**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ УЧРЕЖДЕНИЯ ОБРАЗОВАНИЯ “БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ” КАФЕДРА ИНТЕЛЛЕКТУАЛЬНЫХ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ**

**Лабораторная работа №3**

По дисциплине “Теоретико-множественные основы ИС”

Тема: “Нахождение кратчайшего пути в связном графе”

**Выполнил:**

Студент 1 курса

Группы ИИ-23

Юшкевич А.Ю.

**Проверила:**

Глущенко Т.А.

**Брест 2023**

**Задание:**

- Алгоритмом Дейкстры вычислить кратчайшие пути от вершины ко всем остальным вершинам графа, указав их длины и сам путь для каждой пары вершин (последовательность вершин). Для хранения длин кратчайших путей рекомендуется использовать бинарную кучу (min-heap).
- Алгоритмом Флойда-Уоршелла вычислить кратчайшие пути между всеми парами вершин взвешенного графа, указав их длины и пути.
- Вручную указать 3 итерации прохождения алгоритмов (построить матрицы).
- Написать программу решения задачи поиска с возвратом на выбор:

задача о лабиринте ( входные данные – булева матрица размером

или .. Предусмотреть ситуацию отсутствия выхода из лабиринта).

задача о количестве островов: 200. Number of Islands ресурса https://leetcode.com / (см. файл продолжение).

Код программы:

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#define V 6
#define INF 9999
int minDistance(int dist[], int sptSet[]) {
    int min = INT_MAX, min_index;
    for (int v = 0; v < V; v++)
        if (sptSet[v] == 0 && dist[v] <= min)
            min = dist[v], min_index = v;
    return min_index;}
void printSolution(int dist[], int parent[]){
    printf("TOP \t DISTANCE \t WAY\n");
    for (int i = 0; i < V; i++)    {
        printf("%d \t\t %d \t\t %d", i, dist[i], i);
```

```
        int j = parent[i];
        while (j != -1)        {
            printf(" <- %d", j);
            j = parent[j];        }
        printf("\n");    }}
void dijkstra(int graph[V][V], int src) {
    int dist[V];
    int parent[V];
    int sptSet[V];
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = 0, parent[i] = -1;
    dist[src] = 0;
    for (int count = 0; count < V - 1; count++) {
        int u = minDistance(dist, sptSet);
        sptSet[u] = 1;
        for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u] +
graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v], parent[v] = u;    }
    printSolution(dist, parent);}
void printSolutionfW(int dist[V][V], int next[V][V]) {
    printf("MATRIX OF DISTANCE:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
            if (dist[i][j] == INF)
                printf("%7s", "INF");
            else
                printf("%7d", dist[i][j]);        }
        printf("\n");    }
    printf("MATRIX OF PATH:\n");
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < V; j++) {
```

```
            if (next[i][j] == -1)
                printf("%7s", "-");
            else
                printf("%7d", next[i][j]);        }
        printf("\n");    }}
void floydWarshall(int graph[V][V]) {
    int dist[V][V];
    int next[V][V];
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++) {
            dist[i][j] = graph[i][j];
            if (graph[i][j] != INF && i != j)
                next[i][j] = j;
            else
                next[i][j] = -1;        }
    for (int k = 0; k < V; k++) {
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (dist[i][k] + dist[k][j] < dist[i][j]) {
                    dist[i][j] = dist[i][k] + dist[k][j];
                    next[i][j] = next[i][k];                }           }
}    }
    printSolutionfW(dist, next);}
int main() {
    int graph[V][V] = {{0, 7, INF, 6, INF, INF},
                   {7, 0, 5, INF, INF, INF},
                   {INF, 5, 0, 8, 6, INF},
                   {6, INF, 8, 0, 10, 14},
                   {INF, INF, 6, 10 ,0 ,11},
                   {INF ,INF ,INF ,14 ,11 ,0}};
    dijkstra(graph, 0);
    floydWarshall(graph);
```

