**НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ**

**«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ  
імені ІГОРЯ СІКОРСЬКОГО»**

**Інститут прикладного системного аналізу**

**Кафедра системного проектування**

**Курсова робота**

**з дисципліни «Алгоритмізація та програмування»**

**на тему: «Найкоротші шляхи. Алгоритм Дейкстри»**

Виконала:

студентка I курсу, групи ДА-02

Лесечко Олеся Романівна \_\_\_\_\_\_\_\_\_\_

Керівник:

Романов В.В. \_\_\_\_\_\_\_\_\_\_

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.

Студентка \_\_\_\_\_\_\_\_\_\_\_\_\_

Київ – 2021 рік

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **ІПСА КПІ ім. І. Сікорського** | | | | | | |
| (назва вищого закладу освіти) | | | | | | |
| Кафедра | | ***Системне проектування*** | | | | |
| Дисципліна | | ***Алгоритмізація та програмування*** | | | | |
| Галузь знань | | ***12 Інформаційні технології*** | | | | |
| Курс | ***перший*** | | Група | ***ДА-02*** | Семестр | ***другий*** |

**ЗАВДАННЯ**

**на курсовий проект(роботу) студента**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Лесечко Олеся Романівна | | | | | |
| (прізвище, ім’я, по батькові) | | | | | |
| 1. Тема проекту(роботи) | Найкоротші шляхи. Алгоритм Дейкстри | | | | |
|  | | | | | |
|  | | | | | |
|  | | | | | |
| 2. Строк здачі студентом закінченого проекту(роботи) | | | ***25.05.2021 р.*** | | |
| 3. Вихідні дані до проекту(роботи) | |  | | | |
|  | | | | | |
|  | | | | | |
|  | | | | | |
| 4. Зміст розрахунково-пояснювальної записки (перелік питань, які підлягають розробці) | | | | |  |
| ***1. Постановка задачі*** | | | | | |
| ***2. Алгоритм розв’язку задачі*** | | | | | |
| ***3. Загальна блок-схема алгоритму та опис алгоритму*** | | | | | |
| ***4. Опис програмного продукту.*** | | | | | |
| ***5. Тестування програми*** | | | | | |
| ***6. Висновки.*** | | | | | |
| ***7. Список використаної літератури.*** | | | | | |
| ***Додаток А. Лістинг. Додаток Б. Блок-Схема.*** | | | | | |
| 5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень) | | | |  | |
| ***1. Загальна блок-схема алгоритму.*** | | | | | |
| ***2. Ілюстрації роботи програми.*** | | | | | |
| 6. Дата видачі завдання | 19.03.2021 | | | | |

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| **№/п** | **Назва етапів курсового проекту (роботи)** | **Строк виконання**  **етапів роботи** | **Примітка** |
| 1. | Вибір теми курсової роботи. Опрацювання літератури. Оформлення листа Завдання. | 2,3-й тиждень лютого |  |
| 2. | Аналіз постановки задачі. Узгодження з керівником попереднього плану роботи | 3,4-й тиждень лютого |  |
| 3. | Вибір та дослідження методів та структур даних, Розробка загального алгоритму | 1-й тиждень березня |  |
| 4. | Перше узгодження з керівником. | 1-й тиждень березня |  |
| 5. | Проектування інтерфейсу. Розробка алгоритмів окремих блоків. | 2-й тиждень березня |  |
| 6. | Друге узгодження з керівником. | 2-й тиждень березня |  |
| 7. | Програмна реалізація. | 3,4-й тижні березня |  |
| 8. | Демонстрація першого варіанту.  Трете узгодження з керівником. | 1-й тиждень квітня |  |
| 9. | Доопрацювання програми. Заключне тестування програми. | 2,3-й тижні квітня |  |
| 10. | Аналіз результатів. Оформлення звіту. | до 2-го тижня травня |  |
| 11. | Захист та демонстрація курсової роботи. | до 25.05– «А»  до 30.05 –«В,С»  до 05.06 – «D,E» |  |

|  |  |  |
| --- | --- | --- |
| Студент |  | Лесечко Олеся Романівна |
|  | (підпис) | (прізвище, ім’я, по батькові) |

|  |  |  |
| --- | --- | --- |
| Керівник |  | Безносик Олександр Юрійович |
|  | (підпис) | (прізвище, ім’я, по батькові) |

|  |  |
| --- | --- |
| 25.05.2021 |  |
| (дата) |  |

**ЗМІСТ**

[Вступ 5](#_Toc73474616)

[РОЗДІЛ 1 ПОСТАНОВКА ЗАДАЧІ 6](#_Toc73474617)

[1.1 Теоретичні відомості 6](#_Toc73474618)

[1.2 Обґрунтування вибору інструментів розробки 8](#_Toc73474619)

[1.3 Постановка задачі проектування 9](#_Toc73474620)

[РОЗДІЛ 2 РОЗРОБКА АЛГОРИТМУ РОЗВ’ЯЗКУ ЗАДАЧІ 10](#_Toc73474621)

[2.1 Вибір зберігання графа 10](#_Toc73474622)

[2.2 Алгоритм пошуку в глибину 10](#_Toc73474623)

[2.3 Математичне підґрунтя алгоритму Дейкстри 13](#_Toc73474624)

[2.4 Побудова алгоритму Дейкстри 14](#_Toc73474625)

[2.5 Функції на основі алгоритму Дейкстри та пошуку в глибину 16](#_Toc73474626)

[РОЗДІЛ 3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ 17](#_Toc73474627)

[3.1 Опис програми 17](#_Toc73474628)

[3.2 Інструкція користувачу та приклад роботи 20](#_Toc73474629)

[Висновки 29](#_Toc73474630)

[Література 30](#_Toc73474631)

[Додатки 32](#_Toc73474632)

# Вступ

Існує велика кількість завдань які можна представити у вигляді точок і зв’язків між ними, тобто в термінах теорії графів. Слово «граф» з'явилося в математичній літературі порівняно недавно. Але поняття графа використовується дуже часто не тільки в математиці, але і повсякденному житті під різними назвами: схема, діаграма, карта, лабіринт тощо.

Теорія графів надає виключно зручний апарат для моделювання структур різних систем і відношень між об'єктами найрізноманітнішої природи. Завдяки наочності і простоті цей апарат останнім часом завоював широке визнання і часто використовується при дослідженні та моделюванні різних інформаційних процесів.

Граф являє собою непорожню множину точок та множину дуг, обидва кінці яких належать заданій множині точок. Прикладами графів можуть бути схема метрополітена, схеми залізничних та шосейних доріг, структурні формули молекул, плани виставок та інше, тобто, схеми і плани (чи карти) без масштабу, які показують лише зв'язки між об'єктами.

При зображенні графів на малюнках чи схемах відрізки дуги можуть бути прямолінійними чи криволінійними; довжини дуг та розташування точок є довільним. Точки інакше називають вершинами; дуги - ребрами (дугами) графа. Вершини графа на малюнку виділяються зазвичай кругами хоча б тому, що не всі точки перетину ребер є вершинами графа.

# РОЗДІЛ 1 ПОСТАНОВКА ЗАДАЧІ

## 1.1 Теоретичні відомості

При розв’язанні широкого кола прикладних задач нерідко виникає необхідність знайти маршрут, що зв’язує задані вершини в графі. Маршрутом в графі називається послідовність вершин і ребер, в якій будь-які два сусідні елементи інциденті. Зауважу, що ребра і вершини в маршруті можуть повторюватися. Маршрут називається відкритим, якщо його кінцеві вершини різні, і замкнутим або циклічним в разі. Відкритий маршрут називають ланцюгом (Вершини можуть повторюватися). Ланцюг, якої не повторюється вершини, шляхом (простий). Замкнута ланцюг називається циклом, замкнутий шлях - простий циклом. Ребро графа називається циклічним, якщо в графі існує цикл, постач це ребро. Неорграф без циклів називається ациклічним, орграф без контурів – без контурним [1].

Довжина маршруту дорівнює кількості ребер у ньому (причому кожне ребро вказується стільки разів воно зустрічається в даному маршруті).

