Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
ФІОТ
ІСТ

Лабораторна робота №2

по курсу “Теорія алгоритмів”

на тему “Методи розробки алгоритмів. Частина 1.”

Варіант 3

Виконали ст. групи ІС-31:
Коваль Богдан
Михайлова Софія
Шмигельський Ілля
Перевірив: ст. вик. Дорошенко К.С.

Київ 2024

Комп'ютерний практикум 1

Тема: Методи розробки алгоритмів. Частина 1.

Мета роботи: порівняння алгоритмів розв'язку задачі, побудованих різними методами.

Завдання

1. Для свого варіанту зробити наступні дії:

- ❖ Сформулювати постановку задачі
- ❖ Обрати відповідні 2 алгоритми з теоретичної частини практикуму або на свій розсуд
- ❖ Накреслити блок-схеми, на яких виконано аналіз складності алгоритмів
- ❖ Написати програмний код
- ❖ Провести дослідження продуктивності роботи алгоритмів, зробити результати
- ❖ дослідження у вигляді графіків або діаграм;
- ❖ Зробити висновки про доцільність використання кожного з алгоритмів для типових вхідних даних та про відповідність результатів експериментального дослідження аналітичним оцінкам складності.
- ❖ Скласти таблицю тестування.
- ❖ Навести скрин-шоти роботи програми. для заданої задачі;ю
- ❖ Дати відповіді на контрольні питання

Постановка задачі

Не всі студенти НТУУ «КПІ ім. Ігоря Сікорського» полюбляють гуляти пішки, тому компанія хлопців і дівчат з університету вирішила прокласти транспортні маршрути від свого навчального закладу до інших вищеперерахованих місць за умови найменшої вартості такого проєкту. Допоможіть їм прокласти такі маршрути.

Вихідні дані: масив з заданими маршрутами.

Вихідні дані: мінімальний шлях для обходу всіх вершин графа.

Побудова моделі

1. Алгоритм Флойда - Уоршелла

Основні величини

graph	матриця суміжності	представляє ваги ребер між вершинами
numVertices	ціле число	зберігає кількість вершин у графі
startVertex	ціле число	вершина з якої починається обхід графв
price	ціле число	ціна за 1 км у грн/км

Допоміжні величини:

next	матриця	відстеження шляху між вершинами
floydWarshall()	функція	реалізація алгоритму Флойда - Воршелла для пошуку найкоротших вершин між усіма вершинами графу
printPath()	рекурсивна функція	вивід шляху між двома вершинами
outResult()	функція	вивід результату

2. Алгоритм Дейкстри

Основні величини:

graph	двовимірний масив	містить відстані між різними вершинами графу
SIZE	константа	розмір графу (кількість вершин)
outResult()	функція	виведення результатів алгоритму - найкоротших відстаней та шляхів до всіх вершин
dijkstra()	основна функція	виконує алгоритм Дейкстри

Допоміжні величини:

dist	масив	збереження найкоротшої відстані від початкової вершини до кожної вершини графу

visited	масив	вказує чи була відвідана кожна с вершин графу
path	масив векторів	збереження шляху до кожної вершини графу
findMinDistanceVertex()	функція	знаходження вершини з найменшою відстанню серед тих, що не були відвідані
numeric_limits<double>::max()	функція	ініціалізація початкових значень найкоротших відстаней

Доцільність використання обраних алгоритмів

Для комп’ютерного практикуму 2 ми обрали два алгоритма. Перший - алгоритм Флойда - Уоршелла. **Алгоритм Флойда-Уоршелла** - це ефективний алгоритм для знаходження найкоротших шляхів між усіма парами вершин у вагованому графі. Основними випадками використання цього алгоритму є:

1. Графи з вагами ребер: Алгоритм Флойда-Уоршелла працює з графами, де ребра мають вагу. Це робить його відмінним вибором для задач, пов'язаних з маршрутизацією мереж, плануванням маршрутів у транспортних системах та іншими ситуаціями, де потрібно знайти найкоротший шлях між усіма парами вершин.
2. Ситуації без негативних циклів: Алгоритм Флойда-Уоршелла працює коректно на графах з негативними вагами, але не може правильно обробити цикли в графі, які мають від'ємну суму. Такі ситуації можуть призвести до некоректних результатів.
3. Малі графи: Для невеликих графів, де кількість вершин не дуже велика, алгоритм Флойда-Уоршелла може бути досить ефективним і простим у реалізації.

Однак, варто враховувати, що алгоритм Флойда-Уоршелла має складність O(V^3), де V - кількість вершин у графі. Тому для дуже великих графів цей алгоритм може бути недоцільним з точки зору часу виконання. У таких випадках можуть бути використані інші алгоритми, такі як алгоритм Дейкстри або алгоритм Беллмана-Форда, які мають кращу амортизовану складність для пошуку найкоротших шляхів від однієї вершини до всіх інших.

Другий - **алгоритм Дейкстри**. Алгоритм Дейкстри досить ефективний у випадках, коли потрібно знайти найкоротші шляхи від однієї вершини до всіх інших у графі.

1. Початкова вершина: наш код використовує алгоритм Дейкстри для знаходження найкоротших шляхів від заданої початкової вершини. Це корисно, якщо потрібно знайти найкоротший шлях з конкретної точки до всіх інших у графі.
2. Ваги ребер: Алгоритм Дейкстри працює з графами, де ребра мають додатні ваги. У нашому випадку граф має невід'ємні ваги, тому алгоритм Дейкстри підходить для вирішення цієї задачі.
3. Величина графу: наш граф має 11 вершин, що є достатньою кількістю вершин для ефективної роботи алгоритму Дейкстри. Цей алгоритм працює швидко для невеликих та середніх за розміром графів.
4. Результати: Алгоритм Дейкстри надає інформацію про найкоротший шлях та його вагу від початкової вершини до всіх інших вершин у графі, що відповідає нашим вимогам.

