевезень;
2. Глобальна доступність. Авіаційні мережі охоплюють всі континенти, забезпечуючи глобальний доступ до різних регіонів. Це особливо важливо для міжнародних та дальньомагістральних перевезень;
3. Підвищена комфортність для пасажирів. Сучасні пасажирські літаки оснащені комфортабельними салонами, розвагами та сервісом, що робить подорож приємною та зручною для пасажирів;
4. Реакція на економічні зрушення. Авіаційний транспорт може швидко реагувати на економічні зрушення, забезпечуючи швидку переорієнтацію маршрутів та пунктів призначення відповідно до попиту;
5. Можливість доставки великих обсягів вантажу. Повітряний транспорт може ефективно перевозити великі обсяги вантажу, що робить його важливим для логістичних потреб підприємств та міжнародної торгівлі;
6. Маршрутизація в умовах невеликої інфраструктури. В порівнянні з іншими

видами транспорту, повітряний транспорт може працювати в умовах обмеженої інфраструктури, так як йому не потрібні широкі суходільні маршрути;

7. Сприяння економіці та розвитку регіонів. Розширення авіаційних мереж сприяє економічному розвитку та збільшенню туризму в різних регіонах, що стимулює створення нових робочих місць та підтримує інфраструктурний розвиток;
8. Мінімізація перешкод. Повітряний транспорт мінімізує перешкоди в зв'язку з тим, що він може летіти прямо від точки А до точки Б, не будучи обмеженим територіальною географією.

Щодо моєї задачі, то вона буде узагальнювати деякі системи перевезень, які були наведені у пункті 1.1. В самому застосунку ми будемо вказувати певну кількість міст. Програма буде сама малювати з’єднання між ними та визначати найоптимальніший шлях. Перевагою цього методу є те що він досить простий та його розробка не займає багато часу. Також, його досить легко вдосконалити та адаптувати під потреби певної компанії. Ще можна зазначити про інтеграції з іншими системами.

1.4. Висновки до розділу

У даному розділі ми визначили що таке організація авіаційних перевезень та її ключові етапи. Нище описані аспекти організації авіаційних перевезень:

Комплексність та багатозадачність. Організація авіаційних перевезень є великим та багатозадачним процесом, який охоплює велику кількість аспектів, від планування маршруту до обробки документації.

Важливість планування маршруту. Визначення оптимального маршруту є ключовим етапом, оскільки це впливає на ефективність, зручність та економічність авіаперевезень, також воно впливає на безбеку а також можна врахувати всі обмеження і певні обставини.

Роль бронювання і квитків. Процес бронювання та видачі квитків має важливе

значення для організації пасажирських перевезень, забезпечуючи управління обсягами та доступність місць на польотах.

Забезпечення вантажу та пасажирів. Важливо враховувати аспекти забезпечення вантажів (пакування, маркування) та пасажирів (комфорт, обслуговування) для забезпечення безпроблемного перевезення.

Безпека як пріоритет. Забезпечення безпеки, як для пасажирів, так і для вантажів, є пріоритетом в організації авіаційних перевезень.

Координація та логістика. Ефективна координація літаків, персоналу та вантажів є важливим елементом для забезпечення пунктуальності та ефективності авіаперевезень.

Робота з документами. Збір та обробка документів є необхідною складовою для забезпечення легальності та безпеки авіаперевезень.

Постійний моніторинг та удосконалення. У зв'язку з динамічністю галузі, важливо постійно моніторити та удосконалювати процеси організації авіаційних перевезень для вдосконалення ефективності та забезпечення високого стандарту обслуговування.

В цілому, організація авіаційних перевезень вимагає комплексного підходу, великої уваги до деталей та використання різноманітних навичок та ресурсів для успішної інтеграції всіх етапів цього складного процесу.

Також розглянули декілька прикладів існуючих систем організації авіаційних перевезень.

Далі у розділі ми визначили що таке маршрутизація авіаційних перевезень та визначили кілька методів маршрутизації, які використовуються в авіаційній індустрії. А саме:

1. Метод великого кола. Цей метод базується на великому колі, яке об'єднує дві точки на глобусі;
2. Великі авіаперехрестя (*GR*). Метод використовує географічні точки на глобусі, де літаки можуть змінити напрямок або здійснити посадку. Ці авіаперехрестяможуть мати значущий вплив на світові авіаційні компанії;
3. Метод оптимального рівня. Цей метод полягає в виборі оптимального рівня

польоту для літака в залежності від ряду факторів, таких як метеоумови, планована відстань та вага вантажу;

4. Відомі маршрути. Деякі авіакомпанії використовують відомі маршрути, які є сталими траєкторіями між двома пунктами і їх можна використовувати для планування маршрутів;
5. Метеорологічна інформація. Планування маршруту може включати урахування погоди, турбулентності і інших метеорологічних факторів для забезпечення безпеки і комфорту польоту;
6. Економічні фактори. Вибір маршруту також може бути заснований економічними факторами, такими як вартість пального, збитки часу і вартість авіаподорожі;
7. Навігаційні правила і обмеження. Планування маршруту повинно дотримуватися навігаційних правил і обмежень, які встановлені відповідними авіаційними органами.

Спираючись на ці методи можна зробити висновок, що маршрутизація авіаційних перевезень – це складний і багатогранний процес, який вимагає урахування різних факторів для досягнення ефективних та безпечних перевезень. Точний метод маршрутизації може варіюватися в залежності від конкретних обставин і вимог авіакомпанії.

У кінці розділу були поставлена задача дослідження яка складаються у створенні найоптимальніших шляхів перевезення пасажирів та вантажу з одного місця до іншого за допомогою повітряного транспорту.

РОЗДІЛ 2
ЕВРИСТИЧНИЙ МЕТОД МАРШРУТИЗАЦІЇ АВІАЦІЙНИХ ПЕРЕВЕЗЕНЬ

2.1. Дослідження побудови маршрутів евристичними методами

Евристичні методи – це підходи до розв'язання задач, які базуються на досвіді, інтуїції та спрощених правилах. Ці методи використовуються для знаходження приблизних рішень у складних задачах оптимізації, де знаходження точного рішення може бути непрактичним з точки зору обчислювальних витрат. Деякі загальні види евристичних методів включають:

1) Локальні пошукові методи:

Жадібний алгоритм (*Greedy Algorithm*). Жадібний алгоритм – це метод розв'язання оптимізаційних задач, в якому на кожному кроці вибирається локально найкращий варіант з метою досягнення глобальної оптимальності. Основна ідея полягає в тому, щоб на кожному етапі вибирати оптимальне рішення без урахування можливих подальших наслідків.

Основні кроки жадібного алгоритму:

1. Визначення критерію оптимальності. Визначте критерій, за яким буде визначатися, яке рішення є оптимальним;
2. Вибір локально оптимального варіанту. На кожному кроці обирайте варіант, який найкраще відповідає визначеному критерію оптимальності;
3. Оновлення рішення. Додавання вибраного варіанту до поточного рішення;
4. Повторення або завершення. Повторюйте кроки 2-3 до досягнення визначеного критерію або досягнення завершального стану.

Жадібні алгоритми досить ефективні та прості в реалізації. Однак, через свою простоту, вони не завжди дають глобально оптимальні результати, оскільки на кожному кроці враховується тільки локальний критерій оптимальності. Основна перевага полягає в швидкості вирішення проблеми. І таким чином це може бути критично в певних умовах чи ситуаціях.

Приклад: Задача комівояжера може бути вирішена жадібним методом, де на кожному кроці обирається найближче місто для відвідання, і розв'язок будується додаванням цього міста до маршруту.

Локальний пошук (*Local Search*): Локальний пошук – це евристичний метод розв'язання оптимізаційних задач, який працює на основі поточного рішення та його околиці. На кожному кроці алгоритм переходить до сусіднього рішення, яке краще за поточне, і так продовжується, доки не буде досягнута задовільна якість або вичерпані всі можливості.

Основні кроки алгоритму локального пошуку:

Ініціалізація:

1. Виберіть початкове рішення;
2. Визначте критерій оптимальності.

Локальний пошук:

1. Для поточного рішення обчисліть його сусідні рішення (рішення, що відрізняється невеликою зміною);
2. Оберіть найкраще серед сусідніх рішень за критерієм оптимальності.

Оцінка:

1. Якщо нове рішення краще за поточне, прийміть його як поточне;
2. Перевірте, чи виконано критерій оптимальності. Якщо так, закінчіть алгоритм.

Повторення:

1. Повторюйте кроки 2-3 до досягнення задовільної якості або до вичерпання можливостей.

Локальний пошук може застрягнути в локальному оптимумі, якщо відсутній механізм виходу з тупикових ситуацій. Тому важливо реалізувати додаткові стратегії, такі як випадковий рестарт або зміна параметрів пошуку, щоб вирішити цю проблему.

Цей метод застосовується в різних областях, таких як оптимізація функцій, задачі комівояжера та інші задачі оптимізації, де потрібно шукати локальні максимуми або мінімуми функції;

2) Метаевристичні методи:

Методи генетичних алгоритмів (*Genetic Algorithms*): Метод генетичних алгоритмів (ГА) – це евристичний підхід до розв'язання оптимізаційних задач, інспірований природнім еволюційним процесом. Генетичні алгоритми здатні ефективно працювати у просторах великої розмірності та знаходити приблизні рішення, які задовольняють заданим критеріям оптимальності.

Основні кроки генетичних алгоритмів:

1. Ініціалізація. Створення початкової популяції різних рішень;
2. Оцінка пристосованості. Визначення якості кожного рішення в популяції за допомогою функції пристосованості;
3. Відбір. Вибір рішень для подальшого розмноження на основі їхньої пристосованості. Рішення з більшою пристосованістю мають більше шансів бути обраними;
4. Схрещування (кросовер). Створення нових рішень шляхом об'єднання частин рішень батьків. Це може відбуватися на рівні бітів, параметрів чи структур рішень;
5. Мутація. Випадкова зміна деяких параметрів у рішеннях для збереження різноманітності в популяції;
6. Оцінка пристосованості нової популяції. Перевірка якості нових рішень в популяції;
7. Заміна популяції. Вибір частини популяції для заміни новою популяцією на основі їхньої пристосованості;
8. Повторення. Повторення кроків 3-7 до досягнення критерію зупинки (наприклад, досягнення певної пристосованості або пройдення фіксованої кількості ітерацій).

Генетичні алгоритми застосовуються в різних областях, таких як оптимізація функцій, задачі комівояжера, розкладу, проектування та інші. Вони можуть бути адаптовані до різних типів задач і підлаштовані під конкретні умови.

Методи рою частинок (*Particle Swarm Optimization*): Метод рою частинок (*Particle Swarm Optimization, PSO*) є евристичним алгоритмом оптимізації,

інспірованим колективною поведінкою роїв частинок в природі. У цьому методі кожна частинка представляє можливе рішення у просторі оптимізації, а рух частинок моделюється з урахуванням їхнього власного кращого рішення та найкращого рішення, знайденого роєм.

Основні етапи методу рою частинок:

1. Ініціалізація. Створення початкового рою частинок у просторі оптимізації. Присвоєння початкових швидкостей та позначення кращих рішень кожної частинки;
2. Оцінка пристосованості. Визначення якості кожного рішення (частинки) за допомогою функції пристосованості;
3. Оновлення швидкості та положення. Кожна частинка оновлює свою швидкість та положення, використовуючи інформацію про її власне краще рішення та краще рішення сусідів (всіх частинок або обмеженої кількості найближчих);
4. Оцінка та оновлення найкращого рішення. Порівняння поточного рішення кожної частинки з її найкращим рішенням. Якщо поточне краще, воно оновлюється;
5. Повторення. Повторювати кроки 2-4 до досягнення критерію зупинки (наприклад, досягнення заданого рівня пристосованості або певної кількості ітерацій).

Метод рою частинок ефективний для оптимізації задач у просторі великої розмірності, де традиційні методи можуть виявитися неефективними. Він також добре справляється з задачами, що мають неоднорідні або нелінійні простори оптимізації. Метод рою частинок може бути успішно використаний для різних задач оптимізації, включаючи функціональну оптимізацію та задачі навчання з нагляду.

Методи вімінного пошуку (*Simulated Annealing*): Метод вімінного пошуку (*Simulated Annealing*) є евристичним алгоритмом оптимізації, який імітує процес охолодження металу. Цей метод дозволяє алгоритму виходити з локальних оптимумів та знаходити кращі рішення за допомогою ймовірності.

Основні етапи методу вімінного пошуку:

1. Ініціалізація. Вибір початкового рішення та початкової температури. Визначення критерію зупинки (наприклад, досягнення заданої температури або кількості ітерацій);
2. Генерація нового рішення. Застосування випадкового перетворення до поточного рішення, щоб отримати нове можливе рішення;
3. Оцінка рішення. Визначення функції пристосованості для нового та поточного рішень;
4. Порівняння рішень. Порівняння якості нового рішення з поточним. Якщо нове рішення краще, прийняти його. Якщо гірше, прийняти його з ймовірністю, що залежить від різниці в якості та поточної температури;
5. Оновлення температури. Зменшення температури відповідно до певного розкладу. Зменшення температури дозволяє алгоритму "охолоджуватися" та зменшувати ймовірність прийняття гірших рішень з часом;
6. Повторення. Повторення кроків 2-5 до досягнення критерію зупинки.

Метод вімінного пошуку дозволяє алгоритму досліджувати простір рішень, враховуючи якість рішень та ймовірність прийняття гірших рішень. Цей підхід дозволяє методу виходити з локальних оптимумів та знаходити глобально оптимальні рішення. Вімінний пошук застосовується в різних областях, включаючи оптимізацію функцій та комбінаторні задачі;

3) Методи найближчого сусіда:

Метод ближчого сусіда (*Nearest Neighbor Algorithm*): Метод ближчого сусіда (*Nearest Neighbor Algorithm*) – це жадібний алгоритм, використовуваний для розв'язання задачі комівояжера. Основна ідея полягає в тому, що алгоритм обирає найближче непройдене місто на кожному кроці, сподіваючись, що такий вибір призведе до коротшого загального маршруту.

