МІНІСТРЕСТВО ОСВІТИ Й НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

“Харківський Політехнічний Інститут”

Кафедра управління проєктами в інформаційних технологіях

Звіт з лабораторної роботи №7

“Однозв’язні та двозв’язні списки й робота з ними”

з дисципліни

“Алгоритми та структури даних”

Варіант №2

Перевірив: ст. викл. каф. УПІТ Мошко Є.О.

Виконав: ст. гр. КН-1223г Ставицький А.А.

Харків – 2024

**Мета лабораторної роботи:** Набуття практичних вмінь і навичок при представленні заданих графів різними способами та можливістю їх програмної реалізації. Набуття практичних вмінь та навичок по використанню алгоритмів обходу графів: алгоритму в глибину (DFS- Dipth First Search) та алгоритму в ширину (BFS- Breadth First Search). Набуття практичних вмінь і навичок з використання алгоритмів Дейкстри та Флойда.

**Порядок виконання роботи:**

Для неорієнтованих та орієнтованих графів зі своїх варіантів (варіант вибирається згідно номера по журналу, варіанти завдань наведено у додатку А) виконати наступні завдання:

1. Створити програму, яка буде зберігати в файлі, заданий граф наступними

способами:

а) матрицею суміжності;

б) матрицею інцидентності;

в) списком ребер;

г) списком суміжності.

Передбачити вивід кожного представлення на екран.

2. Написати програму, яка виконуватиме наступні завдання:

1. ﻿﻿за заданою матрицею суміжності побудувати матрицю інцидентності;
2. ﻿﻿за заданою матрицею інцидентності побудувати список ребер;
3. ﻿﻿за заданою матрицею суміжності побудувати список суміжності.
4. ﻿﻿за заданою матрицею інцидентності побудувати матрицю суміжності;

3. Виконати обходи в глибину та в ширину графів зі своїх варіантів, оформивши результати. Номер варіанту вибирати згідно номера по журналу.

Варіанти завдань наведено у додатку А.

4. Для свого варіанту графа знайти найкоротші шляхи до всіх вершин від

вершини а за допомогою алгоритму Дейкстри.

5. Для свого варіанту графа знайти найкоротші шляхи між усіма парами вершин

за допомогою алгоритму Флойда.

1. ﻿﻿﻿Написати програмну реалізацію алгоритмів.