```
    return 0;}
```

Результаты:

Алгоритм Дейкстры:

| **Вершина** | **Метка** | **Посещена** |
|---|---|---|
| 1 | 0 | Нет |
| 2 | ∞ | Нет |
| 3 | ∞ | Нет |
| 4 | ∞ | Нет |
| 5 | ∞ | Нет |
| 6 | ∞ | Нет |

Первая итерация: выбираем вершину 1 с минимальной меткой 0 и помечаем ее как посещенную. Обновляем метки ее соседей 2 и 4:

| **Вершина** | **Метка** | **Посещена** |
|---|---|---|
| 1 | 0 | Да |
| 2 | 7 | Нет |
| 3 | ∞ | Нет |
| 4 | 6 | Нет |
| 5 | ∞ | Нет |
| 6 | ∞ | Нет |

Вторая итерация: выбираем вершину 4 с минимальной меткой 6 и помечаем ее как посещенную. Обновляем метки ее соседей 3, 5 и 6:

| **Вершина** | **Метка** | **Посещена** |
|---|---|---|
| 1 | 0 | Да |

| | | |
|---|---|---|
| 2 | 7 | Нет |
| 3 | 14 | Нет |
| 4 | 6 | Да |
| 5 | 16 | Нет |
| 6 | 20 | Нет |

Третья итерация: выбираем вершину 2 с минимальной меткой 7 и помечаем ее как посещенную. Обновляем метку ее соседа 3:

| | | |
|---|---|---|
| **Вершина** | **Метка** | **Посещена** |
| 1 | 0 | Да |
| 2 | 7 | Да |
| 3 | **12** | Нет |
| 4 | 6 | Да |
| 5 | **16** | Нет |
| **6** | 20 | Нет |

Алгоритм Флойда-Уоршелла:

| | **1** | **2** | **3** | **4** | **5** | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 7 | ∞ | 6 | ∞ | ∞ |
| 2 | 7 | 0 | 5 | ∞ | ∞ | ∞ |
| 3 | ∞ | 5 | 0 | 8 | 6 | ∞ |
| 4 | 6 | ∞ | 8 | 0 | 10 | 14 |
| 5 | ∞ | ∞ | 6 | 10 | 0 | 11 |
| 6 | ∞ | ∞ | ∞ | 14 | 11 | 0 |

В этой итерации проверяется, можно ли уменьшить расстояние между двумя вершинами , если пройти через вершину . Для этого сравнивают элемент матрицы

расстояний с суммой элементов и . Если сумма меньше, чем элемент , то мы обновляем его значением суммы. Например, для пары вершин и мы имеем:

d[2][4] = ∞ d[2][1] + d[1][4] = 7 + 6 = 13 13 < ∞ d[2][4] = d[2][1] + d[1][4] = 13

| | **1** | **2** | **3** | **4** | **5** | 6 |
|---|---|---|---|---|---|---|
| 1 | **0** | **7** | ∞ | **6** | ∞ | ∞ |
| 2 | **7** | **0** | **5** | **13** | ∞ | ∞ |
| 3 | ∞ | **5** | **0** | **8** | **6** | ∞ |
| 4 | **6** | **13** | **8** | **0** | **10** | **14** |
| 5 | ∞ | ∞ | **6** | **10** | **0** | **11** |
| 6 | ∞ | ∞ | ∞ | **14** | **11** | **0** |

d[1][3] = ∞ d[1][2] + d[2][3] = 7 + 5 = 12 12 < ∞ d[1][3] = d[1][2] + d[2][3] = 12

| **1** | **2** | **3** | **4** | **5** | **6** | |
|---|---|---|---|---|---|---|
| 1 | **0** | **7** | **12** | **6** | ∞ | ∞ |
| 2 | **7** | **0** | **5** | **13** | **11** | ∞ |
| 3 | **12** | **5** | **0** | **8** | **6** | ∞ |
| 4 | **6** | **13** | **8** | **0** | **10** | **14** |
| 5 | ∞ | **11** | **6** | **10** | **0** | **11** |
| 6 | ∞ | ∞ | ∞ | **14** | **11** | **0** |

d[1][5] = ∞ d[1][3] + d[3][5] = 12 + 6 = 18 18 < ∞ d[1][5] = d[1][3] + d[3][5] = 18

| | **1** | **2** | **3** | **4** | **5** | 6 |
|---|---|---|---|---|---|---|
| 1 | **0** | **7** | **12** | **6** | **18** | ∞ |
| 2 | **7** | **0** | **5** | **13** | **11** | ∞ |
| 3 | **12** | **5** | **0** | **8** | **6** | ∞ |
| 4 | **6** | **13** | **8** | **0** | **10** | **14** |

| | | | | | | |
|---|---|---|---|---|---|---|
| 5 | **18** | **11** | **6** | **10** | **0** | **11** |
| 6 | ∞ | ∞ | ∞ | **14** | **11** | **0** |

Задача 200 с leetcode.com:

```
void dfs(char** grid, int i, int j, int m, int n) {
    if (i >= 0 && i < m && j >= 0 && j < n && grid[i][j] == '1') {
        grid[i][j] = '0';
        dfs(grid, i - 1, j, m, n);
        dfs(grid, i + 1, j, m, n);
        dfs(grid, i, j - 1, m, n);
        dfs(grid, i, j + 1, m, n);     }}
int numIslands(char** grid, int gridSize, int* gridColSize){
    int count = 0;
    int m = gridSize;
    int n = gridColSize[0];
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == '1') {
                count++;
                dfs(grid, i, j, m, n);            }        }    }
    return count;}
```

Алгоритм решения задачи о 8 ферзях:

- Первый ферзь ставится на первую горизонталь, затем каждый следующий ставится на следующую так, чтобы его не били предыдущие ферзи.
- Если для очередного ферзя нет подходящей клетки на его горизонтали, то алгоритм возвращается к предыдущему ферзю и переставляет его на другую свободную клетку на его горизонтали.
- Если для первого ферзя нет подходящей клетки на первой горизонтали, то алгоритм завершается, так как все возможные расстановки были перебраны.
- Если все восемь ферзей были успешно расставлены, то алгоритм выводит найденное решение и продолжает поиск других решений, возвращаясь к последнему ферзю и переставляя его на другую свободную клетку.

Ответы на вопросы:

1. Жадный алгоритм - это алгоритм, который на каждом этапе принимает локально оптимальный выбор, думая, что конечное решение также будет оптимальным. Из указанных алгоритмов, **алгоритм Дейкстры** является жадным, так как он выбирает вершину с наименьшим расстоянием до начальной вершины на каждом шаге и обновляет расстояния до соседних вершин. Алгоритм Флойда-Уоршелла не является жадным, так как он вычисляет кратчайшие пути между всеми парами вершин с помощью динамического программирования. Сложность алгоритма Дейкстры зависит от реализации очереди с приоритетом и составляет **O(E + V log V)**, где E - число ребер, а V - число вершин в графе. Сложность алгоритма Флойда-Уоршелла составляет **O(V^3)**, где V - число вершин в графе.
2. Классический алгоритм Дейкстры не работает для графов с отрицательными весами, потому что он основан на жадном подходе, который выбирает узел с наименьшей стоимостью на каждом шаге. Однако, если есть отрицательные веса, то кратчайший путь может быть найден не через узел с наименьшей стоимостью, а через узел с отрицательным весом, который уменьшает общую стоимость пути
3. Эффективность алгоритма Дейкстры зависит от способа нахождения вершины с минимальной меткой и способа хранения множества непосещённых вершин. Куча - это структура данных, которая позволяет быстро находить минимальный (или максимальный) элемент и изменять его. Кучи имеют решающее значение в некоторых эффективных алгоритмах на графах, таких, как алгоритм Дейкстры. Существуют разные виды куч, которые отличаются по своим свойствам и реализации. Например, бинарная куча - это куча, в которой каждый узел имеет не более двух потомков. Куча Фибоначчи - это куча, в которой каждый узел имеет произвольное число потомков и поддерживает дополнительную операцию слияния двух куч. 2-4 куча - это куча, в которой каждый узел имеет от двух до четырех потомков и поддерживает балансировку дерева. Использование разных видов куч влияет на сложность алгоритма Дейкстры. Например, если использовать бинарную кучу, то сложность алгоритма составит O(m \log n)O(mlogn), где m — число рёбер, а n — число вершин. Если использовать кучу Фибоначчи, то сложность алгоритма составит O(m + n \log n)O(m + nlogn), что лучше для разреженных графов. Если использовать 2-4 кучу, то сложность алгоритма составит O(m \log_4 n)O(mlog4n), что лучше для плотных графов
4. Из алгоритмов Дейкстры и Флойда-Уоршелла **алгоритм Флойда-Уоршелла** построен на принципе динамического программирования. Этот принцип заключается в разбиении сложной задачи на более простые подзадачи и использовании уже найденных решений подзадач для решения исходной задачи. Алгоритм Флойда-Уоршелла находит кратчайшие пути между всеми парами вершин во взвешенном графе с положительным или отрицательным весом ребер (но без отрицательных циклов). Алгоритм Дейкстры находит кратчайшие пути от одной из вершин графа до всех остальных, но работает только для графов без ребер отрицательного веса.
5. Алгоритм Флойда-Уоршелла не работает для графов с отрицательными циклами, потому что он предполагает, что кратчайший путь между двумя вершинами не может быть улучшен добавлением промежуточных вершин. Однако, если в графе есть отрицательный цикл, то проход по этому циклу уменьшает суммарный вес пути, и так можно делать бесконечно. Да, можно использовать алгоритм Флойда-Уоршелла для проверки наличия отрицательных циклов в графе. Для этого нужно запустить алгоритм и после его окончания проверить, есть ли в матрице расстояний элементы **меньше**

