МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

ЦЕНТР ЗАОЧНОЇ, ДИСТАНЦІЙНОЇ ТА ВЕЧІРНЬОЇ ФОРМ НАВЧАННЯ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

СЕКЦІЯ ІКТ

ВИПУСКНА РОБОТА

на тему:

«Комп'ютерна реалізація алгоритмів Беллмана-Форда та Флойда-Уоршелла та їх порівняльний аналіз»

Завідувач

випускаючої кафедри **Довбиш А. С.**

Керівник роботи **Шаповалов С. П.**

Студента групи ІНз – 61С **Хонтураєв Б.**

СУМИ 2020

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
СУМСЬКИЙ ДЕРЖАВНИЙ УНІВЕРСИТЕТ

Центр заочної, дистанційної і вечірньої форм навчання
Кафедра комп'ютерних наук

Затверджую_____

Зав. кафедрою Довбиш А.С.

"_____" _____2020 г.

ЗАВДАННЯ

до випускної роботи

Студента п'ятого курсу, групи ІНз-61С спеціальності "Інформатика" заочної форми навчання Хонтураєва Бахтіоржона.

Тема: "Комп'ютерна реалізація алгоритмів Беллмана-Форда та Флойда-Уоршелла та їх порівняльний аналіз"

Затверджена наказом по СумДУ

№_____ от _____2020 г.

Зміст пояснювальної записки: 1) аналітичний огляд методів пошуку оптимальних шляхів в графах; 2) постановка завдання й формування завдань дослідження; 3) опис основних положень, математичних моделей і алгоритмів, що використовуються для рішення поставленого завдання; 5) розробка інформаційного й програмного забезпечення; 6) аналіз результатів моделювання.

Дата видачі завдання "_____"_____2020 г.

Керівник випускної роботи _____ Шаповалов С. П.

Завдання приняв до виконання _____ Хонтураєв Б.

РЕФЕРАТ

Записка: 43 стор., 15 рис., 3 табл., 1 додаток, 8 джерел інформації.

Об’єкт дослідження — проблема знаходження найкоротшого шляху в графі.

Мета роботи — комп’ютерний порівняльний аналіз алгоритмів знаходження найкоротших шляхів в графах.

Методи дослідження — математичне моделювання, компютерна реалізація алгоритмів на ЕОМ.

Результати — розроблено інформаційне та програмне забезпечення комп’ютерного порівняльного аналізу алгоритмів знаходження найкоротших шляхів в графі. Проведено огляд алгоритмів знаходження найкоротших шляхів у графах, обрано два алгоритми для порівняльного аналізу. Розроблено комп’ютерну реалізацію алгоритмів Беллмана-Форда та Флойда-Уоршелла, створену за допомогою алгоритмічної мови програмування С++.

ЗНАХОДЖЕННЯ НАЙКОРОТШОГО ШЛЯХУ В ГРАФАХ, АЛГОРИТ БЕЛЛМАНА-ФОРДА, АЛГОРИТМ ФЛОЙДА-УОРШЕЛЛА, ПОРІВНЯЛЬНИЙ КОМП’ЮТЕРНИЙ АНАЛІЗ, МОВА ПРОГРАМУВАННЯ С++

ЗМІСТ

ВСТУП

В останні роки графи отримала статус однієї з найважливіших структур даних в будь яких галузях науки. За її допомоги представляється можливим вирішення великої кількості проблем з області економіки, техніки і багатьох інших сферах людської діяльності, що визначає графи по суті основними об'єктами, на яких будується сучасна теорія алгоритмів [1-4]. Особливо важливий взаємозв'язок існує між теорією графів і теоретичної кібернетикою (теорією автоматів, дослідженням операцій, теорією кодування, теорією ігор).

Розвиток інформаційно-комунікаційних технологій полегшив усі сфери життя, одна з яких - простота отримання географічної інформації. Використання географічної інформації може змінюватися залежно від потреб, наприклад, потреби в пересуваннях при подорожуванні, потреби при перевезенні товарів та багато іншого. За підтримки належної інфраструктури майже ніхто не загубиться до місця призначення навіть до чужих місць або тих, яких раніше не відвідував [5-7].

На перший план виходить рішення оптимізаційних задач з застосуванням структури даних графи. Достовірно і те, що графи служать математичною моделлю для всякої системи, що містить бінарне відношення.

Задача про найкоротший шлях полягає у знаходженні найкоротшого шляху від заданої початкової вершини до заданої кінцевої вершини. Формулювання задач про знаходження відстаней таке:

1. Для заданої початкової вершини a знайти найкоротші шляхи від a до всіх інших вершин.
2. Знайти найкоротші шляхи між усіма парами вершин.

Виявляється, що майже всі методи розв’язання задачі про найкоротший шлях від заданої початкової вершини до заданої кінцевої вершини, також дають змогу знайти й найкоротші шляхи від вершини a до всіх інших вершин графа. Отже, за їх допомогою можна розв’язати будь задачу оптимального пересування із невеликими додатковими обчислювальними витратами.

1 АНАЛІЗ ПРОБЛЕМИ. ПОСТАНОВКА ЗАДАЧІ ДОСЛІДЖЕННЯ

1.1 Аналітичній огляд методів пошуку оптимальних шляхів в графах

В теорії алгоритмів та структур даних, знаходження оптимальних шляхів на структурі даних «графи» полягає в знаходженні такого шляху між двома вершинами (або вузлами) графу, що задовольняє умовам оптимальності. 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 відрізку[1-4].

В англомовних джерелах інформації ця проблема одержала назву «the shortest path problem» [5, 8].

Для подолання таких проблем, ми повинні мати вхідні дані, що складаються з заданого зваженого графа (це набір вершин V і ребер E з дійсно-значимою функцією ваги $f : E \rightarrow R$), і задану функцію цілі, оптимальне значення якої ми будемо відшукувати в рамках потрібного.

Задача «the shortest path problem» 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.

Ці узагальнення вимагають значно дієвіших алгоритмів ніж спрощений підхід із запуском алгоритму пошуку найкоротшого шляху між всіма значимими парами вершин.

На рисунку 1.1 представлено рішення задачі пошуку найкоротшого шляху у графі.

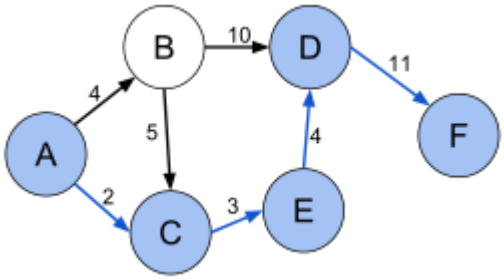

Рисунок 1.1 – Рішення проблеми пошуку найкоротшого шляху у графі

Такого роду рішення можуть прийматися для задач на відкритій місцевості, якщо та задана картою (рис. 1.2), або в ігрових завданнях на пошук шляхів в лабіринтах (рис. 1.3).

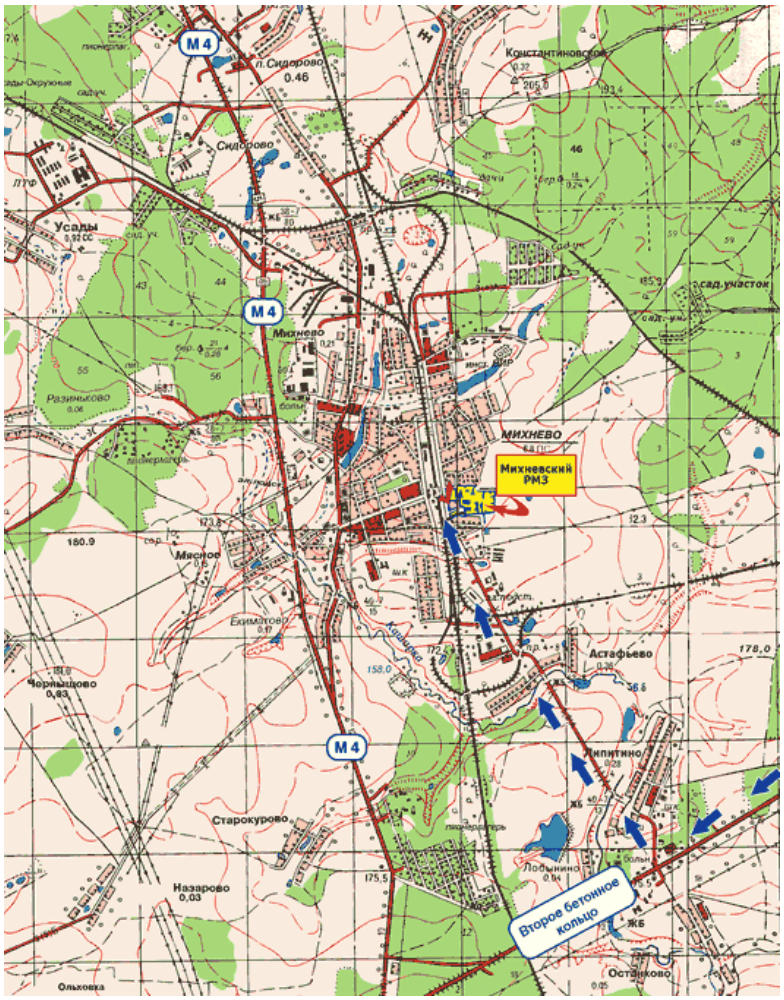

Рисунок 1.2 – Карта місцевості для задачі пошуку оптимальних шляхів

Серед існуючих алгоритмів розв'язання цієї задачі визначимо наступні:

- Алгоритм Дейкстри розв'язує задачі з однією парою, одним входом і одним виходом.
- Алгоритм Беллмана-Форда розв'язує задачі з одним входом, якщо ваги ребер можуть бути від'ємні.