Основні кроки методу ближчого сусіда:

1. Вибір початкового міста. Вибір будь-якого міста як початкового пункту маршруту;
2. Пошук найближчого міста. Обирання найближчого непройденого міста до поточного місця перебування;

3. Додавання до маршруту. Додавання вибраного міста до маршруту;
4. Оновлення поточного місця перебування. Переміщення до вибраного міста та оновлення поточного місця перебування;
5. Повторення. Повторення кроків 2-4 до тих пір, поки всі міста не будуть відвідані;
6. Завершення маршруту. Повернення до початкового міста, завершення маршруту.

Метод ближчого сусіда є простим і швидким способом розв'язання задачі комівояжера, але не завжди гарантує оптимальний результат. В результаті вибору найближчого міста на кожному кроці може виникнути ситуація, коли алгоритм "застрягає" в локальному оптимумі та не знаходить глобально оптимальний маршрут. Також важливо відзначити, що метод ближчого сусіда може бути чутливим до вибору початкового міста;

4) Методи пошуку в глибину і ширину:

Пошук в глибину (*Depth-First Search*): Пошук в глибину (*Depth-First Search* або *DFS*) – це алгоритм обходу графа чи дерева, який досліджує який-небудь гілковий шлях, поки не доходить до кінця, а тільки тоді повертається назад та обходить інші гілки. *DFS* можна застосовувати для вирішення різних задач, таких як пошук шляху, визначення зв'язності графа, топологічне сортування, інші задачі на графах.

Основні кроки алгоритму пошуку в глибину:

1. Початок. Вибрати початковий вузол графа;
2. Відмітка відвіданих вузлів. Відзначити поточний вузол як відвіданий;
3. Рекурсивний обхід сусідів. Рекурсивно викликати *DFS* для всіх сусідів поточного вузла, які ще не були відвідані;
4. Повернення назад. Після того, як всі сусіди були відвідані, повернутися назад на рівень вище;
5. Повторення. Повторювати кроки 2-4 для інших вузлів, які ще не були відвідані для досягнення бажаного результату та повного проходження заданого дерева чи графа.

Пошук в глибину може бути реалізований за допомогою рекурсивної або нерекурсивної стратегії. У рекурсивній реалізації використовується стек викликів, а в нерекурсивній – стек або інша структура даних для збереження і керування станами.

DFS часто використовується в алгоритмах обробки графів, таких як знаходження шляху, знаходження циклів, визначення зв'язності графа та інших завдань, де важливо досліджувати глибину графу.

Пошук в ширину (*Breadth-First Search*): Пошук в ширину (*Breadth-First Search* або *BFS*) — це алгоритм обходу графа чи дерева, який вивчає всі вершини одного рівня перед переходом до вершини наступного рівня. Він використовує структуру даних "черга" для збереження вершин, які мають бути відвідані.

Основні кроки алгоритму пошуку в ширину:

1. Початок. Вибрати початковий вузол графа;
2. Ініціалізація. Додати початковий вузол до черги та позначити його як відвіданий;
3. Пошук сусідів та додавання до черги:
    1) Поки черга не пуста. Вилучити вершину з початку черги. Додати всі сусідні невідвідані вершини до черги та позначити їх як відвідані;
4. Повторення. Повторювати крок 3, поки не будуть відвідані всі доступні вершини або досягнуто мету.

Пошук в ширину гарантує, що вершини вивчаються у порядку їхнього рівня віддаленості від початкової вершини. Цей алгоритм широко використовується для вирішення задач, таких як знаходження найкоротшого шляху в невагованому графі, визначення зв'язності графа, топологічне сортування та інші.

Ці методи можуть бути використані для різних задач оптимізації, включаючи задачі комівояжера, задачі розміщення, задачі розкладу, тощо. Вибір конкретного методу залежить від характеристик конкретної задачі та обмежень на обчислювальні ресурси.

Також евристичні методи дозволяють здійснити розв’язання складних задач, для яких людина не може дати точний (математичний) алгоритм. Як й кожен метод

виведення рішень, евристичні методи відтворюють логіку міркувань людини в процесі розв’язання конкретної задачі. Однак, на відміну від традиційних методів, які здійснюють виведення рішень на основі моделей представлення знань про предметну область, евристичні методи не вимагають наявності таких моделей, а логіка міркувань людини відбивається (моделюється) самим алгоритмом. Для побудови маршрутів авіаційних перевезень можна скористатися наприклад відомим евристичним методом комівояжера. Задача комівояжера полягає у знаходженні найвигіднішого маршруту, який проходить через всі вказані міста. В умовах задачі вказується критерій вигідності маршруту (найкоротший, найдешевший) і відповідні матриці відстаней, вартості тощо. Зазвичай задано, що маршрут повинен проходити через кожне місто тільки один раз.

У математичній постановці задача комівояжера передбачає побудову такого маршруту, який мінімізує деяку критеріальну функцію F. Існує багато різновидів узагальненої постановки задачі, зокрема геометрична задача комівояжера (коли матриця відстаней відображає відстані між точками на площині), трикутна задача комівояжера (коли на матриці вартостей виконується нерівність трикутника), симетрична та асиметрична задачі комівояжера. Точний розв’язок задачі комівояжера може бути отриманий за допомогою математичних методів оптимізації рішень, таких як метод спрямованого перебору варіантів або методу гілок та меж. Але їх використання вимагає побудови доволі складних математичних моделей та значних обсягів обчислень. Тому на практиці для розв'язання задачі комівояжера частіше застосовуються евристичні алгоритми, які дозволяють встановити не оптимальний, але достатньо наближений до нього розв'язок. Задачу комівояжера зазвичай представляють у графічній постановці, тобто у вигляді зваженого неорієнтованого графа, вершини якого відповідають містам, які повинен відвідати комівояжер, а ребра – шляхам між містами(див. рис. 2.1).

Кожному ребру приписується його вага – дійсне число, яке характеризує ступень переваги відповідної ділянки маршруту (тобто велечини, що вказують на відстань між містами, вартість пересування , час проходження відстані чи інші можливі характеристики).

Вхідні дані задачі комівояжера такі:

I – множина номерів міст, які має відвідати комівояжер (множина номерів вершин графа):

$$I = \{i:\ i = \overline{1, m}\}, \tag{2.1}$$

де m – кількість таких міст (кількість вершин);

i^{HK} – номер міста (вершини графа), з якого комівояжер починає рух і куди він повинен повернутися (зазвичай приймається $i^{HK} = 1$);

C– множина дійсних чисел c_{ij}, які характеризують ступень доцільності включення ділянок між містами у маршрут руху комівояжера (множина ваг ребер графа):

$$C = \{c_{ij} \mid i = \overline{1, m};\ j \in I \backslash \{i\}\}; \tag{2.2}$$

Найпростіший евристичний алгоритм розв’язання задачі комівояжера передбачає послідовне (покрокове) приєднання чергової ділянки (ребра графа) до побудованого вже часткового маршруту руху комівояжера.

Введемо додаткові умовні позначення:

k – номер кроку алгоритму;

$$k = \overline{1, m}; \tag{2.3}$$

$I^H(k)$ – множина номерів міст (вершин графа), які вже увійшли у частковий маршрут, побудований на початок k-го кроку;

$I^K(k)$ – множина номерів міст (вершин графа), які ще не увійшли у частковий маршрут, побудований на початок k-го кроку:

$$I^K(k) = I \backslash I^H(k); \tag{2.4}$$

$i^H(k)$ – номер міста (вершини графа), яке є останнім у частковому маршруті, що побудований на початок k -го кроку;

$i^K(k)$ – номер міста (вершини графа), яке обирається на k - у кроці алгоритму для приєднання до часткового маршруту, побудованого на початок цього кроку;

V – вектор, елементами якого є номери міст (вершин графа), розташовані у порядку їх проходження комівояжером:

$$V = (v_k \mid k = \overline{1,m}); \quad v_1 = v_{i^{HK}}; \tag{2.5}$$

F – критеріальна функція, значення якої відбиває сумарну довжину маршруту.

На кожному k-у ($k = \overline{1,m-1}$) кроці алгоритму послідовно виконуються такі дії.

1) Фіксується вершина графа $i^H(k)$.

   На першому ($k = 1$) кроці $i^H(k) = i^{HK}$.

   На подальших ($1 < k \leq m - 1$) кроках $i^H(k) = i^K(k-1)$.

2) Формується підмножина вершин $I^K(k)$, які ще не входять до часткового маршруту, побудованого на початок k-го кроку:

$$I^K(k) = I(k-1) \backslash \{i^H(k)\}; \quad I^K(0) = I; \tag{2.6}$$

3) Формується підмножина $C(k)$ ваг ребер, які поєднують вершину $i^H(k)$ з вершинами

   $i \in I^K(k)$:

$$C(k) = \{c_{i^H(k),j} \mid j \in I^K(k)\}. \tag{2.7}$$

4) Визначається мінімальний елемент підмножини $C(k)$:

$$c_{i^H(k),j} | \ j \in I^K(k)_{i^H(k),j^*\ min}. \tag{2.8}$$

Другий індекс j^* є номером вершини графа, яка з'єднана з вершиною $i^H(k)$ ребром з мінімальною вагою.

5) Вага ребра $c_{i^H(k),j^*}$ додається до суми ваг ребер маршруту, накопиченої на початок k-го кроку:

$$F(k) = F(k-1) + c_{i^H(k),j^*}, F(0) = 0. \tag{2.9}$$

6) Фіксується кінцева вершина визначеного ребра графа:

$$i^K(k) = i^H(k+1) = j^*. \tag{2.10}$$

7) Номер кінцевої вершини $i^K(k)$ заноситься у k- у позицію вектору послідовності вершин маршруту V:

$$v_k = i^K(k). \tag{2.11}$$

Якщо $k < m-1$, реалізується наступний $(k+1)$- й цикл алгоритму, починаючі з пункту 1.

Якщо $k = m-1$, виконується наступний пункт алгоритму.

8) Вага $c_{i^K(k),i^{HK}}$ ребра графа, яке з'єднує вершину $i^K(k)$ з початковою вершиною i^{HK}, додається до суми ваг ребер маршруту, накопиченої на початок k -го кроку, і обчислюється значення критеріальної функції:

$$F = F(k) + c_{i^K(k),i^{HK}}. \tag{2.12}$$

На цьому обчислювальний процес завершується.

Шуканий маршрут комівояжера (тобто, послідовність міст, які він повинен відвідати) описується вектором:

$$V = (v_k \mid k = \overline{1, m}). \quad (2.13)$$

Також, наприклад, алгоритм лексикографічного порядку використовується для визначення порядку між об'єктами, такими як рядки або числа, на основі їх лексичного представлення. Цей порядок відображає той, який ми спостерігаємо при алфавітному чи цифровому сортуванні. Цей алгоритм може мати практичні застосування у різних областях. Деякі з них включають:

Комбінаторна оптимізація: Генерація всіх перестановок може бути корисною для пошуку оптимальних або найкращих рішень у комбінаторних задачах, наприкладі, у задачках певної маршрутизації, розкладання задач, або при розв'язанні графових задач.

Тестування програмного забезпечення: Генерація перестановок може бути використана для створення тестових випадків для програм, де порядок вхідних або вихідних даних має значення.

Криптографія: У деяких алгоритмах криптографії використовуються перестановки, і генерація їх може бути важливою для аналізу безпеки та тестування вразливостей.

Генетичні алгоритми: У генетичних алгоритмах важливим етапом може бути генерація нових особин шляхом випадкових або систематичних перестановок генів.

Аналіз даних: Генерація перестановок може бути використана для статистичного аналізу даних, де важливо розглядати всі можливі комбінації.

Оптимізація розміщення ресурсів: У задачах розміщення об'єктів або ресурсів може бути важливим розглядати різні перестановки для знаходження оптимального розташування.

Основний принцип лексикографічного порядку:

1) Порівняння перших елементів. Порівнюється перший елемент об'єктів. Якщо вони різні, порядок визначений;

2) Якщо перші елементи однакові. Переходимо до наступних елементів і повторюємо процес порівняння;

3) Якщо один з об'єктів закінчується. Якщо один із об'єктів закінчується, інший об'єкт розглядається більшим. Наприклад, «*abc*» менше за «*abcd*»;

4) Довжина об'єктів. Якщо всі елементи досягли кінця, і об'єкти не відрізняються, то порівнюється їхня довжина. Об'єкт із більшою довжиною вважається більшим;

5) Порядок в алфавіті або числовий порядок: Якщо всі елементи досягли кінця, і довжина однакова, то порядок визначається в алфавітному порядку для рядків або числовому порядку для чисел. Наприклад, «*apple*» менше за «*banana*», а «123» менше за «456».

Цей процес триває до тих пір, поки не буде визначений повний лексикографічний порядок між об'єктами. Алгоритм широко використовується в програмуванні для сортування, порівняння та різних операцій з даними, де важливо визначити взаємний порядок між об'єктами.

Ось алгоритм на прикладі. Припустимо, що P = (5, 1, 7, 6, 3, 9, 8, 4, 2). Алгоритм роботи наступний:

Починаючи з кінця, ми бачимо 4 > 2, потім 8 > 4, потім 9 > 8 і, нарешті, 3 < 9. Отже, наш x дорівнює 5. (Для цього x ми маємо $P[x]$ = 3 і $P[x+1]$ = 9)

Починаючи з $x + 1$, ми бачимо, що 3 < 9, 3 < 8, 3 < 4, але 3 > 2. Останнє значення, яке все ще було більше за 3, було 4, а його індекс був y = 8.

