МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Київський національний університет будівництва і архітектури

Кафедра прикладної математики

**Лабораторна робота №5**

**Реалізація алгоритмів Дейкстри та Флойда.**

Виконала ст.групи

ІСТ-21 факультету АІТ

Пасічник М.С

**Мета роботи:** Дослідити задачу знаходження найкоротших шляхів:

а) від джерела до всіх інших вершин заданого графа (алгоритм Дейкстри);

б) між усіма парами вершин заданого графа (алгоритм Флойда).

**Завдання:**

1. Навести орієнтований граф G(V,E)

2. Для заданого графа знайти найкоротший шлях від джерела до всіх інших вершин заданого графа, використовуючи алгоритм Дейкстри.

3. Для заданого графа знайти найкоротші шляхи між усіма парами вершин за допомогою алгоритму Флойда.

4. Написати програму реалізації алгоритмів Дейкстри та Флойда.

Алгоритм Дейкстри

#include<iostream>

#include<stdio.h>

#include< queue >

#include<list>

using namespace std;

#define INF 0x3f3f3f3f

typedef pair<int, int> iPair;

class Graph

{

int V;

list<pair<int, int>>\* adj;

public:

Graph(int V)

{

this->V = V;

adj = new list<iPair>[V];

}

void addEdge(int u, int v, int w);

// prints shortest path from s

void shortestPathingraph(int s);

};

void Graph::addEdge(int u, int v, int w)

{

adj[u].push\_back(make\_pair(v, w));

adj[v].push\_back(make\_pair(u, w));

}

void Graph::shortestPathingraph(int src)

{

priority\_queue<iPair, vector<iPair>, greater<iPair>> pq;

vector<int> dist(V, INF);

pq.push(make\_pair(0, src));

dist[src] = 0;

while (!pq.empty()) {

int u = pq.top().second;

pq.pop();

list<pair<int, int>>::iterator i;

for (i = adj[u].begin(); i != adj[u].end(); ++i) {

int v = (\*i).first;

int weight = (\*i).second;

if (dist[v] > dist[u] + weight) {

dist[v] = dist[u] + weight;

pq.push(make\_pair(dist[v], v));

}

}

}

printf("Vertex \tDistance from Source\n");

for (int i = 0; i < V; ++i)

printf("%d \t\t %d\n", i, dist[i]);

}

int main()

{

int V = 6;

Graph g(V);

g.addEdge(0, 1, 4);

g.addEdge(0, 5, 8);

g.addEdge(1, 2, 8);

g.addEdge(1, 3, 11);

g.addEdge(2, 3, 7);

g.addEdge(2, 5, 4);

g.addEdge(3, 4, 9);

g.addEdge(3, 5, 14);