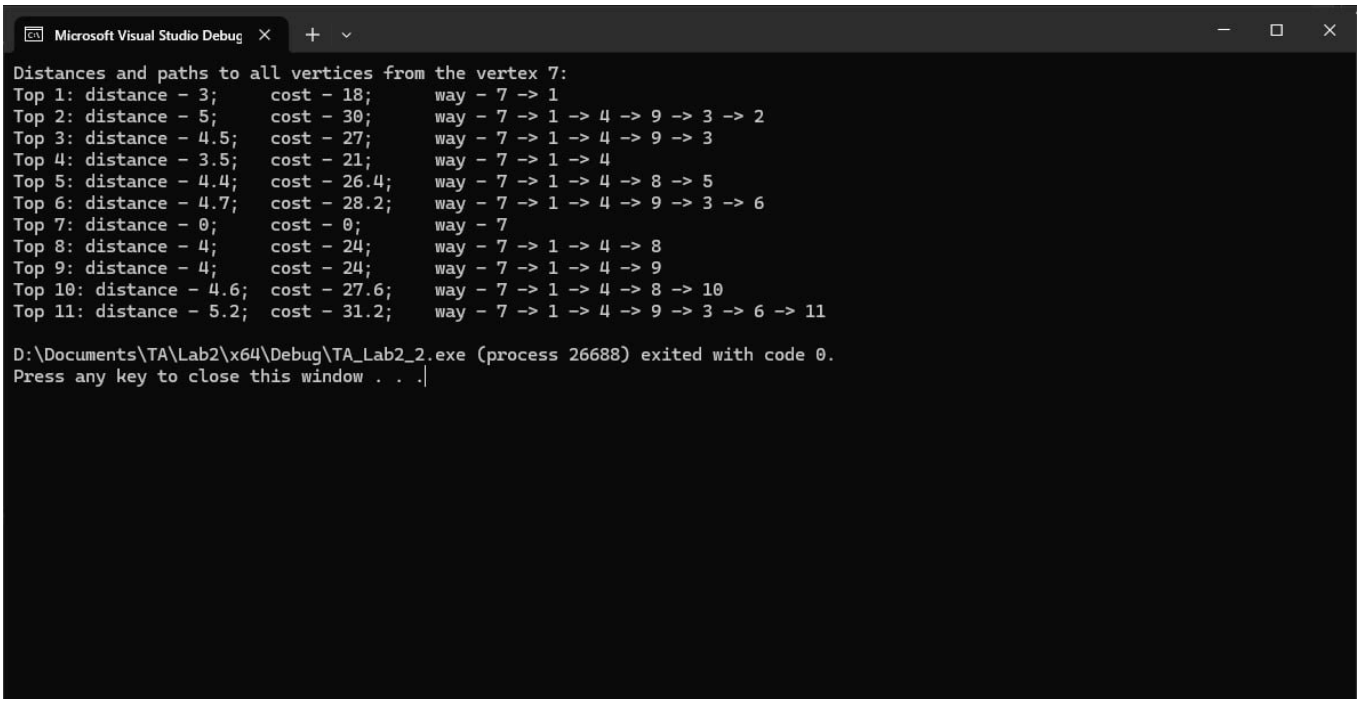
Інтерфейс користувача

Алгоритм Флойда - Уоршелла і алгоритм Дейкстри видають в консолі мінімальні відстані між вершинами, мінімальний шлях для обходу всіх вершин, починаючи з 6 вершини. Також відстані і вартість для кожного з варіантів.

Флойда - Уоршелла:

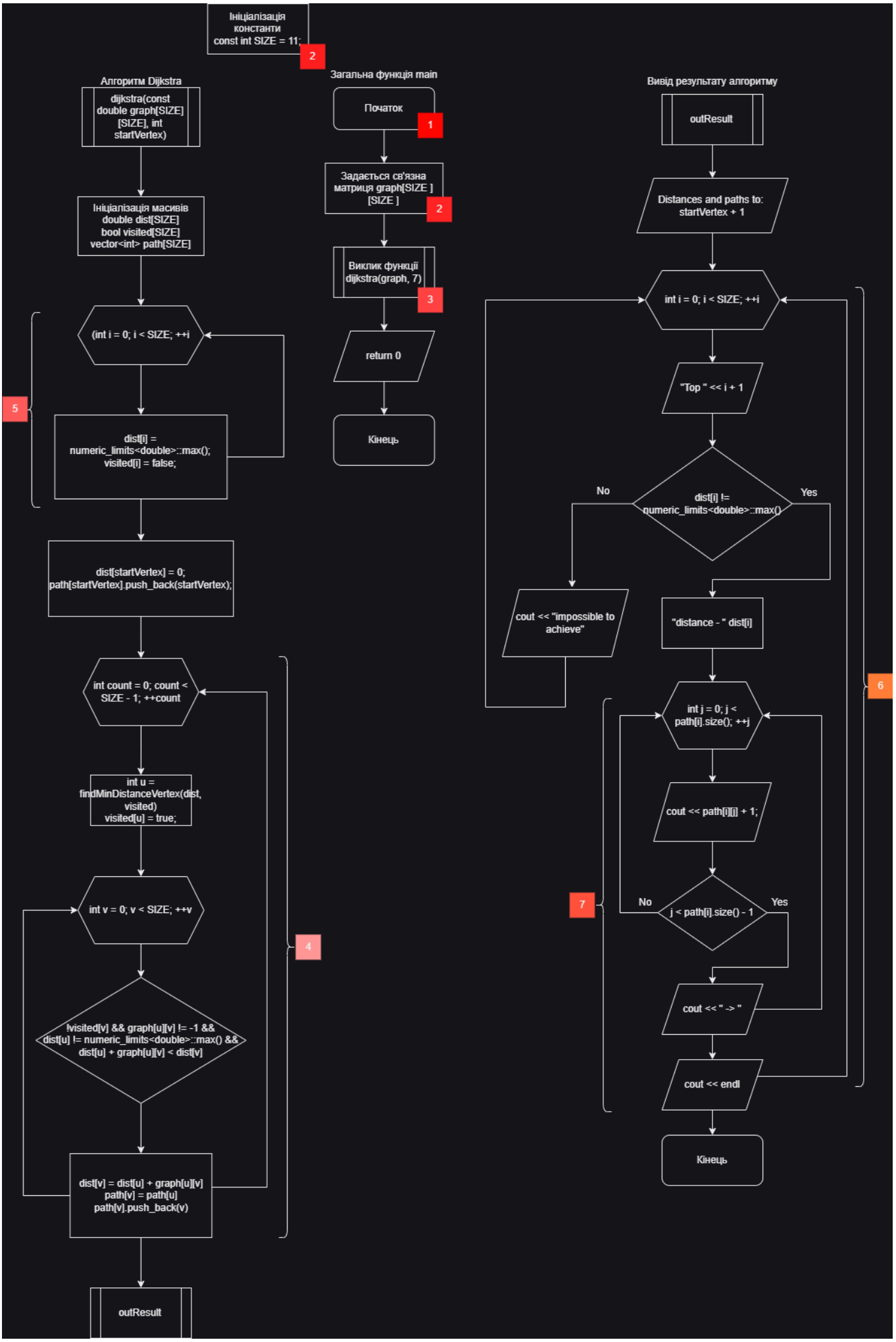
```
Мінімальні відстані між вершинами:
1       2       1.5     0.5     1.4     1.7     3       1       1       1.6     2.2
2       0.6     0.5     1.5     1       0.4     5       1.4     1       1.3     0.3
1.5     0.5     0.4     1       0.5     0.2     4.5     0.9     0.5     0.8     0.7
0.5     1.5     1       1       0.9     1.2     3.5     0.5     0.5     1.1     1.7
1.4     1       0.5     0.9     0.6     0.7     4.4     0.4     0.5     0.3     1.2
1.7     0.4     0.2     1.2     0.7     0.4     4.7     1.1     0.7     1       0.5
3       5       4.5     3.5     4.4     4.7     6       4       4       4.6     5.2
1       1.4     0.9     0.5     0.4     1.1     4       0.8     0.9     0.6     1.6
1       1       0.5     0.5     0.5     0.7     4       0.9     1       0.8     1.2
1.6     1.3     0.8     1.1     0.3     1       4.6     0.6     0.8     0.6     1.5
2.2     0.3     0.7     1.7     1.2     0.5     5.2     1.6     1.2     1.5     0.6

Мінімальний шлях для обходу всіх вершин, починаючи з 6 вершини:
7 -> 1: 7 ->1           || Відстань: 3 km       вартість - 18 UAH
7 -> 2: 7 ->9 -> 3 -> 2                 || Відстань: 5 km       вартість - 30 UAH
7 -> 3: 7 ->9 -> 3              || Відстань: 4.5 km     вартість - 27 UAH
7 -> 4: 7 ->1 -> 4              || Відстань: 3.5 km     вартість - 21 UAH
7 -> 5: 7 ->8 -> 5              || Відстань: 4.4 km     вартість - 26.4 UAH
7 -> 6: 7 ->9 -> 3 -> 6                 || Відстань: 4.7 km     вартість - 28.2 UAH
7 -> 8: 7 ->4 -> 8              || Відстань: 4 km       вартість - 24 UAH
7 -> 9: 7 ->4 -> 9              || Відстань: 4 km       вартість - 24 UAH
7 -> 10: 7 ->8 -> 10            || Відстань: 4.6 km     вартість - 27.6 UAH
7 -> 11: 7 ->9 -> 6 -> 11               || Відстань: 5.2 km     вартість - 31.2 UAH
```