Помінявши місцями $P[x]$ і P[y], ми отримаємо перестановку (5, 1, 7, 6, 4, 9, 8, 3, 2).

Перевернувши частину від індексу $x+1$ до кінця, ми отримаємо наш результат: перестановку (5, 1, 7, 6, 4, 2, 3, 8, 9). Це справді перестановка, яка безпосередньо слідує за вхідною перестановкою в лексикографічному порядку.

Початок розглядання визначення лексикографічного порядку. Ось є дві перестановки чисел від 1 до 9: одна P = (5, 1, 7, 6, 4, 9, 8, 3, 2), інша Q = (5, 1, 7, 8, 2, 4, 6, 3, 9).

Перший менший за другий. Це тому, що вони мають спільний префікс (5, 1, 7), а наступний елемент у першій перестановці менший за наступний елемент у другій перестановці (6 < 8).

Це перше важливе спостереження: кожна перестановка, яка більша за дану перестановку *P*, має спільний префікс із *P[x]*, а потім має більше значення, ніж *P[x+1]* у наступній позиції.

З усіх цих перестановок ми хочемо лексикографічно найменшу. Як це виглядає? Хто з них? Як ми можемо це знайти?

Якщо припустити, що ми розглядаємо перестановки, які більші, ніж у нашому прикладі перестановки *P* = (5, 1, 7, 6, 3, 9, 8, 4, 2). Однією з таких перестановок є перестановка *Q1* = (5, 1, 7, 8, 2, 4, 6, 3, 9). Інша така перестановка *Q2* = (5, 1, 7, 6, 9, 2, 4, 8, 3).

Якщо оглянути загальні префікси, які вони спільні з *P*. Загальний префікс між *P* і *Q1* це (5, 1, 7). Загальний префікс між *P* і *Q2* довший: (5, 1, 7, 6). І лише цієї інформації достатньо, щоб визначити, що *Q2* має бути меншим за *Q1*. У першій позиції, де *Q1* і *Q2* відрізняються, *Q2* все ще має однакове значення з *P*, тоді як *Q1* має строго більше значення.

Звідси витікає наше друге спостереження: з усіх перестановок, які більші за *P*, найменша має найдовший можливий префікс, спільний з *P*.

Перш ніж фактично побудувати бажану наступну перестановку, давайте знайдемо цю довжину загального префікса. Це підводить нас до наступного запитання: якщо задано перестановку *P* і ціле число x, чи існує інша перестановка, яка більша за *P* і має перші x (чи більше) елементів спільно з *P*?

Щоб відповісти на це питання, розглянемо спочатку приклад. Припустимо, що *P* = (5, 1, 7, 6, 3, 9, 8, 4, 2) і *x* = 6. Чи є інша перестановка, яка більша за *P* і також починається з (5, 1, 7, 6, 3, 9)? Отже, у кожній перестановці, яка має цей префікс, ми маємо елементи 2, 4 і 8 у певному порядку. І якщо ми подивимося на перестановку *P*, ми побачимо, що ці три елементи розташовані в порядку спадання. І це робить *P* лексикографічно найбільшою з усіх перестановок, які починаються з (5, 1, 7, 6, 3, 9).

Отже, спостереження №3: припустимо, що ми розбиваємо перестановку *P* на префікс *P1* і суфікс *P2*. Перестановка *P* є лексикографічно найбільшою з усіх перестановок з префіксом *P1* тоді і тільки тоді, коли *P2* відсортовано в порядку спадання.

І ми можемо відразу прослідкувати це за допомогою спостереження №4: якщо ми хочемо створити перестановку, яка більша за *P*, і ми хочемо залишити найдовший можливий префікс *P* незмінним, ми повинні знайти найкоротший можливий суфікс, який не сортується за спаданням порядок.

Саме це робить крок 1 нашого алгоритму.

Ось де ми знаходимося в нашому прикладі. Ми виявили, що для *P* = (5, 1, 7, 6, 3, 9, 8, 4, 2) правильним значенням *x* є 5. Іншими словами, ми можемо зберегти префікс (5, 1, 7, 6), але ми повинні збільшити значення 3, яке зараз знаходиться на *P[5]*.

Як ми можемо збільшити це значення? Ми повинні замінити його на одне з інших значень, які не використовуються у фіксованому префіксі. Тобто ми можемо змінити наше 3 на 9, на 8, на 4 або на 2.

Тепер ми маємо пам’ятати, що нам потрібна більша перестановка (яка виключає 2 у нашому прикладі, як 2 < 3). Крім того, ми хочемо, щоб найменша з усіх перестановок була більшою за *P*. У нашому прикладі це виключає 8 і 9. Наприклад, будь-яка перестановка, яка починається з (5, 1, 7, 6, 4), є меншою за будь-яку перестановку що починається з (5, 1, 7, 6, 8).

Це дає нам наше спостереження №4: при збільшенні *P* нам потрібно замінити *P[x]* наступним більшим значенням.

Порівняння еврестичних методів зображено в таб. 2.1.

Таблиця 2.1

Порівняння еврестичних методів

Характеристика	**Жадібний алгоритм**	**Локальний пошук**	**Генетичні алгоритми**	**Методи рою частинок**

Загальний принцип	Вибір на кожному кроці найбільш оптимального рішення	Пошук оптимального рішення в околиці поточного рішення	Імітація еволюційних процесів для пошуку оптимального рішення	Моделювання поведінки частинок у просторі для пошуку оптимального рішення
Глобальний пошук	Не гарантовано	Не гарантовано	Залежить від параметрів	Не гарантовано
Локальний мінімум	Так	Так	Так	Так
Вибірковість	Так	Залежить від алгоритму	Так	Так
Залежність від початкового стану	Так	Так	Так	Так
Підходить для задач великої розмірності	Залежить від природи задачі	Не завжди	Залежить від параметрів	Так

Закінчення таблиці 2.1

Характеристика	**Жадібний алгоритм**	**Локальний пошук**	**Генетичні алгоритми**	**Методи рою частинок**
Підходить для задач з обмеженнями	Залежить від природи задачі	Залежить від природи задачі	Так	Залежить від природи задачі

2.2. Застосування алгоритму генерації перестановок за допомогою лексикографічного впорядкування

Мною був обраний саме алгоритм генерації перестановок за допомогою лексикографічного впорядкування через ряд переваг над задачею комівояжера. А саме через :

1) Ефективність. Лексикографічне впорядкування дозволяє генерувати перестановки відразу в правильному порядку без необхідності перебору всіх можливих варіантів. Це робить алгоритм більш ефективним порівняно з іншими методами генерації перестановок;
2) Простота реалізації. Алгоритм лексикографічного впорядкування перестановок виглядає просто і може бути легко реалізований в багатьох мовах програмування. Він використовує прості операції порівняння та обміну для отримання наступної перестановки;
3) Можливість роботи зі згенерованими перестановками. Після того, як перестановка згенерована, можна легко визначити вартість маршруту для задачі комівояжера та визначити, чи є цей маршрут оптимальним;
4) Пам'ять. Алгоритм працює ітеративно і не вимагає зберігання усіх можливих перестановок у пам'яті одразу. Це особливо важливо, коли масштаби задачі великі, і обсяг пам'яті обмежений.

Алгоритм генерації перестановок за допомогою лексикографічного впорядкування використовує лексикографічний порядок для послідовного генерування всіх можливих перестановок об'єктів, таких як числа або символи. Основна ідея полягає в тому, щоб послідовно переходити від однієї перестановки до наступної так, щоб кожна нова перестановка була "наступною" у відповідному лексикографічному порядку.

Нище описано приблизні кроки алгоритм:

1) Ініціалізація. Розташуйте об'єкти у початковому лексикографічному порядку;
2) Пошук точки обміну. Почніть з кінця послідовності і знайдіть перший елемент, який може бути обмінений з наступним елементом так, щоб збільшити послідовність у лексикографічному порядку. Цей елемент і буде точкою обміну;
3) Пошук найменшого елемента для обміну. Почніть з кінця послідовності і знайдіть найменший елемент, який є більшим за точку обміну. Це буде елемент, з яким точка обміну буде обмінюватися;

4) Обмін елементів. Обміняйте точку обміну і найменший елемент;
5) Обертання залишку послідовності. Після обміну точки обміну і найменшого елемента, оберніть залишок послідовності після точки обміну, щоб забезпечити лексикографічний порядок;
6) Повторення. Повторюйте кроки 2 – 5 до тих пір, поки не буде згенеровано всі можливі перестановки.

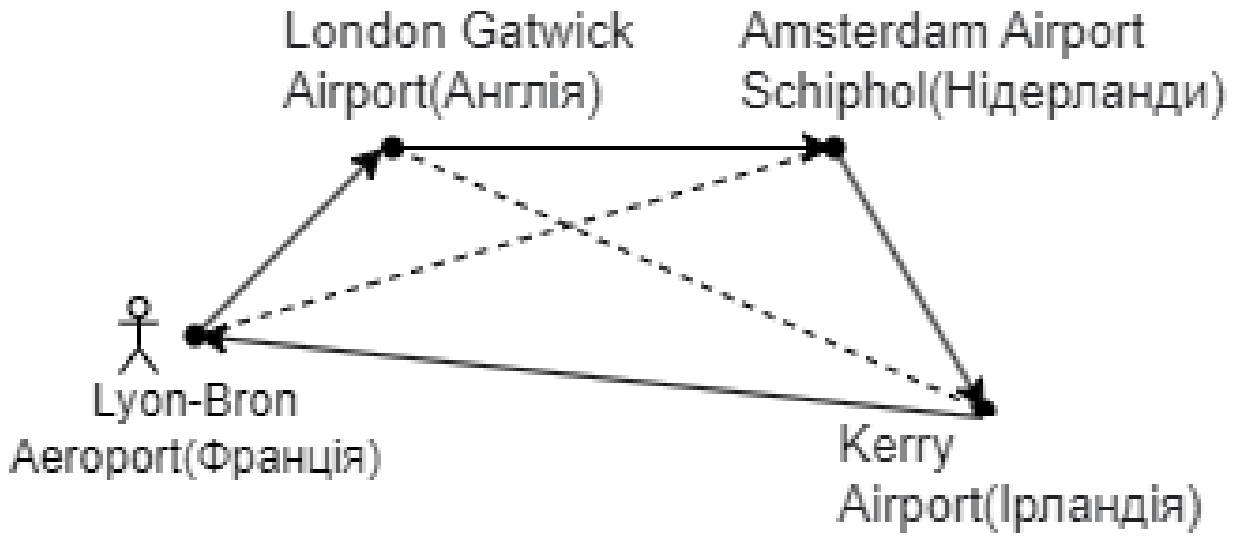

Рис. 2.1. Приклад уявлення методу комівояжера

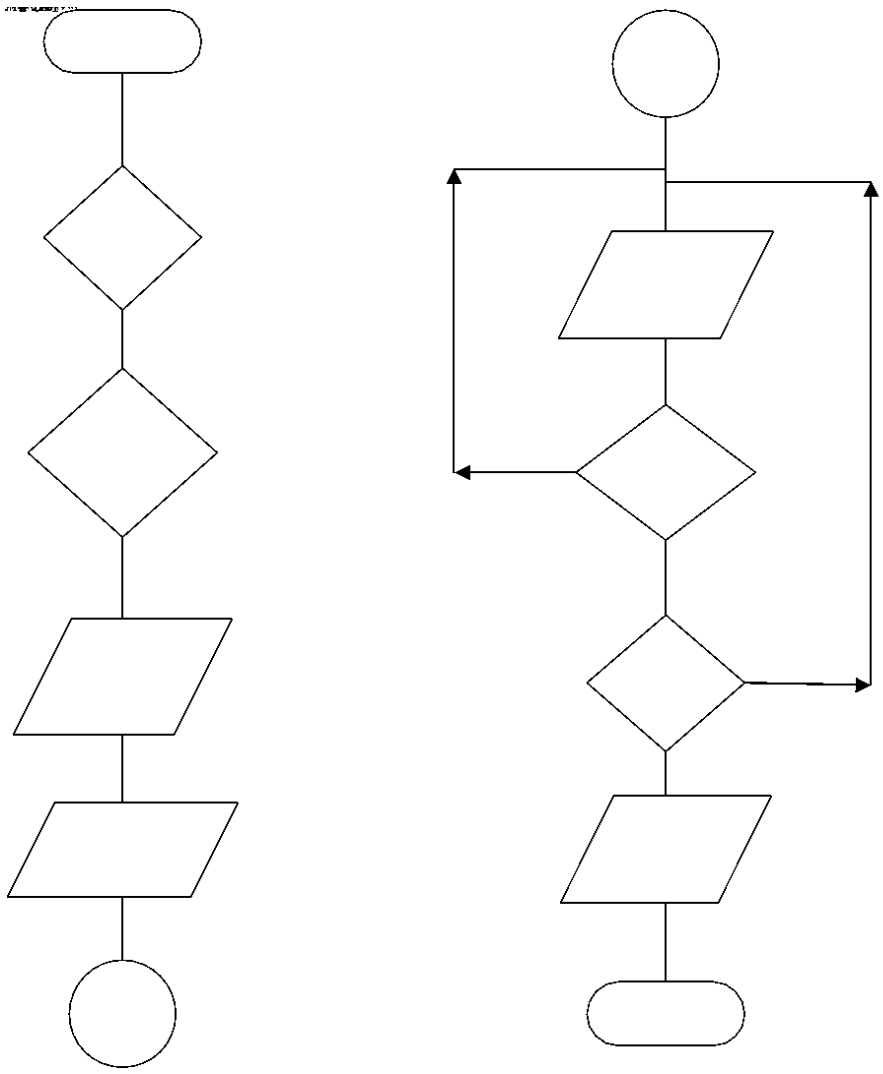

Рис. 2.2. Схема алгоритму виконання програми

Цей алгоритм, що зображено на рис. 2.2 ефективно генерує всі можливі перестановки в лексикографічному порядку без зайвого обчислення всіх можливих перестановок наперед. Якщо програма запущена то на екрані починають малюватися точки, які потім з’єднуються. Після цього координати з’єднань записуються у масив для подальшої обробки. Рахується відстань і визначається найкращою. Після цього йде наступний крок(з’єднання) після якого рахується відстань і якщо вона краще за попередню – попереднє найкраще значення перезаписується. І так відбувається допоки не буде виконаний останій крок програми. Після завершення на екран виводиться найкращий шлях для проходження. Також виконання цього алгоритму у вигляді діаграми класів зображено на рис. 2.3.