- Алгоритм пошуку A* розв'язує задачу для однієї пари із використанням евристики в спробі пришвидшити пошук.
- Алгоритм Флойда-Воршелла розв'язує задачу для всіх пар.
- Алгоритм Джонсона розв'язує задачу для всіх пар, і може бути швидшим за алгоритм Флойда-Воршелла на розріджених графах.
- Теорія збурень знаходить (в найгіршому випадку) локально найкоротший шлях.

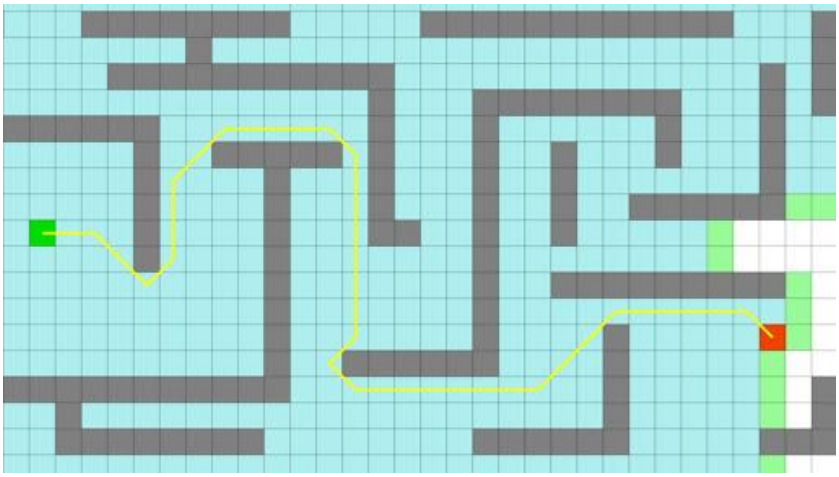

Рисунок 1.3 – Пошук найкоротшого шляху в лабіринті

У реальних задачах прикладного характеру, що моделюються структурою даних графи часто потрібно брати до уваги додаткову інформацію – фактичну віддаль між окремими пунктами, вартість проїзду, час проїзду тощо.

Для цього використовують поняття зваженого графа.

Зваженим називають граф, кожному ребру e якого приписано дійсне число w(e). Це число називають вагою ребра e. Аналогічно означають зважений орієнтований граф: це такий орієнтований граф, кожній дузі e якого приписане дійсне число w(e), яке називається вагою дуги.

Розглянемо два способи зберігання зваженого графа G = (V, E) в пам’яті комп’ютера. Нехай |V| = n, |E| = m.

Перший спосіб – подання графа матрицею ваг W, яка являє собою аналог матриці суміжності. Елементи цієї матриці $w_{ij} = w(v_i , v_j)$, якщо ребро або дуга $(v_i , v_j) \in E$. Якщо ж ребро або дуга (v_i , v_j) не входить да множини заданих ребер E, то $w_{ij} = 0$ чи $w_{ij} = \infty$ залежно від розв’язуваної задачі. Наприклад, нижче представлений граф та матриця суміжності ваг цього графу.

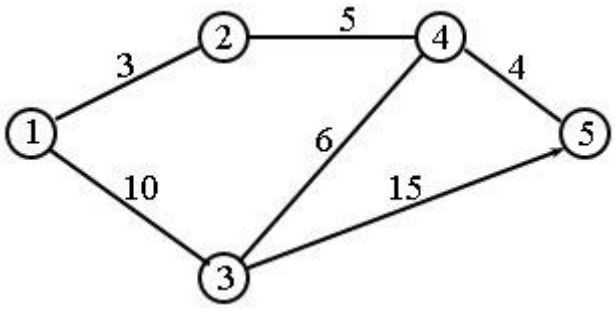

Рисунок 1.4 – Приклад графа

	1	2	3	4	5
1	—	3	10	∞	∞
2	3	—	∞	5	∞
3	10	∞	—	6	15
4	∞	5	6	—	4
5	∞	∞	∞	4	—

Рисунок 1.5 – Матриця суміжності графа, поданого рис. 1.4

Другий спосіб – поданням графа списком ребер. Для зваженого графу під кожний елемент списку E можна відвести три комірки – дві для ребра й одну для його ваги, тобто всього потрібно 3m комірок.

Довжиною шляху в зваженому графі називають суму ваг ребер (дуг), які утворюють цей шлях. Якщо граф не зважений, то вагу кожного ребра (кожної дуги) уважають рівною одиниці та отримають поняття довжини шляху як кількості ребер (дуг) у ньому.

Наступна порівняльна таблиця (див. табл. 1.1) взята з досліджень [Schrijver, Alexander (2004). Combinatorial Optimization — Polyhedra and Efficiency. Algorithms and Combinatorics] . В цій таблиці літерою L – позначено максимальну довжину (або вагу) серед всіх ребер, за умови цілочисельності всіх ваг ребер.

Алгоритми найкоротшого шляху застосовуються для автоматичного пошуку маршрутів між фізичними точками, такими як маршрути проїзду на веб-сайтах веб-картографування, таких як MapQuest або Google Maps. Для цього додатка доступні швидкі спеціалізовані алгоритми. [1]

Якщо уявити недетермінірованного абстрактну машину у вигляді графа, в якому вершини описують стану, а ребра описують можливі переходи, можна використовувати алгоритми найкоротшого шляху, щоб знайти оптимальну послідовність варіантів вибору для досягнення певного цільового стану або встановити нижні межі часу, необхідного для досягти цього стану. Наприклад, якщо вершини представляють стану головоломки, такі як кубик Рубика, і кожне спрямоване ребро відповідає одному ходу або повороту, алгоритми найкоротшого шляху можна використовувати для пошуку рішення, яке використовує мінімально можливу кількість ходів.

В мережевому або телекомунікаційному мисленні цю проблему найкоротшого шляху іноді називають проблемою мінімальної затримки і зазвичай пов'язані з проблемою самого широкого шляху. Наприклад, алгоритм може шукати найкоротший (мінімальна затримка) найширший шлях або найдовший найкоротший (мінімальна затримка) шлях.

Таблиця 1.1 Порівняльна таблиця алгоритмів пошуку найкоротших шляхів у графі

Algorithm	Time complexity	Author
	$O(V^2EL)$	Ford 1956
Bellman–Ford algorithm	$O(VE)$	Shimbel 1955, Bellman 1958, Moore 1959
	$O(V^2 \log V)$	Dantzig 1960
Dijkstra's algorithm with list	$O(V^2)$	Leyzorek et al. 1957, Dijkstra 1959, Minty (see Pollack & Wiebenson 1960), Whiting & Hillier 1960
Dijkstra's algorithm with binary heap	$O((E + V) \log V)$	Johnson 1977
. . .	. . .	. . .

Dijkstra's algorithm with Fibonacci heap	$O(E + V \log V)$	Fredman & Tarjan 1984, Fredman & Tarjan 1987
	$O(E \log \log L)$	Johnson 1981, Karlsson & Poblete 1983
Gabow's algorithm	$O(E \log_{E/V} L)$	Gabow 1983, Gabow 1985
	$O(E + V \sqrt{\log L})$	Ahuja et al. 1990
Thorup	$O(E + V \log \log V)$	Thorup 2004

Directed graphs with arbitrary weights without negative cycles[edit]

Algorithm	Time complexity	Author
	$O(V^2EL)$	Ford 1956
Bellman–Ford algorithm	$O(VE)$	Shimbel 1955, Bellman 1958, Moore 1959

1.2 Постановка задачі

Розглянемо наступну постановку задачі. *Для заданого зваженого графа розв'язати завдання знаходження шляхів найменшої довжини. Для досягнення мети знайти рішення підзадач:*

1. *Створити комп'ютерну реалізацію алгоритму Беллмана-Форда.*
2. *Створити комп'ютерну реалізацію алгоритму Флойда-Уоршелла.*
3. *Провести порівняльний аналіз цих алгоритмів на тестових завданнях.*

2 МАТЕМАТИЧНА ПОСТАНОВКА ЗАВДАННЯ ТА ВИБІР МЕТОДУ ЇЇ РІШЕННЯ

2.1 Короткий огляд відомих рішень

Завдяки широкому застосуванню, дослідження про знаходження найкоротших шляхів у графах інтенсивне розвиваються та вдосконалюються, захоплюючи все нові та нові практичні впровадження– від знаходження шляхів між об'єктами на місцевості, до знаходження оптимальних маршрутів в автопілотах та системах комунікації інформаційних пакетів в INTERNET [1-8].

На рис. 2.1 представлений один із практичних застосувань знаходження найкоротший шляхів в мережі.