﻿﻿﻿

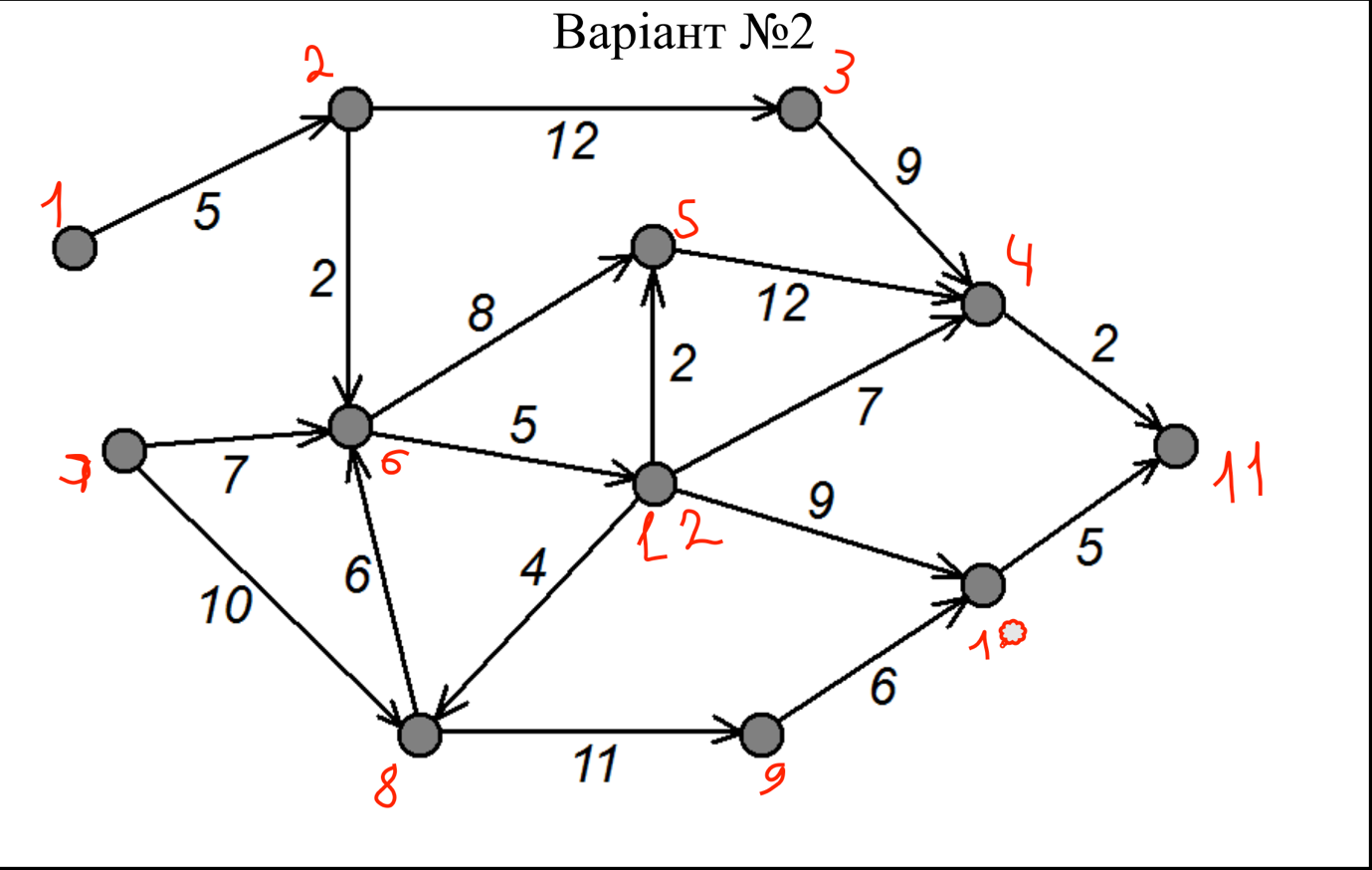


Рисунок 1 - граф 2 варіанту

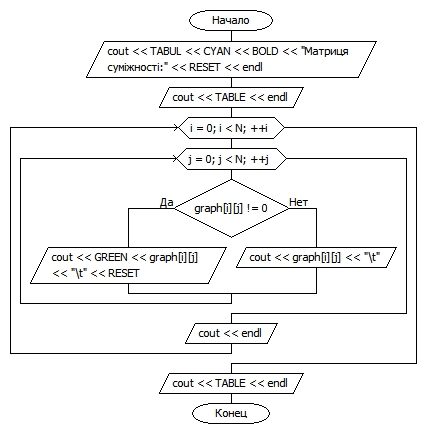


Рисунок 2 – Блок-схема до функції printAdjacencyMatrix

Функція printAdjacencyMatrix призначена для виводу матриці суміжності графа у консоль з використанням кольорового форматування. Вона приймає як параметр двовимірний масив graph[N][N], який представляє матрицю суміжності, і виводить її на екран. Ненульові елементи матриці розфарбовуються в зелений колір для легшого сприйняття, а нульові залишаються у стандартному кольорі. Спочатку виводиться заголовок із текстом “Матриця суміжності”, відформатований за допомогою кольорів і стилів (CYAN, BOLD). Потім починається вивід самої матриці: для кожного рядка і стовпця перевіряється значення елемента. Якщо значення ненульове, воно виводиться зеленим (GREEN), а потім скидається стиль (RESET). Нульові елементи виводяться без форматування. Кожен елемент розділений табуляцією для збереження таблиці у вигляді матриці. Після виводу всіх рядків і стовпців завершується таблиця. Функція зручно візуалізує матрицю суміжності, підкреслюючи важливі зв’язки між вершинами графа ненульовими значеннями.



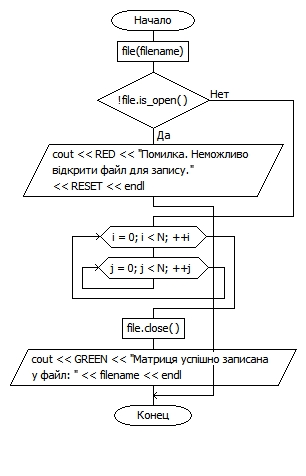


Рисунок 3 – Блок-схема до функції saveAdjacencyMatrixToFile

Функція saveAdjacencyMatrixToFile призначена для запису матриці суміжності графа у файл. Вона отримує два параметри: двовимірний масив graph[N][N], який містить матрицю суміжності, і рядок filename, що задає ім’я файлу для збереження даних. Спочатку функція відкриває файл за допомогою об’єкта ofstream. Якщо файл не вдалося відкрити, програма виводить повідомлення про помилку червоним кольором і припиняє виконання функції. У разі успішного відкриття у файл додається заголовок “Матриця суміжності:”, щоб позначити тип даних. Далі здійснюється подвійний цикл: зовнішній проходить по рядках матриці, а внутрішній — по стовпцях. Кожен елемент записується у файл із розділенням табуляцією для зручного форматування. Після запису всіх даних файл закривається, а користувач отримує повідомлення зеленим кольором про успішне завершення операції із зазначенням імені файлу.