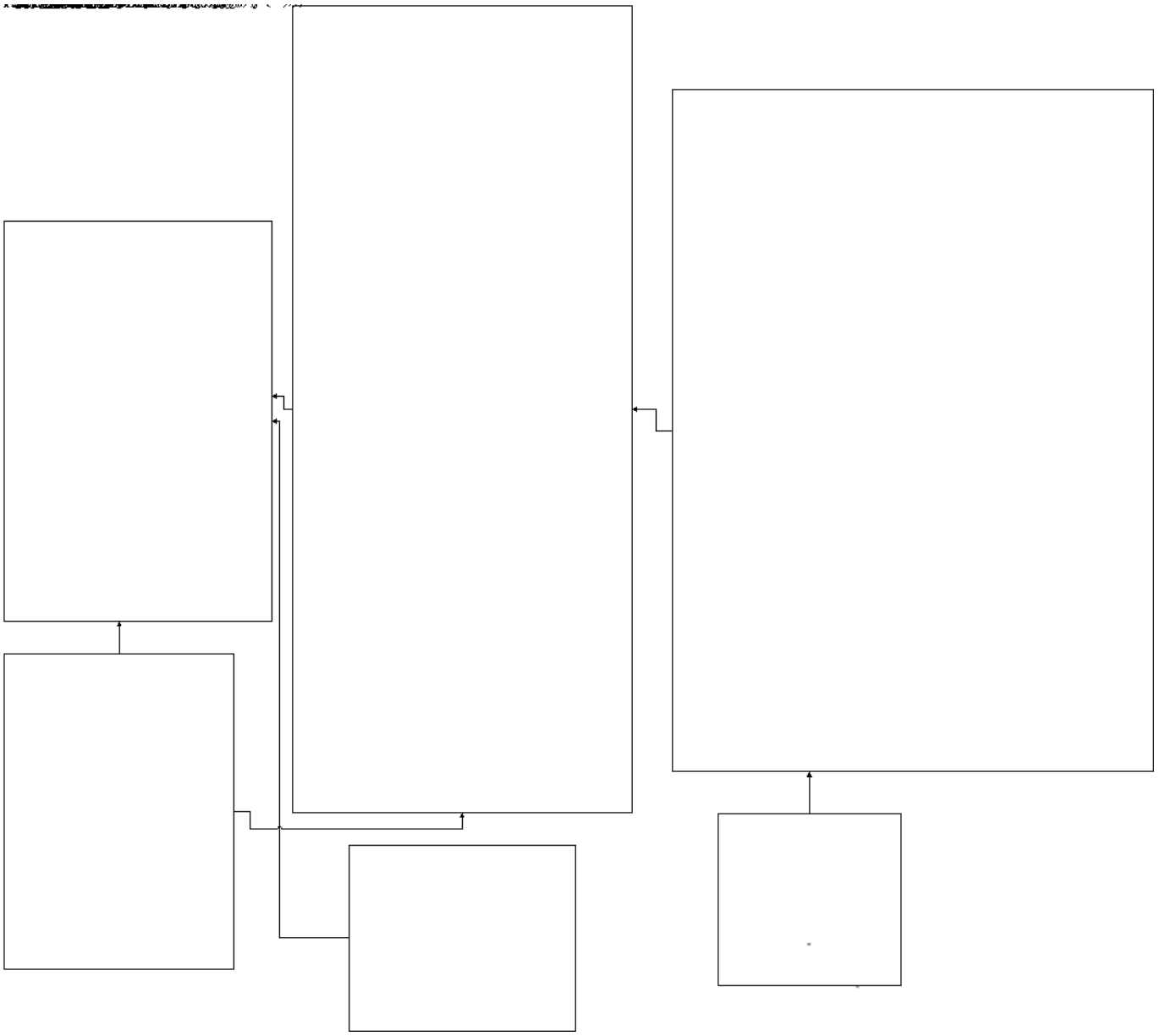

Рис. 2.3. Діаграма класів додатку

2.3. Висновки до розділу

У цьому розділі ми досліди можливості побудови маршрутів різними евристичними методами.

Евристичні алгоритми є методами розв'язання задач, які можуть не гарантувати знаходження оптимального рішення, але зазвичай працюють досить ефективно в областях складних задач або тих, для яких немає точного рішення. Висновки щодо евристичних алгоритмів можна сформулювати наступним чином у такій послідовності:

Швидкість розв'язання. Евристичні алгоритми часто володіють високою швидкістю розв'язання, що робить їх відмінним вибором для великих або складних задач.

Не гарантована оптимальність. Однак ці алгоритми не завжди гарантують знаходження оптимального рішення. Вони базуються на евристичних правилах та стратегіях, які можуть призводити до локально оптимальних, а не глобально оптимальних рішень.

Використання в оптимізації. Евристичні алгоритми широко використовуються в задачах оптимізації, де важко або неможливо знаходити точні рішення за прийнятний час.

Гнучкість та адаптивністью Ці алгоритми зазвичай виявляють велику гнучкість та адаптивність до змінних умов задачі. Вони можуть пристосовуватися до нової інформації та змінюваних умов, що робить їх корисними в реальних сценаріях.

Використання у комбінаторних задачах. Евристичні алгоритми широко застосовуються у комбінаторних задачах, таких як задачі комівояжера, розкладання задач на машини тощо.

Спрощення складних задач. Вони можуть допомагати спрощувати складні задачі, знаходячи прийнятні, як правило, рішення.

Можливість ускладнення задач. Іноді евристичні методи можуть ускладнити задачу, якщо не враховувати всі можливі аспекти та обмеження.

Загалом, евристичні алгоритми є потужними інструментами для розв'язання задач, де важко або неможливо застосувати точні методи. Вони знаходять застосування в різних галузях, де швидкість та адаптивність є ключовими факторами.

Застосували алгоритм генерації перестановок за допомогою лексикографічного впорядкування до поставленої задачі, саме через його ряд переваг:

1) Ефективність. Лексикографічне впорядкування дозволяє генерувати перестановки відразу в правильному порядку без необхідності перебору всіх можливих варіантів;

2) Простота реалізації. Алгоритм лексикографічного впорядкування перестановок виглядає просто і може бути легко реалізований в багатьох мовах програмування;
3) Можливість роботи зі згенерованими перестановками. Після того, як перестановка згенерована, можна легко визначити вартість маршруту для задачі комівояжера та визначити, чи є цей маршрут оптимальним;
4) Пам'ять. Алгоритм працює ітеративно і не вимагає зберігання усіх можливих перестановок у пам'яті одразу.

РОЗДІЛ 3
ПРОГРАМНА РЕАЛІЗАЦІЯ МОДУЛЯ МАРШРУТИЗАЦІЇ АВІАЦІЙНИХ ПЕРЕВЕЗЕНЬ

3.1. Інформаційне забезпечення модуля

Так як *JavaScript (JS)* є мовою програмування, яка використовується для розробки веб-додатків та взаємодії з веб-сторінками. Нижче перераховано кілька типових програм та проектів, які можна розробити за допомогою *JavaScript*:

Веб-сайти:

Розробка статичних або динамічних веб-сайтів.

Використання фреймворків, таких як *React, Angular* або *Vue*, для покращення роботи зі структурою інтерфейсу.

Інтерактивні веб-елементи:

Створення віджетів, плавних анімацій та ефектів для поліпшення взаємодії користувача.

Ігри:

Розробка браузерних ігор за допомогою бібліотек, таких як *Phaser* або *Three.js*.

1) *Phaser* – це *JavaScript*-фреймворк для створення ігор, який використовується для розробки веб-ігор та інших інтерактивних веб-додатків. *Phaser* надає набір інструментів та функцій, які полегшують розробку ігор, таких як керування анімаціями, обробка введення гравців, робота з графікою, аудіо та фізикою.

   Основні можливості *Phaser* включають:

   Анімації та спрайтию Легка інтеграція та керування анімаціями об'єктів та героїв гри.

   Фізикаю Підтримка фізичних ефектів, таких як реалістичне рухання та взаємодія об'єктів.

Звук. Вбудована підтримка аудіо, яка дозволяє легко додавати та керувати звуковим ефектом у грі.

Введення гравців. Обробка введення гравців, такого як клавіатура та миша, для реалізації взаємодії гравців з грою.

Фази гри. Можливість розділення гри на фази, такі як "завантаження", "меню" та "гра", для легкості управління логікою гри.

Phaser є відкритим програмним забезпеченням і доступний для використання безкоштовно. Він широко використовується серед розробників веб-ігор та надає гнучкість та продуктивність для швидкого створення ігор в середовищі веб-браузера;

2) *Three.js* – це бібліотека *JavaScript* для створення *3D*-графіки у веб-браузерах. Вона надає інструменти для створення та візуалізації тривимірних сцен, об'єктів та ефектів безпосередньо в браузері. *Three.js* розширює можливості *WebGL*, що є стандартом для відображення *3D*-графіки у веб-середовищі.

Основні функції та можливості *Three.js* включають:

Створення сцени. *Three.js* дозволяє створювати *3D*-сцени, які можуть включати об'єкти, камери та світло.

Створення та робота з об'єктами. Ви можете легко створювати та маніпулювати *3D*-об'єктами, такими як куби, сфери, моделі та інші.

Матеріали та текстури. *Three.js* підтримує різні матеріали для об'єктів, а також можливість додавати текстури для забезпечення більш реалістичного вигляду.

Анімація. Можливість створення анімацій об'єктів та камер для живого та динамічного відображення сцени.

Обробка введення. Підтримка обробки введення, такого як клікання мишею або клавіші, для інтерактивності *3D*-сцени.

VR та *AR*. *Three.js* також підтримує віртуальну та розширену реальність, що дозволяє розробникам створювати вражаючі віртуальні та розширені реальності у веб-браузерах.

Three.js використовується великою кількістю веб-розробників для створення відмінної *3D*-графіки в інтерактивних веб-додатках та іграх.

Розширення для браузера. Створення розширень для браузера, таких як *Chrome Extensions* або *Firefox Add-ons*.

Мобільні додатки. Використання фреймворків, таких як *React Native* або *Ionic*, для створення крос-платформових мобільних додатків.

Серверна розробка. Використання *Node.js* для створення серверів та розробки серверних додатків.

Робота з *API*. Взаємодія зі сторонніми *API* для отримання або відправлення даних.

Автоматизація завдань. Створення скриптів для автоматизації рутинних завдань на власному комп'ютері або в браузері.

Веб-додатки в реальному часі. Розробка додатків, які використовують технології веб-сокетів для реального часу.

Форми та валідація. Розробка форм для збору даних в інтерактивному вигляді та валідація введених користувачем даних.

Мапи та геолокація. Використання *API* карт для відображення мап та взаємодії з геолокацією.

Співпраця з базами даних. Взаємодія з базами даних, такими як *Firebase*, або використання *AJAX* для асинхронного отримання та відправлення даних.

3) *Firebase* – це платформа для розробки мобільних та веб-додатків, яка надає ряд сервісів та інструментів для швидкої та легкої розробки та впровадження додатків. *Firebase* пропонує різноманітні сервіси для роботи з базами даних, аутентифікації, зберігання та обробки файлів, реального часу зв'язку, хостингу та інших функцій.

   Основні складові *Firebase* включають:

   Realtime Database. *NoSQL* база даних в реальному часі, яка дозволяє синхронізувати дані між користувачами та пристроями у реальному часі.
   Firestore. Розширена *NoSQL* база даних, яка надає додаткові можливості та

масштабується краще ніж *Realtime Database*.

Authentication. Сервіс для автентифікації користувачів, що дозволяє використовувати різні методи входу, такі як електронна пошта, *Google, Facebook, Twitter* і т. д.

Cloud Functions. Сервіс, який дозволяє розробникам написати та використовувати функції в хмарі, реагуючи на події та виклики *API*.

Cloud Storage. Сховище для зберігання та обробки файлів, таких як зображення, відео, аудіо тощо.

Cloud Messaging (*FCM*). Сервіс для надсилання повідомлень на мобільні пристрої та веб-додатки.

Hosting. Сервіс для хостингу статичних та динамічних веб-сайтів та додатків.

Firebase дозволяє розробникам швидко створювати та впроваджувати функціонал своїх додатків без значного напруження з боку інфраструктури. Вона є популярним вибором для стартапів та розробників, які шукають швидке та зручне рішення для хмарової розробки;

4) *AJAX* (*Asynchronous JavaScript* and *XML*) – це техніка веб-розробки, яка дозволяє взаємодіяти з сервером асинхронно. Основна ідея полягає в тому, що дані можуть передаватися між клієнтом і сервером без необхідності перезавантаження сторінки веб-сайту.

Основні компоненти *AJAX* включають:

JavaScript. Використовується для виклику серверних *API* та обробки відповіді.

XMLHttpRequest або *Fetch API*: Об'єкт, який використовується для взаємодії з сервером та відправки асинхронних запитів.

HTML (или інші формати даних): Дані можуть передаватися у форматі *XML*, але частіше використовуються інші формати, такі як *JSON*.