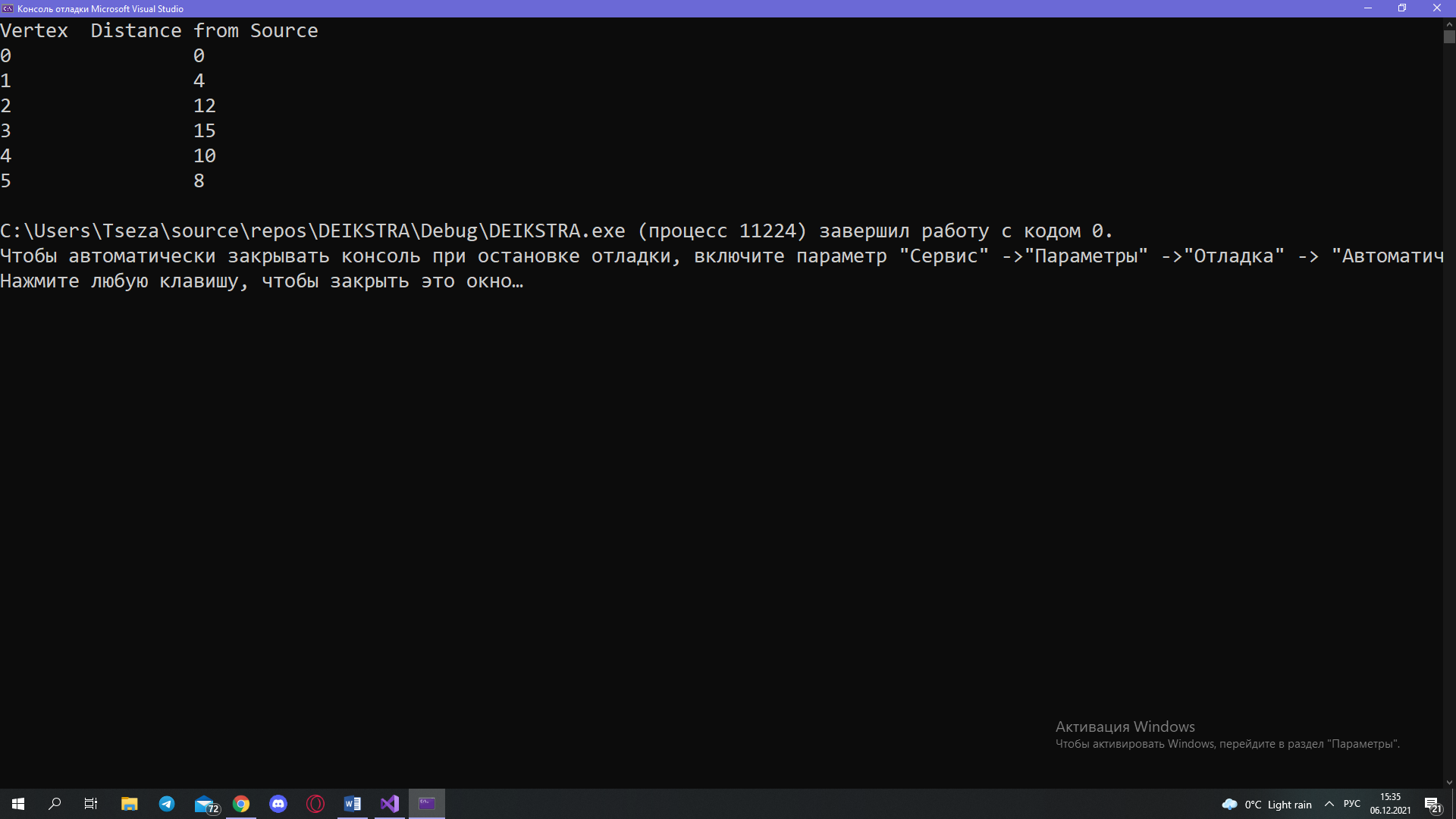
g.addEdge(4, 2, 10);

g.addEdge(5, 4, 2);

g.shortestPathingraph(0);

return 0;

}



Початок з вершини v0

v0 ->v1 (v1=4)

v1->v2 (v2=12)

v0->v1->v3(v3=15)

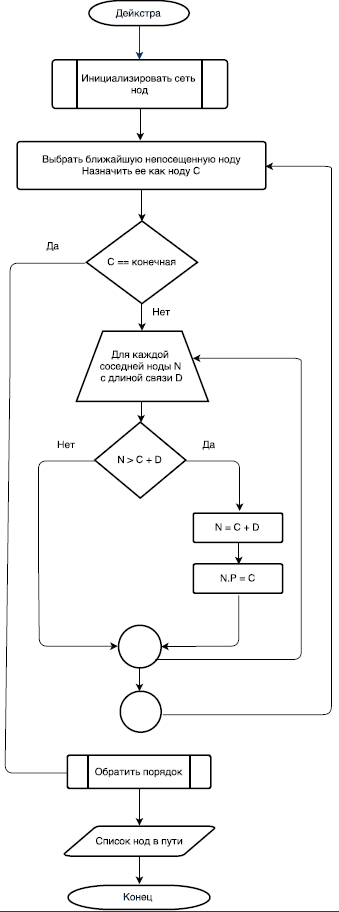
v0->v5->v4 (v4=10)

v0->v5 (v5=8)