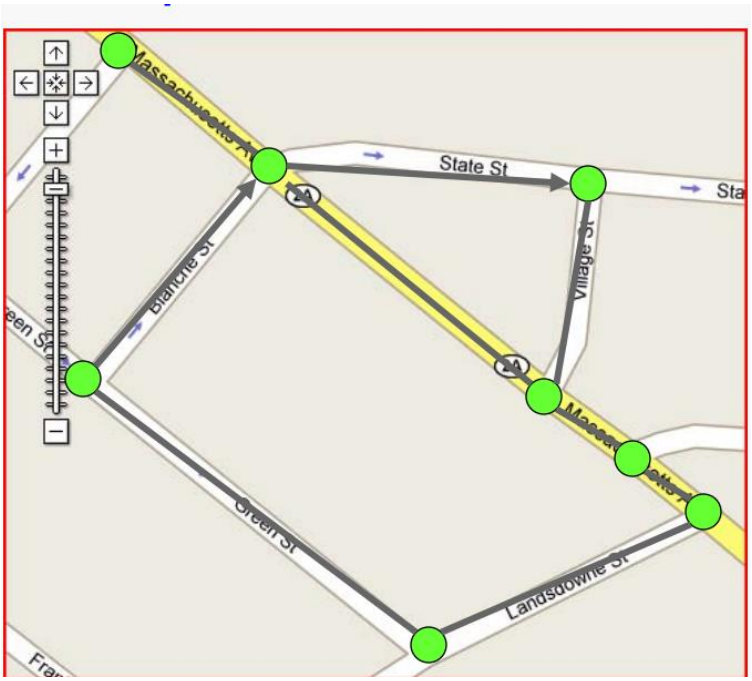

Рисунок 2.1 – Знаходження найкоротшого шляху в транспортній мережі

Нехай задано граф G = (V, E), в якому V – множина вершин, E – множина ребер, з заданими вартостями c_{ij} на кожному ребрі (i, j). Потрібно знайти найкоротший шлях між вибраними вершинами.

Математично це означає, що потрібно знайти такий шлях P між вершинами v_1 та v_n $P(v_1, v_2, \ldots, v_n)$, щоб

$$\sum_{i=1}^{n} c_{ij} \rightarrow \min \qquad (2.1)$$

Ця проблема в англомовних наукових джерелах має назву «The Shortest Path Problem» (SPP).

Існує кілька різних, але пов'язаних між собою проблем, в яких шукають найкоротші шляхи в графі:

- ❖ Найкоротший s - t шлях: знайти найкоротший шлях від s до t (однієї вибраної пари вершин).
- ❖ Пошук найкоротших шляхів з одним джерелом у графі: знайти найкоротший шлях від s до всіх вузлів.
- ❖ Пошук для всіх пар вершин найкоротших шляхів: знайти SP між кожною парою вузлів.

Крім того, ми можемо також розрізняти проблеми найкоротших шляхів залежно від типу графів, який ми отримуємо в якості вхідних даних.

Розглянемо кілька найбільш популярних алгоритмів пошуку найкоротшого шляху в графі:

1) алгоритм Дейкстри;
2) алгоритм Беллмана-Форда;
3) алгоритм пошуку A *;
4) алгоритм Флойда-Уоршелла;
5) алгоритм Лі (хвильової алгоритм).

На рис. 2.1 показаний найкоротший шлях у графі, який знайдений за алгоритмом, що працює на жадібній стратегії, тобто обирає на кожному кроці найменше рішення. Очевидно, що рішення цього алгоритму неоптимальне (тобто такого роду алгоритми не придатні до рішення SPP).

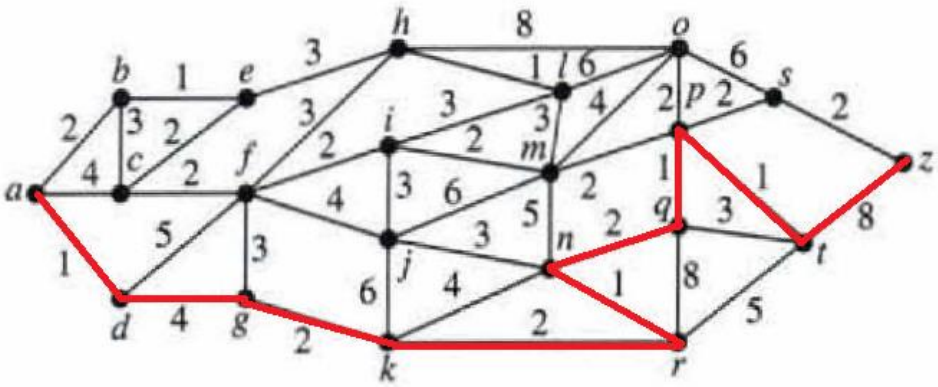

Рисунок 2.1 – Приклад, як жадібний алгоритм невірно розв’язує «The Shortest Path Problem»

Постає питання який алгоритм використовувати для рішення конкретної практичної задачі. Вибір буде залежати від властивостей графа, а також від часу виконання алгоритму. Всі дослідження повинні відповісти на наступні конкретні питання:

Чи містить граф негативні ваги ребер?

Чи містить граф позитивні ваги ребер?

Чи містить граф цикли?

Швидкість реалізації важливіше часу виконання?

В табл. 2.1 приводиться порівняльний аналіз алгоритмів, що наданий в англомовній літературі [5]. В таблиці використанні позначення n = |V|, e= |E|.

Таблиця 2.1 Порівняльний асимптотичний аналіз алгоритмів пошуку найменших шляхів в графах

Algorithm	Negative Edge Weights	Positive Edge Weights > 1	Cyclic	Runtime
DFS	✗	✗	✗	O(n + e)
BFS	✗	✗	✓	O(n + e) or $O(g^d)$
Bidirectional Search	✗	✗	✓	O(n + e) or $O(g^{(d/2)})$
Dijkstra	✗	✓	✓	O(e + n log(n))
Bellman-Ford	✓	✓	✓	O(n * e)

На підставі тих питань, що приведені вище та використання табл. 2.1 ви можете визначити потрібний алгоритм для використання. Ось дерево рішень:

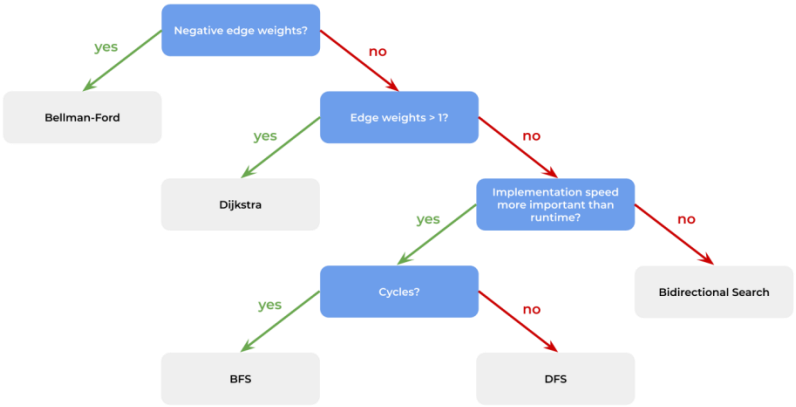

Рисунок 2.2 – Дерево рішень для вибору алгоритму

2.2 Алгоритм Беллмана –Форда та його опис

Поява алгоритму віддячує працям трьох математиків Лестера Форда, Річарда Беллмана і Едварда Мура тому іноді він має повне ім’я - алгоритм Беллмана - Форда – Мура, але більш він відомий саме як Беллмана-Форда.

І хоча його поява визначається 1956-1958 роками, сучасне своє застосування він знайшов в протоколах дистанційно-векторної маршрутизації, наприклад в RIP (Routing Information Protocol - Протокол маршрутної інформації). Алгоритм розподілений, тому що він включає в себе кілька вузлів (маршрутизаторів) всередині автономної системи, сукупність IP-мереж, зазвичай належать Інтернет-провайдеру. Він складається з наступних кроків:

1. Кожен вузол обчислює відстані між собою і всіма іншими вузлами в AS і зберігає цю інформацію у вигляді таблиці.
2. Кожен вузол відправляє свою таблицю всіх сусідніх вузлів.
3. Коли вузол отримує таблиці відстаней від своїх сусідів, він обчислює найкоротші маршрути до всіх інших вузлів і оновлює свою власну таблицю, щоб відбити будь-які зміни.

Основними недоліками алгоритму Беллмана - Форда в цих застосуваннях є:

- Це не добре масштабується.
- Зміни в топології мережі не відображаються швидко, оскільки поновлення поширюються по вузлах.
- Вважайте до нескінченності, якщо помилки каналу або вузла роблять вузол недоступним з деякого набору інших вузлів, ці вузли можуть витратити назавжди, поступово збільшуючи свої оцінки відстані до нього, і в той же час можуть виникати цикли маршрутизації.

Математичний опис алгоритму.

Вхідним завданням алгоритму є граф G = (V, E) з вагами ребер f (e). Виділимо в графі вершину-джерело *u*. Позначимо через d (v) найкоротшу відстань від джерела *u* до деякої вершини в графі *v*.

Алгоритм Беллмана-Форда відшукає функцію d (v) як розв’язання рівняння

$$d(v) = \min \{d(w) + f(e) \mid e = (w, v) \in E\}, \forall v \neq u, \qquad (2.1)$$

з початковою умовою d (u) = 0.

Основною операцією алгоритму є релаксація ребра: якщо e = (w, v) ∈ E і d (v)> d (w) + f (e), то проводиться присвоювання d (v) ← d (w) + f (e).

Псевдокод алгоритму.

```
// Ініціалізація
 для кожної вершини v ∈ G// Основна частина
 для i=1 до |G.v|-1     для кожного ребра  ∈якщо  // зберігаємо попередню вершину
 // Перевірка на наявність циклів з від'ємною вагою
 для кожного ребра  ∈ якщо повернути ХИБА
 повернути ІСТИНА
```

На рисунку 2.1 представлений інформаційний граф алгоритму, який демонструє описані рівні паралелізму.