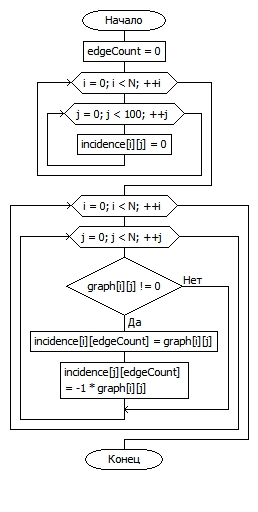


Рисунок 4 – Блок-схема до функції

Функція fillIncidenceMatrix призначена для заповнення матриці інцидентності графа на основі заданої матриці суміжності. Вона приймає три параметри: двовимірний масив graph[N][N], який містить матрицю суміжності, двовимірний масив incidence[N][100] для збереження матриці інцидентності та посилання на змінну edgeCount, яка зберігає кількість ребер графа. На початку функції змінна edgeCount ініціалізується нулем, а матриця incidence заповнюється нулями, що забезпечує початкову коректність структури.Далі функція здійснює подвійний цикл: зовнішній цикл проходить по рядках матриці суміжності, внутрішній — по стовпцях. Коли знаходиться ненульовий елемент в матриці суміжності, це означає наявність ребра між вершинами. У матриці інцидентності цей елемент відображається: у відповідному рядку для вершини-джерела зберігається позитивна вага ребра, а у рядку для вершини-приймача — від’ємна вага. Після обробки кожного ребра лічильник edgeCount збільшується на одиницю.Таким чином, функція формує матрицю інцидентності, де кожен стовпець відповідає одному ребру графа, а рядки — вершинам.



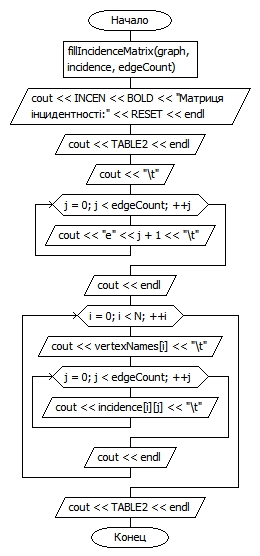


Рисунок 5 – Блок-схема до функції printIncidenceMatrix

Функція printIncidenceMatrix призначена для виведення матриці інцидентності графа на екран. Спочатку викликається функція fillIncidenceMatrix(graph, incidence, edgeCount), щоб заповнити матрицю інцидентності, ґрунтуючись на матриці суміжності. Потім виводиться заголовок “Матриця інцидентності” із застосуванням кольору (CYAN) і жирного шрифту для кращої видимості. Далі йде виведення заголовків стовпців, які позначають ребра графа у вигляді “e1”, “e2”, “e3” і так далі, до загального числа ребер edgeCount. Після цього для кожної вершини виводиться її ім’я з масиву vertexNames[i], після чого йде виведення елементів відповідного рядка з матриці інцидентності. Якщо елемент матриці не нульовий, він виводиться зеленим кольором для покращення видимості. В іншому випадку, елемент виводиться звичайним чином без кольорового оформлення. Завершується виведення таблиці за допомогою спеціальної змінної TABLE2, яка, ймовірно, використовується для завершення оформлення таблиці. Це дозволяє користувачеві чітко побачити, як кожна вершина і кожне ребро пов’язані між собою в графі.