Основні переваги *AJAX*:

Асинхронність. Запити відправляються асинхронно, що дозволяє сторінці продовжувати цю взаємодію з користувачами без очікування відповідей від

сервера.

Динамічність. Дозволяє оновлювати лише певні частини сторінки, замість перезавантаження всієї сторінки.

Зменшення навантаження на сервер: Так як тільки частина сторінки оновлюється, це може зменшити обсяг даних, які передаються між клієнтом та сервером.

AJAX використовується для реалізації різних функціональностей, таких як завантаження даних у фоновому режимі, автозаповнення поля пошуку, динамічне оновлення частин сторінок та багато іншого. Це стало стандартною практикою у веб-розробці для покращення користувацького досвіду та ефективного взаємодії з серверами.

Програма починається з того, що в неї задаються міста, які потім малюються на випадково заданому місці мапи. Вона перевіряє усі можливі випадки проходження і визначає найкращій з них.

Для прикладу візмемо порядоки «*ABC*» та «123» і пройдемо всі можливі варіанти перестановок(див. рис. 3.1).

ABC	123
ACB	132
BAC	213
BCA	231
CAB	312
CBA	321

Рис. 3.1. Приклад перебору всіх можливих перестановок

Використовує програма алгоритм генерації перестановок за допомогою лексикографічного впорядкування:

Для цього маємо таку формулу:

Припустимо таке, що *P[1..n]* є перестановкою від 1 до *n*. Ми можемо створити

наступну перестановку в лексикографічному порядку, дотримуючись цих простих кроків:

1) Знайти найбільше x таке, що *P[x] < P[x+1]*.
   (Якщо такого *x* немає, *P* є останньою перестановкою.)
2) Знайти найбільше *y* таке, що *P[x] < P[y]*.
3) Замінити *P[x]* і *P[y]*.
4) Перестановка *P[x+1 .. n]*.

Приклад такого алгоритму. Отже, *P* = (5, 1, 7, 6, 3, 9, 8, 4, 2). Алгоритм роботи буде наступний:

1) Починаючи з кінця, ми бачимо 4 > 2, потім 8 > 4, потім 9 > 8 і, нарешті, 3<9. Отже, наш *x* дорівнює 5. (Для цього *x* ми маємо *P[x]* = 3 і *P[x+1]* = 9.)
2) Починаючи з *x + 1*, ми бачимо, що 3 < 9, 3 < 8, 3 < 4, але 3 > 2. Останнє значення, яке все ще було більше за 3, було 4, а його індекс був *y* = 8.
3) Помінявши місцями *P[x]* і *P[y]*, ми отримаємо перестановку (5, 1, 7, 6, 4, 9, 8, 3, 2).
4) Перевернувши частину від індексу *x + 1* до кінця, ми отримаємо наш результат: перестановку (5, 1, 7, 6, 4, 2, 3, 8, 9). Це і є та перестановка, яка безпосередньо слідує за вхідною перестановкою в лексикографічному порядку.

Для прикладу візьмемо 3 міста у вигляді *A*, *B*, *C* та розподілимо їх хаотично та візьмемо для прикладу порядок 1, 2, 0 (див. рис. 3.2).

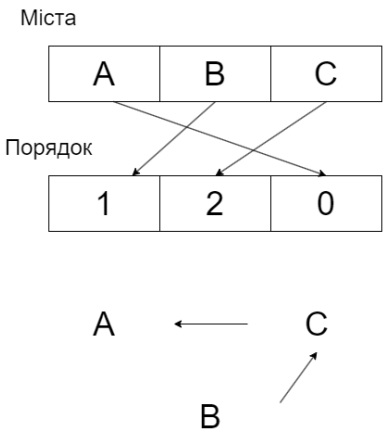

Рис. 3.2. Приклад проходження порядку

Так як порядок в програмуванні йде з 0, то під одиницю підпадає *B*, це наш початок подорожі. Наступним йде 2 – це *C*, а 0 – *A*. Тобто рухаємося так: *B→C→A*.

Найкращим кроком буде, за допомогою функції *draw()* виконується весь процес вимальовування наступного порядку маршруту між містами та процентного виконання програми, завдяки знаходженню факторіалу кількості міст. Повним сенсу, кожного разу роблячи заміну програма перевіряє чи не найкращій зараз варіант і проходить далі. По завершенню програми, виводиться найкращий результат у вигляді маршрута.

3.2. Програмне забезпечення модуля

Програма буде писатися мовою програмування *JavaScript* з використанням *HTML* та *CSS* у середовищі програмування *Visual Studio Code*.

Мною була обрана саме мова програмування *JS* через певні переваги над іншими мовами програмування, таких як *C#*, *Java*, *C++*.

JavaScript (JS) і *C# (C Sharp), C++*, та *Java* – це різні мови програмування, які використовуються для різноманітних цілей та в контексті різних технологій. Нижче розглянемо деякі можливі переваги *JavaScript* порівняно з *C#*, *C++* та *Java*, але слід пам'ятати, що це лише загальні розгляди, і кожна мова має свої унікальні сценарії застосування.

Веб-розробка.

JavaScript одна з основних мов для розробки веб-інтерфейсів та фронтенду. Застосовується в браузерах для динамічного оновлення сторінок.

C#, C++, Java. Хоча ці мови також можуть використовуватися для веб-розробки, вони більше спеціалізовані в інших областях, таких як серверні додатки, десктопні програми тощо.

Кросплатформенність.

JavaScript зазвичай використовується для написання коду, який виконується в браузерах, що робить його кросплатформенним за природою.

C#, C++, Java. Зазвичай використовуються для розробленя кросплатформених

додатків за допомогою відповідних технологій, таких як *.NET Core (C#)*, *JavaFX* (*Java*), або *Qt (C++)*.

Швидкість виконання.

JavaScript зазвичай повільніший порівняно із компільованими мовами, такими як *C++*.

C#, C++, Java. Можуть бути швидшими завдяки використанню компіляції чи виконанню на віртуальній машині з оптимізацією виконання.

Синтаксис та простота використання.

JavaScript має простий синтаксис, що робить його привабливим для початківців та швидше вивчається.

C#, C++, Java. Може виглядати більш складним для новачків, але це також залежить від індивідуального досвіду та підходу до програмування.

Відкритість та екосистема.

JavaScript. Відкритий стандарт, широка екосистема бібліотек та фреймворків (наприклад, *React, Angular, Vue*).

C#, C++, Java. Мають свої власні екосистеми та фреймворки (наприклад, *.NET* для *C#*, *Qt* для *C++, Java EE* для *Java*).

Типізація.

JavaScript. Як правило, має слабку динамічну типізацію.

C#, C++, Java. Мають більш сильну статичну типізацію, що може полегшити виявлення помилок на етапі компіляції.

Обираючи мову програмування, важливо враховувати конкретні потреби проекту та власні вподобання розробника. Кожна з цих мов має свої власні сильні та слабкі сторони, і оптимальний вибір залежить від конкретного контексту використання.

JS (або *JavaScript*) – це високорівнева, універсальна мова програмування, яка в основному використовується для розробки веб-сайтів та веб-додатків. Ось ключові аспекти *JavaScript*:

Призначення. *JavaScript* головним чином використовується для фронтенд-розробки веб-сайтів. Вона дозволяє самим розробникам створювати інтерактивні та

динамічні веб-сторінки, які дозволяють користувачам взаємодіяти з веб-контентом.

Вона також може бути використана на серверному боці завдяки середовищу виконання під назвою *Node.js*, що робить її підходящою для розробки *fullstack* веб-додатків.

Синтаксис. *JavaScript* має синтаксис у стилі *C*, який вважається досить простим для читання та написання. Зазвичай вона вбудовується в *HTML*-код за допомогою тегів <*script*> для веб-додатків.

Інтерактивність. *JavaScript* додає інтерактивність на веб-сторінки, дозволяючи реалізувати функції, такі як валідація форм, анімації, завантаження динамічного контенту та інше.

Вона реагує на дії користувачів, такі як кліки мишею, введення з клавіатури та натискання кнопок.

Типи даних. *JavaScript* підтримує різні типи даних, включаючи рядки, числа, булеві значення, об'єкти, масиви, функції та інше. Вона використовує динамічне типізування, що означає, що типи змінних можуть змінюватися під час виконання.

Потік керування. *JavaScript* надає конструкції керування, такі як оператори *if*, цикли *for* та *while*, та оператори *switch* для умовної та ітеративної програми.

Функції. Функції є фундаментальною частиною *JavaScript*. Ви можете визначати та викликати функції, передавати їх як аргументи та повертати їх з інших функцій.

Об'єкти та прототипи. *JavaScript* є прототипною мовою. Об'єкти можна створювати за допомогою конструкторів або об'єктних літералів. Вона підтримує спадкування через прототипи.

Асинхронне програмування. *JavaScript* відмінно справляється з асинхронними завданнями. Вона використовує зворотні виклики, обіцянки (*Promises*) та синтаксис *async/await* для керування операціями, такими як отримання даних з серверів та обробка введення користувача.

Бібліотеки та фреймворки. Для спрощення і прискорення веб-розробки доступний широкий спектр бібліотек та фреймворків. Деякі популярні з них включають *jQuery*, *React*, *Angular*, *Vue.js* та *Express.js*.

Безпека. Код *JavaScript* може бути вразливим до загроз безпеці, таких як *Cross-Site Scripting* (*XSS*) та *Cross-Site Request Forgery* (*CSRF*). Розробники повинні дотримуватися найкращих практик для захисту своїх додатків.

ES6 та інше. *JavaScript* продовжує розвиватися. *ECMAScript 6* (*ES6*) внесла сучасні можливості, такі як стрілкові функції, літерали шаблонів та ключові слова *let* та *const*, і подальші версії продовжують покращувати мову.

JavaScript є фундаментальною мовою в світі веб-розробки та відіграє ключову роль у створенні динамічних та інтерактивних веб-додатків. Вона широко підтримується веб-браузерами та має велику та активну спільноту розробників, що робить її цінним ресурсом для тих, хто цікавиться веб-розробкою та іншими галузями програмування.

Фреймворк у *JavaScript* – це певна структура або набір правил, які надають зручний і підготовлений до використання фундамент для розробки програмного забезпечення на *JavaScript*. Фреймворки розробляються для того, щоб спростити процес розробки, забезпечити кращу організацію коду, підтримувати найкращі практики та стандарти, і включати готові рішення для часто виникаючих завдань.

Ось декілька типових фреймворків та бібліотек у світі *JavaScript*:

Angular: *Angular* – це популярний веб-фреймворк, який розробляється і підтримується командою *Google*. Це високопродуктивний і потужний інструмент для створення односторінкових додатків (*Single Page Applications* або *SPA)* на базі мови програмування *TypeScript*.

Основні характеристики *Angular* включають:

Компонентний підхід. *Angular* побудований на основі компонентного підходу, де весь інтерфейс користувача розбивається на невеликі та повторно використовувані компоненти.

Двостороннє зв'язування даних. *Angular* використовує концепцію двостороннього зв'язування даних, що дозволяє автоматично оновлювати користувацький інтерфейс при зміні даних і навпаки.

Залежності та ін'єкція залежностей. *Angular* використовує систему ін'єкції залежностей для кращого керування залежностями між компонентами та сервісами.

Маршрутизація. Для управління переходами між сторінками додатку використовується вбудований модуль маршрутизації.

Типізація з *TypeScript*. *Angular* написаний на *TypeScript*, що дозволяє використовувати типізацію для покращення стабільності та розробки коду.

Розширені можливості. *Angular* надає різноманітні інструменти та можливості для розробки великих та складних веб-додатків, такі як модульність, сервіси, анімації та інше.

Активна спільнота та підтримка. *Angular* має широку та активну спільноту розробників, а також офіційну документацію та підтримку від *Google*.

Angular є одним з популярних фреймворків для веб-розробки, і його використання розповсюджене для створення сучасних та потужних веб-додатків.

React: *React* – це бібліотека для створення інтерфейсів користувача (*UI*), розроблена Facebook. Вона використовується для розробки односторінкових додатків (*Single Page Application*s або *SPA*) та для реалізації компонентів інтерфейсу в багаторівневих додатках. *React* є відкритим програмним забезпеченням та знаходить широке застосування в індустрії веб-розробки.

Основні характеристики та концепції *React* включають:

Компоненти. *React* базується на концепції компонентів, які є самостійними і відтворюють певні частини інтерфейсу користувача. Компоненти можна вкладати один в одного для створення складних інтерфейсів.

Віртуальний *DOM*. React використовує віртуальний *DOM* для оптимізації роботи з реальним *DOM*. Віртуальний *DOM* є проміжним шаром між логікою програми та реальним *DOM*, що дозволяє оптимізувати процес оновлення інтерфейсу.

Єдина сторона даних. Реактивний підхід *React* дозволяє організовувати потік даних так, щоб зміни даних автоматично відображалися на інтерфейсі, що спрощує розробку та управління станом додатка.

JSX (*JavaScript XML*). *JSX* є розширенням *JavaScript*, яке дозволяє описувати структуру користувацького інтерфейсу за допомогою синтаксису, що нагадує *HTML*. Код перетворюється на певні виклики у *React.createElement*, щоб створювати *React*-елементи.

Декларативний підхід. *React* використовує декларативний підхід до розробки, де розробник описує те, що має бути відображено, і *React* бере на себе управління тим, як це робити.

Екосистема та спільнота. Реакт має широку екосистему, що включає в себе різноманітні бібліотеки та інструменти, такі як *Redux* для управління станом, *React Router* для маршрутизації та багато іншого. Велика та активна спільнота сприяє поширенню знань та рішень.

React є однією з найпопулярніших бібліотек для розробки веб-інтерфейсів, і вона використовується в численних великих та успішних проектах.