На наведеному інформаційному графі нижній рівень паралелізму позначений в горизонтальних площинах. Множина всіх площин являє собою верхній рівень паралелізму (операції в кожній площині можуть виконуватися паралельно).

Нижній рівень паралелізму на графі алгоритму розташований на рівнях [2] і [3], відповідними операціями ініціалізації масиву дистанцій [2] і поновлення масиву с використанням даних масиву ребер [3]. Операція [4] - перевірка того, чи були зміни на останній ітерації і вихід з циклу, якщо таких не було.

Верхній рівень паралелізму, як уже говорилося, полягає в паралельному підрахунку дистанцій для різних вершин-джерел, і на малюнку відзначений різними площинами.

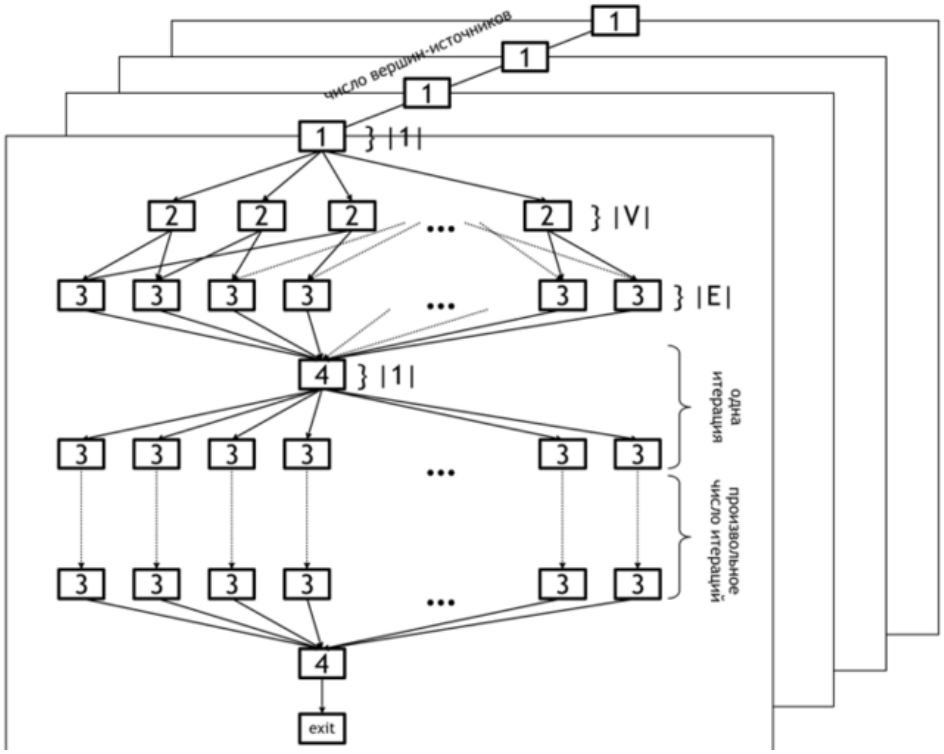

Рисунок 2.1 - Інформаційний граф алгоритму Беллмана-Форда

Послідовний алгоритм реалізується наступним псевдокодом:

Вхідні дані: граф з вершинами V, ребрами E з вагами f (e);

вершина-джерело u.

Вихідні дані: відстані d (v) до кожної вершини v ∈ V від вершини u.

```
for each v ∈ V do d (v): = ∞
d (u) = 0
for i from 1 to | V | - 1:
    for each e = (w, v) ∈ E:
        if d (v)> d (w) + f (e):
            d (v): = d (w) + f (e)
```

Алгоритм Беллмана-Форда діє шляхом релаксації, в якій наближення до оптимального значення відстані замінюються кращими, поки вони в кінцевому підсумку не досягнуть рішення. При цьому, приблизна відстань до кожної вершини завжди є завищеною оцінкою істинної відстані і замінюється мінімумом його старого значення і довжиною знову знайденого шляху.

Дія алгоритму Беллмана - Форда - він послаблює всі ребра, і робить це

| V | -1 разів, де |V| це число вершин в графі. У кожному з цих повторень зростає число вершин з правильно розрахованими відстанями, з чого випливає, що в кінцевому підсумку все вершини будуть мати свої правильні відстані.

Цей метод дозволяє застосовувати алгоритм Беллмана - Форда до більш широкого класу вхідних даних, ніж наприклад, алгоритм Дейкстра.

Асимптотична оцінка алгоритму Беллмана – Форда набуває значень

O (| V | | E |), де | V | і | E | кількість вершин і ребер відповідно.

В англомовних версіях представлення алгоритму Беллмана – Форда відповідно наступне

```
function BellmanFord(list vertices, list edges, vertex source) is
    ::distance[], predecessor[]

    // This implementation takes in a graph, represented as
    // lists of vertices and edges, and fills two arrays
    // (distance and predecessor) about the shortest path
    // from the source to each vertex

    // Step 1: initialize graph
    for each vertex v in vertices do
        distance[v] := inf             // Initialize the distance to all
vertices to infinity
        predecessor[v] := null         // And having a null predecessor

    distance[source] := 0              // The distance from the source to
itself is, of course, zero

    // Step 2: relax edges repeatedly
    for i from 1 to size(vertices)-1 do //just |V|-1 repetitions; i is never
referenced
        for each edge (u, v) with weight w in edges do
            if distance[u] + w < distance[v] then
                distance[v] := distance[u] + w
                predecessor[v] := u

    // Step 3: check for negative-weight cycles
    for each edge (u, v) with weight w in edges do
        if distance[u] + w < distance[v] then
            error "Graph contains a negative-weight cycle"

    return distance[], predecessor[]
```

Продемонструємо покрокове виконання алгоритму Беллмана-Форда на прикладі. Нехай задано вхідний граф (рис. 2.2)

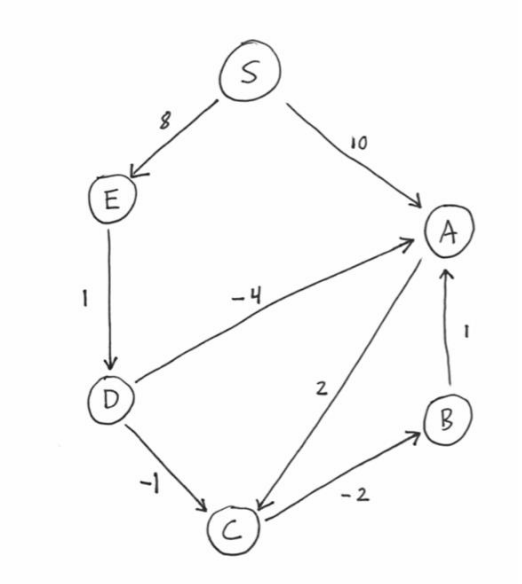

Рисунок 2.2 – Вхідний граф

1 Ітерація

S	A	B	C	D	E
0	-	-	-	-	-

S	A	B	C	D	E
0	10	-	-	-	8

A.d + w(S, A) = 0 + 10 = 10

E.d + w(S, E) = 0 + 8 = 8

S	**A**	B	C	D	E
0	10	-	12	-	8

C.d + w(A, C) = 10 + 2 = 12

S	A	**B**	C	D	E
0	10	-	12	-	8

Для B поки що немає шляху

S	A	B	**C**	D	E
0	10	10	12	-	8

B.d + w(C, B) = 12 - 2 = 10

S	A	B	C	**D**	E
0	10	-	12	-	8

Для D поки що немає шляху

S	A	B	C	D	**E**
0	10	-	12	9	8

D.d + w(E, D) = 8 + 1 = 9

Даним способом проходять всі ітерації до тих пір, поки можливо покращити значення шляху для хоча б для 1 вершини.

2.3 Алгоритм Флойда-Уоршелла та його опис

Хоча цей алгоритм має назву Флойда-Уоршелла, і його поява в інформаційному просторі датується початком 1960-х років, він по суті включає в собі наробки раніше опублікованих праць Б. Роя (1959 р.) та тісно пов'язаний з алгоритмом Кліні (опубліковано в 1956) для перетворення детермінованого кінцевого автомата в регулярний вираз. Сучасна формулювання алгоритму у вигляді трьох вкладених циклів for була вперше описана П. Інгерманом також в 1962 році [7].

Алгоритм Флойда-Уоршелла в своїй основі має парадигму динамічного програмування, що визнано в роботах Р. Флойда. Це дозволяє використовувати його для вирішення багатьох проблем:

- Оптимальна маршрутизація;
- Швидке обчислення мереж Pathfinder;
- Знаходження регулярного виразу, що позначає регулярна мова, що приймається кінцевим автоматом;
- Найкоротші шляхи в графах;
- Транзитивне замикання графів;
- Обчислення подібності між графіками та інші.

Алгоритм Флойда - Уоршелла порівнює всі можливі шляхи через граф між кожною парою вершин. Асимптотична складність такої процедури становить $\Theta(|V|^3)$. Якщо використати структуру даних «фібоначієві кучі» то складність можна оптимізувати до $O(V*E*\log(V))$, де V - кількість вершин, а E — кількість ребер («О»-велике).

Навіщо в цьому алгоритмі використовується парадигма динамічного програмування? Динамічне програмування - це альтернатива вирішення завдань методом «в лоб», тобто brute forc'ом або жадібними алгоритмами. В загальній інтерпретації цю парадигму можна подати наступне:

1. Розбиття задачі на підзадачі меншого розміру.