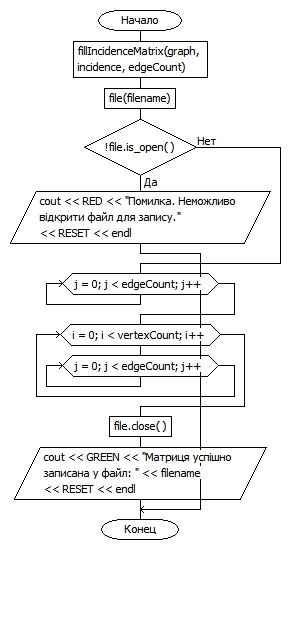


Рисунок 6 – Блок-схема до функції

Функція saveIncidenceMatrixToFile виконує збереження матриці інцидентності графа в файл. Ось кроки її роботи: 1. Спочатку викликається функція fillIncidenceMatrix(graph, incidence, edgeCount), яка заповнює матрицю інцидентності, базуючись на матриці суміжності, і визначає кількість ребер. 2. Використовуємо ofstream file(filename), щоб відкрити файл для запису. Якщо файл не відкривається (наприклад, через проблеми з доступом або правами на запис), виводиться повідомлення про помилку. 3. Записується заголовок “Матриця інцидентності”, що позначає, що наступні дані будуть стосуватися матриці інцидентності. Потім йде заголовок стовпців, які позначають ребра графа у вигляді “e1”, “e2”, “e3” і так далі. Це забезпечує зручний і зрозумілий вигляд файлу. 4. Для кожної вершини (V1, V2, …, Vn) у файлі записується її інцидентність з усіма ребрами. Для кожної вершини у файлі будуть значення, що вказують на її зв’язок з певними ребрами графа, і це робиться по всіх ребрах, що містяться в графі. Після завершення запису в файл, функція закриває його, і виводиться повідомлення, що матриця була успішно записана в файл.



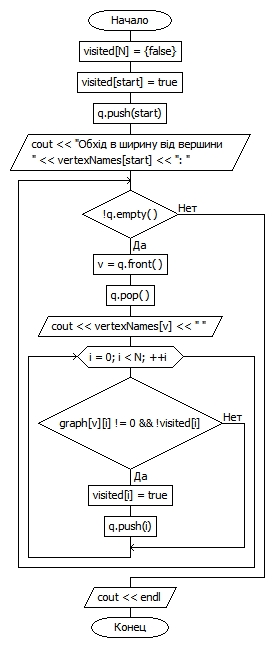


Рисунок 7 – Блок-схема до функції breadthFirstSearch

Функція breadthFirstSearch реалізує алгоритм обходу в ширину для графа, починаючи з заданої вершини. Спочатку створюється масив visited[N], який містить логічні значення, що вказують, чи була відвідана відповідна вершина (всі значення ініціалізуються як false). Потім ініціалізується черга q, яка використовується для зберігання вершин, що потребують відвідування. Початкова вершина позначається як відвідана і додається в чергу. Далі виводиться повідомлення, що вказує на початок обходу, і початкова вершина виводиться. Після цього, поки черга не порожня, з черги береться перша вершина, і для кожної суміжної вершини перевіряється, чи є між ними ребро та чи була вона вже відвідана. Якщо ребро існує і вершина ще не відвідана, то вона позначається як відвідана і додається до черги.



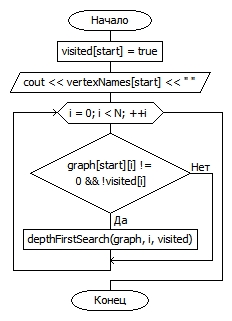


Рисунок 8 – Блок-схема до функції

Функція depthFirstSearch реалізує алгоритм обходу в глибину для графа, починаючи з заданої вершини. Параметри функції включають: graph[N][N] — матриця суміжності для графа, start — індекс початкової вершини, з якої починається обхід, та visited[N] — масив логічних значень, що позначають, чи була відвідана та чи інша вершина. Спочатку поточна вершина start позначається як відвідана, після чого вона виводиться на екран. Потім функція рекурсивно викликає себе для всіх сусідів поточної вершини, якщо вони не були відвідані (тобто, якщо є шлях до них і вони ще не були помічені як відвідані). Цей процес триває до тих пір, поки всі доступні сусіди не будуть перевірені, і обхід не завершиться.