Найменша з довжин простих циклів називається обхватом графа [2].

Подання графів в пам'яті - це спосіб зберігання інформації в ребрах графа, що дозволяє вирішувати наступні завдання:

* + - Для двох даних вершин і перевірити, чи пов'язані вершини і ребром.
    - Перебрати всі ребра, що виходять з даної вершини.

Основними способами зберігання графа є

* + - Матриця суміжності.
    - Списки (або безлічі) суміжних вершин.
    - Список ребер.

Матриця суміжності графа G зі скінченною кількістю вершин n (пронумерованих числами від 1 до n) — це квадратна матриця A розміру n, в якій значення елементу aij рівне числу ребер з i-ї вершини графа в j-у вершину.

Іноді, особливо у разі неорієнтованого графа, петля (ребро з i-ї вершини в саму себе) вважається за два ребра, тобто значення діагонального елементу aii в цьому випадку рівне подвоєному числу петель навколо i-ї вершини [3].

У випадку, якщо у графі багато ребер, списки суміжності на практиці дають мало переваг порівняно з матрицею суміжності. Зате якщо в графі відносно небагато ребер, то списки суміжності часто виявляються кориснішими й ефективнішими.

Цей спосіб має певні переваги, такі як ефективніше використання пам’яті і швидший обхід суміжних вершин, але має також і недоліки.

Списки суміжності і такі операції над ними, як додавання й вилучення ребер із графа та проходження по суміжних вершинах, можна реалізувати на базі динамічних масивів.

Списки суміжності доцільно використовувати при розв’язанні широкого класу задач, зокрема в задачах пошуку на графі [1].

В теорії графів, задача про найкоротший шлях полягає в знаходженні шляху між двома вершинами (або вузлами) такого графу, що сума ваг ребер з яких він складається мінімальна.

Прикладом може бути знаходження найкоротшого шляху між двома пунктами на дорожній мапі, в цьому випадку вершинами є пункти, а ребрами - відтинки дороги із вагами, рівними часу, необхідному для подолання цього відтинку. Така задача іноді згадується як задача про найкоротший шлях між парою вершин, щоб відрізнити від наступних узагальнень:

* Задача про найкоротші шляхи з одного входу, тут ми маємо знайти найкоротші шляхи між вхідною вершиною v та всіма іншими вершинами графа.
* Задача про найкоротші шляхи з одним виходом, тут ми маємо знайти найкоротші шляхи з усіх вершин графа до однієї вихідної v. Може бути зведена до задачі з одним входом шляхом зміни на зворотні ребер графа.
* Задача про найкоротші шляхи для всіх пар, тут ми маємо знайти найкоротші шляхи між кожною парою вершин v, v' в графі.

Є два найбільш ефективні алгоритми знаходження найкоротшого шляху:

* алгоритм Дейкстри;
* алгоритм Флойда;

Ці алгоритми легко виконуються при малій кількості вершин у графі. При збільшенні їх кількості завдання пошуку найкоротшого шляху ускладняється.

## 1.2 Обґрунтування вибору інструментів розробки

Програма була написана на мові C++.

Переваги мови C++

* Швидкодія. Швидкість роботи програм на С++ практично не поступається програмам на С, хоча програмісти отримали в свої руки нові можливості і нові засоби;
* Масштабованість. На мові C++ розробляють програми для найрізноманітніших платформ і систем;
* Можливість роботи на низькому рівні з пам'яттю, адресами, портами. (Що, при необережному використанні, може легко перетворитися на недолік);
* Можливість створення узагальнених алгоритмів для різних типів даних, їхня спеціалізація, і обчислення на етапі компіляції, з використанням шаблонів;
* Підтримуються різні стилі та технології програмування, включаючи традиційне директивне програмування, ООП, узагальнене програмування (шаблони, макроси).

Середовище розробки IDE Microsoft Visual Studio, тому що зручний та інтуїтивний інтерфейс.

## 1.3 Постановка задачі проектування

Для того, щоб зробити цю роботу треба, щоб програма робила наступні дії:

* Можливість користувача задавати кількість міст та створювати між ними дороги.
* Пошук найкоротшого шляху з міста а в місто б.
* Пошук всіх можливих шляхів з міста а в місто б.
* Пошук мінімальної відстані від а до будь-якого міста.
* Пошук всіх міст, які не пов'язані між собою дорогами.

# РОЗДІЛ 2 РОЗРОБКА АЛГОРИТМУ РОЗВ’ЯЗКУ ЗАДАЧІ

## 2.1 Вибір зберігання графа

Розглянувши основні способи зберігання графа, було прийнято рішення використовувати матрицю суміжності через наявність таких переваг:

1. Швидкий пошук і перевірка наявності або відсутності певного ребра між двома вершинами.
2. Швидке додавання або видалення ребра між двома вершинами.

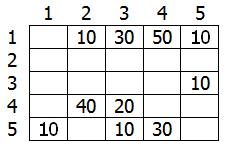


Рис. 2.1 Матриця суміжності

Матриця буде реалізована як динамічний масив.

## 2.2 Алгоритм пошуку в глибину

Алгоритм пошуку в глибину - алгоритм для обходу дерева, структури подібної до дерева, або графа. Робота алгоритму починається з кореня дерева (або іншої обраної вершини в графі) і здійснюється обхід в максимально можливу глибину до переходу на наступну вершину.

Нехай  - простий зв'язний граф, усі вершини якого позначено попарно різними символами. У процесі пошуку вглиб вершинами графа G надають номери (DFS-номери) та певним способом даних для збереження множин, яку називають стеком. Зі стеку можна вилучити тільки той елемент, який було додано до нього останнім: стек працює за принципом «останній прийшов — перший вийшов» (LIFO). Інакше кажучи, додавання й вилучення елементів у стеку відбувається з одного кінця, який називається верхівкою стеку. DFS-номери вершини х позначають DFS(х).

Ініціалізація:

1. Почати з довільної вершини v. Виконати DFS(v) =1. Включити цю вершину в стек.
2. Розглянути вершину у верхівці стеку: нехай це вершина х. Якщо всі ребра, інцидентні вершині х, позначено, то перейти до кроку 4, інакше - до кроку 3.
3. Нехай {x, y} - непозначене ребро. Якщо DFS(у) уже визначено, то позначимо ребро {x, y} потовщено суцільною лінією, визначити DFS(у) як черговий DFS-номер, включити цю вершину в стек і перейти до кроку 2.
4. Виключити вершину х зі стеку. Якщо стек порожній, то зупинитись, інакше - перейти до кроку 2 [4].

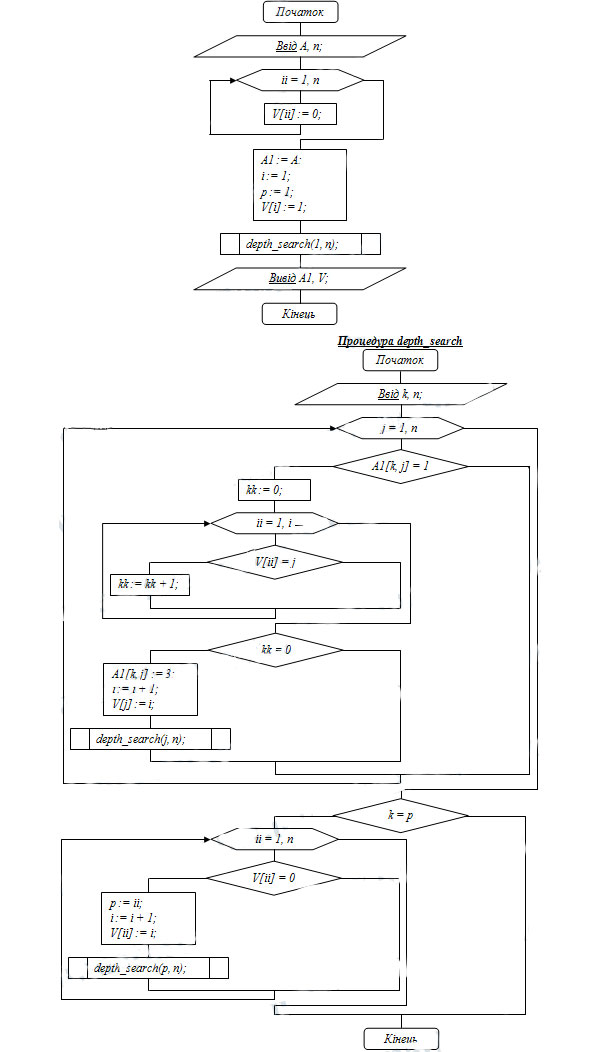


Рис. 2.2 Узагальнена блок-схема алгоритму пошуку в глибину

## 2.3 Математичне підґрунтя алгоритму Дейкстри

Основне твердження, на якому заснована коректність алгоритму Дейкстри, наступне. Стверджується, що після того як будь-яка вершина v стає поміченою, поточна відстань до неї  вже є найкоротшою, і більше не змінюватиметься. Далі опишемо доказ цього твердження.