2. Знаходження оптимального рішення підзадач рекурсивно.

3. Використання отриманого рішення підзадач для конструювання рішення вихідної задачі.

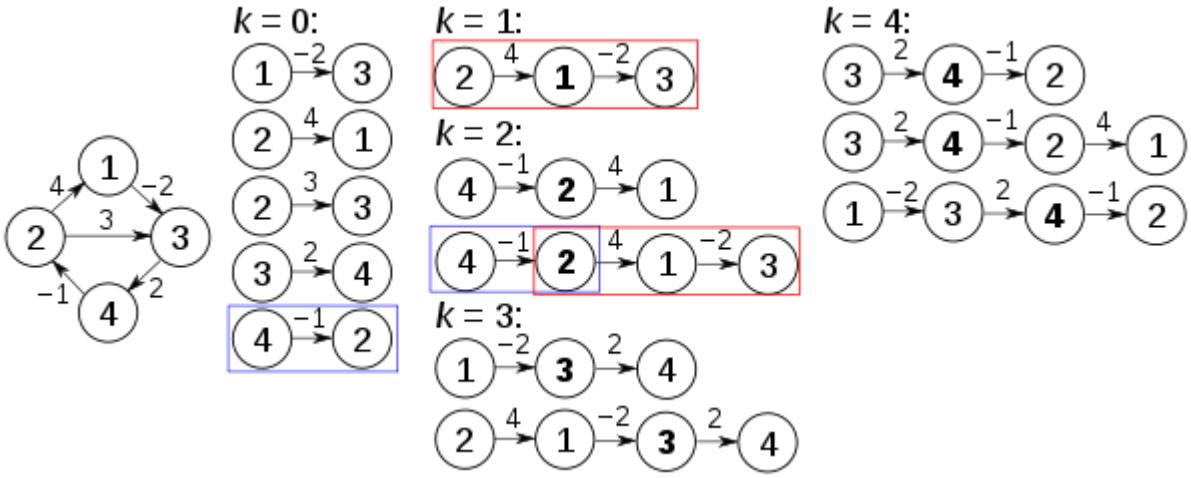

Рисунок 2.3 – Покрокове виконання алгоритму Флойда – Уоршелла з використанням парадигми динамічного програмування

Для знаходження найкоротших шляхів між усіма вершинами графа використовується не перебір всіх можливостей, що призведе до великого часу роботи і зажадає більше пам'яті, а висхідне динамічне програмування, тобто всі підзадачі, які згодом знадобляться для вирішення вихідної задачі, прораховуються заздалегідь і потім використовуються.

Псевдокод та блок-схема алгоритму Флойда-Уоршелла надані нижче. Він вирішує для всіх пар вузлів у графі проблему найкоротшого шляху, навіть для графів з негативними вагами ребер.

Алгоритм реалізується в три цикли for, зовнішній вибирає вершину, через яку ми пробуємо знайти кращий шлях, внутрішні два перебирають пари вершин, між якими ми шукаємо оптимальний шлях. У внутрішньому циклі, якщо між зупинками існує шлях і він не містить пересадок (прямий шлях), то залишаємо його і переходимо до наступної кінцевої вершині, інакше відбувається розрахунок складності маршрутів і їх порівняння. Якщо новий шлях через проміжну зупинку по складності такої ж, як і існуючий, то додаємо його як новий варіант шляху, якщо краще, то замінюємо їм існуючий шлях, якщо гірше, то залишаємо старий шлях. Якщо існуючого шляху між двома розглянутими зу-

пинками немає, то створюємо новий шлях через проміжну зупинку і поміщаємо в матрицю.

Pseudocode

Input: An $n \times n$ matrix $[c_{ij}]$
Output: An $n \times n$ matrix $[d_{ij}]$ is the shortest distance from i to j under $[c_{ij}]$
An $n \times n$ matrix $[e_{ij}]$ is a node in the path from i to j.

begin
  for all $i \neq j$ **do** $d_{ij} := c_{ij}$;
  for $i = 1, \ldots, n$ **do** $d_{ii} := \infty$;
  for $j = 1, \ldots, n$ **do**
    for $i = 1, \ldots, n, i \neq j,$ **do**
      for $k = 1, \ldots, n, k \neq j,$ **do**

$$d_{ik} := \min\{d_{ik}, d_{ij} + d_{jk}\}$$

$$e_{ik} := \begin{cases} j & \text{if } d_{ik} > d_{ij} + d_{jk} \\ e_{ik} & \text{otherwise} \end{cases}$$

end

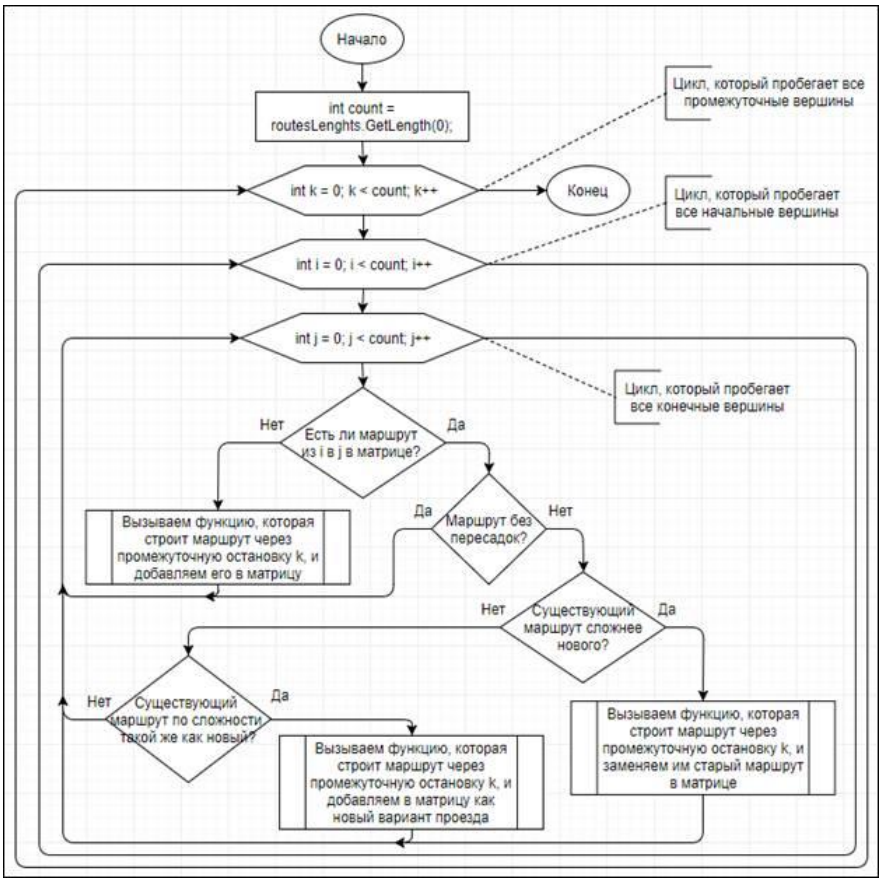

Рисунок 2.4 – Блок-схема алгоритму Флойда-Уоршелла

3 РОЗРОБКА ІНФОРМАЦІЙНОГО ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ТА ТЕСТОВІ РОЗРАХУНКИ

3.1Формування вхідних даних

Нехай задано граф G = (V, E), в якому V – множина вершин, E – множина ребер, з заданими вартостями c_{ij} на кожному ребрі (i, j). Потрібно знайти найкоротший шлях між вибраними вершинами.

Математично це означає, що потрібно знайти такий шлях P між вершинами v_1 та v_n $P(v_1, v_2, \ldots, v_n)$, щоб

$$\sum_{i=1}^{n} c_{ij} \rightarrow \min \qquad (3.1)$$

Для знаходження найкоротшого шляху та проведення комп'ютерного порівняльного аналізу швидкодії алгоритмів Флойда-Уоршелла та Беллмана - Форда було використовувано алгоритмічну мову програмування С ++ - сучасну, гнучку, мобільну мову програмування високого рівня, що використовує об'єктно-орієнтований підхід з його величезними можливостями.

Тестування проводилися на комп'ютері з наступною конфігурацією:

- ПРОЦЕССОР Intel Core i3 3240 CPU@3.40 GHz;
- ОПЕРАТИВНА ПАМ'ЯТЬ (ОРЕ) 4.0 ГБ (3.11 ГБ досяжно);
- ОПЕРАЦІЙНА СИСТЕМА Windows 10.

Вхідний інформацією є кількість вершин і їх позначення, матриця зв'язків між вершинами, яку задає користувач ІС. Дана вхідна інформація використовується на всіх етапах комп'ютерного порівняльного аналізу.

```
#define MaxNodes n  //Кількість вершин
//Описе типу вузла стеку
typedef struct Zveno *svqz;
typedef struct Zveno
{
  int Element;
  svqz Sled;
};
```

```
class Spisok
{
  private:
        int Mas[MaxNodes][MaxNodes];  //Матриця ваг ребер
        int DD[MaxNodes][MaxNodes];   //Матриця відстаней
        int SS[MaxNodes][MaxNodes];   //Матриця послідовних вершин
```

Вихідною інформацією в процесі пошуку найкоротшого шляху в графі є: граф і позначенням найкоротших шляхів, довжини найкоротших шляхів.

Виконавцями процесу є користувач ІС і інформаційна система (ІС *).