Алгоритм Дейкстри:

* Встановлює відстань(вагу ребра) до початкового вузла рівним нулю v0 = 0;
* Встановлює відстань(вагу ребра ) до решти вершин на нескінченне значення v1,v2,v3,v4,v5 = ∞;
* Вибирає невідвідану вершину графа, що знаходиться на найменшій відстані від початкового вузла, та позначає її v0->v1 (v2=12);
* Розраховує відстань(вагу ребер ) до суміжних вершин, вибираючи найменшу відстань за кожної оцінки;
* Відвідує наступний вузол;

Алгоритм діє за цим сценарієм до тих пір, поки всі вершини не будуть відвідані



**Складність алгоритму Дейкстри**

Складність алгоритму Дейкстри залежить від способу знаходження вершини v, а також способу зберігання безлічі невідвіданих вершин та способу оновлення міток. Позначимо через n кількість вершин, а через m – кількість ребер у графі G.

У найпростішому випадку, коли пошуку вершини з мінімальним d[v] проглядається все безліч вершин, а зберігання величин d — масив, час роботи алгоритму є O(n2 + m). Основний цикл виконується порядку n разів, у кожному їх перебування мінімуму витрачається порядку n операцій, плюс кількість (змін міток, яке перевищує кількості ребер у вихідному графі.

Для розріджених графів (тобто таких, для яких m набагато менше n2) невідвідані вершини можна зберігати в двійковій купі, а як ключ використовувати значення d[i], тоді час вилучення вершини з U стане log n, при тому, що час модифікації d[i] зросте до log n. Так як цикл виконується порядку n разів, а кількість релаксацій не більше m, швидкість роботи такої реалізації O(n \* log n + m \* log n)

Якщо для зберігання невідвіданих вершин використовувати купу фібоначчія, для якої видалення відбувається в середньому за O(log n), а зменшення значення в середньому за O(1), то час роботи алгоритму складе O(n\*log n + m)

**Алгоритм Флойда**

Потрібно знайти матрицю найкоротших відстаней d, у якій елемент dij або дорівнює довжині найкоротшого шляху з i в j, або дорівнює +∞, якщо вершина j недоступна з i. На кожному кроці алгоритму ми братимемо чергову вершину (нехай її номер — i) і для всіх пар вершин u і v обчислювати найкоротший шлях з u в v, що містить тільки вершини з множини (вершин графа ) -> {1..i}, проходить через вершину i, то найкоротшим шляхом з u в є короткий шлях з u в i, об'єднаний з найкоротшим шляхом з i в v. В іншому випадку, коли цей шлях не містить вершини i, найкоротший шлях з u в v, що містить тільки вершини з множини (вершин графа ) -> {1..i} є найкоротшим шляхом з u в v, що містить тільки вершини з множини {1..i−1 }. У результаті отримуємо, що матриця d(n) і є шуканою матрицею найкоротших шляхів, оскільки містить у собі довжини найкоротших шляхів між усіма парами вершин, що мають вершини з множини з множини {1..n}, що є просто всі вершини графа .

Ми використовуємо двумірний масив.

Необхідно знайти всі пари вершин (x, y), з'єднаних деяким шляхом. Іншими словами, потрібно побудувати нове відношення, яке складатиметься з усіх пар (x, y)

#include <iostream>

#include <climits>

#include <iomanip>

using namespace std;

#define N 4

#define M INT\_MAX

void printPath(int path[][N], int v, int u)

{

if (path[v][u] == v)

return;

printPath(path, v, path[v][u]);

cout << path[v][u] << " ";

}

void printSolution(int cost[N][N], int path[N][N])

{

for (int v = 0; v < N; v++)

{

for (int u = 0; u < N; u++)