Для першої ітерації справедливість його очевидна - для вершини s маємо , що і є довжиною найкоротшого шляху до неї. Нехай тепер це твердження виконано для всіх попередніх ітерацій, тобто всіх уже помічених вершин; доведемо, що воно не порушується після виконання поточної ітерації. Нехай v - вершина, обрана на поточній ітерації, тобто вершина, яку алгоритм збирається помітити. Доведемо, що дійсно дорівнює довжині найкоротшого шляху до неї (позначимо цю довжину через).

Розглянемо найкоротший шлях P до вершини v. Зрозуміло, цей шлях можна розбити на два шляхи: P\_1, що складається тільки з позначених вершин (як мінімум стартова вершина s буде в цьому шляху), і інша частина шляху P\_2 (вона теж може включати помічені вершини, але починається обов'язково з непомічених). Позначимо через p першу вершину шляху P\_2, а через q - останню вершини шляху P\_1.

Доведемо спочатку наше твердження для вершини p, тобто доведемо рівність . Однак це практично очевидно: адже на одній з попередніх ітерацій ми вибирали вершину q і виконували релаксацію з неї. Оскільки (в силу самого вибору вершини p) найкоротший шлях до p дорівнює найкоротшому шляху до q плюс ребро (p, q), то при виконанні релаксації з q величина  дійсно встановиться в потрібне значення.

Внаслідок невід'ємності вартостей ребер довжина найкоротшого шляху  (а вона по щойно доведеним дорівнює) не перевищує довжини найкоротшого шляху до вершини v. З огляду на те  (адже алгоритм Дейкстри не міг знайти більш короткого шляху, ніж це взагалі можливо), в результаті отримуємо співвідношення: .

З іншого боку, оскільки і p, і v - вершини непомічені, то так як на поточної ітерації була обрана саме вершина v, а не вершина p, то отримуємо іншу нерівність: .

З цих двох нерівностей укладаємо рівність , а тоді зі знайдених до цього співвідношень отримуємо:  що і потрібно було довести [6].

## 2.4 Побудова алгоритму Дейкстри

Алгоритм знаходить найкоротшу відстань від однієї з вершин графа до всіх інших і працює тільки для графів без ребер негативного ваги. Кожній вершині приписується вага - це вага шляху від початкової вершини до даної. Також кожна вершина може бути виділена.

Якщо вершина виділена, то шлях від неї до початкової вершини найкоротший, якщо немає - то тимчасовий.

Обходячи граф, алгоритм вважає для кожної вершини маршрут, і якщо він виявляється найкоротшим, виділяє вершину.

Вагою даної вершини стає вага шляху. Алгоритм закінчує свою роботу, дійшовши до кінцевої вершини, і вагою найкоротшого шляху стає вага кінцевої вершини.

Ініціалізація:

1. Всім вершинам, за винятком першої, присвоюється вага, яка дорівнює нескінченності, а першій вершині - 0.
2. Всі вершини не виділено.
3. Перша вершина оголошується поточною.
4. Вага всіх невиділених вершин перераховується по формулі: вага невиділеної вершини є мінімальне число з старого ваги даної вершини, суми ваги поточної вершини і ваги ребра, що з'єднує поточну вершину з невиділеної.
5. Серед невиділених вершин шукаємо вершину з мінімальною вагою. Якщо така не знайдена, тобто вага всіх вершин дорівнює нескінченності, то маршрут не існує. Отже, вихід. Інакше, поточної стає знайдена вершина. Вона ж виділяється.
6. Якщо поточної вершиною виявляється кінцева, то шлях знайдений, і його вага є вага кінцевої вершини.
7. Перехід на крок 4.

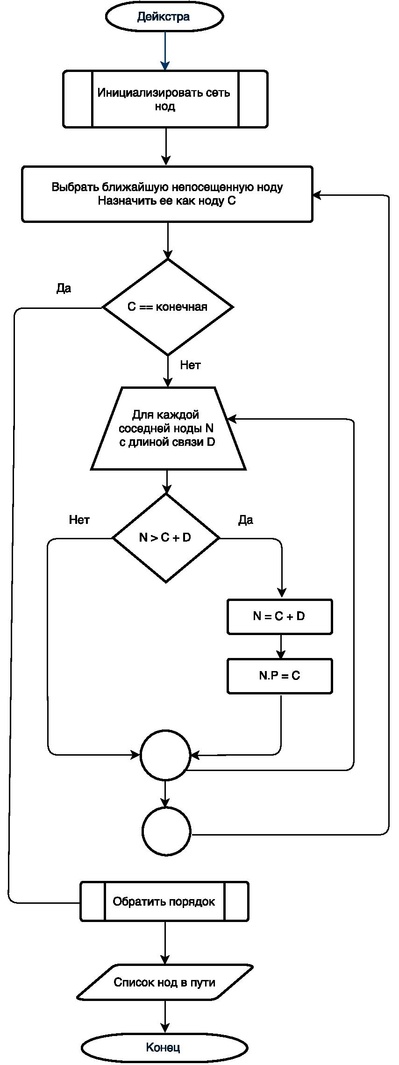


Рис. 2.3 Узагальнена блок-схема алгоритму Дейкстри

На кожному кроці до множини S додається і з решти вершин, відстань до якої від початкової вершини менше, ніж для інших, що залишилися вершин.

При цьому будемо використовувати масив D, в Який записуються довжини найкоротших шляхів для кожної вершини.

Коли безліч S буде містити всі вершини графа, тоді масив D міститиме довжини найкоротших шляхів від початкової вершини до кожної вершині [5].

## 2.5 Функції на основі алгоритму Дейкстри та пошуку в глибину

* Пошук всіх можливих шляхів з міста а в місто б.

Методом перебору знаходимо всі можливі шляхи переходу з однієї вершини в іншу, якщо вершина в якій ми знаходимось дорівнює кінцевій вершині, тоді виводимо шлях який зберігається в списку. Інакше за допомогою функції пошуку в глибину шукаємо вершину у яку ми зможемо потрапити далі.

* Пошук мінімальної відстані від а до будь-якого міста.

Принцип дії такий самий як і в функції для пошуку мінімальної відстані алгоритмом Дейкстри з міста а в місто б, різниця тільки в тому що нам на вихід повертається масив з усіма мінімальними відстанями від вершини а, а не одне число.

* Пошук всіх міст, які не пов'язані між собою дорогами.

За допомогою функції пошук в глибину, відмічаємо всі вершини в які ми можемо потрапити і виводимо ті в які не зможемо потрапити.

# РОЗДІЛ 3 РЕАЛІЗАЦІЯ ПРОГРАМНОГО ПРОДУКТУ

## 3.1 Опис програми

Розробка програми в основному базується на алгоритмі, вказаному у попередньому розділі.

Програма була написана в одному основному файлі main.cpp. У ньому розміщено абсолютне все, від ініціалізації до самих обчислень.