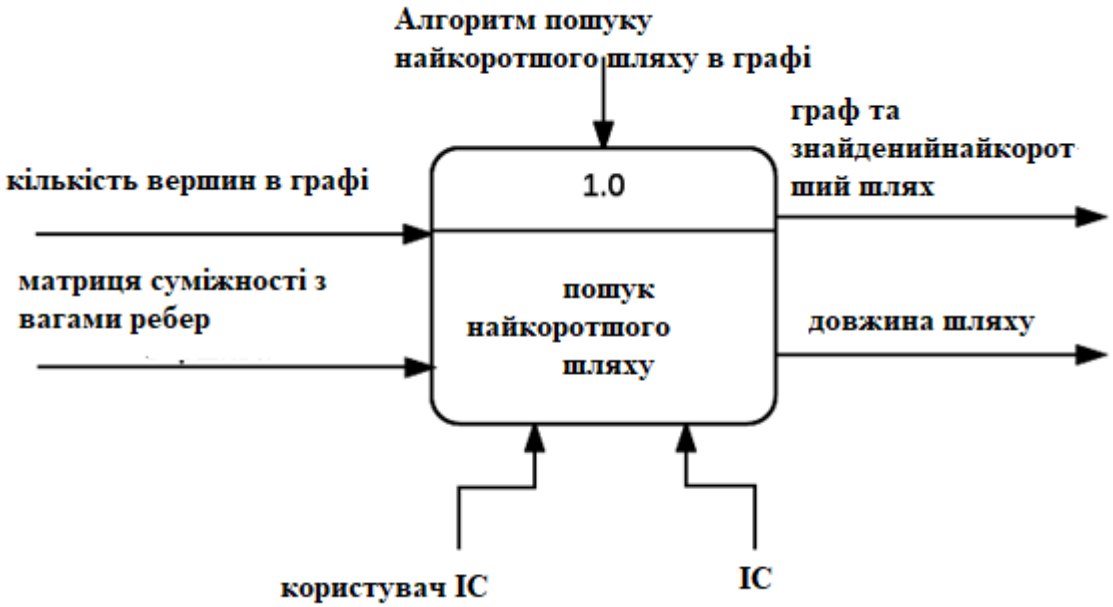

Рисунок 3.1 – Загальна схема визначення найкоротшого шляху в порівняльному аналізі алгоритмів

Матриця суміжності заданого графа, що відобращає звязки між вершинами та ваги ребер задається в програмному коді наступне

```
void Spisok::Vvod_Ves()
//Введення матриці ваг ребер заданого графу
{
  cout << "Введення по рядкам:\n";
  for (int i=0;i<MaxNodes;i++)
   for (int j=0;j<MaxNodes;j++)
```

```
    {
      cout << "Введите Mas[" << (i+1) << "," << (j+1) << "]: ";
      cin >> Mas[i][j];
    }
}
```

Знайдений найкоротший шлях та його довжина в програмному коді виводиться наступне

```
    cout << "\n Найкоротший шлях : ";
    Small_Put (one, two);
    cout << "Довжина шляху: " << DD[one][two];
```

3.2 Комп'ютерна реалізація алгоритмів та основні її складові

Управління процесом порівняльного комп'ютерного аналізу здійснюється на підставі практичної реалізації обраних алгоритмів на ЕОМ та їх тестування на різних вхідних даних, де варіюється кількості вершин на ребер в графі.

Цей процес можна охарактеризувати наступне (див. рис. 3.2):

1.1. Заповнення масиву вершин графа - на даному етапі користувач системи вносить в систему інформацію про кількість вершин аналізованого графа і позначенні даних вершин для подальшого відображення результатів розрахунків;

1.2. Заповнення матриці суміжності графа »- на даному етапі виконується заповнення матриці зв'язків вершин графа;

1.3. Реалізація алгоритму Флойда »- на даному етапі виконується розрахунок найкоротших шляхів між вершинами графа (при попарном їх порівнянні), обчислені значення зберігаються у внутрішні змінні системи;

1.4. Реалізація алгоритму Прима »- на даному етапі виконується розрахунок найкоротших шляхів між вершинами графа (при попарном їх порівнянні), обчислені значення зберігаються у внутрішні змінні системи;

1.5. Порівняльна оцінка довжин шляхів, отриманих за алгоритмами »- на даному етапі виконується порівняння довжин шляхів, знайденим по кожному

алгоритму, також виконується аналіз точності кожного алгоритму; результати виконаного аналізу надаються користувачеві у вигляді самого графа з відміченими шляхами і у вигляді інформаційного повідомлення

На рисунку 3.2 надано деталізацію процесу «Пошук найкоротшого шляху в графі».

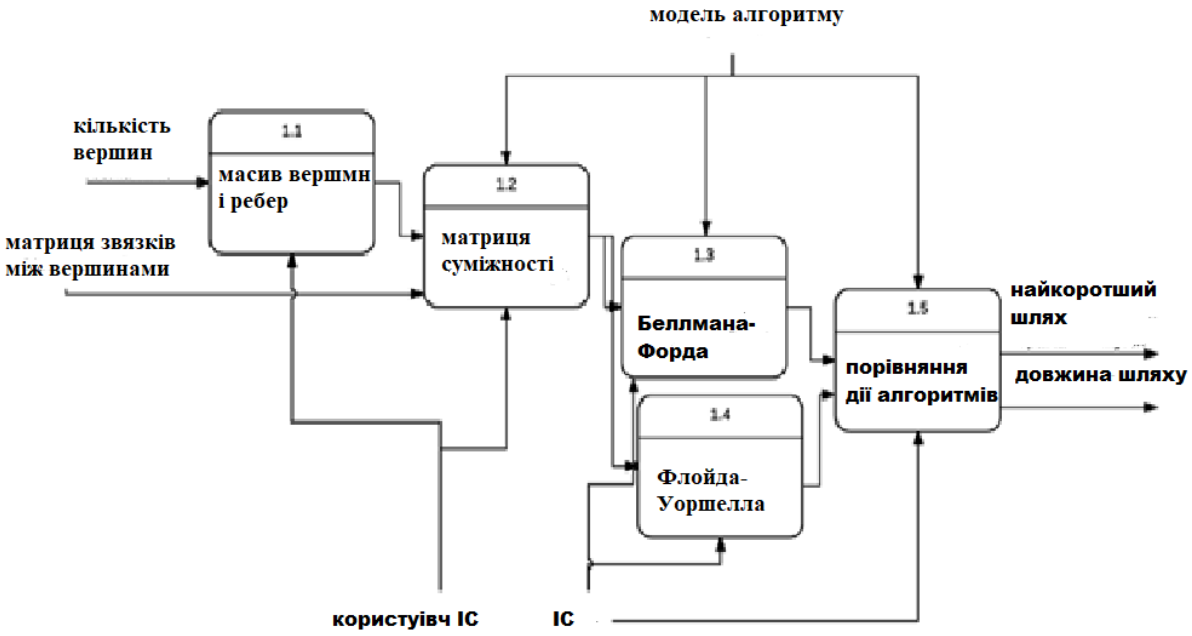

Рисунок 3.2 – Деталізація процесу проведення порівняльного комп’ютерного аналізу дії алгоритмів

3.3 Порівняльний аналіз тестових прикладів

Порівняльний аналіз проводився на однакових графах і основною метою, окрім тестування правильності відпрацювання алгоритмів, мав знаходження відповіді на питання – який з алгоритмів має переваги над іншим по швидкодії. Відмітимо основну принципову відмінність алгоритмів між собою.

Принцип роботи алгоритму Беллмана-Форда.

1) Алгоритм знаходить мінімальні шляхи від однієї вершини графа до всіх інших.

v.d – повний шлях до вершини v;

u.d – шлях від початкової до вершини u;

w(u,v) – вага ребра між вершинами u і v;

2) Алгоритм при своїй реалізації представляє із себе кілька фаз. На кожній фазі переглядаються всі ребра графа, і алгоритм намагається «поліпшити»

значення шляху v.d значенням u.d + w(u,v). Фактично це означає, що ми намагаємося поліпшити значення для вершини v користуючись ребром (u,v) і поточним значенням для вершини u.

3) Якщо граф заданий списком ребер: ініціалізація потребує O(V) часу, кожен з |V|-1 проходів потребує O(E) часу, прохід по усім ребрам для перевірки наявності негативного циклу займає O(E) часу. Отже алгоритм працює за O(V*E) часу.

Якщо граф заданий матрицею суміжності, то алгоритм буде виконуватись за O(E^3) часу.

Принцип роботи алгоритму Флойда-Уоршелла.

1) Алгоритм знаходить мінімальні шляхи між парами вершин графа.

k – кількість ітерацій (номер допоміжної вершини);

i – індекс рядка матриці (номер вершини);

j – індекс стовбця матриці (номер вершини);

a(i, j) – шлях між вершинами i та j;

2) На кожному кроці порівнюється значення шляху між вершинами i та j, та шлях між ними через допоміжну вершину k, a(i, k) + a(k, j). Якщо шлях через допоміжну вершину менше ніж поточне значення шляху між вершинами, значення мінімального шляху між цими вершинами оновлюється.

3) Оскільки граф заданий матрицею суміжності і ми маємо 3 вкладених цикли, то алгоритм має складність O(E^3).

Специфіка алгоритмів відпрацьовувалася на простих прикладах, для яких результат (найкоротший шлях у графі) одержати не громіздко.