Дейкстри :

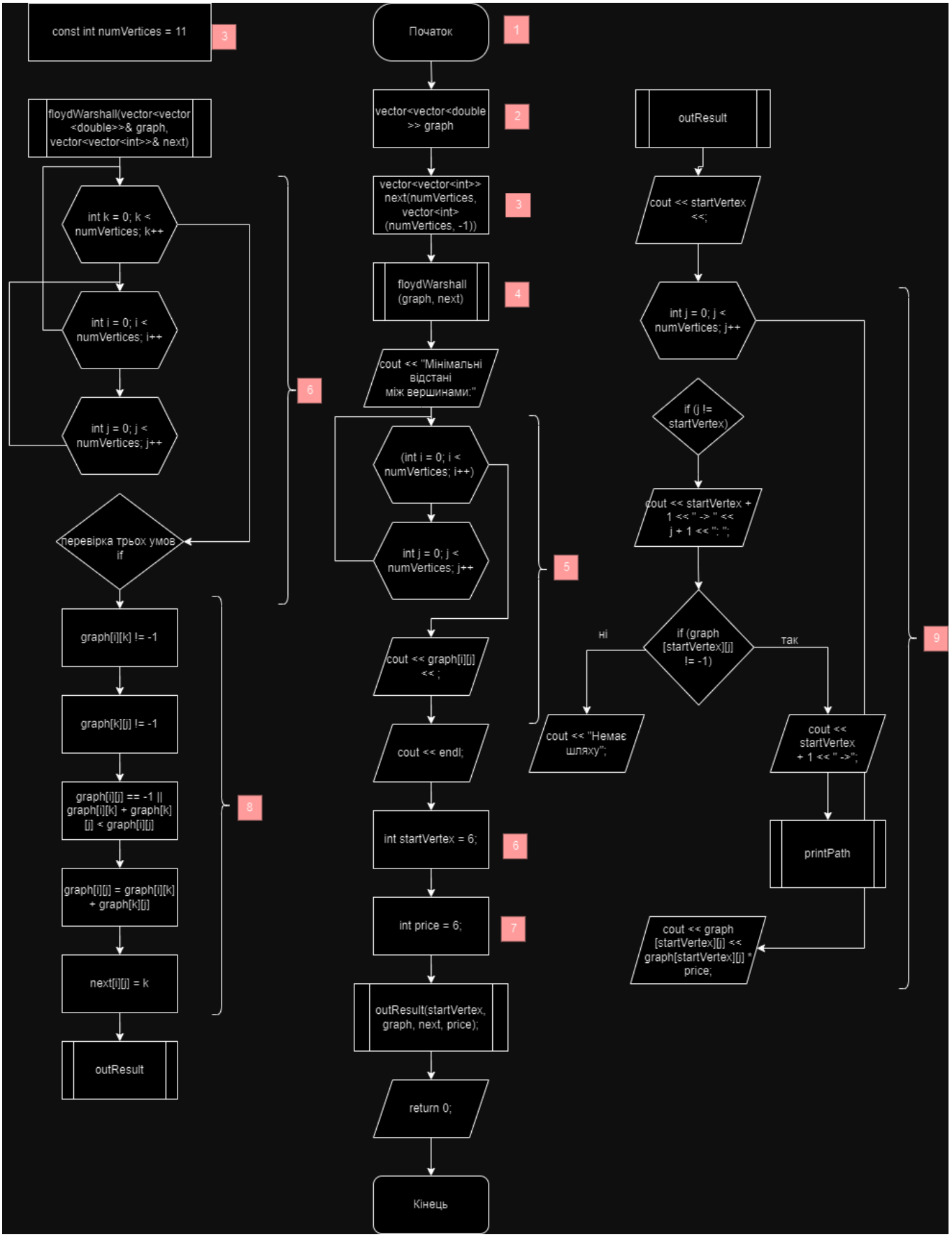


Розробка алгоритму

Алгоритм Дейкстри :


Ініціалізація константи
const int SIZE = 11;
2
Загальна функція main
Початок
1
Задається св'язна матриця graph[SIZE] [SIZE]
2
Виклик функції dijkstra(graph, 7)
3
return 0
Кінець
Алгоритм Dijkstra
dijkstra(const double graph[SIZE] [SIZE], int startVertex)
Ініціалізація масивів
double dist[SIZE]
bool visited[SIZE]
vector<int> path[SIZE]
(int i = 0; i < SIZE; ++i
5
dist[i] = numeric_limits<double>::max();
visited[i] = false;
dist[startVertex] = 0;
path[startVertex].push_back(startVertex);
int count = 0; count < SIZE - 1; ++count
4
int u = findMinDistanceVertex(dist, visited)
visited[u] = true;
int v = 0; v < SIZE; ++v
!visited[v] && graph[u][v] != -1 && dist[u] != numeric_limits<double>::max() && dist[u] + graph[u][v] < dist[v]
dist[v] = dist[u] + graph[u][v]
path[v] = path[u]
path[v].push_back(v)
outResult
Вивід результату алгоритму
outResult
Distances and paths to: startVertex + 1
int i = 0; i < SIZE; ++i
6
"Top " << i + 1
dist[i] != numeric_limits<double>::max()
No
Yes
cout << "impossible to achieve"
"distance - " dist[i]
int j = 0; j < path[i].size(); ++j
7
cout << path[i][j] + 1;
No
j < path[i].size() - 1
Yes
cout << " -> "
cout << endl
Кінець

Алгоритм Флойда - Уоршелла



Правильність алгоритму

Алгоритм Дейстри:

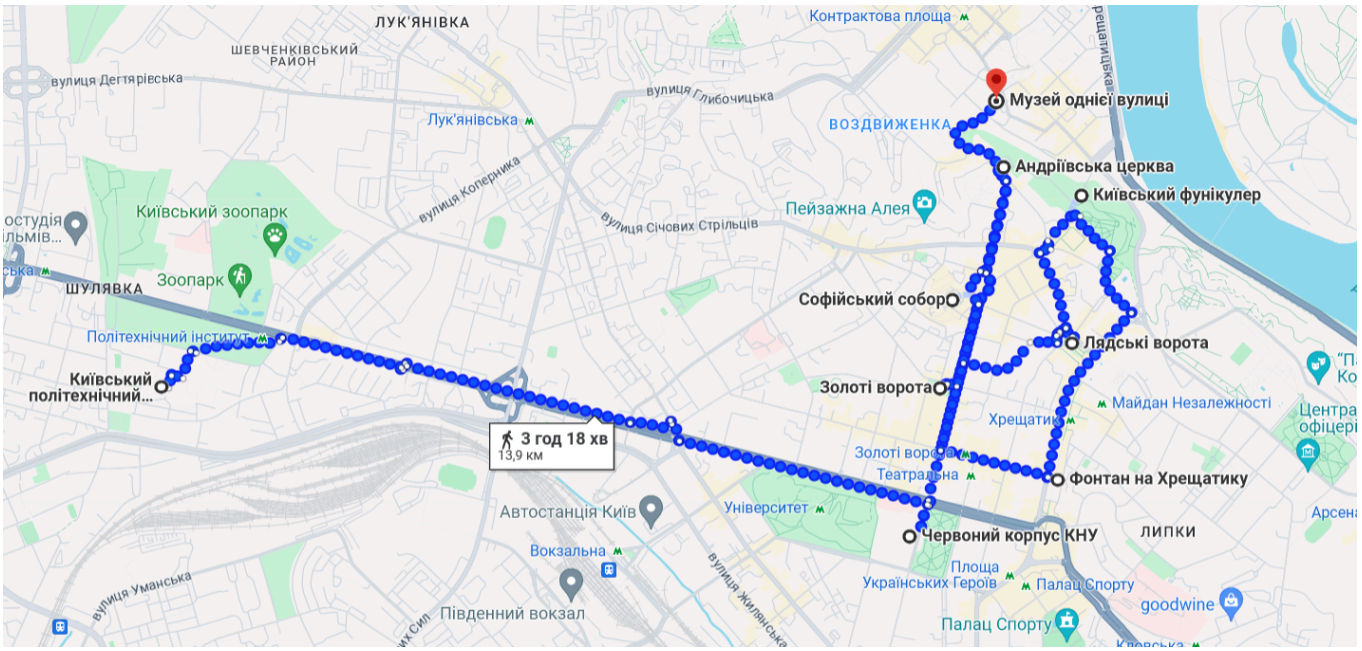
1. Обґрунтування правомірності кожного кроку: блок ***1*** - початок програми, блок ***2*** відповідає за введення початкових даних – без них алгоритм не працюватиме; блок ***3*** відповідає за виклик головної функції, який відповідає за пошук за алгоритмом Дейкстрі.
2. Доведення кінцевості алгоритму: кінцевість алгоритму залежить від блоків 4**, *5***. У наданому коді кількість елементів у матриці визначається за допомогою константної змінної SIZE, яка задається прямо в програмі Отже, кількість вершин у матриці є скінченною, оскільки вона обмежена значенням, заданою в програмі. Таким чином, матриця завжди буде містити кінцеву кількість вершин. **Використання властивостей вхідних даних**: Якщо вхідні дані обмежені за певними параметрами, можна аналізувати ці властивості для підтвердження кінцевості алгоритму.
3. Для виведення отриманих даних алгоритмом використані блоки ***6, 7.***

Флойда - Уоршелла

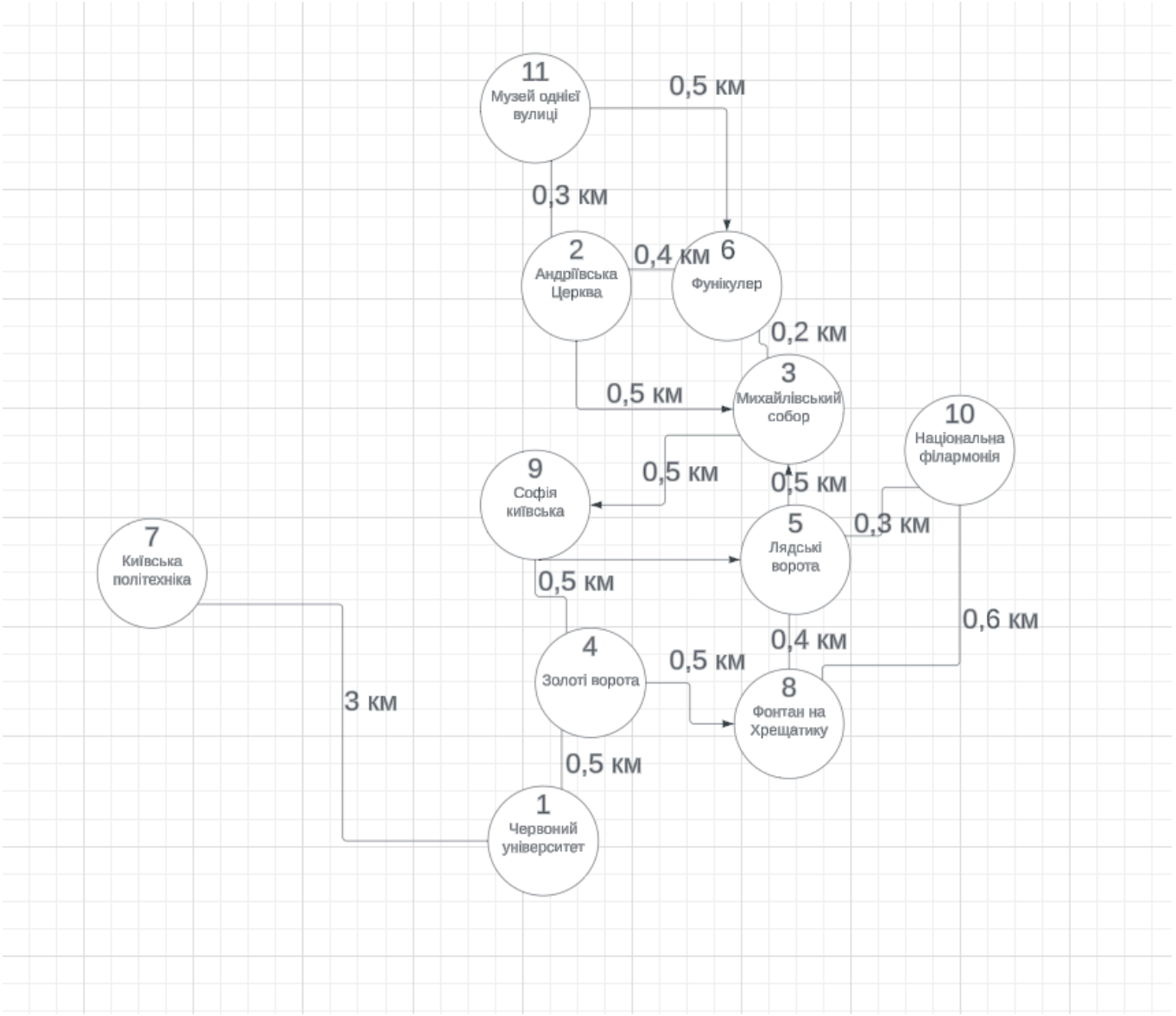
1. Обґрунтування правомірності кожного кроку : блок ***1*** - початок програми, блок ***2*** відповідає за введення початкових даних – без них алгоритм не працюватиме; блок ***4*** відповідає за виклик головної функції, який відповідає за пошук за алгоритмом Флойда - Уоршелла.