Під час розробки програми були використані такі бібліотеки:

* #include <iostream> для роботи зі змінними вводу-виводу
* #include <ctime> для створення випадкових елементів
* #include <queue> std::queue
* #include <list> std::list
* #include <windows.h> для роботи кирилиці

1. Функція початкового введення значень

int input(int& n, int& m) отримує змінні n, m що являються посиланням на аргумент. Оскільки посилання на змінну обробляється так само, як і сама змінна, то будь-які зміни, внесені в посилання, приведуть до змін вихідного значення аргументу. Користувач вводить кількість міст n, і кіль шляхів m, потім обирає ввести дороги самостійно або створити їх випадково. Функція повертає значення “1” якщо користувач обрав ввести дороги самостійно, або “2”, якщо обрав створити дороги випадково.

1. Функція підготовки матриці суміжності

void arrayPreparation(int\*\* arr, int n) функція отримує покажчик на двовимірний масив та розмір матриці. Спочатку ми виділяємо масив покажчиків, а потім перебираємо кожен елемент масиву покажчиків і виділяємо динамічний масив для кожного елемента цього масиву. Та присвоюємо всім елементам значення “-1”.

1. Функція випадкового заповнення матриці суміжності

void createRandom(int n, int m, int\*\* data) функція отримує розмір матриці n, кількість шляхів m та покажчик на двовимірний масив. Функція випадково вибере m шляхів від однієї вершини до іншої та відстань між ними.

1. Функція самостійного заповнення матриці суміжності

void createPersonally(int n, int m, int\*\* data) функція отримує розмір матриці n, кількість шляхів m та покажчик на двовимірний масив. Користувач сам обирає вершини та відстань між ними.

1. Функція Алгоритм Дейкстри

int Dijkstra\_Algorithm(int n, int a, int b, int\*\* data) функція отримує розмір матриці n, початкову вершину a, кінцеву вершину b та покажчик на двовимірний масив. Всім вершинам, за винятком першої, присвоюється вага, яка дорівнює “-1”, а першій вершині - 0. Всі вершини не виділено. Перша вершина оголошується поточною. Вага всіх невиділених вершин перераховується по формулі: вага невиділеної вершини є мінімальне число з старого ваги даної вершини, суми ваги поточної вершини і ваги ребра, що з'єднує поточну вершину з невиділеної. Серед невиділених вершин шукається вершина з мінімальною вагою. Якщо така не знайдена, тобто вага всіх вершин дорівнює нескінченності, то маршрут не існує. Отже, вихід. Інакше, поточної стає знайдена вершина. Вона ж виділяється. Якщо поточної вершиною виявляється кінцева, то шлях знайдений, і його вага є вага кінцевої вершини. Функція повертає мінімальну відстань від вершини a до вершини b.

1. Функція знаходження всіх можливих шляхів з вершини a у вершину b

void paths\_a\_to\_b(int n, int a, int b, int\*\* data) функція отримує розмір матриці n, початкову вершину a, кінцеву вершину b та покажчик на двовимірний масив. Функція викликає рекурсивну функію paths\_a\_to\_b findAllWays(n, a, b, data, relations, &way), яка методом перебору знайде всі можливі шляхи виходу з однієї вершини в іншу.

1. Рекурсивна функція перебору всіх шляхів від вершини a до b

void findAllWays(int n, int ver, int b, int\*\* data, int\*& relations, list <int>\* way) функція отримує розмір матриці n, вершину ver в якій ми знаходимося у цей момент, кінцеву вершину b, покажчик на двовимірний масив та список в якому зберігатися шлях який ми вже пройшли від вершини а до вершини ver. Якщо вершина в якій ми знаходимось дорівнює кінцевій вершині, тоді виводити шлях який зберігається в списку way. Інакше в циклі шукаємо вершину у яку ми зможемо потрапити та викликаємо цю функцію заново.

1. Функція для знаходження найкоротшого шляху з а в будь-яку вершину

void Dijkstra\_Algorithm2(int n, int a, int\*\* data) функція отримує розмір матриці n, початкову вершину a та покажчик на двовимірний масив. Всім вершинам, за винятком першої, присвоюється вага, яка дорівнює “-1”, а першій вершині - 0. Всі вершини не виділено. Перша вершина оголошується поточною. Вага всіх невиділених вершин перераховується по формулі: вага невиділеної вершини є мінімальне число з старого ваги даної вершини, суми ваги поточної вершини і ваги ребра, що з'єднує поточну вершину з невиділеної. Серед невиділених вершин шукається вершина з мінімальною вагою. Якщо така не знайдена, тобто вага всіх вершин дорівнює нескінченності, то маршрут не існує. Отже, вихід. Інакше, поточної стає знайдена вершина. Вона ж виділяється. Якщо поточної вершиною виявляється кінцева, то шлях знайдений, і його вага є вага кінцевої вершини. В кінці виводитися мінімальна відстань від вершини а до решти вершин.

1. Функція для знаходження всіх вершин не пов'язаних між собою

void unrelated\_cities(int n, int\*\* data) функція отримує розмір матриці n та покажчик на двовимірний масив. За допомогою функції пошук в глибину, відмічаємо всі вершини в які ми можемо потрапити і виводимо ті в які не зможемо потрапити.

1. Функція пошук у глибину

void DFS(int n, int ver, int\*\* data, int\*& relations) функція отримує розмір матриці n, покажчик на двовимірний масив та посилання на масив в якому ми будемо відзначати вершини в які є можливість потрапити. Вибираємо будь-яку вершину з ще не пройдених, позначимо її як u. Запускаємо процедуру dfs (u) Позначаємо вершину ver як пройдену. Для кожної вершини суміжній з вершиною ver (назвемо її v) запускаємо dfs (v). Повторюємо поки всі вершини не опиняться пройденими.

1. Очищення пам'яті, виділеної під двовимірний динамічний масив

void freeing\_up\_memory(int n, int\*\* data) функція отримує розмір матриці n та покажчик на двовимірний масив. В циклі звільняється пам'ять з-під кожного рядка динамічного масиву і потім звільняється пам'ять з-під масиву покажчиків.

1. Функція зміни кількості вершин

void change\_number\_vertices(int& n, int\*\*& data) функція отримує розмір матриці n та покажчик на двовимірний масив. При введенні нової кількості вершин, стара матриця замінюється на нову, з новою розмірністю

1. Функція зміни кількості ребер

change\_number\_ribs(int n,int &m, int\*\*& data) функція отримує розмір матриці n, кількість шляхів m та покажчик на двовимірний масив. Користувач вибирає додати ребро або видалити. при додаванні обирає вершини і відстань між ними, при видаленні відстань між обраними вершин становиться “-1”.

## 3.2 Інструкція користувачу та приклад роботи

1. Запустити програму. На екрані буде виведено привітання та коротке ознайомлення з її функціоналом (Рис. 3.1).

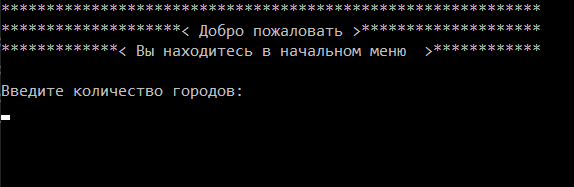


Рис. 3.1 Крок №1

1. Ввести кількість міст (Рис. 3.2).

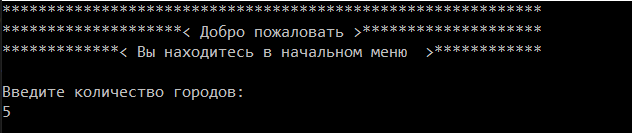


Рис. 3.2 Крок №2

1. Натиснути Enter. На екрані з’явиться надпис про те, що треба ввести друге число (Рис. 3.3).

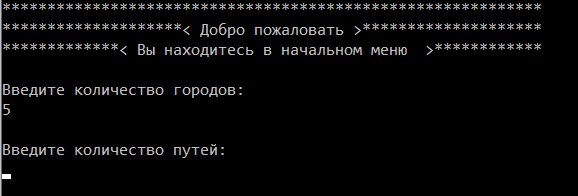


Рис. 3.3 Крок №3

1. Ввести кількість шляхів (Рис. 3.4).