Vue.js: *Vue.js* – це прогресивний *JavaScript*-фреймворк для розробки інтерфейсів користувача. Він є легким, гнучким та простим у використанні, і часто використовується для створення односторінкових додатків (*SPA*) та динамічних інтерфейсів взагалі. *Vue.js* орієнтований на підрахунок, і його легко інтегрувати в існуючі проекти.

Основні характеристики та концепції *Vue.js* включають:

Реактивний підхід. *Vue.js* використовує реактивний підхід, що означає, що інтерфейс автоматично оновлюється при зміні даних. Розробник визначає залежності між даними та представленням, і *Vue.js* відстежує та оновлює відповідні частини інтерфейсу.

Компоненти. *Vue.js* базується на компонентній архітектурі. Інтерфейс розбивається на маленькі та самостійні компоненти, які можна легко вкладати один в одного.

Директиви. Директиви в *Vue.js* – це спеціальні атрибути, які розширюють можливості *HTML* елементів. Наприклад, директива *v-if* визначає умову відображення елемента.

Декларативний синтаксис шаблонів. *Vue.js* використовує декларативний синтаксис шаблонів, який надає зручний спосіб опису *HTML*-структури та зв'язків між даними та інтерфейсом.

Vue CLI. Vue CLI – є інструментом командного рядка для швидкого створення та налаштування проектів *Vue*. Він має вбудовану систему для керування залежностями, розгортанням та іншими завданнями.

Легка інтеграція. *Vue.js* легко інтегрується в існуючі проекти, і ви можете поступово впроваджувати його в окремих частинах додатку.

Активна спільнота. Як і в інших популярних фреймворках, *Vue.js* має широку та активну спільноту розробників, яка підтримує вас як у вигляді документації, так і в аспекті розв'язання проблем.

Vue.js використовується як для невеликих проектів, так і для розробки великих та масштабних додатків. Його легкість вивчення та гнучкість роблять його популярним вибором для розробників веб-додатків.

Express.js: *Express.js* – це мінімалістичний та гнучкий веб-фреймворк для розробки веб-додатків та веб-служб на мові програмування *JavaScript*, яка використовується в середовищі *Node.js*. *Express* надає набір інструментів для роботи з *HTTP*-запитами та відповідями, обробки маршрутів та інших задач, пов'язаних з веб-розробкою.

Основні характеристики та концепції *Express.js* включають:

Маршрутизація. *Express* дозволяє визначати маршрути, які визначають, як сервер обробляє *HTTP*-запити. Це дозволяє розробникам визначати, як обробляти різні шляхи (*URL*).

Middleware. *Middleware* – це функції, які мають доступ до об'єктів запиту та відповіді, а також можуть модифікувати їх або завершувати обробку запиту. *Middleware* може бути використано для реалізації різних функціональностей, таких як логування, обробка сесій, автентифікація та інше.

Шаблонізатори. *Express* не включає в себе шаблонізатор, але ви можете легко інтегрувати будь-який шаблонізатор, для створення виглядів (наприклад, *EJS, Pug, Handlebars*).

Статичні файли. *Express* дозволяє легко обслуговувати статичні файли (зображення, таблиці стилів, скрипти) за допомогою вбудованого *middleware express.static*.

HTTP-запити та відповіді. *Express* дає дуже спрощений спосіб обробки *HTTP* - запитів та відповідей, що дозволяє розробникам ефективно створювати веб-додатки.

Розширюваність. *Express* є дуже гнучким та розширюваним. Ви можете використовувати багато сторонніх модулів (*middleware*) для додавання функціональності.

Спрощений сервер. *Express* виводить *Node.js HTTP*-сервер на більш вищий рівень, дозволяючи швидко створювати веб-додатки з меншим обсягом коду.

Express.js є одним з найпопулярніших веб-фреймворків для розробки серверної частини додатків на мові *JavaScript*. Він широко використовується для створення *API*, односторінкових додатків та інших веб-проектів на платформі *Node.js*.

Ember.js: *Ember.js* – це відкритий веб-фреймворк, який дозволяє розробляти амбіційні веб-додатки, які просто обмінюються даними та взаємодіють зі службами ззовні. Він покликаний допомагати розробникам створювати масштабовані та легко обслуговувані веб-додатки, використовуючи конвенції та підходи, що сприяють продуктивності.

Основні концепції та характеристики *Ember.js* включають:

Компонентна архітектура. *Ember* використовує компонентну архітектуру для розділення інтерфейсу користувача на малинькі та повторно використовувані компоненти.

Конвенції перед конфігураціями. *Ember* розповсюджує конвенції перед конфігураціями, що означає, що ви можете швидко почати роботу, дотримуючись певних стандартів та назв, але ви, звісно ж, можете вибрати свої шляхи, якщо це необхідно.

Handlebars та *HTMLBars*. *Ember* використовує шаблонізатор *Handlebars* (а в новіших версіях – *HTMLBars*), який є виразним та легким у використанні, дозволяючи вбудовувати динамічні дані у *HTML*.

Автоматична оновлення інтерфейсу користувача. *Ember* використовує підхід двостороннього зв'язування даних, завдяки чому зміни в моделі автоматично оновлюють інтерфейс користувача та навпаки.

Роутинг. *Ember* має класну вбудовану систему роутингу, яка дозволяє їй легко управляти станом додатка та визначати, які екрани повинні бути відображені для різних *URL*-шляхів.

Очікувані *HTTP*-запити (*Conventional JSON API*). *Ember* надає підтримку для очікуваних *HTTP*-запитів, що відповідають конвенціям *JSON API* для легкого взаємодії з сервером.

Ember.js спрощує створення амбіційних веб-додатків, забезпечуючи структуру та конвенції, які полегшують розробку та утримання великих проектів. Його підходи допомагають розробникам зосередитися на функціональності, замість налаштувань та деталей.

Meteor: *Meteor* – це відкритий фреймворк для розробки повностекових веб-додатків. Він надає інструменти та конвенції для швидкого створення як клієнтської, так і серверної частини веб-додатків, використовуючи мову програмування *JavaScript*. Основною особливістю *Meteor* є підтримка реального часу (*real-time*) взаємодії між клієнтом та сервером.

Основні характеристики та концепції *Meteor* включають:

Повностековий фреймворк. *Meteor* надає повний набір інструментів для розробки як клієнтської, так і серверної частини веб-додатків.

Реальний час. *Meteor* підтримує реальний час взаємодії, що означає, що зміни на сервері автоматично відображаються на клієнтському інтерфейсі без необхідності оновлення сторінки.

Двостороннє зв'язування даних. *Meteor* використовує двостороннє зв'язування даних, що дозволяє автоматично синхронізувати дані між сервером та клієнтом.

Пакетна система: *Meteor* має потужну пакетну систему, яка дозволяє легко використовувати сторонні пакети та розширювати функціональність додатка.

Швидкий розгортання: *Meteor* надає простий спосіб розгортання веб-додатків за допомогою хмарних платформ, таких як *Galaxy*, або на власних серверах.

Інтеграція з базами даних: *Meteor* підтримує інтеграцію з різними базами даних, включаючи *MongoDB*, за допомогою якого можна ефективно працювати з даними.

Однаковий код: *Meteor* дозволяє писати код, який може використовуватися як на клієнті, так і на сервері, що полегшує розробку та утримання коду.

Meteor є інтегрованим та продуктивним фреймворком, призначеним для швидкої розробки веб-додатків, зокрема тих, які вимагають реального часу та інтенсивно використовують *JavaScript* на обох сторонах.

Фреймворки та бібліотеки у *JavaScript* допомагають розробникам прискорити роботу, скоротити час розробки та зменшити кількість помилок завдяки готовим рішенням та структурованому підходу до розробки. Розробники можуть вибирати між різними фреймворками в залежності від своїх потреб та уподобань.

Бібліотеки:

1) *jQuery*: Легковагова бібліотека, яка полегшує взаємодію з *DOM* та виконання анімацій;
2) *D3.js*. Бібліотека для створення динамічних та інтерактивних діаграм та графіків на веб-сторінках;
3) *Lodash*. Утилітарна бібліотека, яка надає багато корисних функцій для роботи з масивами, об'єктами та іншими структурами даних;
4) *axios*: Бібліотека для здійснення *HTTP*-запитів з використанням обіцянок;
5) *Redux*. Контейнер для стану додатку в *React*-додатках, який полегшує управління станом додатку;
6) *RxJS*. Бібліотека для роботи з асинхронним програмуванням та реактивним програмуванням;
7) *Three.js*. Бібліотека для роботи з *3D*-графікою у веб-додатках.

Після визначиня фреймворків та технологій порівняємо їх щоб дізнатися який краще в тому чи іншому аспекті(див. таб. 1.1).

Таблиця 3.1

Порівняння фреймворків

Фреймворк Характеристика	***React***	***Angular***	***Vue.js***	***Express***
Розробник	*Facebook*	*Google*	*Evan You*	Спільнота

Закінчення таблиці 3.1

Фреймворк Харак.	***React***	***Angular***	***Vue.js***	***Express***
Версія	Остання стабільна	Остання стабільна	Остання стабільна	Остання стабільна
Архітектура	Компонентна	Модульна	Компонентна	–
Розмір бандлу	Залежить від застосування	Середній або великий	Малий або середній	Не застосовується
Швидкодія	Висока	Середня	Висока	Висока
Підходи до розробки	Декларативний	Декларативний	Декларативний	Імперативний, Мінімалістичний
Документація	Добра	Добра	Добра	Задовільна
Виртуальний *DOM*	Так	Так	Так	–
Реактивність	Частково	Так	Так	–
Спільнота	Велика	Велика	Велика	Велика

Підтримка *TypeScript*	Так	Так	Так	Ні

CSS (*Cascading Style Sheets*) – це мова стилів, яка використовується для визначення вигляду та форматування веб-документів, написаних мовою *HTML* (або іншими мовами розмітки). За допомогою *CSS* можна контролювати зовнішній вигляд елементів сторінки, таких як шрифти, кольори, розташування, відступи, рамки та інші стилізаційні аспекти.

Основні риси *CSS*. Відокремлення стилів від структури: *CSS* дозволяє відокремлювати стилі від *HTML-коду*, що полегшує управління та підтримку веб-сайту. Стилі можна оголошувати в окремих *CSS-файлах* і підключати їх до *HTML-*
документів.

Каскадування (*Cascading*). У *CSS* існує концепція каскаду, яка визначає порядок застосування стилів, які можуть бути оголошені в різних місцях. Це дозволяє контролювати пріоритети стилів та їх спадкування від батьківських елементів.

Використання селекторів. Селектори в *CSS* визначають, до яких елементів будуть застосовані конкретні стилі. Селектори можуть бути класами, ідентифікаторами, тегами, атрибутами та іншими визначеними критеріями.

Різні типи одиниць вимірювання. *CSS* підтримує різні типи одиниць вимірювання, такі як пікселі, відсотки, *rem*, і багато інших, для забезпечення гнучкості визначення розмірів та відступів.

Анімації та переходи. *CSS* дозволяє створювати анімації та переходи, щоб робити сторінку більш динамічною та привабливою для користувача.

Можливості адаптації до різних пристроїв. *CSS* має функціонал для адаптації веб-сайтів до різних пристроїв і розмірів екранів, що дозволяє розробляти адаптивні та мобільно-дружні веб-додатки.

Ось деякі з переваг *CSS*:

Розділення структури та стилів. *CSS* дозволяє розділити *HTML*-код (структуру сторінки) та стилі (вигляд сторінки). Це сприяє покращенню читабельності, обслуговуваності і можливості змінювати вигляд сторінок без змін в *HTML*-структурі.

Гнучкість та каскадність. Каскадність у *CSS* дозволяє визначити пріоритети стилів, що дозволяє легко наслідувати стилі та визначати їх на різних рівнях, шарах веб-сторінки.

Повторне використання коду. Стилі в *CSS* можна використовувати повторно для різних елементів на сторінці чи навіть на різних сторінках, що спрощує утримання та зменшує обсяг коду.

Відділення від пристосуваності. *CSS* дозволяє створювати адаптивний дизайн, розміщаючи стилі окремо від *HTML*. Це дозволяє створювати один і той же контент для різних пристроїв, просто змінюючи стилі.

Різноманітність властивостей та значень. *CSS* має широкий спектр властивостей та значень, що дозволяє розробникам точно керувати виглядом елементів. Властивості можуть стосуватися шрифтів, кольорів, розмірів, відступів та багатьох інших аспектів.

Анімація та переходи. *CSS* дозволяє створювати анімацію та плавні переходи між різними станами елементів. Це сприяє покращенню користувацького досвіду та візуально покращує веб-сайти.

Сумісність та виправлення помилок. *CSS* є стандартизованою мовою, і вона добре підтримується більшістю сучасних браузерів. Крім того, веб-розробники можуть використовувати інструменти для виявлення та виправлення помилок у стилях.

Підтримка різних мов та пристроїв. *CSS* дозволяє розробникам створювати мультимовні та мультимедійні сайти, а також легко пристосовувати вигляд веб-сторінок до різних пристроїв та розмірів екрану.

Взагалі, *CSS* володіє рядом переваг, які роблять його важливим інструментом для розробників, які працюють над веб-проектами.

Git – це розподілена система керування версіями (*Version Control System* або *VCS*), яка використовується для відстеження змін в програмному коді під час розробки програмного забезпечення. Розроблена Лінусом Торвальдсом для керування розробкою ядра операційної системи *Linux*, *Git* став однією з найпопулярніших систем керування версіями та широко використовується в інших проектах програмного забезпечення.

Основні концепції та функціональності *Git* включають:

Розподілена керівництво версіями. *Git* є розподіленою системою керування версіями, що означає, що кожен розробник має повний репозитарій з повною історією проекту на своєму комп'ютері. Це сприяє ефективній роботі в команді та можливості працювати офлайн.