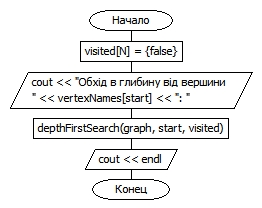
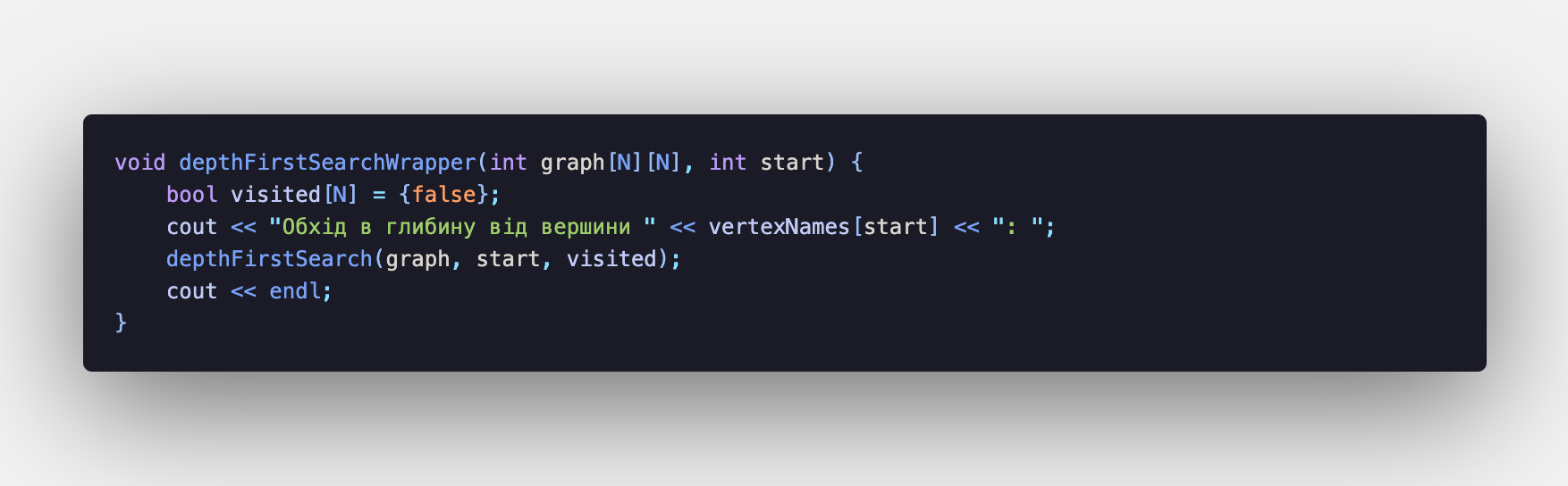


Рисунок 9 – Блок-схема до функції depthFirstSearchWrapper

Функція depthFirstSearchWrapper є обгорткою для функції depthFirstSearch. Вона викликає обхід в глибину, ініціюючи масив відвіданих вершин і передаючи його разом з графом і початковою вершиною в основну функцію пошуку. Спочатку масив visited[N] ініціалізується значеннями false, що означає, що всі вершини ще не відвідані. Потім виводиться повідомлення з назвою початкової вершини, і викликається функція depthFirstSearch для виконання самого обходу. Після завершення обходу виводиться новий рядок для кращого оформлення виведення.



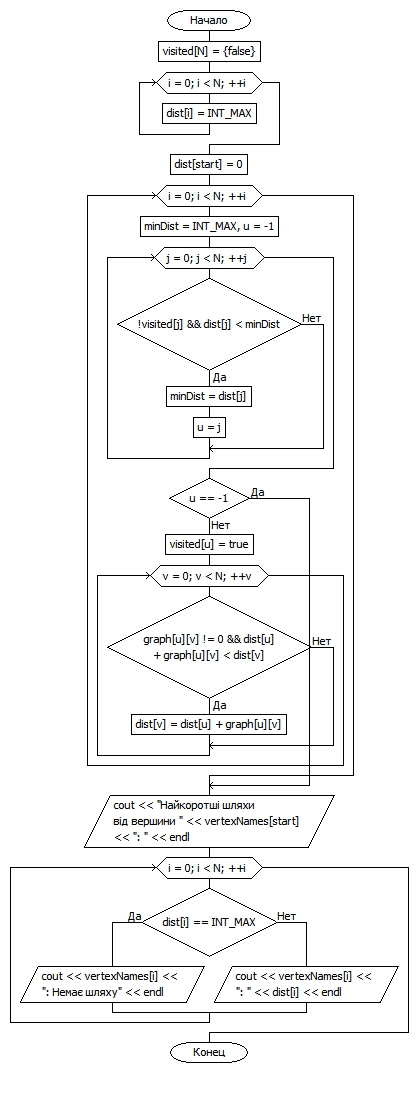


Рисунок 10 – Блок-схема до функції dijkstra

Функція dijkstra реалізує алгоритм Дейкстри для знаходження найкоротших шляхів від заданої вершини до всіх інших вершин в графі. Ось як працює цей алгоритм: 1. **Ініціалізація**: - Масив dist[N] використовується для зберігання найкоротших відстаней від початкової вершини до всіх інших. Спочатку всі значення в масиві встановлюються в INT\_MAX (означає нескінченність), за винятком початкової вершини, яка має відстань 0. - Масив visited[N] відслідковує, які вершини вже відвідані. 2. **Основний цикл**: - Для кожної вершини шукається вершина з мінімальною відстанню серед тих, які ще не відвідані. Це робиться в циклі по всіх вершинах графа. - Якщо знайдена вершина (вона не була відвідана і її відстань менша за мінімальну), ця вершина вважається поточною, і її відстань стає мінімальною.