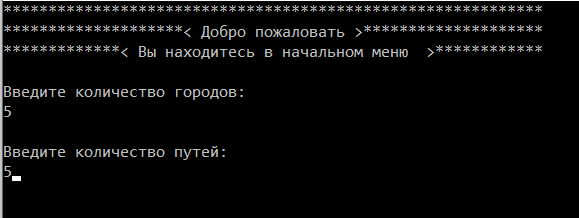


Рис. 3.4 Крок №4

1. Натиснути Enter. На екрані з’явиться меню з переліком можливих операцій (вручну обрати дороги і створити дороги випадково), серед яких користувач може обрати одну. Є необмежена кількість спроб правильно ввести відповідь. Варіанти: 1 – вручну обрати дороги, 2 – створити дороги випадково (Рис. 3.5).

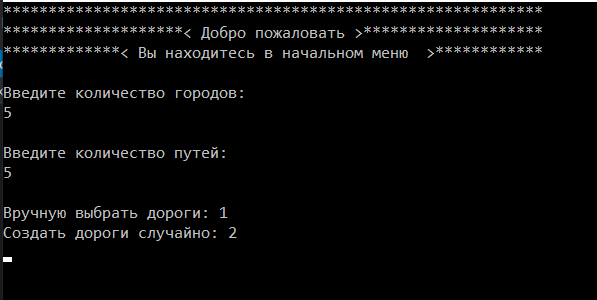


Рис. 3.5 Крок №5

1. Обрати опцію, ввівши 1 або 2. Для прикладу виберемо вручну обрати дороги (1). Натиснути Enter. На екран буде виведено n (n – кількість шляхів) запитів створити дорогу від, до і відстань між ними (Рис. 3.6-3.8).

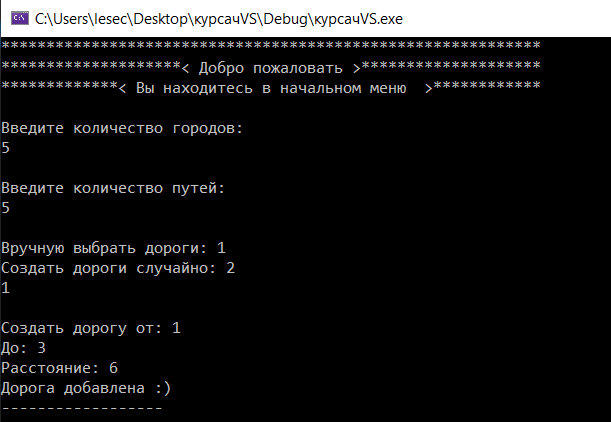


Рис. 3.6 Крок №6

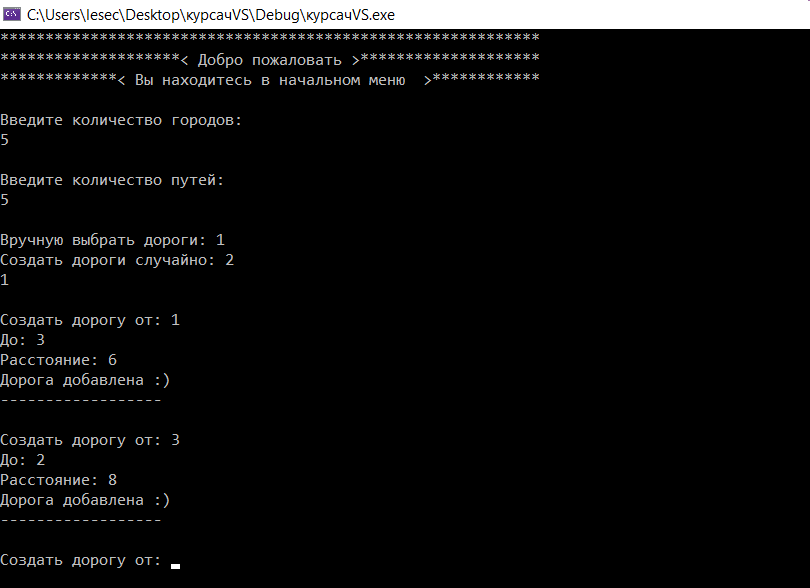


Рис. 3.7 Крок №7

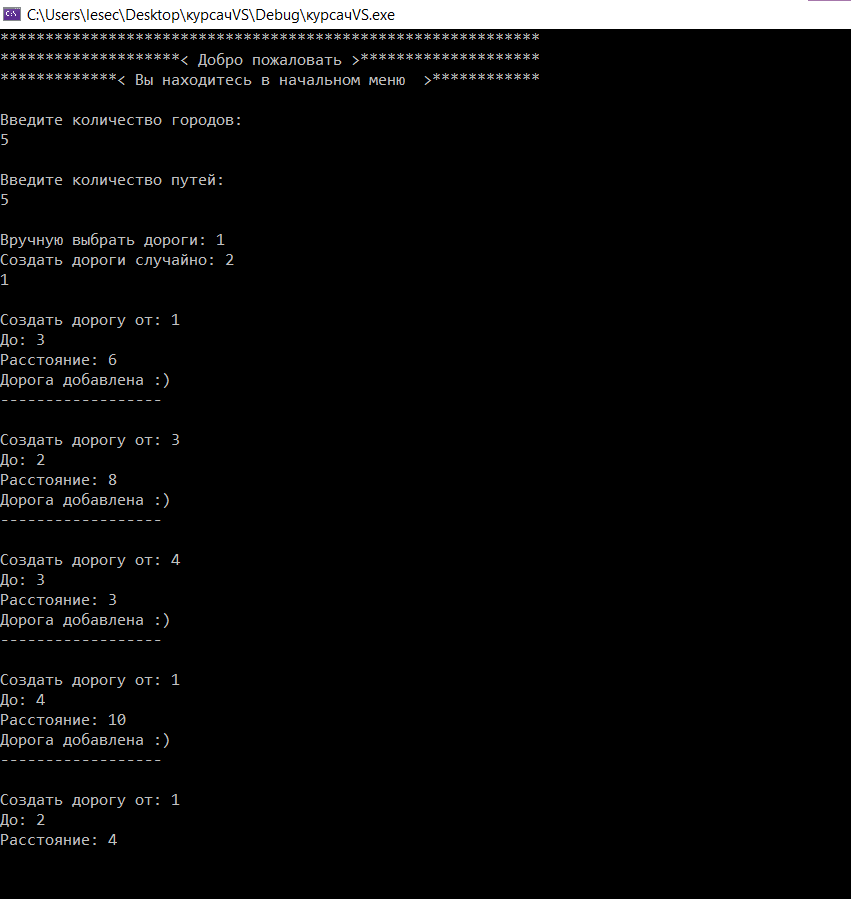


Рис. 3.8 Крок №8

1. Натиснути Enter. На екрані з’явиться меню з переліком можливих операцій (найкоротший шлях з а в б, всі можливі шляхи з а в б, мінімальні відстані від а до будь-якого з міст, всі міста, які пов'язані між собою дорогами, показати матрицю суміжності, вихід), серед яких користувач може обрати одну. Є необмежена кількість спроб коректно ввести відповідь. Варіанти: 1 - найкоротший шлях з а в б, 2 - всі можливі шляхи з а в б, 3 - мінімальні відстані від а до будь-якого з міст, 4 - всі міста, які пов'язані між собою дорогами, 5 - показати матрицю суміжності, 6- змінити кількість ребер, 7 – змінити кількість вершин, 8 - вихід (Рис. 3.9).

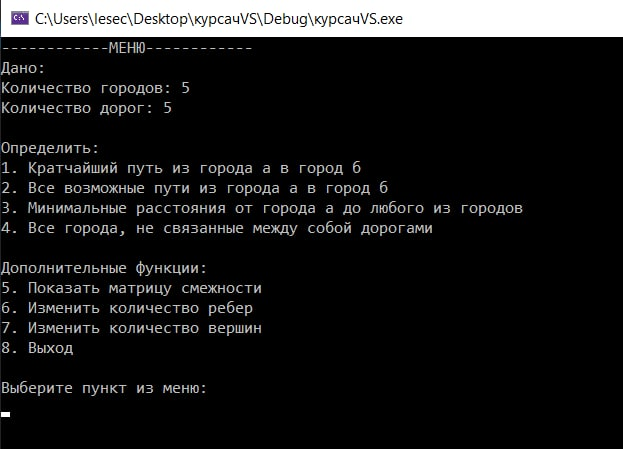


Рис. 3.9 Крок №9

1. Перевіримо функцію для знаходження найкоротшого шляху з міста а в місто б, виберемо пункт 1, на екрані буде запит для вводу вершини а та вершини б (Рис 3.10-3.11).