Коміти. Коміти представляють собою зміни в коді або інших файлах, які вносяться в репозитарій. Кожен коміт має унікальний ідентифікатор та повідомлення, що пояснює зміни.

Відгалуження (*Branching*). *Git* дозволяє створювати відгалуження (гілки) для роботи над окремими функціями чи змінами, не впливаючи на основну лінію розробки. Після завершення можливо злити відгалуження з основною гілкою.

Злиття (*Merging*). *Git* дозволяє об'єднувати різні гілки або відгалуження, щоб об'єднати зміни та розвинути проект.

Відстеження змін. *Git* зберігає повну історію змін, тому розробники можуть відстежувати, хто і коли вніс конкретні зміни, а також переглядати попередні версії проекту.

Віддалені репозитарії. *Git* дозволяє спільно використовувати проекти, використовуючи віддалені репозитарії, які розташовані на серверах. Це полегшує спільну роботу великих команд та співпрацю над великими проектами.

Git став стандартом у світі розробки програмного забезпечення, і його використання поширене серед розробників для керування версіями свого коду.

Microsoft Visual Studio Code (*VS Code*) – це безкоштовний і відкритий текстовий редактор, розроблений компанією *Microsoft*. Він призначений для розробки програмного забезпечення та підтримується на різних операційних

системах, включаючи *Windows*, *macOS* і *Linux*. Ось декілька ключових аспектів *Microsoft Visual Studio Code*:

Легкий та Надійний. *VS Code* відзначається швидкістю та легкістю використання. Він працює швидко навіть на менш потужних комп'ютерах і не завантажує системні ресурси.

Розширення та Екосистема. *VS Code* має широкий спектр розширень, які дозволяють розробникам налаштовувати і розширювати функціональність редактора. Розширення доступні для різних мов програмування, фреймворків і інструментів розробки.

Інтеграція з *Git*. *VS Code* має вбудовану підтримку системи контролю версій *Git*, що робить спільну роботу в команді набагато простіше.

Автодоповнення та *IntelliSense*. Редактор надає автодоповнення для коду та функцію *IntelliSense*, яка допомагає розробникам швидше та точніше писати код.

Відладка. *VS Code* надає можливості відладки з використанням різних мов та платформ. Він підтримує відладку на віддалених серверах і вбудований відладчик для популярних мов та фреймворків.

Термінал. Редактор має вбудований термінал, який дозволяє виконувати команди безпосередньо з редактора.

Робота з різними мовами та платформами: *VS Code* підтримує багато мов програмування і фреймворків, що дозволяє розробникам працювати з різними технологіями та платформами.

Спільнота та Підтримка. *VS Code* має велику та активну спільноту користувачів, що означає, що завжди є доступ до довідки, плагінів та розширень.

Microsoft Visual Studio Code дуже популярний серед розробників, оскільки він поєднує в собі простоту текстового редактора з можливостями інтегрованого середовища розробки *(IDE*). Він допомагає розробникам писати, тестувати та налагоджувати код більш продуктивно та зручно.

Так як мовою програмування була обрана *JS*, то можна зробити висновок, що найкращим доступним вибором середовища програмування буде *Microsoft Visual Studio Code*. Мій вибір впав на створення сторінки на якій будуть проходити

розрахунки для знаходження найоптимальнішого шляху для літака між містами використовуючи лексикографічний метод.

Visual Studio Code (VS Code), *WebStorm* і *Atom* – це редактори коду та інтегровані середовища розробки, які використовуються програмістами для написання коду. Кожен з цих інструментів має свої переваги та недоліки, і вибір між ними зазвичай залежить від індивідуальних потреб користувача. Нижче наведено деякі можливі переваги *Visual Studio Code* порівняно із *WebStorm* та *Atom*:

Швидкість та легкість.

Visual Studio Code. Відомий своєю швидкістю та легкістю. Запускається швидко, має менше витрат пам'яті та може легко працювати з різними типами проектів.

WebStorm. Багатофункціональне середовище розробки, але може вимагати більше ресурсів в порівнянні з *VS Code*.

Atom: Також може бути дещо важким у порівнянні з *VS Code*.

Розширюваність та екосистема.

Visual Studio Code. Має широкий вибір розширень (*extensions*) з великою спільнотою. Дозволяє користувачам налаштовувати редактор за своїми потребами.

WebStorm. Включає в себе багато функцій "з коробки", але може бути менш гнучким у плані розширюваності порівняно із *VS Code*.

Atom. Також має розширюваність, але в порівнянні з *VS Code* може бути менше розширень та менша спільнота.

Багатомовність та вбудована підтримка:

Visual Studio Code. Має велику кількість плагінів для підтримки різних мов програмування та фреймворків. Підтримка вбудованих інструментів для розробки.

WebStorm. Спеціалізований на веб-технологіях та має вбудовану підтримку багатьох фреймворків.

Atom. Підтримує різні мови програмування, але може вимагати встановлення плагінів для повноцінної підтримки.

Інтеграція та взаємодія з іншими інструментами.

Visual Studio Code. Добре інтегрується з іншими інструментами та сервісами розробки. Має вбудовану підтримку *Git*.

WebStorm. Також надає інтеграцію з багатьма іншими інструментами, але може бути менше універсальним порівняно із *VS Code*.

Atom: Забезпечує інтеграцію з іншими інструментами, але може вимагати деяких додаткових налаштувань.

Обираючи між цими редакторами коду, важливо враховувати конкретні потреби та особливості вашого проекту, а також власні вподобання щодо інтерфейсу та робочого процесу.

Створення застосунку починаємо з оголошення всіх змінних які в подальшому будуть використані.

Наступним кроком буде оголошення функцію *setup()*, яка викликається тільки один раз, під час запуску програми. Завдяки цій функції визначається початкові властивості середовища, такі як наприклад розмір екрану чи колір заднього фону.

Наступний крок це розміщення точок, які слугують умовними містами.

Щоб візуалізувати точки(міста), викличемо функцію *draw()*. Ця функція викликається одночасно після *setup()*, функція *draw()* безперервно виконує рядки коду, що містяться в її блоці, поки програма не буде зупинена або не буде викликано *noLoop()*. *draw()* автоматично виводиться і ніколи не повинен виявлятися явно.

Робота функціїя повина завжди контролюватися за допомогою *noLoop()*, *redraw()* і *loop()*. Після того, як *noLoop()* зупиняє виконання коду у функції *draw()*, *redraw()* змушує код всередині *draw()* виконується один раз, а *loop()* змушує код всередині *draw()* продовжувати безперервне виконання.

Після того як ми взяли координати які вибрані навмання та створимо з них кулі для кожного міста завдяки циклу та припишемо їм порядковий номер.

Після того я точки сформовані, з’єднаємо її завдяки методу *vertex()*.

Усі фігури будуються шляхом з’єднання ряду вершин. *vertex()* використовується для визначення координат вершин для точок, ліній, трикутників, квадратів і багатокутників. Він використовується виключно у функціях *beginShape()* і *endShape()*.

Код модуля:

```
// Оголошуємо змінні
let cities = [];
let totalCities = 6;
let order = [];
let totalPermutations;
let count = 0;
let recordDistance;
let bestEver;
// Створюємо прямокутник та створюємо точки
function setup() {
  createCanvas(800, 615);
  for (let i = 0; i < totalCities; i++) {
    let v = createVector(random(width), random(height / 2));
    cities[i] = v;
    order[i] = i;
  }
  let d = calcDistance(cities, order);
  recordDistance = d;
  bestEver = order.slice();
  totalPermutations = factorial(totalCities);
  console.log(totalPermutations);
}
function draw() {
  background(0);
  fill(100, 255, 23);
  for (let i = 0; i < cities.length; i++) {
    ellipse(cities[i].x, cities[i].y, 20, 20);
    text("\n" + i, cities[i].x, cities[i].y);
  }
```

```
stroke(255, 0, 255);
strokeWeight(4);
noFill();
beginShape();
for (let i = 0; i < order.length; i++) {
  let n = bestEver[i];
  vertex(cities[n].x, cities[n].y);
}
endShape();
translate(0, height / 2);
stroke(255);
strokeWeight(1);
noFill();
beginShape();
for (let i = 0; i < order.length; i++) {
  let n = order[i];
  vertex(cities[n].x, cities[n].y);
}
endShape();
let d = calcDistance(cities, order);
if (d < recordDistance) {
  recordDistance = d;
  bestEver = order.slice();
}
textSize(32);
fill(255);
let percent = 100 * (count / totalPermutations);
text(nf(percent, 0, 2) + "% виконано", 20, height / 2 - 100);
text("найкращий шлях " + bestEver, 22, height / 2 - 50);
nextOrder();
```

```
}
```

Наступним кроком має сенс перейти до самого алгоритму, завдяки якому все працює.

1. Для початку знаходимо такий *x*, який буде найбільшим але буде менше ніж наступне *P[x] < P[x+1]*, а якщо такого *x* немає, то програму завершено.

Тобто, треба знайти порядковий номер числа у масиві, яке буде ментше ніж наступне. Для прикладу візьмемо масив з 6 цифр: [0,1,5,6,9,3]. Найменшим буде число 6 з порядковим номером 3.

Код пункту 1:

```
let largestI = -1;
 for (let i = 0; i < order.length - 1; i++) {
   if (order[i] < order[i + 1]) {
     largestI = i;
   }
 }
 if (largestI == -1) {
   noLoop();
   fill(200, 139, 4);
   text("Програму завершено,\n найкращий шлях -> " + bestEver, 30, 3);
 }
```

2. Треба знайти найбільше *y* яке буде більше за *P[x] > P[y]*. Для [0,1,5,6,9,3] таким буде 3, з порядковим номером 5. Тобто 3 < 5.

Код пункту 2:

```
let largestJ = -1;
 for (let j = 0; j < order.length; j++) {
   if (order[largestI] < order[j]) {
     largestJ = j;
   }
 }
```

3. Замінити *P[x]* та *P[y]*. Для цього створимо функцію для заміни та викличемо її підставивши наші данні з двох попередніх пунктів.

Код пункту 3:

```
function swap(a, i, j) {
  let temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}

swap(order, largestI, largestJ);
```

4. Перевернути *P[x+1 .. n]*.

Код пунту 4:

```
let endArray = order.splice(largestI + 1);
  endArray.reverse();
  order = order.concat(endArray);
```

Повний код програми:

Код з файлу *index.html*:

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/0.7.3/p5.min.js"></script>
    <script
src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/0.7.3/addons/p5.dom.min.js">
</script>
<script
src="https://cdnjs.cloudflare.com/ajax/libs/p5.js/0.7.3/addons/p5.sound.min.js">
</script>
    <link rel="stulesheet" href="css/style.css" />
```

```
    <meta charset="utf-8" />
    <title>Lexicographic Order</title>
  </head>
  <body>
    <script src="js/script.js"></script>
  </body>
</html>
```

Код з файлу *script.js*:

```
// Оголошуємо змінні
let cities = [];
let totalCities = 6;
let order = [];
let totalPermutations;
let count = 0;
let recordDistance;
let bestEver;
// Функція в якій задається початкове
function setup() {
  createCanvas(800, 615);
  for (let i = 0; i < totalCities; i++) {
    let v = createVector(random(width), random(height / 2));
    cities[i] = v;
    order[i] = i;
  }
  let d = calcDistance(cities, order);
  recordDistance = d;
  bestEver = order.slice();
  totalPermutations = factorial(totalCities);
  console.log(totalPermutations);
```

```
}
// Функція яка вимальовує точки та з'єднання
function draw() {
  background(0);
  fill(100, 255, 23);
  for (let i = 0; i < cities.length; i++) {
    ellipse(cities[i].x, cities[i].y, 20, 20);
    text("\n" + i, cities[i].x, cities[i].y);
  }
  stroke(255, 0, 255);
  strokeWeight(4);
  noFill();
  beginShape();
  for (let i = 0; i < order.length; i++) {
    let n = bestEver[i];
    vertex(cities[n].x, cities[n].y);
  }
  endShape();
  translate(0, height / 2);
  stroke(255);
  strokeWeight(1);
  noFill();
  beginShape();
  for (let i = 0; i < order.length; i++) {
    let n = order[i];
    vertex(cities[n].x, cities[n].y);
  }
  endShape();
  let d = calcDistance(cities, order);
  if (d < recordDistance) {
```

```
    recordDistance = d;
    bestEver = order.slice();
  }
  textSize(32);
  fill(255);
  let percent = 100 * (count / totalPermutations);
  text(nf(percent, 0, 2) + "% виконано", 20, height / 2 - 100);
  text("найкращий шлях " + bestEver, 22, height / 2 - 50);
  nextOrder();
}
// Функція заміни значень
function swap(a, i, j) {
  let temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
// Функція яка визначає відстань
function calcDistance(points, order) {
  let sum = 0;
  for (let i = 0; i < order.length - 1; i++) {
    let cityAIndex = order[i];
    let cityA = points[cityAIndex];
    let cityBIndex = order[i + 1];
    let cityB = points[cityBIndex];
    let d = dist(cityA.x, cityA.y, cityB.x, cityB.y);
    sum += d;
  }
  return sum;
}
// Алгоритм лексичного порядку
```

```
function nextOrder() {
  count++;
  // Пункт 1
  let largestI = -1;
  for (let i = 0; i < order.length - 1; i++) {
    if (order[i] < order[i + 1]) {
      largestI = i;
    }
  }
  if (largestI == -1) {
    noLoop();
    fill(200, 139, 4);
    text("Програму завершено,\n найкращий шлях -> " + bestEver, 30, 3);
  }
  // Пункт 2
  let largestJ = -1;
  for (let j = 0; j < order.length; j++) {
    if (order[largestI] < order[j]) {
      largestJ = j;
    }
  }
  // Пункт 3
  swap(order, largestI, largestJ);
  // Пункт 4
  let endArray = order.splice(largestI + 1);
  endArray.reverse();
  order = order.concat(endArray);
}
// Визначення факторіалу
function factorial(n) {
```

```
  if (n == 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

Код з файлу *style.css*:

```
html, body {
  margin: 0;
  padding: 0;
}
canvas {
  display: block;
}
```

3.3. Інтерфейс модуля

Зображення процесу виконання обчислення програми(див. рис. 3.3).