2. Доведення кінцевості алгоритму: кінцевість алгоритму залежить від блоків ***6*****, *9***. У наданому коді кількість елементів у матриці визначається за допомогою константної змінної SIZE, яка задається прямо в програмі Отже, кількість вершин у матриці є скінченною, оскільки вона обмежена значенням, заданою в програмі. Таким чином, матриця завжди буде містити кінцеву кількість вершин. **Використання властивостей вхідних даних**: Якщо вхідні дані обмежені за певними параметрами, можна аналізувати ці властивості для підтвердження кінцевості алгоритму.

Загальна карта маршрутів



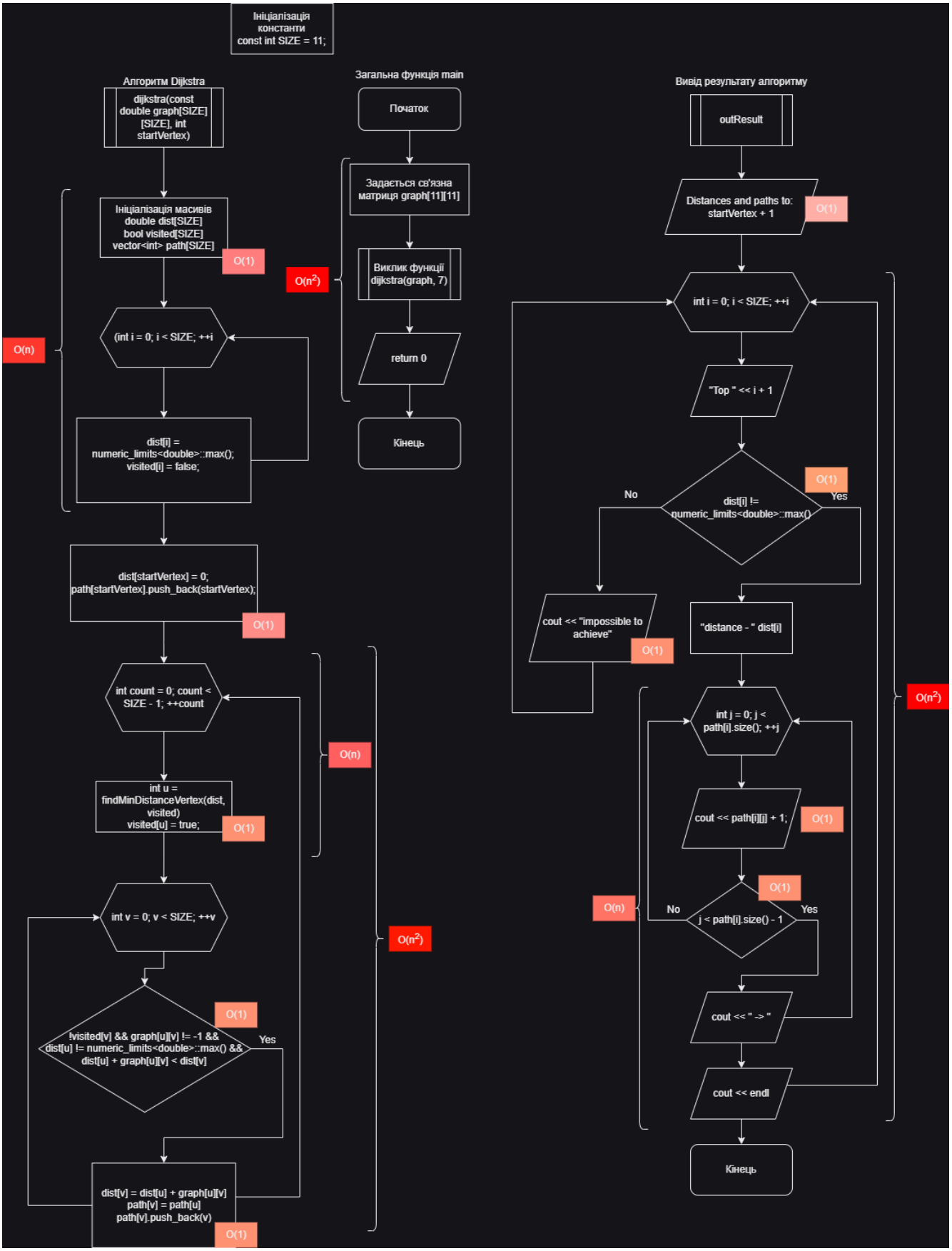
Блок-схема відстаней



Аналіз алгоритму та його складності

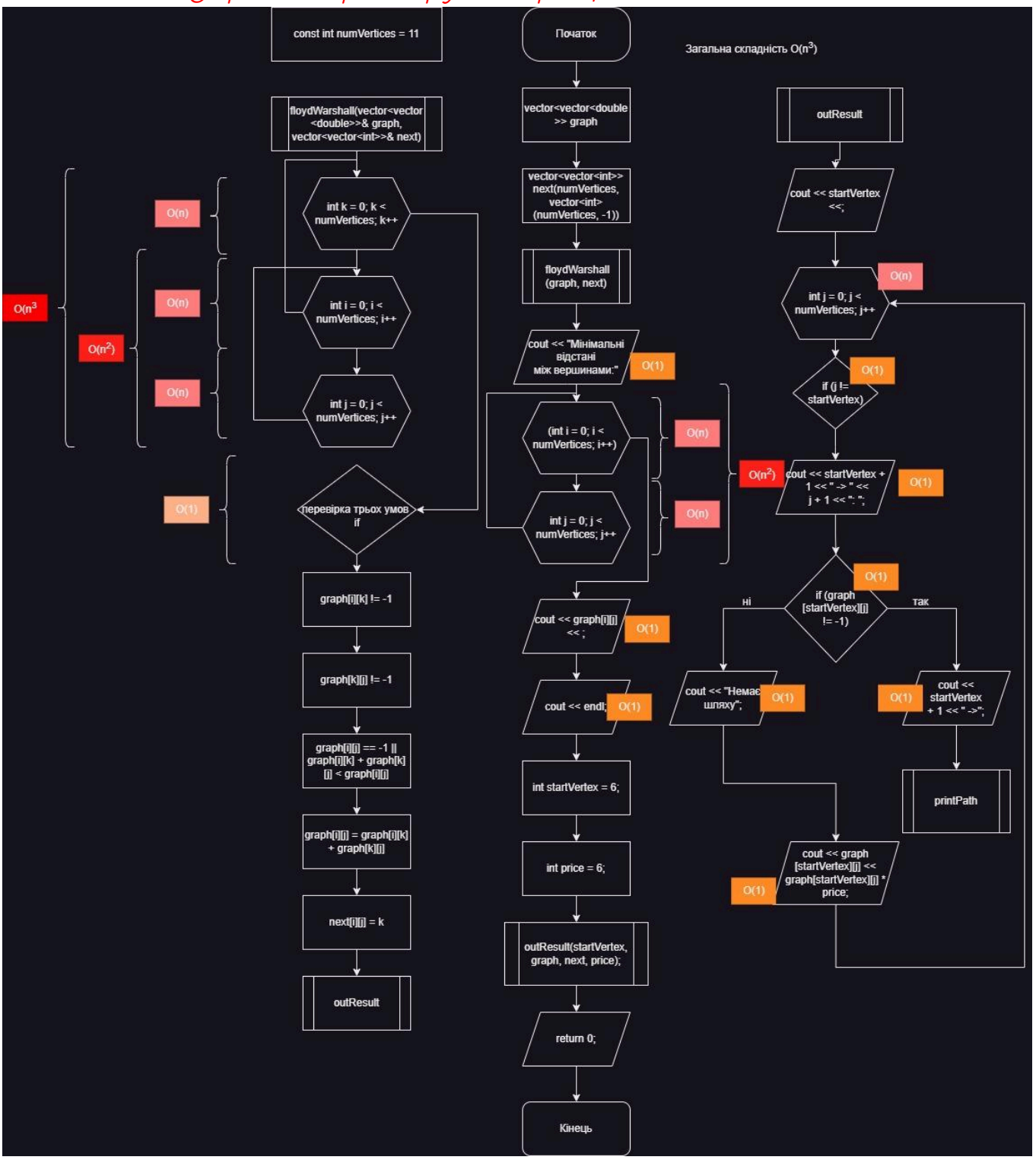
Алгоритм Дейкстри

Загальна складність алгоритму *O(n^2). n - В даному випадку статична і дорівнює розміру матриці*



Алгоритм Войда - Уоршелла

Загальна складність алгоритму *O(n^3). n - В даному випадку статична і дорівнює розміру матриці*



Порівняльний висновок :

Обидва алгоритма цілком доцільно використовувати для розв'язання задачі з. Алгоритм Дейкстри **доцільніше** підходить для пошуку найкоротших шляхів, оскільки має меншу оцінку складності O(n^2), тоді як Алгоритм Войда-Уоршелла має оцінку (O^3). Також Дейкстри простіший в плані реалізації коду.

Реалізація алгоритму

Алгоритм Флойда - Уоршелла

```
#include <iostream>
#include <vector>
#include <windows.h>

using namespace std;


const int numVertices = 11;

void floydWarshall(vector<vector<double>>& graph,
vector<vector<int>>& next) {
     for (int k = 0; k < numVertices; k++) {
     for (int i = 0; i < numVertices; i++) {
          for (int j = 0; j < numVertices; j++) {
               if (graph[i][k] != -1 && graph[k][j] != -1 &&
                    (graph[i][j] == -1 || graph[i][k] +
graph[k][j] < graph[i][j])) {
                    graph[i][j] = graph[i][k] + graph[k][j];
                    next[i][j] = k;
               }
          }
     }
     }
}

void printPath(int i, int j, const vector<vector<int>>& next) {
     if (next[i][j] == -1) {
     cout << j+1 << " ";
     return;
     }


     cout << next[i][j] +1<< " -> ";
     printPath(next[i][j], j, next);
}