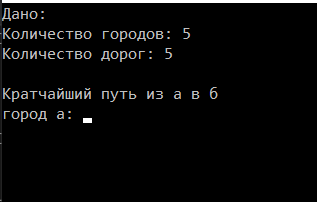


Рис. 3.10 Крок №10

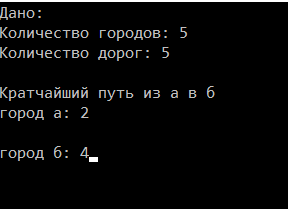


Рис. 3.11 Крок №11

1. Натиснути Enter. На екрані буде показано мінімальний шлях з міста в місто б.(Рис. 3.12).

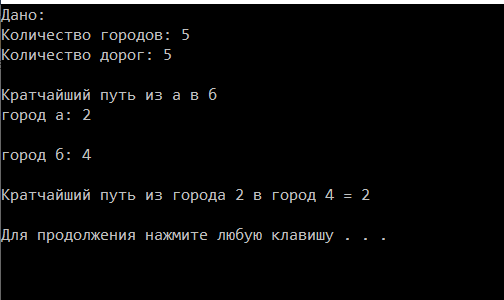


Рис.3.12 Крок №12

1. Далі для прикладу виберемо в меню функцію мінімальна відстань з а до будь якого міста. На екрані буде запит для вводу вершини а (Рис. 3.13).

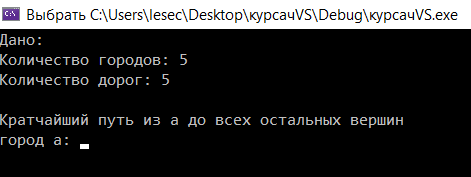


Рис 3.13 Крок № 13

1. Виберемо вершину “1” Натиснути Enter. На екран будуть мінімальні відстані від міста “1” (Рис 3.14).

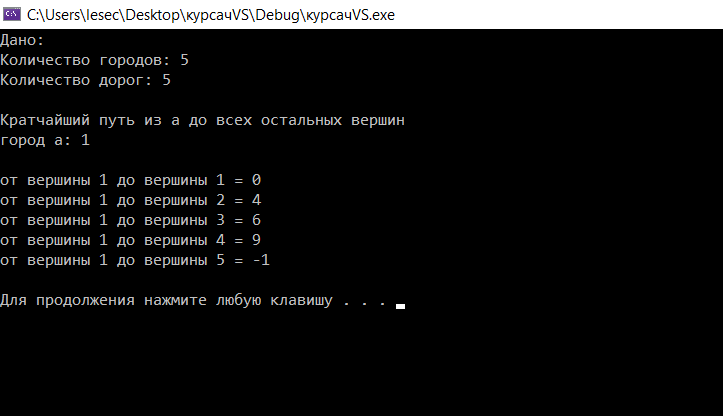


Рис 3.14 Крок №14

1. Натиснути Enter. Ми повернемося до меню (Рис 3.15).

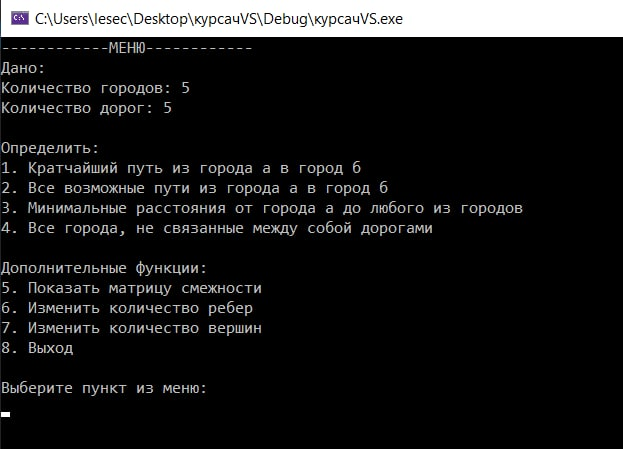


Рис 3.15 Крок №15

1. Для перевірки усіх можливих шляхів з а в б, виберемо в меню пункт “2”, на екрані буде запит для вводу вершини а та вершини б (Рис 3.16-3.17).

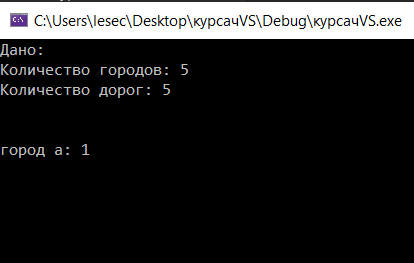


Рис.3.16 Крок №16

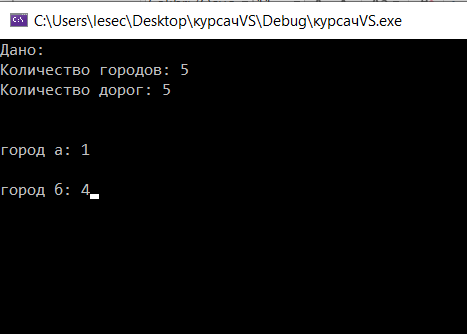


Рис. 3.17 Крок №17

1. Натиснути Enter. На екрані буде показано усі можливі шляхи з міста в місто б. (Рис. 3.18).

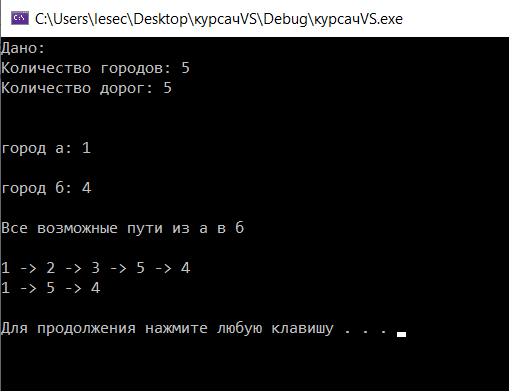


Рис. 3.18 Крок№18

1. Натиснути Enter. Ми повернемося до меню (Рис 3.19).

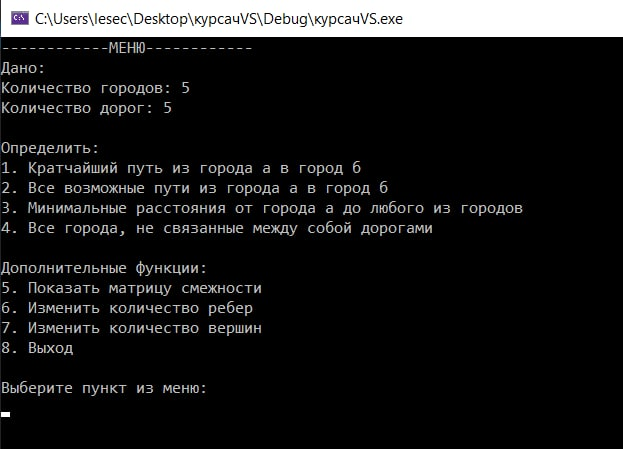


Рис. 3.19 Крок№ 19

1. Для виходу з програми виберемо пункт 8. На екран буде виведено завершальні слова (Рис 3.20).

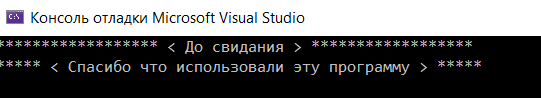


Рис 3.20 Крок №20

# Висновки

У ході виконання курсової роботи було розроблено програму обходу графа, пошуку шляхів між вершинами і знаходження мінімального шляху.

В першому розділі були надані короткі теоретичні відомості про графи, способи їх зберігання та задачу на знаходження найкоротшого шляху між парою вершин.

В другому розділі було описано вибір для зберігання графа, алгоритм пошуку в глибину, математичну складову та алгоритм Дейкстри.