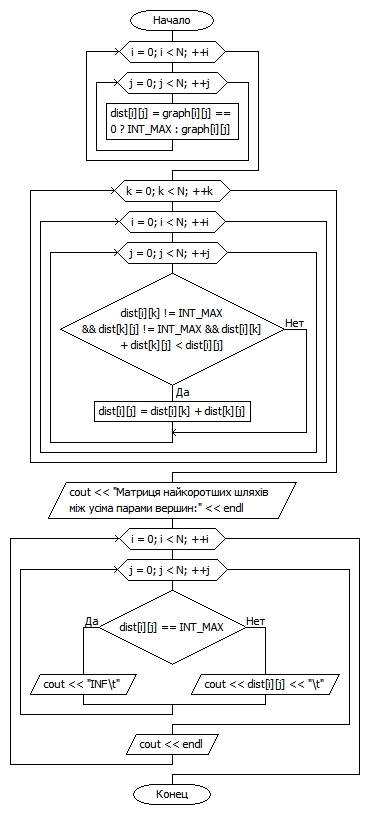


Рисунок 11 – Блок-схема до функції floydWarshall

Функція floydWarshall реалізує алгоритм Флойда-Уоршала для знаходження найкоротших шляхів між усіма парами вершин у графі. Спочатку створюється двовимірний масив dist[N][N], який містить відстані між парами вершин. Якщо між двома вершинами є шлях (граф орієнтований, а нуль означає відсутність шляху), то відстань ініціалізується відповідним значенням з матриці суміжності. Якщо шляху немає, відстань встановлюється рівною INT\_MAX, що означає нескінченність. Далі для кожної пари вершин (i, j) перевіряється, чи можна зменшити відстань, проходячи через проміжну вершину k. Якщо можна, то відстань між парами вершин оновлюється. Після виконання алгоритму матриця dist містить найкоротші відстані між усіма парами вершин. Якщо між парами немає шляху, виводиться “NO”, в іншому випадку виводиться відстань між вершинами.



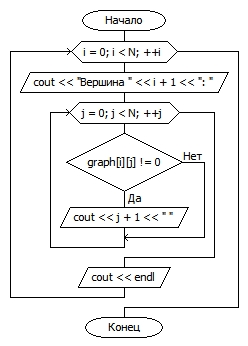


Рисунок 12 – Блок-схема до функції printAdjacencyList

Ось код функції printAdjacencyList, яка виводить список суміжності для заданої матриці суміжності. Для кожної вершини перевіряється, чи є зв’язок з іншими вершинами, і якщо такий зв’язок є, виводиться номер відповідної сусідньої вершини. Для кожної вершини виводиться її список суміжних вершин. Ця функція використовує два цикли: перший цикл проходить по кожній вершині, а другий цикл перевіряє, з якими саме вершинами є зв’язок. Якщо зв’язок є (значення в матриці суміжності не нульове), то виводиться номер вершини, з якою є зв’язок. Після виведення всіх сусідів для поточної вершини переходить на новий рядок.



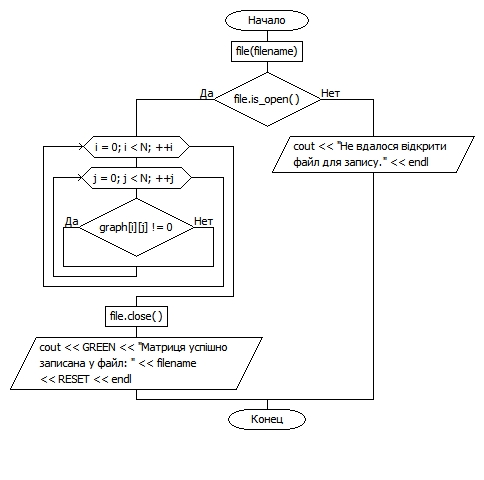


Рисунок 13 – Блок-схема до функції saveAdjacencyListToFile

Ось опис функції saveAdjacencyListToFile, яка зберігає список суміжності графа в файл. Ця функція приймає два аргументи: матрицю суміжності graph і ім’я файлу filename. Спочатку вона відкриває файл для запису за допомогою ofstream. Якщо файл успішно відкрито, то для кожної вершини графа виводиться її номер, а потім для кожної сусідньої вершини (де елемент у матриці суміжності не нульовий) записується номер сусідньої вершини. Після цього записується переведення рядка для кожної вершини, і файл закривається. Якщо файл не вдалося відкрити, виводиться відповідне повідомлення про помилку. Наприкінці функція виводить повідомлення про успішне збереження матриці в файл або повідомляє про невдачу.