void outResult(int startVertex, vector<vector<double>>& graph,
const vector<vector<int>>& next, const int price)
{
```

```
    // Виведення мінімального шляху для обходу всіх вершин,
починаючи зі стартової вершини
    cout << "\nМінімальний шлях для обходу всіх вершин,
починаючи з " << startVertex << " вершини:\n";
    for (int j = 0; j < numVertices; j++) {
    if (j != startVertex) {
        cout << startVertex + 1 << " -> " << j + 1 << ": ";
        if (graph[startVertex][j] != -1) {
            cout << startVertex + 1 << " ->";
            printPath(startVertex, j, next);
        }
        else {
            cout << "Немає шляху";
        }
        cout << "\t\t || Відстань: " << graph[startVertex][j]
<<" km" <<"\t вартість - "<< graph[startVertex][j]*price<< " UAH"
<< "\n";
    }

    }
}

int main() {
    // підключення української локалізації
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    // Ініціалізація графу з вагами ребер
    vector<vector<double>> graph = {

    {-1, -1, -1, 0.5, -1, -1, 3, -1, -1, -1, -1},
    {-1, -1, 0.5, -1, -1, 0.4, -1, -1, -1, -1, 0.3},
    {-1, 0.5, -1, -1, 0.5, 0.2, -1, -1, 0.5, -1, -1},
    {0.5, -1, -1, -1, -1, -1, -1, 0.5, 0.5, -1, -1},
    {-1, -1, 0.5, -1, -1, -1, -1, 0.4, 0.5, 0.3, -1},
    {-1, 0.4, 0.2, -1, -1, -1, -1, -1, -1, -1, 0.5},
    {3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, 0.5, 0.4, -1, -1, -1, -1, 0.6, -1},
    {-1, -1, 0.5, 0.5, 0.5, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, 0.3, -1, -1, 0.6, -1, -1, -1},
    {-1, 0.3, -1, -1, -1, 0.5, -1, -1, -1, -1, -1}
```

```
    };

    // Ініціалізація матриці next для відстеження шляху
    vector<vector<int>> next(numVertices,
vector<int>(numVertices, -1));

    floydWarshall(graph, next);

    // Виведення мінімальних відстаней для кожної пари вершин
    cout << "Мінімальні відстані між вершинами:\n";
    for (int i = 0; i < numVertices; i++) {
    for (int j = 0; j < numVertices; j++) {
        cout << graph[i][j] << "\t";
    }
    cout << endl;
    }

    int startVertex = 6; // Вершина, з якої починається обхід
    int price = 6; // ціна за 1 км у грн/км

    outResult(startVertex, graph, next, price);

    return 0;
}
```