В третьому розділі, фінальному, було розроблений програмний продукт, основні функції та інтерфейс.

Отже, розроблене програмне забезпечення працює правильно, воно є зручним і зрозумілим для користувача. Всі функції, що були передбачені технічним завданням, реалізовані. Всі умови технічного завдання виконані.

# Література

* + 1. Дискретна Математика :: Основні поняття теорії графів.

https://evgavrilenko.ucoz.ru/DS/LEKCIYA\_6-7.pdf

* + 1. Інтернет-енциклопедія «Вікіпедія». Маршрут (теорія графів).

<https://uk.wikipedia.org/wiki/Маршрут_(теорія_графів)>.

* + 1. Інтернет-енциклопедія «Вікіпедія». Матриця суміжності.

[https://uk.wikipedia.org/wiki/Матриця\_суміжності](https://uk.wikipedia.org/wiki/Алгоритм_Дейкстри).

* + 1. Інтернет-енциклопедія «Вікіпедія». Пошук у глибину.

<https://uk.wikipedia.org/wiki/Пошук_у_глибину>.

* + 1. Інтернет-енциклопедія «Вікіпедія». Алгоритм Дейкстры.

<https://uk.wikipedia.org/wiki/Алгоритм_Дейкстри>.

* + 1. Знаходження найкоротших шляхів від заданої вершини до всіх інших вершин алгоритмом Дейкстри.

<https://e-maxx.ru/algo/dijkstra>

* + 1. Бакланов И.Г. Технологии измерений первичной сети. Часть 1. Системы Е1, PDH, SDH. М.: ЭКО-ТРЕНДЗ, 2000. 142 с.
    2. Кормен, Томас X., Лейзерсон, Чарльз И., Ривест, Рональд Л., Штайн, Клиффорд. Алгоритмы: построение и анализ, 2-е издание. М.: Издат. дом «Вильямс», 2005. 1296 с.
    3. Лаврухина Н.В. Методы и модели оценки инвестиционной привлекательности предприятия // Теория и практика общественного развития. 2014. №8. С. 113–117.
    4. Степанов С.Е., Хамер Г.В. Модифицированная модель Фулмера как инструмент диагностики вероятности наступления банкротства компании // В мире научных открытий. 2013. №4. С. 236–247.
    5. Харари Ф. Теория графов / Пер. с англ. и предисл. В.П. Козырева. Под ред. Г.П. Гаврилова. Изд. 2-е. М.: Едиториал УРСС, 2003. 296 с.
    6. Diestel R. Graph Theory - Springer, 2005. 410 p.

# Додатки

## Додаток A. Лістинг

main.cpp

#include <iostream>

#include <ctime>

#include <queue>

#include <list>

#include <windows.h>

using namespace std;

void freeing\_up\_memory(int n, int\*\* data) { // Освобождение памяти

for (int i = 0; i < n ; i++) {

delete[] data[i];

}

delete[] data;

}

int input(int& n, int& m) { // функция изначального ввода значений

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*< Добро пожаловать >\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*< Вы находитесь в начальном меню >\*\*\*\*\*\*\*\*\*\*\*\*\n\n";

cout << "Введите количество городов:\n";

cin >> n;

cout << "\nВведите количество путей:\n";

cin >> m;

cout << "\nВручную выбрать дороги: 1\nСоздать дороги случайно: 2\n";

int opt;

do{

cin >> opt;

}while (opt != 1 && opt != 2);

return opt == 1 ? 1 : 2;

}

void arrayPreparation(int\*\* arr, int n) { // функция заполнения матрицы значениями -1

for (int i = 0; i < n; i++) {

arr[i] = new int[n];

for (int j = 0; j < n; j++) {

arr[i][j] = -1;

}

}

}

void createPersonally(int n, int m, int\*\* data) { // функция заполнения персонально матрицы смежности

for (int i = 0; i < m; i++) {

int x, y, dist;

while (1) {

cout << "\nСоздать дорогу от: ";

cin >> x;

cout << "До: ";

cin >> y;

cout << "Расстояние: ";

cin >> dist;

if (x <= n && y <= n && data[x - 1][y - 1] == -1) {

data[x - 1][y - 1] = dist;

data[y - 1][x - 1] = dist;

cout << "\n--------------------\n";

cout << "Дорога добавлена :) \n";

cout << "--------------------\n\n";

break;

}

if (x > n || y > n) {

cout << "\n-----------------------------------\n";

cout << "Одного из городов не существует :( \n";

cout << "Пожалуйста перепишите... \n";

cout << "-----------------------------------\n\n";

continue;

}

if (data[x - 1][y - 1] != -1) {

cout << "Дорога между городами уже есть, заменить ?(y/n) \n";

char otv;

cin >> otv;

if (otv == 'y') {

data[x - 1][y - 1] = dist;

data[y - 1][x - 1] = dist;

cout << "\n--------------------\n";

cout << "Дорога изменена :) \n";

cout << "--------------------\n\n";

break;

}

else {

continue;

}

}

}

}

}

void createRandom(int n, int m, int\*\* data) { // функция случайного заполнения матрицы смежности

for (int i = 0; i < m; i++) {

int x, y, dist;

while (1) {

x = rand() % n;

y = rand() % n;

dist = (rand() % 10) + 1;

if (x < n && y < n && y != x && data[x][y] == -1) {

break;

}

}

data[x][y] = dist;

data[y][x] = dist;

}

}

int Dijkstra\_Algorithm(int n, int a, int b, int\*\* data) { // функция для нахождения кратчайшого пути из а в б;

queue <int> vertex\_queue; // очередь для хранения всех вершин какие мы должны обойти

int\* dist = new int[n]; // массив дистанций от вершини а

vertex\_queue.push(a);

for (int i = 0; i < n; i++) {

dist[i] = -1; // заполнение массива дистанций значениями -1

}

dist[a] = 0; // почначальная дистанция

while (!vertex\_queue.empty()) {

int ver = vertex\_queue.front();

vertex\_queue.pop();

for (int i = 0; i < n; i++) {

if (data[ver][i] != -1 && (dist[i] == -1 || dist[i] > dist[ver] + data[ver][i])) {

dist[i] = dist[ver] + data[ver][i];

vertex\_queue.push(i);

}

}

}

return dist[b];

}

void findAllWays(int n, int ver, int b, int\*\* data, int\*& relations, list <int>\* way) { // функция перебора всех путей от вершины а до б

way->push\_back(ver);

relations[ver] = 1;

if (ver == b) {

for (auto iter = way->begin(); iter != way->end(); iter++)

{

cout << \*iter + 1;

if (iter != way->end()) {

iter++;

if (iter != way->end()) {

cout << " -> ";

}

iter--;

}

}

cout << endl;

return;

}

for(int i = 0;i < n;i++){

if(data[ver][i] != -1 && relations[i] != 1){

findAllWays(n, i, b, data, relations,way);

way->pop\_back();

relations[i] = 0;

}

}

}

void paths\_a\_to\_b(int n, int a, int b, int\*\* data){ // функция нахождения всех возможных путей из а в б

int\* relations = new int[n]; // связи

list <int> way;

findAllWays(n, a, b, data, relations, &way);

}

void DFS(int n, int ver, int\*\* data, int\*& relations) { // поиск в глубину

relations[ver] = 1;

for (int i = 0; i < n; i++) {

if (data[ver][i] != -1 && relations[i] != 1) {

DFS(n, i, data, relations);

}

}

}

void Dijkstra\_Algorithm2(int n, int a, int\*\* data) { // функция для нахождения кратчайшего пути из а в любую вершину

queue <int> vertex\_queue; // очередь для хранения всех вершин какие мы должны обойти

int\* dist = new int[n]; // массив дистанций от вершини а

vertex\_queue.push(a);

for (int i = 0; i < n; i++) {

dist[i] = -1; // заполнение массива значениями -1

}

dist[a] = 0; // почначальная дистанция

while (!vertex\_queue.empty()) {

int ver = vertex\_queue.front();

vertex\_queue.pop();

for (int i = 0; i < n; i++) {

if (data[ver][i] != -1 && (dist[i] == -1 || dist[i] > dist[ver] + data[ver][i])) {

dist[i] = dist[ver] + data[ver][i];

vertex\_queue.push(i);