Наприклад, нехай задано граф для тестування

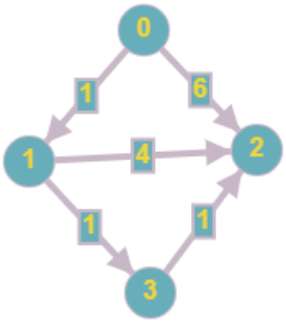

Рисунок 3.3 – Приклад для тестування

Специфіка відпрацювання алгоритму Флойда-Уоршелла.

1 Ітерація (k = 0)

	0	**1**	**2**	**3**
0	0	1	6	-
1	-	0	4	1
2	-	-	0	-
3	-	-	1	0

a(i, j) > a(i, k) + a(k, j)

a(0, 0) > a(0, 0) + a(0, 0)

0 > 0 + 0 – ні

a(i, j) > a(i, k) + a(k, j)

a(0, 1) > a(0, 0) + a(0, 1)

1 > 0 + 1 – ні

a(i, j) > a(i, k) + a(k, j)

a(0, 2) > a(0, 0) + a(0, 2)

6 > 0 + 6 – ні

Перейдемо до моменту зміни

2 Ітерація (k = 1)

	0	**1**	**2**	**3**
0	0	1	6	∞
1	∞	0	4	1
2	∞	∞	0	∞
3	∞	∞	1	0

a(i, j) > a(i, k) + a(k, j)

$$a(0, 3) > a(0, 1) + a(1, 3)$$

$$\infty > 1 + 1 – так$$

$$a(0, 3) = 2$$

	0	**1**	**2**	**3**
0	0	1	6	2
1	∞	0	4	1
2	∞	∞	0	∞
3	∞	∞	1	0

І так доки не будуть знайдені всі мінімальні шляхи.

Проведемо порівняльну оцінку алгоритмів між собою на швидкодію в залежності від кількості вершин в графі. Основні рішення представлені табл. 3.1.

Таблиця 3.1 Порівняльний аналіз алгоритмів на швидкодію

Кількість вершин графа	**Час (сек)**	
	Беллмана-Форда	**Флойда-Уоршела**
4	0.00000	0.00000
11	0.00000	0.00000
20	0.00047	0.00015
40	0.00438	0.00125

Покажемо результати таблиці графічно.

Рисунок 3. 4 – Алгоритм Беллмана-Форда

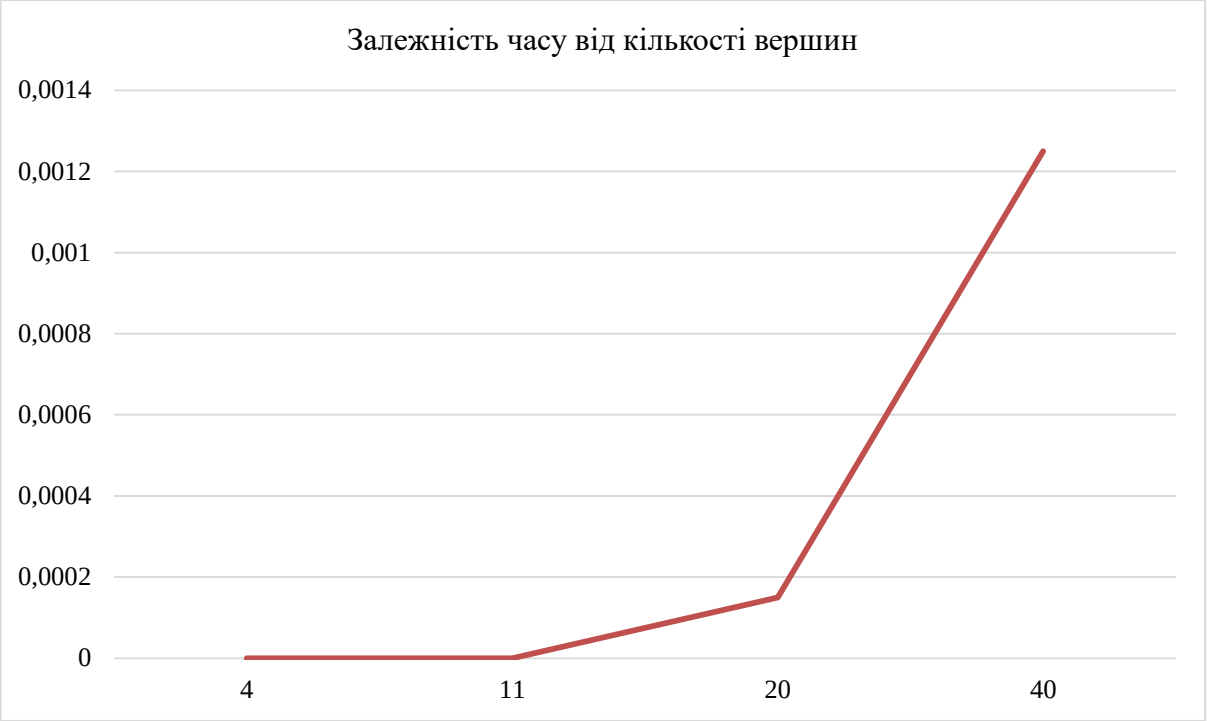

Рисунок 3.5 – Алгоритм Флойда-Уоршелла

Проведемо обробку одержаних порівняльних оцінок часу комп'ютерної реалізації алгоритмів Флойда-Уоршелла та Беллмана-Форда засобами табличного редактора Excel. На рисунку 3.5 представлені дані таблиці 3.1 та проведений трендовий аналіз одержаних функцій часу виконання алгоритмів від числа вершин в графах.

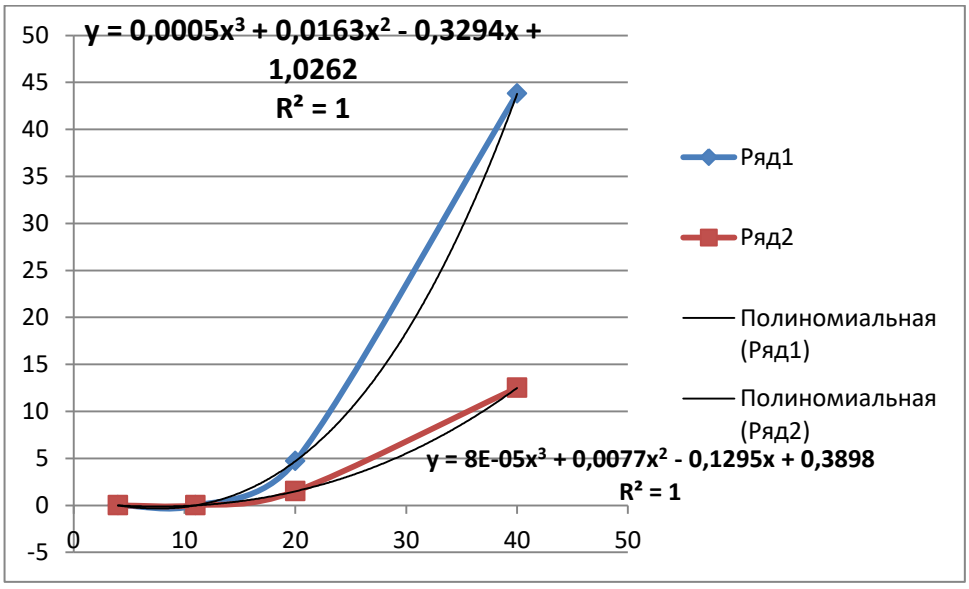

Рисунок 3.5 – Залежність часу виконання алгоритмів Флойда-Уоршелла та Беллмана-Форда від кількості вершин в графі

На діаграмі по осі абсцис відкладено кількість вершин в графі, по осі ординат – час виконання алгоритмів в форматі $t*10^{-4}$ с. В легенді ряд 1 представляє дані швидкодії алгоритму Беллмана-Форда, ряд 2 – швидкодія алгоритму Флойда-Уоршелла.

Одержані графіки функцій були апроксимовані функціями кубічного поліному (це обумовлено насамперед асимптотичними оцінками цих алгоритмів), та становлять:

Алгоритм Беллмана-Форда $T(n) = 0{,}0005n^3 + 0{,}0163n^2 + 0{,}3294n + 1{,}0262$;

Алгоритм Флойда-Уоршелла $T(n) = 8*10^{-5}n^3 + 0{,}0077n^2 - 0{,}1295n + 0{,}3898$;

де $n = |V|$ - кількість вершин в графі.

Порівняльний аналіз алгоритмів в явному вказує на перевагу другого ряду над першим, тобто алгоритм Флойда-Уоршелла за швидкодією має переваги над алгоритмом Беллмана-Форда.

ВИСНОВКИ

В випускній роботі проведені дослідження по застосуванні алгоритмів пошуку найкоротших шляхів в графах. Для порівняльного аналізу швидкодії таких алгоритмів між собою було обрано алгоритми Беллмана-Форда і Флойда-Уоршелла, що не представили труднощі у реалізації.

За результатами досліджень можна зробити наступні висновки.

1. Для незначної кількості вершин у графі обидва алгоритми по швидкодії практично співпадають.

2. Зі значному збільшенні вершин графа алгоритм Флойда-Уоршелла має переваги над алгоритмом Беллмана-Форда.

3. Одержані компютерні асимптотичні оцінки часу виконнання алгоритмів, шляхом їх апроксимації кубічними поліномами, вони складають:

Алгоритм Беллмана-Форда $T(n) = 0{,}0005n^3 + 0{,}0163n^2 + 0{,}3294n + 1{,}0262$;

Алгоритм Флойда-Уоршелла $T(n) = 8*10^{-5}n^3 + 0{,}0077n^2 - 0{,}1295n + 0{,}3898$;

де $n = |V|$ - кількість вершин в графі.