Алгоритм Дейкстри

```
#include <iostream>
#include <limits>
#include <vector>

using namespace std;

const int SIZE = 11;

// Функція для знаходження вершини з найменшою відстанню, яка
ще не була відвідана
int findMinDistanceVertex(const double dist[], const bool
visited[]) {
    double minDist = numeric_limits<double>::max();
```

```
    int minIndex = -1;

    for (int i = 0; i < SIZE; ++i) {
    if (!visited[i] && dist[i] < minDist) {
        minDist = dist[i];
        minIndex = i;
    }
    }

    return minIndex;
}

void outResult( int startVertex, double dist[SIZE], vector<int>
path[SIZE]) {

    // Виводимо результати
    cout << "Distances and paths to all vertices from the vertex
" << startVertex + 1 << ":" << endl;
    for (int i = 0; i < SIZE; ++i) {
    cout << "Top " << i + 1 << ": ";
    if (dist[i] != numeric_limits<double>::max()) {
        cout << "distance - " << dist[i] << "; \t way - ";
        for (int j = 0; j < path[i].size(); ++j) {
            cout << path[i][j] + 1; // Додати 1 до індексів для
нумерації з 1
            if (j < path[i].size() - 1) {
                cout << " -> ";
            }
        }
        cout << endl;
    }
    else {
        cout << "impossible to achieve" << endl;
    }
    }
}

// Функція для виведення найкоротших шляхів до всіх вершин з
використанням алгоритму Дейкстри
void dijkstra(const double graph[SIZE][SIZE], int startVertex) {
    double dist[SIZE]; // Масив для зберігання найкоротших
```

```
відстаней до вершин
    bool visited[SIZE]; // Масив, що вказує, чи була відвідана
вершина
    vector<int> path[SIZE]; // Масив для зберігання шляхів до
вершин

    // Ініціалізуємо масиви
    for (int i = 0; i < SIZE; ++i) {
    dist[i] = numeric_limits<double>::max(); // Встановлюємо
відстань до всіх вершин на початку як нескінченність
    visited[i] = false; // Жодна вершина ще не відвідана
    }

    // Відстань до стартової вершини завжди 0
    dist[startVertex] = 0;
    path[startVertex].push_back(startVertex); // Шлях до
початкової вершини

    // Знаходимо найкоротший шлях для кожної вершини
    for (int count = 0; count < SIZE - 1; ++count) {
    int u = findMinDistanceVertex(dist, visited); // Знаходимо
вершину з найменшою відстанню
    visited[u] = true; // Позначаємо вершину як відвідану

    // Оновлюємо відстані до сусідніх вершин, якщо вони ще не
відвідані і відстань до них через поточну вершину коротша
    for (int v = 0; v < SIZE; ++v) {
        if (!visited[v] && graph[u][v] != -1 && dist[u] !=
numeric_limits<double>::max() && dist[u] + graph[u][v] <
dist[v]) {
            dist[v] = dist[u] + graph[u][v];
            path[v] = path[u]; // Копіюємо шлях до вершини u
            path[v].push_back(v); // Додаємо вершину v до
шляху
        }
    }
    }

    // Виводимо результати
    cout << "Distances and paths to all vertices from the vertex
" << startVertex + 1 << ":" << endl;
```

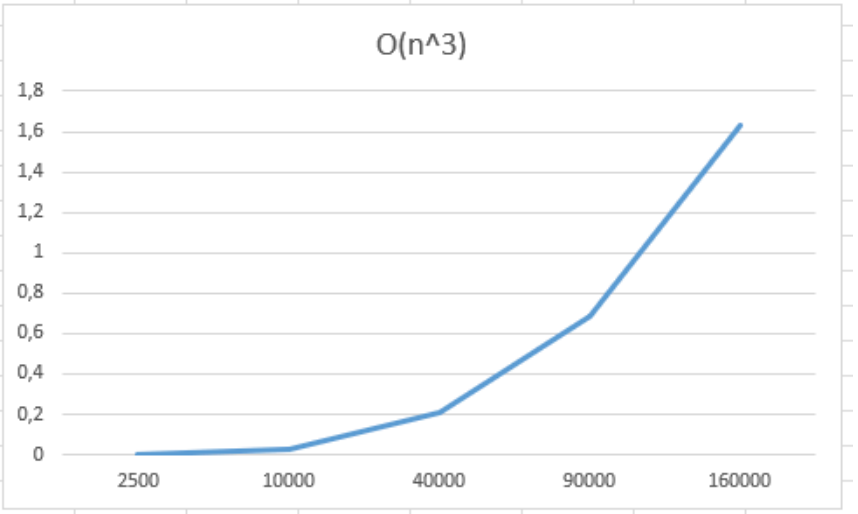
```
    for (int i = 0; i < SIZE; ++i) {
    cout << "Top " << i + 1 << ": ";
    if (dist[i] != numeric_limits<double>::max()) {
        cout << "distance - " << dist[i] << ";\t cost - "<<
dist[i] * 6 << "; \t way - ";
        for (int j = 0; j < path[i].size(); ++j) {
            cout << path[i][j] + 1; // Додати 1 до індексів для
нумерації з 1
            if (j < path[i].size() - 1) {
                cout << " -> ";
            }
        }
        cout << endl;
    }
    else {
        cout << "impossible to achieve" << endl;
    }
    }

    //outResult(startVertex, &dist[SIZE], &path[SIZE]);
}

int main() {
    double graph[SIZE][SIZE] = {
    {-1, -1, -1, 0.5, -1, -1, 3, -1, -1, -1, -1},
    {-1, -1, 0.5, -1, -1, 0.4, -1, -1, -1, -1, 0.3},
    {-1, 0.5, -1, -1, 0.5, 0.2, -1, -1, 0.5, -1, -1},
    {0.5, -1, -1, -1, -1, -1, -1, 0.5, 0.5, -1, -1},
    {-1, -1, 0.5, -1, -1, -1, -1, 0.4, 0.5, 0.3, -1},
    {-1, 0.4, 0.2, -1, -1, -1, -1, -1, -1, -1, 0.5},
    {3, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, 0.5, 0.4, -1, -1, -1, -1, 0.6, -1},
    {-1, -1, 0.5, 0.5, 0.5, -1, -1, -1, -1, -1, -1},
    {-1, -1, -1, -1, 0.3, -1, -1, 0.6, -1, -1, -1},
    {-1, 0.3, -1, -1, -1, 0.5, -1, -1, -1, -1, -1}
    };

    // Застосовуємо алгоритм Дейкстри для знаходження
найкоротших шляхів від вершини 1
    dijkstra(graph, 6);
```

```
    return 0;
}
```

Скласти таблицю тестування

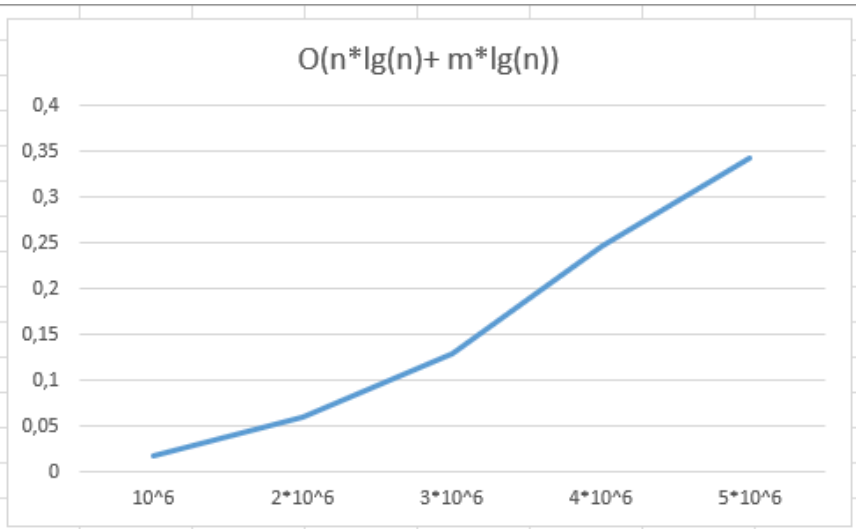
Алгоритм Флойда-Уоршела

к-ть вершин графу	час виконання (с)
2500	0,004
10000	0,028
40000	0,205
90000	0,689
160000	1,63



Алгоритм Дейкстри

к-ть вершин графу	час виконання (с)
10^6	0,017
2*10^6	0,06
3*10^6	0,129
4*10^6	0,247
5*10^6	0,344



Контрольні запитання

1. Перерахуйте відомі вам методи розробки алгоритмів. Докладніше розкажіть про один з них.