}

}

}

cout << endl;

for (int i = 0; i < n; i++) {

cout << "от вершины " << a + 1 << " до вершины " << i + 1 << " = " << dist[i] << endl;

}

cout << endl;

}

void unrelated\_cities(int n, int\*\* data) { // функция для нахождения всех вершин не связанных между собой

int\*\* relations = new int\* [n]; // связи

arrayPreparation(relations, n);

for (int i = 0; i < n; i++) {

DFS(n, i, data, relations[i]);

}

cout << endl;

for (int i = 0; i < n; i++) {

cout << "вершина " << i + 1 << " не имеет дорог к: ";

for (int j = 0; j < n; j++) {

if (relations[i][j] == -1 && i != j) {

cout << j + 1 << " ";

}

}

cout << endl;

}

cout << endl;

}

void print(int n, int\*\* data) {

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

cout << data[i][j] << " ";

}

cout << endl;

}

}

void change\_number\_vertices(int& n, int\*\*& data){ // Изменить количество вершин

cout << "Введите сколько вершин должно быть в графе: \n";

int newN;

cin >> newN;

int\*\* newData = new int\*[newN];

arrayPreparation(newData, newN);

int tmp = min(n, newN);

for (int i = 0; i < tmp; i++) {

for (int j = 0; j < tmp; j++) {

newData[i][j] = data[i][j];

}

}

swap(data,newData);

freeing\_up\_memory(n , newData);

n = newN;

cout << endl;

}

void change\_number\_ribs(int n,int &m, int\*\* data) { // Изменить количество ребер

cout << "1. Добавить ребро\n2. Удалить ребро\n\n";

int opt;

do {

cin >> opt;

} while (opt != 1 && opt != 2);

if (opt == 1) {

int x, y, dist;

while (1) {

cout << "\nСоздать дорогу от: ";

cin >> x;

cout << "До: ";

cin >> y;

cout << "Расстояние: ";

cin >> dist;

if (x <= n && y <= n && data[x - 1][y - 1] == -1) {

data[x - 1][y - 1] = dist;

data[y - 1][x - 1] = dist;

cout << "\n--------------------\n";

cout << "Дорога добавлена :) \n";

cout << "--------------------\n\n";

break;

}

if (x > n || y > n) {

cout << "\n-----------------------------------\n";

cout << "Одного из городов не существует :( \n";

cout << "Пожалуйста перепишите... \n";

cout << "-----------------------------------\n\n";

continue;

}

if (data[x - 1][y - 1] != -1) {

cout << "Дорога между городами уже есть, заменить ?(y/n) \n";

char otv;

cin >> otv;

if (otv == 'y') {

data[x - 1][y - 1] = dist;

data[y - 1][x - 1] = dist;

cout << "\n--------------------\n";

cout << "Дорога изменена :) \n";

cout << "--------------------\n\n";

break;

}

else {

break;

}

}

}

}

else {

int x, y;

cout << "\nУдалить дорогу от: ";

cin >> x;

cout << "До: ";

cin >> y;

if (data[x - 1][y - 1] == -1) {

cout << "\n--------------------------------\n";

cout << "Нельзя удалить дорогу, какой нет\n";

cout << "----------------------------------\n\n";

}

else {

data[x - 1][y - 1] = -1;

data[y - 1][x - 1] = -1;

cout << "\n--------------------------------------------------\n";

cout << "Дорога от " << x << " до " << y << " была удалена \n";

cout << "--------------------------------------------------\n\n";

}

}

}

int menu(int n, int m, int\*\* data) { // функция меню

system("cls");

cout << "------------МЕНЮ------------\n";

cout << "Дано:\n";

cout << "Количество городов: " << n << "\n";

cout << "Количество дорог: " << m << "\n\n";

cout << "Определить:\n";

cout << "1. Кратчайший путь из а в б\n";

cout << "2. Все возможные пути из а в б\n";

cout << "3. Минимальные расстояния от а до любого из городов\n";

cout << "4. Все города, не связанные между собой дорогами\n\n";

cout << "Дополнительные функции:\n";

cout << "5. Показать матрицу смежности\n";

cout << "6. Изменить количество ребер\n";

cout << "7. Изменить количество вершин\n";

cout << "8. Выход\n\n";

cout << "Выберите пункт из меню: \n";

int item;

cin >> item;

switch (item) { // выбираем пункт из меню

case 1: {

system("cls");

cout << "Дано:\n";

cout << "Количество городов: " << n << "\n";

cout << "Количество дорог: " << m << "\n\n";

cout << "Кратчайший путь из а в б\n";

int a, b;

while (1) {

cout << "город а: ";

cin >> a;

cout << "город б: ";

cin >> b;

if (a <= n && b <= n) {

break;

}

cout << "Что-то пошло не так. Пожалуйста, попробуйте еще раз.\n";

}

cout << "Кратчайший путь из города " << a << " в город " << b << " = " << Dijkstra\_Algorithm(n, a - 1, b - 1, data) << "\n\n";

system("pause");

system("cls");

menu(n, m, data);

break;

}

case 2: {

system("cls");

cout << "Дано:\n";

cout << "Количество городов: " << n << "\n";

cout << "Количество дорог: " << m << "\n\n";

cout << "\n";

int a, b;

while (1) {

cout << "город а: ";

cin >> a;

cout << "город б: ";

cin >> b;

if (a <= n && b <= n) {

break;

}

cout << "Что-то пошло не так. Пожалуйста, попробуйте еще раз.\n";

}

cout << "Все возможные пути из а в б\n\n";

paths\_a\_to\_b(n, a - 1, b - 1, data);

system("pause");

system("cls");

menu(n, m, data);

break;

}

case 3: {

system("cls");

cout << "Дано:\n";

cout << "Количество городов: " << n << "\n";

cout << "Количество дорог: " << m << "\n\n";

cout << "Кратчайший путь из а до всех остальных вершин\n";

int a;

while (1) {

cout << "город а: ";

cin >> a;

if (a <= n) {

break;

}

cout << "Что-то пошло не так. Пожалуйста, попробуйте еще раз.\n";

}

Dijkstra\_Algorithm2(n, a - 1, data);

system("pause");

system("cls");

menu(n, m, data);

break;

}

case 4: {

system("cls");

cout << "Дано:\n";

cout << "Количество городов: " << n << "\n";

cout << "Количество дорог: " << m << "\n\n";

cout << "Все города, не связанные между собой дорогами\n";

unrelated\_cities(n, data);

system("pause");

system("cls");

menu(n, m, data);

break;

}

case 5: {

system("cls");

cout << "Дано:\n";

cout << "Количество городов: " << n << "\n";

cout << "Количество дорог: " << m << "\n\n";

cout << "МАТРИЦА СМЕЖНОСТИ:\n";

print(n, data);

cout << "\n-1 - нет дороги";

cout << "\nиначе просто расстояние от города i до j\n\n";

system("pause");

system("cls");

menu(n, m, data);

break;

}

case 6: {

system("cls");

change\_number\_ribs(n, m, data);

system("pause");

system("cls");

menu(n, m, data);

break;

}

case 7: {

system("cls");

change\_number\_vertices(n, data);

system("pause");

system("cls");

menu(n, m, data);

break;

}

case 8: {

system("cls");

cout << "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* < До свидания > \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n";

cout <<"\*\*\*\*\* < Cпасибо что использовали эту программу > \*\*\*\*\*\n";

freeing\_up\_memory(n, data);

return 0;

break;

}

default: {

system("cls");

cout << "Что-то пошло не так. Пожалуйста, попробуйте еще раз.\n\n";

system("pause");

system("cls");

menu(n, m, data);

break;

}

}

}

int main() {

setlocale(LC\_ALL, "Russian");

srand(time(NULL));

int n, m; // количество городов, количество путей

int fill = input(n, m);

int\*\* data = new int\* [n];

arrayPreparation(data, n);

fill == 1 ? createPersonally(n, m, data) : createRandom(n, m, data);

menu(n, m, data);

return 0;

}

## Додаток Б. Блок-схема