СПИСОК ЛІТЕРАТУРИ

1. Joshi B. K. Data Structures and Algorithms in C++ / Tata McGraw-Hill Education. – 2010 – 363 p.
2. Мелешко Є.В., Якименко М.С., Поліщук Л.І. Алгоритми та структури даних: Навчальний посібник для студентів технічних спеціальностей денної та заочної форми навчання. – Кропивн.: Видавець – Лисенко В.Ф., 2019. – 156 с.
3. Shinichi Shirakawa, Yasushi Iwata, Youhei Akimoto Dynamic Optimization of Neural Network Structures Using Probabilistic Modeling // The Thirty-Second AAAI Conference on Artificial Intelligence (AAAI-18), 2018. – p. 4074-4082.
4. Кононюк А. Е. Дискретно-непрерывная математика. Графы — В 12-и кн. Кн 7, ч.4— К:"Освіта України", 2015. — 494 с.
5. Dorit S. Hochbaum Lecture Notes for IEOR 266: Graph Algorithms and Network Flows [Електронний ресурс] – Режим доступу до ресурсу: https://hochbaum.ieor.berkeley.edu/files/ieor266-2014.pdf
6. Devi B. R., Rao K. K., Rani M. A. Application of Modified Bellman-Ford Algorithm for Cooperative Communication[Електронний ресурс] – Режим доступу до ресурсу: https://link.springer.com/article/10.1007/s11277-019-06666-7 (Published: 19 August 2019).
7. Ramadiani, Bukhori D., Azainil, Dengen N. Floyd-warshall algorithm to determine the shortest path based on android// 1st International Conference on Tropical Studies and Its Application (ICTROPS), Series: Earth and Environmental Science 144 (2018) [Електронний ресурс] – Режим доступу до ресурсу: https://www.researchgate.net/publication/325017484_Floyd-warshall_algorithm_to_determine_the_shortest_path_based_on_android
8. Williams R. Faster All-Pairs Shortest Paths Via Circuit Complexity// Proceedings of the 2014 ACM Symposium on Theory of Computing, 2014. – p. 664-673.

ДОДАТОК

```
#include <iostream.h>
#define TRUE 1
#define FALSE 0
#define MaxNodes n  //Кількість вершин
//Описе типу вузла стеку
typedef struct Zveno *svqz;
typedef struct Zveno
{
  int Element;
  svqz Sled;
};

class Spisok
{
  private:
        int Mas[MaxNodes][MaxNodes];  //Матриця ваг ребер
        int DD[MaxNodes][MaxNodes];   //Матриця відстаней
        int SS[MaxNodes][MaxNodes];   //Матриця послідовних вершин
          svqz Stack; //Вказівник на робочий стек
        void UDALENIE (svqz *, int *);
        void W_S (svqz *, int);
        void Small_Put (int,int);
  public:
          Spisok() {Stack = NULL;}
        void Vvod_Ves();
        void Reshenie ();
};

void main()
```

```
{
  Spisok A;

  A.Vvod_Ves();
  A.Reshenie();
}

void Spisok::Small_Put (int one, int two)
//Знаходження найкоротшого шляху
{
  svqz St=NULL; //Вказівник на додатковий стек
  svqz UkZv;
  int Flag=FALSE; //Флаг побудови найкоротшого шляху
  int elem1,elem2,k;
  //Переміщення в стек кінцевої та початкової вершин
  W_S (&Stack,two);
  W_S (&Stack,one);
  while (!Flag)
  {
    UDALENIE(&Stack,&elem1);
    UDALENIE(&Stack,&elem2);
    if (SS[elem1][elem2]==elem2) //Якщо є шлях...
      if (elem2==two) // і це кінцевий вузел...
      {
       Flag = TRUE;   //то найкоротштй шлях знайдено
       W_S (&St,elem1);
       W_S (&St,elem2);
      }
      else //це не кінцевий вузел...
      {
```

```
      W_S (&St,elem1); //В додатковий  стек
      W_S (&Stack,elem2); //В робочий стек
     }
   else //Якщо шляху  немає
   {
     W_S (&Stack,elem2); //В робочий стек
     k = SS[elem1][elem2];
     W_S (&Stack,k);     //Запам’ятати проміжну вершину
     W_S (&Stack,elem1); //В робочий стек
   }
 }
 UkZv = St;
 while ( UkZv != NULL )
 {  cout << (UkZv->Element+1) << " ";
    UkZv = UkZv->Sled;  }
 cout << endl;
}

void Spisok::W_S (svqz *stk, int Elem)
//Переміщення Elem в стек stk.
{
 svqz q=new (Zveno);
 (*q).Element = Elem;
 (*q).Sled = *stk; *stk = q;
}

void Spisok::UDALENIE (svqz *stk, int *Klad)
//Видалення з стеку, заданого вказівником *stk.
{
 svqz q;
```

```
  if (*stk==NULL) cout<<"Спроба вибору з порожнього стеку!\n";
  else
         { *Klad = (**stk).Element;
           q = *stk; *stk = (**stk).Sled; delete q; }
}


void Spisok::Vvod_Ves()
//Введення матриці ваг ребер заданого графу
{
  cout << "Введення по рядкам:\n";
  for (int i=0;i<MaxNodes;i++)
   for (int j=0;j<MaxNodes;j++)
     {
       cout << "Введите Mas[" << (i+1) << "," << (j+1) << "]: ";
       cin >> Mas[i][j];
     }
}

void Spisok::Reshenie()
{
  int one,two;
  int i,j;

  //Иніціалізація
  for (i=0;i<MaxNodes;i++)
   for (j=0;j<MaxNodes;j++)
   {
     if (Mas[i][j]>0)  SS[i][j]=j;
```

```
  else SS[i][j]=0;
  DD[i][j]=Mas[i][j];
}
cout << "\nПочаткова вершина: ";
cin >> one; one--;
cout << "Кцнцева вершина: ";
cin >> two; two--;

int ved=0;
while (ved<MaxNodes)
{
  for (i=0;i<MaxNodes;i++)
   for (j=0;j<MaxNodes;j++)
     if (i!=j && i!=ved && j!=ved &&
        DD[i][ved]>0 && DD[ved][j]>0)
       if (DD[i][ved]+DD[ved][j]<DD[i][j] || DD[i][j]==0)
       {
         DD[i][j]=DD[i][ved]+DD[ved][j];
         SS[i][j]=ved;
       }
       ved++;
}
i=one;
if (SS[i][two]!=two && SS[i][two]!=0)
  while (SS[i][two]!=two)
  {
    j=SS[i][two];
    while (SS[i][j]!=j)  j=SS[i][j];
    i=j;
  }
```

```
    cout << "\n Найкоротший шлях : ";
    Small_Put (one, two);
    cout << "Довжина шляху: " << DD[one][two];
}

#include <iostream>
#define inf 100000
using namespace std;

struct Edges {
        int u, v, w;
};

const int Vmax = 1000;
const int Emax = Vmax*(Vmax-1)/2;
int i, j, n, e, start;
Edges edge[Emax];
int d[Vmax];

void bellman_ford(int n, int s) {
        int i, j;

        for(i=0; i<n; i++)
                d[i] = inf;
        d[s] = 0;

        for(i=0; i<n-1; i++)
                for(j=0; j<e; j++)
```

```
                    if(d[edge[j].v] + edge[j].w < d[edge[j].u])
                        d[edge[j].u] = d[edge[j].v] + edge[j].w;

    for(i=0; i<n; i++)
        if (d[i]==inf)
            cout << " " << "-1";
        else
            cout << " " << d[i];
    cout << endl;

}

/* Example
4
0 1 6 0
0 0 4 1
0 0 0 0
0 0 1 0
*/


int main() {
    int w;

    cout << "Number of verticies: "; cin>>n;
    e = 0;
    for(i=0; i<n; i++)
        for(j=0; j<n; j++) {
```

```
                cin >> w;
                if(w!=0) {
                        edge[e].v = i;
                        edge[e].u = j;
                        edge[e].w = w;
                        e++;
                }
        }

    cout << "\n Adjacency matrix:\n";
    for(start=1; start<n+1; start++)
        bellman_ford(n, start-1);
    system("pause");
}


#include <cstdlib>
#include <iostream>
using namespace std;

int main() {
    int n, i, j, k;
    cout<<"Number of vertecies: ";
    cin >> n;
    int a[n][n];
    for(i=0; i<n; i++) {
        for(j=0; j<n; j++) {
            cin >> a[i][j];
```

```
                }
        }
/*
4
0 1 6 -1
-1 0 4 1
-1 -1 0 -1
-1 -1 1 0
*/
        for(k=0; k<n; k++) {
                for(i=0; i<n; i++) {
                        for(j=0; j<n; j++) {
                                if(i != j && a[i][k] != -1 && a[k][j] != -1) {
                                        if(a[i][j] == -1) {
                                                a[i][j] = a[i][k] + a[k][j];
                        }
                        else {
                                                a[i][j] = min(a[i][j], a[i][k] + a[k][j]);
                        }
                }
          }
       }
    }

        cout << "\n Adjacency matrix:\n";
        cout << endl;
        for(i=0; i<n; i++) {
                for(j=0; j<n; j++) {
```

```
                cout <<" "<< a[i][j];
            }
        cout<<endl;
        }

        return 0;
}
```