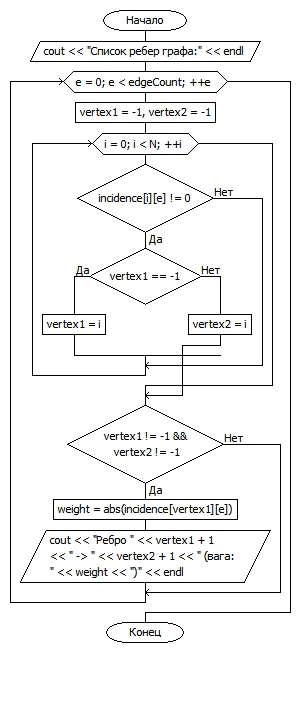


Рисунок 14 – Блок-схема до функції printEdgeListFromIncidenceMatrix

Ось опис функції printEdgeListFromIncidenceMatrix, яка виводить список ребер графа на основі матриці інцидентності.

Функція приймає два аргументи: матрицю інцидентності incidence та кількість ребер edgeCount. Вона перебирає всі елементи матриці інцидентності для кожного ребра. Для кожного ребра функція знаходить дві вершини, що інцидентні до цього ребра (тобто, мають ненульові значення в матриці). Якщо обидві вершини знайдені, то функція виводить інформацію про ребро: з якої вершини воно виходить і в яку веде, а також його вагу (яка дорівнює абсолютному значенню елемента в матриці інцидентності).

• Якщо ребро не знайдено (матриця має помилки або неточні значення), воно не виводиться.

• Якщо ребра правильно визначено, то виводиться його представлення у вигляді: “Ребро [вершина1] -> [вершина2] (вага: [вага])”.



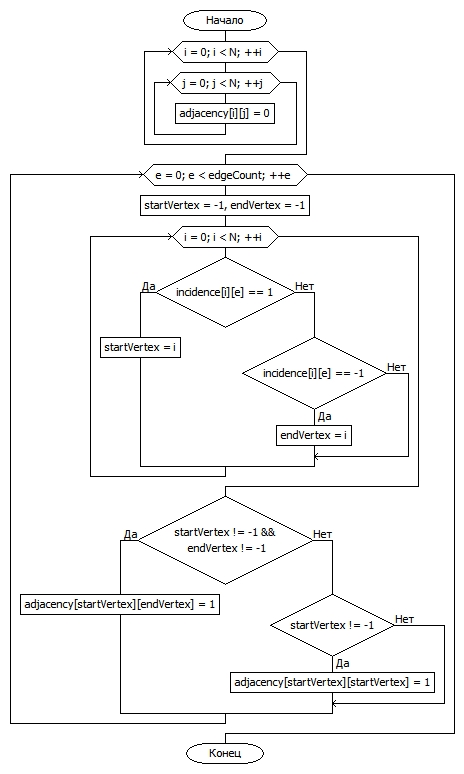


Рисунок 15 – Блок-схема до функції

Цей код реалізує перетворення матриці інцидентності в матрицю суміжності для орієнтованого графа. Спочатку ініціалізується матриця суміжності розміру N x N, заповнена нулями, що вказує на відсутність зв’язків між вершинами. Далі, через цикл обробляється кожне ребро (їх кількість задана змінною edgeCount). Для кожного ребра перевіряється, які дві вершини йому відповідають, за допомогою додаткового циклу по матриці інцидентності. Якщо значення елемента матриці інцидентності для певної вершини та ребра дорівнює 1, це означає, що дана вершина є початком ребра, і вона зберігається в змінній startVertex. Якщо значення елемента дорівнює -1, то вершина є кінцем ребра і зберігається в змінній endVertex. Після цього, якщо обидві вершини визначені, між ними в матриці суміжності ставиться одиничне значення, що означає наявність зв’язку між ними. Якщо кінцева вершина не визначена, то граф може бути петлею, і в такому разі зв’язок ставиться між тією ж самою вершиною.