{

if (u != v && path[v][u] != -1)

{

cout << "Shortest Path from " << v << " -> " << u << " is ("

<< v << " ";

printPath(path, v, u);

cout << u << ")" << endl;

}

}

}

}

void floydWarshall(int adjMatrix[][N])

{

int cost[N][N], path[N][N];

for (int v = 0; v < N; v++)

{

for (int u = 0; u < N; u++)

{

cost[v][u] = adjMatrix[v][u];

if (v == u)

path[v][u] = 0;

else if (cost[v][u] != INT\_MAX)

path[v][u] = v;

else

path[v][u] = -1;

}

}

for (int k = 0; k < N; k++)

{

for (int v = 0; v < N; v++)

{

for (int u = 0; u < N; u++)

{

if (cost[v][k] != INT\_MAX && cost[k][u] != INT\_MAX

&& cost[v][k] + cost[k][u] < cost[v][u])

{

cost[v][u] = cost[v][k] + cost[k][u];

path[v][u] = path[k][u];

}

}

if (cost[v][v] < 0)

{

cout << "Negative Weight Cycle Found!!";

return;

}

}

}

printSolution(cost, path);

}

int main()

{

int adjMatrix[N][N] =

{

{ 0, M, -2, M },

{ 4, 0, 3, M },

{ M, M, 0, 2 },

{ M, -1, M, 0 }

};

floydWarshall(adjMatrix);

return 0;