Метод частинних цілей (“Розділяй та володарюй”)

Цей метод полягає у тому, що глобальна велика задача ділиться (якщо це можливо) на окремі задачі.

Динамічне програмування

Динамічним програмуванням (в найбільш загальній формі) називають процес покрокового розв'язку задачі.

Метод сходження

Даний метод полягає у тому, щоби протягом пошуку найкращого розв'язку алгоритм відшукував все кращі та кращі варіанти розв'язку.

Програмування з поверненнями назад

Іноді доводиться мати справу з задачами пошуку оптимального розв'язку, коли неможливо застосувати жоден з відомих алгоритмів, які здатні допомогти відшукати оптимальний варіант розв'язку, і залишається застосувати останній засіб — повний перебір.

Метод спроб та помилок

Багато задач не допускають аналітичного розв'язку, а тому їх доводиться вирішувати методом спроб та помилок, тобто перебираючи усі можливі варіанти та відкидаючи їх у випадку невдачі.

2. Перерахуйте переваги та недоліки наступних методів розробки алгоритмів: методу часткових (проміжних) цілей, методу підйому (локального пошуку), методу відпрацювання назад.

Метод часткових (проміжних) цілей:

- Переваги:
    - Дозволяє розбити складну задачу на кілька менших підзадач, що полегшує розробку.
    - Забезпечує можливість поетапного вирішення завдання, підвищуючи зрозуміння процесу розробки.
    - Сприяє структуризації коду та підвищує його читабельність.
- Недоліки:
    - Може призвести до виникнення додаткової складності, якщо підзадачі потребують великої кількості взаємодії або синхронізації.
    - Не завжди ефективний для деяких типів задач, які не легко розбити на підзадачі.

Метод підйому (локального пошуку):

- Переваги:
    - Дозволяє знаходити локальні максимуми/мінімуми функції шляхом поступового зміщення в напрямку оптимального розв'язку.
    - Ефективний для задач, де можливо використання ітеративного покращення результату.
    - Зазвичай простий у реалізації та швидкий у виконанні.
- Недоліки:

    - Може застрягати в локальних максимумах/мінімумах, не знаходячи глобально оптимального розв'язку.
    - Не гарантує знаходження оптимального розв'язку для всіх випадків.

Метод відпрацювання назад:

- Переваги:
    - Ефективний для задач з оптимальною підструктурою, таких як задачі розбиття на частини або найкоротший шлях.
    - Забезпечує гарантований знаходження оптимального розв'язку, якщо проблема має оптимальну підструктуру.
- Недоліки:
    - Може бути дуже витратним з точки зору пам'яті та обчислень для деяких складних задач.
    - Не ефективний для задач з великою кількістю підзадач або залежностей між ними.

3. Який тип алгоритмів називають «жадібними» і чому?

"Жадібні" алгоритми вирішують проблеми, обираючи на кожному кроці локально найкращий варіант, не дбаючи про майбутні наслідки. Це може бути доцільно в тих випадках, коли вибір на кожному кроці не впливає на глобальний оптимум. Однак жадібні алгоритми не завжди забезпечують найкращий глобальний результат, оскільки вони не розглядають всі можливі варіанти.

4. Дайте характеристику евристичним алгоритмам. В яких випадках доцільно використовувати цей тип алгоритмів? Опишіть загальний підхід до побудови евристичних алгоритмів.

Евристичні алгоритми - це підхід який базується на експертному інтуїції або простих правилах. Евристичні алгоритми зазвичай не гарантують знаходження оптимального розв'язку, але часто є швидкими та ефективними у випадках складних задач.

Евристичні алгоритми доречні в таких випадках:

1. Коли вимагається **швидка відповідь** або наближене розв'язання, і **точний розв'язок** вимагає значних обчислювальних ресурсів.
2. У випадках, коли не існує ефективного або практичного алгоритму для точного розв'язку проблеми.
3. Коли можливі різні варіанти розв'язку, і не завжди зрозуміло, який є оптимальним.

Загальний підхід до побудови евристичних алгоритмів включає наступні етапи:

1. **Формулювання проблеми**: Ретельне визначення постановки задачі та критеріїв оцінки розв'язку.
2. **Вибір евристик**: Визначення простих правил або стратегій, які можуть допомогти при прийнятті рішення.

3. **Розробка алгоритму**: Реалізація алгоритму на основі вибраних евристик та стратегій.
4. **Тестування та оцінка**: Проведення експериментів для оцінки ефективності та точності роботи алгоритму, а також для визначення його робочих меж.
5. **Покращення та оптимізація**: Удосконалення алгоритму шляхом виправлення недоліків та впровадження нових ідей або евристик.

5. Проаналізуйте, що спільного мають та чим відрізняються алгоритми, що використовують пошук з поверненням, та алгоритми, що використовують метод гілок та границь.

6. Поясніть, для чого можна використовувати метод альфа-бета відсікання.

Метод альфа-бета відсікання є ефективним алгоритмом для оптимізації пошуку в грі на кшталт шахи або шашки. Основна мета цього методу - зменшити кількість вузлів, які потрібно перевірити в дереві гри, шляхом відсікання гілок дерева, які точно не приведуть до оптимального розв'язку.

7. Поясніть термін «структурне програмування». Для чого воно застосовується?

Структурне програмування - це методологія програмування, основна ідея якої полягає в тому, щоб програма складалася зі структурованих блоків (наприклад, послідовностей, виборів, циклів), які можна легко розуміти та підтримувати.

Висновки

У цій частині роботи ми дослідили різні методи розробки алгоритмів і порівняли їх ефективність у розв'язанні задачі. Ми розглянули такі методи, як декомпозиція, жадібні алгоритми, динамічне програмування та евристичні методи.

Декомпозиція дозволяє розбити складну задачу на менші підзадачі, що спрощує її розв'язання та робить код більш структурованим. Жадібні алгоритми приймають локально оптимальні рішення на кожному кроці, що може призвести до знаходження підходящого розв'язку, але не завжди оптимального. Динамічне програмування забезпечує ефективний пошук оптимальних рішень шляхом зберігання результатів підзадач. Евристичні методи дозволяють знаходити наближені розв'язки у випадках, коли точний розв'язок є недосяжним через обмеженість ресурсів або складність задачі.

Вибір методу залежить від конкретної задачі, її складності та обмежень. Для найкращого результату може знадобитися поєднання декількох методів або адаптація їх до конкретної ситуації.