**Аналіз алгоритмів**

|  |  |  |  |
| --- | --- | --- | --- |
| **Алгоритм** | **Найкращий випадок** | **Середній випадок** | **Найгірший випадок** |
|  |
| Дейкстри | O((V+E)\*log(V)) | O((V+E)\*log(V)) | O(V^2 \* log(V)) |  |
| Флойда | O(V^3) | O(V^3) | O(V^3) |  |

**Порівняльний аналіз алгоритму Дейкстри і алгоритму Флойда-Воршелла**

**Алгоритм Дейкстри:**

1. **Призначення**: Знаходить найкоротші шляхи від однієї початкової вершини до всіх інших у графі.

2. **Принцип роботи**:

• Використовує жадібний підхід.

• Поступово вибирає вершину з найменшою поточною відстанню.

• Оновлює відстані до сусідніх вершин.

3. **Складність**:

• **З матрицею суміжності**: O(N^2), де N — кількість вершин.

• **З чергою з пріоритетом** (список суміжності): O((V + E) \log V), де V — кількість вершин, E — кількість ребер.

4. **Обмеження**:

• Працює лише з графами без негативних ваг ребер.

5. **Переваги**:

• Ефективний для пошуку шляху від однієї вершини до всіх інших в розріджених графах.

6. **Недоліки**:

• Потребує окремого запуску для кожної вершини, якщо необхідно знайти всі найкоротші шляхи (не глобальний).

**Алгоритм Флойда-Воршелла:**

1. **Призначення**: Знаходить найкоротші шляхи між усіма парами вершин у графі.

2. **Принцип роботи**:

• Використовує динамічне програмування.

• Послідовно перевіряє, чи можна скоротити шлях між вершинами через проміжну вершину.

3. **Складність**:

• **Завжди**: O(N^3), де N — кількість вершин.

4. **Обмеження**:

• Не підходить для графів з негативними циклами.

5. **Переваги**:

• Забезпечує глобальне рішення для всіх пар вершин.

• Простота реалізації.

6. **Недоліки**:

• Містить кубічну складність, тому менш ефективний у великих графах.

• Потребує великої кількості пам’яті для матриць.

**Висновки:** Під час виконання лабораторної роботи я здобув практичні навички роботи з різними способами представлення графів, зокрема матрицями суміжності та інцидентності. Особливу увагу було приділено алгоритмам пошуку найкоротших шляхів, таким як алгоритм Дейкстри та Флойда-Воршелла. У процесі реалізації цих алгоритмів я зміг краще зрозуміти їхні переваги, принципи роботи та сфери застосування. Завдяки цьому я поглибив свої знання теорії графів і отримав досвід практичного вирішення задач оптимізації на графах.

Серед ключових переваг роботи можна виділити:

1. **Практичне освоєння алгоритму Дейкстри**, який є ефективним для знаходження найкоротших шляхів від однієї вершини до всіх інших. Його перевага полягає у швидкості роботи на розріджених графах, особливо при використанні черги з пріоритетом. У процесі реалізації цього алгоритму я переконався в його зручності для роботи з реальними задачами, такими як маршрутизація в мережах.

2. **Застосування алгоритму Флойда-Воршелла** для знаходження найкоротших шляхів між усіма парами вершин. Цей алгоритм показав себе як надійний інструмент для роботи з повними графами, де його універсальність і простота реалізації є вагомими перевагами.

3. Глибше розуміння теоретичних основ алгоритмів і їхньої складності. Алгоритм Дейкстри продемонстрував перевагу в швидкості, тоді як алгоритм Флойда-Воршелла забезпечує глобальне рішення для всіх пар вершин, хоч і вимагає більше обчислювальних ресурсів.

4. Розвиток навичок оптимізації коду та розуміння того, як структура даних впливає на продуктивність алгоритмів.

**До складнощів можна віднести:**

1. Потребу в значних обчислювальних ресурсах при роботі з великими графами, особливо для алгоритму Флойда-Воршелла.

2. Ретельність у відлагодженні коду, оскільки навіть незначні помилки можуть впливати на коректність результатів.

3. Додатковий час на тестування алгоритмів з різними структурами графів, щоб забезпечити їхню універсальність.

Загалом, лабораторна робота дозволила глибше зрозуміти переваги та обмеження алгоритмів Дейкстри та Флойда-Воршелла. Реалізація цих методів дала змогу оцінити їхню ефективність і використання для різних типів задач, що значно покращило мої навички в алгоритмах та програмуванні.