**нуля** на главной диагонали. Это означает, что существует вершина, из которой можно выйти и вернуться с отрицательным суммарным весом.

6. Алгоритм Дейкстры - находит кратчайшие пути от одной вершины до всех остальных в графе с неотрицательными весами ребер. Алгоритм Беллмана-Форда - находит кратчайшие пути от одной вершины до всех остальных в графе с произвольными весами ребер и определяет наличие отрицательных циклов. Алгоритм Флойда-Уоршелла - находит кратчайшие пути между всеми парами вершин в графе с произвольными весами ребер и без отрицательных циклов. *Алгоритм A (A-star)** - находит кратчайший путь от одной вершины до другой в графе с неотрицательными весами ребер, используя эвристическую функцию для оценки расстояния до цели.
7. Если решать их полным перебором (brute force), то есть перебирать все возможные комбинации ячеек суши и воды, то временная сложность такой реализации могла бы быть равна O(2^(M*N)), где M и N - размеры матрицы.
8. В этих задачах можно применять как поиск в глубину, так и поиск в ширину. Оба этих алгоритма позволяют обойти все ячейки матрицы и пометить принадлежность к определенному острову или лабиринту. Разница между ними заключается в порядке обхода соседних ячеек. Поиск в глубину идет вглубь графа, то есть переходит к следующей ячейке только после того, как обошел все соседние ячейки текущей. Поиск в ширину идет по слоям графа, то есть переходит к следующему уровню только после того, как обошел все ячейки на текущем уровне.
9. Временная сложность поиска в глубину зависит от количества узлов и ребер в графе. Если обозначить количество узлов как N, а количество ребер как E, то временная сложность поиска в глубину составляет O(N+E). Это означает, что алгоритм поиска в глубину посещает каждый узел и каждое ребро не более одного раза.
10. Временная сложность задачи о количестве островов зависит от того, какой алгоритм используется для ее решения. Если применять поиск в глубину или поиск в ширину, то временная сложность будет равна O(M*N), где M и N - размеры матрицы. Это потому, что алгоритм посещает каждую ячейку матрицы не более одного раза. Если же применять полный перебор, то временная сложность будет равна O(2^(M*N)), что очень неэффективно.
11. Выбор варианта выхода из лабиринта зависит от того, какой алгоритм используется для его поиска. Существует несколько способов найти выход из лабиринта, например: Правило правой (или левой) руки: положите руку на правую (или левую) стену от входа в лабиринт и идите, поддерживая контакт между рукой и стеной. Этот метод гарантирует выход из лабиринта, если все стены соединены и нет островов. Алгоритм Люка-Тремо: используйте вещь, которую можно использовать, чтобы помечать каждую тропу. Отмечайте тропы по мере их прохождения. Если вы идете по тропе в первый раз, вам нужно сделать на ней одну пометку. Если вы идете по тропе во второй раз, отметьте ее еще раз. Если вы попадаете на тропу с двумя пометками, поверните назад. Этот метод позволяет найти выход из любого лабиринта. Поиск в глубину или поиск в ширину: используйте структуру данных, такую как стек или очередь, чтобы хранить тропы, которые нужно посетить. Посещайте тропы по одной и помечайте их как посещенные. Добавляйте соседние тропы в структуру данных и продолжайте поиск до тех пор, пока не найдете выход или не закончатся тропы. В зависимости от выбранного алгоритма вы можете найти разные выходы из лабиринта или даже несколько выходов одновременно.
12. Задача о 8 ферзях - это широко известная комбинаторная задача по расстановке фигур на шахматной доске. Исходная формулировка: «Расставить на стандартной 64-

клеточной шахматной доске 8 ферзей так, чтобы ни один из них не находился под боем другого». Подразумевается, что ферзь бьёт все клетки, расположенные по вертикалям, горизонталям и обеим диагоналям. Альтернативная формулировка задачи может быть такая: «Заполнить матрицу размером 8x8 нулями и единицами таким образом, чтобы сумма всех элементов матрицы была равна 8, при этом сумма элементов ни в одном столбце, строке или диагональном ряде матрицы не превышала единицы»[1]. Каждая единица в матрице соответствует ферзю на доске, а каждый ноль - пустой клетке. Общее число возможных расположений 8 ферзей на 64-клеточной доске равно (формула сочетаний ). Общее число возможных расположений, удовлетворяющих условию задачи, равно 92. Множество этих 92 расположений разбивается на 12 подмножеств (11 подмножеств по 8 расположений и одно из четырёх оставшихся), в каждом из которых все расположения получаются друг из друга путём преобразований симметрии: отражения от вертикальной и горизонтальной осей, отражения от диагоналей доски и поворотов на 90, 180 и 270 градусов.

**Вывод:** В ходе данной лабораторной работы мы изучили основные алгоритмы нахождения кратчайшего пути в связном графе.