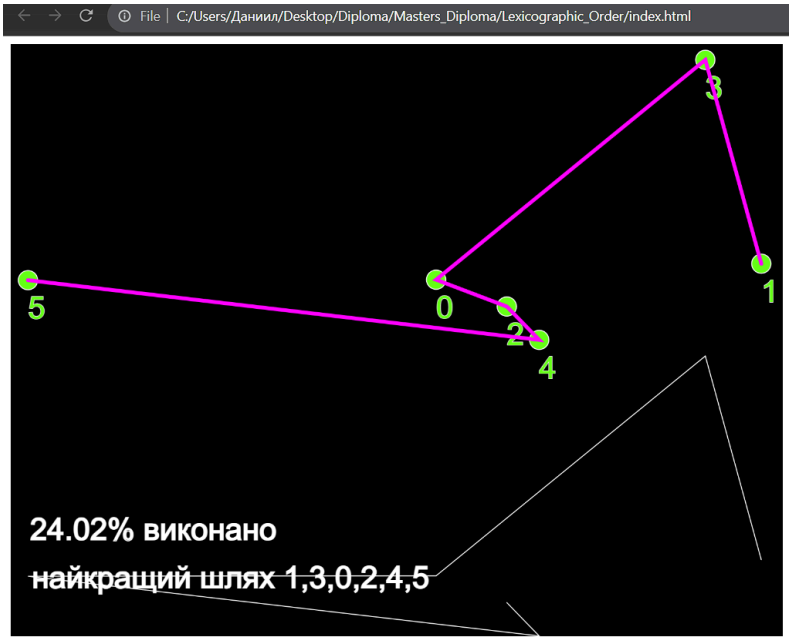

Рис. 3.3. Скріншот виконання програми

Успішне виконання обчислення і виводом шляху на екран(див. рис. 3.4).

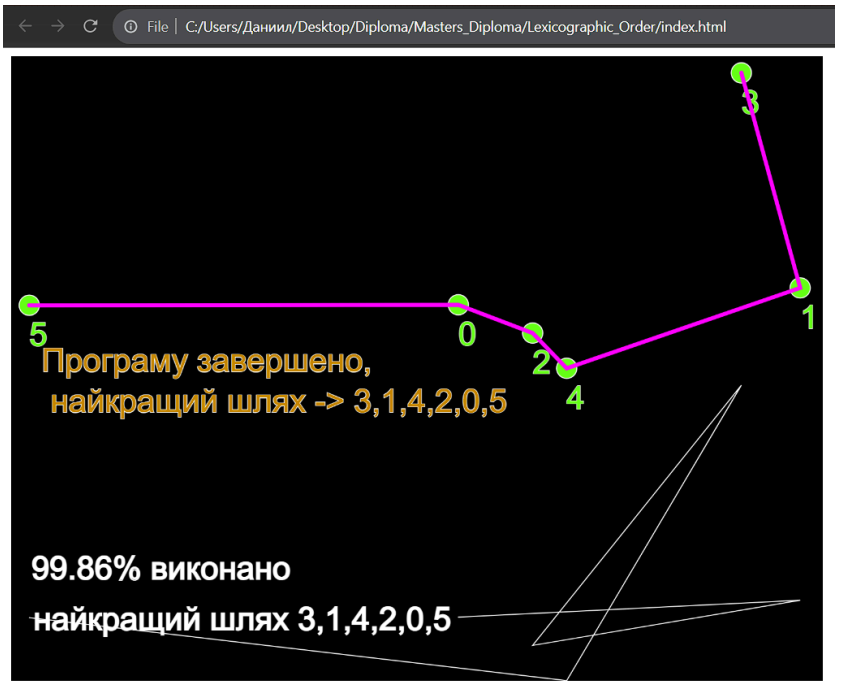

Рис. 3.4. Скріншот успішного виконання програми

3.4. Висновки до розділу

У цьому розділі було визначено що таке *JavaScript* та визначено популярні фреймфорки цієї мови програмування. Було визначенно для яких цілей можна використати *JS*. Його можна використати для:

1) Розробка статичних або динамічних веб-сайтів;
2) Створення віджетів, плавних анімацій та ефектів для поліпшення взаємодії користувача;
3) Розробка браузерних ігор за допомогою бібліотек, таких як *Phaser* або *Three.js*.

Наведений алгоритм у пункті 3.1 для створення наступної перестановки в лексикографічному порядку є ефективним та чітко структурованим. Описані кроки дозволяють систематично міняти елементи перестановки, гарантуючи, що кожна нова перестановка буде наступною за попередньою в лексикографічному порядку.

Використовуючи приклад з трьома містами (A, B, C) та початковою перестановкою (1, 2, 0), було продемонстровано, як вказаний алгоритм працює. Визначені значення x та y, проведено обмін елементів, а потім виконано обертання

частини перестановки. Результатом була нова перестановка (5, 1, 7, 6, 4, 2, 3, 8, 9), яка слідує за вхідною перестановкою в лексикографічному порядку.

Отже, описаний алгоритм може бути успішно використаний для генерації наступних перестановок за лексикографічним порядком і знаходження їхнього відображення на прикладі конкретної ситуації з містами.

Далі було визначенно програмне забезпечення модуля, яке складалося з опису мов програмування та технологій які були використані при розробці застосунку. Також детально була описана сама розробка застосунку з покроковим виконанням пунктів алгоритму у вигляді програмного коду.

В кінці було показано скріншоти застосунку. Процеси виконання програми та та її завершення.

4)

ВИСНОВКИ

1. Досліджено технологію організації авіаційних перевезень.

1) А саме: Планування маршруту. Визначення оптимального маршруту для пасажирів або вантажу, враховуючи місця призначення, доступність аеропортів та інші фактори;
2) Бронювання і квитки. Організація процесу бронювання місць на польотах і видача авіаквитків пасажирам;
3) Забезпечення вантажу. Для вантажних перевезень, це включає в себе пакування, маркування та обробку вантажу перед відправленням;
4) Відправлення і прийом вантажу. Організація процесу завантаження вантажу на борт літака та його вивантаження після приземлення;
5) Пасажирське обслуговування. Забезпечення пасажирів комфортним перельотом, включаючи реєстрацію, обслуговування на борту і послуги в аеропорту;
6) Безпека. Забезпечення безпеки пасажирів і вантажу під час авіаперельотів. Це включає в себе безпеку бортового обладнання і персоналу;
7) Логістика. Координація руху літаків, обслуговуючого персоналу і вантажів для забезпечення пунктуальності і ефективності авіаперевезень;
8) Документація. Збір та обробка всіх необхідних документів, таких як літні дані, митні декларації тощо.

2. Здійснено аналіз існуючих методів побудови маршрутів авіаційних перевезень. Здійснений аналіз в ключає в себе декілька прикладів систем організації авіаційних перевезень:

– Інтернаціональна асоціація повітряного транспорту (*IATA*);
– Європейська організація з безпеки повітряного транспорту (*Eurocontrol*);
– Федеральна авіаційна адміністрація (*FAA*);
– *Air Navigation Service Providers* (*ANSPs*);
– Системb АБС (*Airport Operational Control Centre);*

– *SITA* (*Société Internationale de Télécommunications Aéronautiques*).

3. Доведена можливість застосування до розв’язання задачі маршрутизації авіаційних перевезень алгоритму генерації перестановок за допомогою лексикографічного впорядкування.

Застосування алгоритму генерації перестановок за допомогою лексикографічного впорядкування має ряд переваг, а саме:

1) Ефективність. Лексикографічне впорядкування дозволяє генерувати перестановки відразу в правильному порядку без необхідності перебору всіх можливих варіантів. Це робить алгоритм більш ефективним порівняно з іншими методами генерації перестановок;
2) Простота реалізації. Алгоритм лексикографічного впорядкування перестановок виглядає просто і може бути легко реалізований в багатьох мовах програмування. Він використовує прості операції порівняння та обміну для отримання наступної перестановки;
3) Можливість роботи зі згенерованими перестановками. Після того, як перестановка згенерована, можна легко визначити вартість маршруту для задачі комівояжера та визначити, чи є цей маршрут оптимальним;
4) Пам'ять. Алгоритм працює ітеративно і не вимагає зберігання усіх можливих перестановок у пам'яті одразу. Це особливо важливо, коли масштаби задачі великі, і обсяг пам'яті обмежений. Таким чином астосування алгоритму генерації перестановок за допомогою лексикографічного впорядкування є раціональним та доцільним.

4. Визначено технічне та інформаційне забезпечення модуля. Маршрутизації авіаційних перевезень.

5. Здійснено програмну реалізацію модуля маршрутизації авіаційних перевезень на основі евристичного алгоритму генерації перестановок за допомогою лексикографічного впорядкування (операційне середовище *Microsoft Windows*, мова програмування *JavaScript*, *CSS*, *HTML*, інтегроване середовище програмування *Microsoft Visual Studio Code*) з використанням програмного інструментарію вбудованих бібліотек *JavaScript* та бібліотеки

p5.js.

6. Розроблено інтерфейс модуля маршрутизації авіаційних перевезень.

7. Пропонований модуль може бути використаний у складі автоматизованих системи управління польотами, які автоматизують навігацію та управління польотами літака. Вона включає в себе автопілот, системи управління польотом, навігаційні комп'ютери та інші елементи.

СПИСОК БІБЛІОГРАФІЧНИХ ПОСИЛАНЬ ВИКОРИСТАНИХ ДЖЕРЕЛ

1) Бойченко С.В., Іванченко О.В. Положення про дипломні роботи (проєкти) випускників Нацiанального авіаційного університету. – К.: НАУ, 2017. – 63с.
2) ДСТУ ГОСТ 7.1:2006 «Бібліографічний запис. Бібліографічний опис. Загальні вимоги та правила складання»;
3) ДСТУ 3582: 2013 «Бібліографічний опис скорочення слів і словосполучень в українській мові»;
4) ДСТУ 8302-2015 «Інформація та документація. Бібліографічне посилання. Загальні положення та правила складання».
5) *Aditya Bhargava Grokking Algorithms. An illustrated guide for programmers and other curious people. Manning; First Edition, 2016, 256 p.*
6) *Taha, H. A. Operations Research: An Introduction. Pearson Education, 2011. 120 p.*
7) *Robert C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. Pearson; 1st edition, 2008, 464 p.*
8) *Paul A. Longley, Michael F. Goodchild, David J. Maguire, David W. Rhind Geographic Information Science and Systems. John Wiley & Sons, 2015, 496 p.*
9) *David G. Luenberger Optimization by Vector Space Methods.Wiley-Interscience, 1997, 352 p.*
10) *Gormley C., Tong Z. Mastering Elasticsearch: Building a Scalable, Open Source Search Engine. O'Reilly Media,* 2013, 354 *p.*
11) *Grossman D., Frieder O., Karlgren J. Information Retrieval: Algorithms and Heuristics. Springer,* 2004, 292 *p.*
12) *Papadimitriou, C. H., & Steiglitz, K. Combinatorial Optimization: Algorithms and Complexity. Dover Publications,* 1998, 200 *p.*
13) *Blum, C., & Roli, A. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. ACM Computing Surveys (CSUR)* . 2003, 268-308 *p.*
14) *Introduction to Air Transport Economics: From Theory to Applications by Bijan Vasigh, Ken Fleming, and Tom Tacker,* Routledge, 2018, 520 p.

15) *Osman, I. H. Metastrategy simulated annealing and tabu search algorithms for the vehicle routing problem. Annals of Operations Research,* 2003, 421-451 *p.*

16) *Barnhart, C., Johnson, E. L., Nemhauser, G., Savelsbergh, M., & Vance, P. H.. Branch-and-price: Column generation for solving huge integer programs. Operations Research, 2010*, 316-329 *p.*

17) *Bierwirth, C., & Mattfeld, D. C. A parallel genetic algorithm for the resource-constrained project scheduling problem.* 2014, 634-644 *p.*

18) *Salhi, S., & Nagy, G. . A cluster insertion heuristic for the VRP. Computers & Operations Research,* 2007, 1025-1037 *p.*

19) *Toth, P., & Vigo, D. (Eds.).The Vehicle Routing Problem (Monographs on Discrete Mathematics and Applications). Society for Industrial and Applied Mathematics. 1987, 386 p.*

20) Сучасний підручник з *JavaScript - 2023. Access mode*: uk.javascript.info. *(lastaccess. 05.12.23). – Title from the screen.*

21) *CSS basics – 2023. Access mode: developer.mozilla.org . (lastaccess. 05.12.23). – Title from the screen.*

22) *HTML basics - 2023. Access mode: developer.mozilla.org. (lastaccess. 04.12.23). – Title from the screen.*

23) Початок роботи з *VS Code*. *Access mode*: code.visualstudio.com.

24) *Get Started with p5.js – 2023. Access mode: p5js.org. (lastaccess. 07.12.23). – Title from the screen.*

25) Масиви у *JS* та як їх використовувати. *URL*: *developer.mozilla.org. (lastaccess. 07.12.23). – Title from the screen.*

26) Посібник з *JavaScript – 2023. Access mode: w3schools.com. (lastaccess. 10.12.23). – Title from the screen.*

27) Посібник з *HTML. – 2023. Access mode: w3schools.com. (lastaccess. 10.12.23). – Title from the screen.*

28) Посібник з *CSS – 2023. Access mode w3schools.com. (lastaccess. 10.12.23). – Title from the screen.*