}

При першій ітерації проходимо через вершину 0

1->0->2 sum path 4+2

Друга ітерація проходимо через вершини 0,1

3->1->0 sum path 1+4

3->1->0->2 sum path 1+4+2

Третя ітерація проходимо через вершини 0,1,2

0->2->3 sum path 2+2

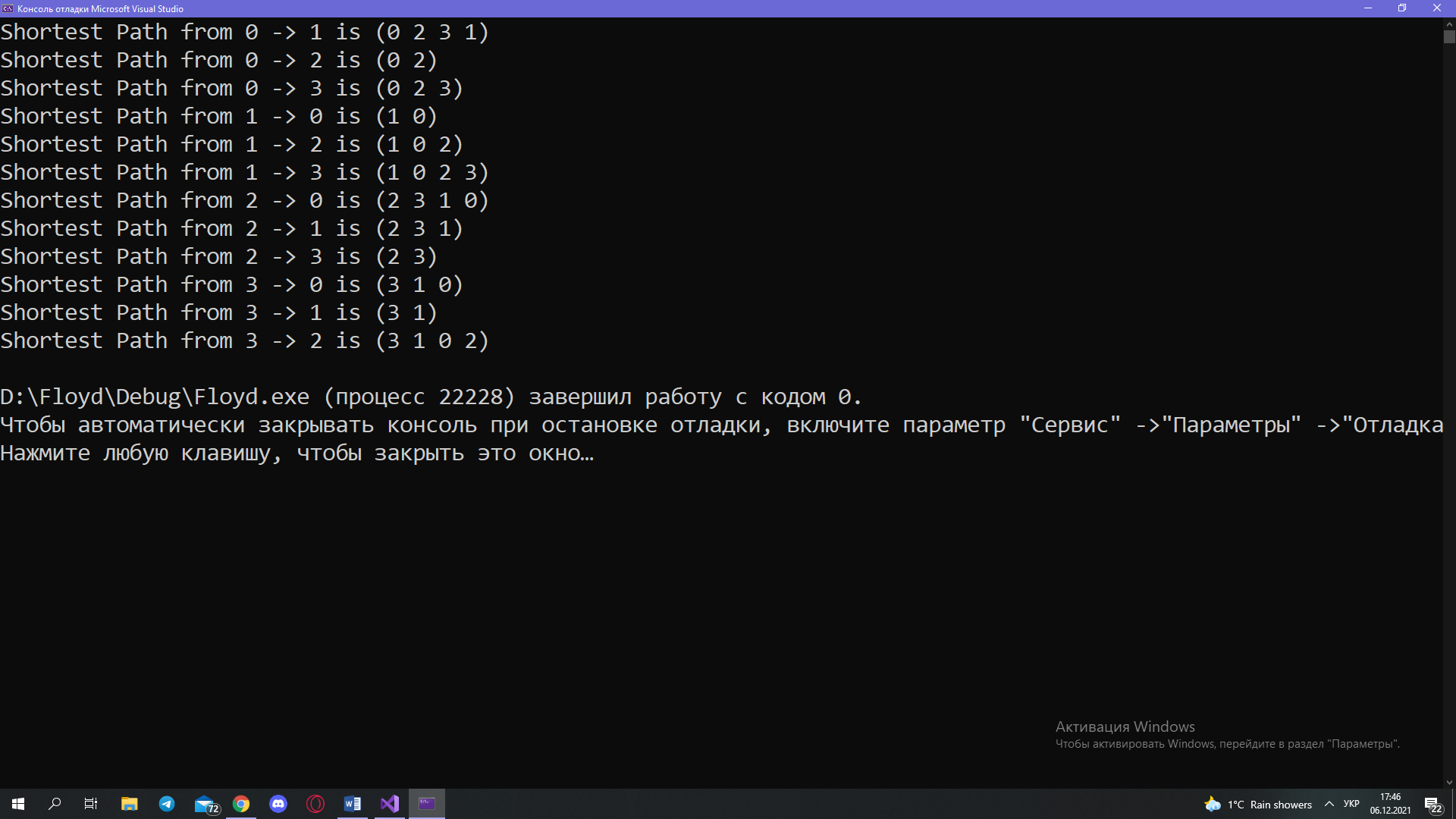
1->0->2->3 sum path 4+2+2+3

Четверта ітерація проходимо через всі вершини

2->3->1 sum path 2+1

2->3->1->0 sum path 2+1+4

0->2->3->1 sum path 2+2+1



**Оцінка складності алгоритмів**

Для розріджених графів з невід'ємними вагами ребер краще використовувати алгоритм Дейкстри для кожної можливої вихідної вершини, оскільки складність алгоритму Дейкстри ({\displaystyle O(|E||V|+|V|^{2}\log |V|)}O(n\*log n + m) ) є кращою, ніж {\displaystyle O(|V|^{3})} — складність алгоритму Флойда-Воршелла, коли {\displaystyle |E|} набагато менше від {\displaystyle |V|^{2}}. Для розріджених графів з негативними ребрами, але без негативних циклів, може бути використаний алгоритм Джонсона з тим самим асимптотичним часом роботи, що й повторюваний алгоритм Дейкстри.

**Контрольні запитання**.

1. Чи будуть коректно працювати алгоритми Дейкстри та Флойда з графом, у якого є ребра з від’ємною вагою?

Ні , тому що дані алгоритми розраховані на роботу с графами що не маюоть від’ємних ребер.

За наявності циклу негативної ваги у матриці D з'являться негативні числа на головній діагоналі.

Для пошуку циклу негативної ваги (флойд) необхідно після завершення роботи алгоритму знайти вершину i, для якої d[i][i]<0, і вивести найкоротший шлях між парою вершин (i,i) . Для запобігання переповненню всі обчислення варто обмежувати знизу величиною −∞ або перевіряти наявність негативних чисел на головній діагоналі під час підрахунку

1. В чому полягає ідея алгоритмів Дейкстри та Флойда?

Алгоритм Дейкстри дозволяє визначити як довжину найкоротшого шляху, а й сам шлях. Це досягається за допомогою покажчика, який для кожної вершини з найкоротшого шляху показує попередню вершину шляху.

Алгоритм Флойда-Уоршолла обчислює мінімальну відстань між усіма вершинами графа, використовуючи матриці і розглядаючи по етапах спочатку 1-шляху, потім 2-шляху і до л-шляхів включно (л - число вершин графа).