МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Построение и анализ алгоритмов»
Тема: Алгоритмы поиска пути в графах

Студент гр. 9382 _____ Иерусалимов Н.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2021

Цель работы.

Познакомиться с алгоритмами по поиску пути в графе. Получить навыки решения задач на такие алгоритмы.

Задание.

Вар. 5. Реализовать алгоритм Дейкстры поиска пути в графе (на основе кода A*).

Задача на жадный алгоритм:

Разработайте программу, которая решает задачу построения пути в *ориентированном* графе при помощи **жадного алгоритма**. Жадность в данном случае понимается следующим образом: на каждом шаге выбирается последняя посещённая вершина. Переместиться необходимо в ту вершину, путь до которой является самым дешёвым из последней посещённой вершины. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес.

Пример входных данных

a e
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0

В первой строке через пробел указываются начальная и конечная вершины Далее в каждой строке указываются ребра графа и их вес

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет abcde

Описание алгоритмов.

Жадный алгоритм.

Начинается со стартовой вершины, смотрятся соединённые вершины к текущей. Среди них выбирается та, вес которой наименьший среди остальных соединенных вершин. Эта выбранная вершина кладется в стек, а у вершины которой смотрели соседей устанавливается флаг на использованную. Далее все повторяется с новой вершиной в стеке. Когда все соседние вершины пройдены включая текущую, то надо вернуться на одну вершину назад по стеку. Как только текущая вершина будет равняться конечной и все соседи рассмотрены, то алгоритм завершается.

Сложность:

На каждой итерации перебираются все соседние вершины, из этого выходит что сложность по времени будет O(I * J). Где J – количество соседних вершин, I – количество вершин.

Граф храниться в векторе структур. Где в каждой структуре присутствует имя графа и вектор ребер. Тогда сложность по памяти будет O(I + J) Где J – количество ребер, I – количество вершин.

Алгоритм Дейкстры.

На старте мы инициализируем новый вектор структур, куда вноситься все наши вершины с начальным весом INT32_MAX (пародия на бесконечность). Далее нашей начальной вершине присваиваем ноль, так как из начальной вершины, до начальной, можно добраться самым коротким путем за 0. Ищется минимальная не используемая вершина. После чего для новой вершины проходимся по всем ее соседям, ставим новый вес, который равняется сумме прошлого веса с текущем, флаг что вершина пройдена и указываем с какой вершины мы пришли. Если случилось так, что у новой вершины нет соседей, то

устанавливаем флаг что вершина пройдена и возвращаемся к предыдущей. Алгоритм заканчивается когда не смогли найти не использованную вершину. Для вывода начинаем проход с конечной вершины и идем по указанным предыдущем вершинам пока не дойдем до начальной.

Сложность:

В этом алгоритме асимптотика работы зависит от реализации.

Разделяют три случая реализации.

1)Наивная реализация. - n раз осуществляем поиск вершины с минимальной величиной d среди O(n) непомеченных вершин и m раз проводим релаксацию за O(1). И тогда скорость будет O($n^2 + m$).

2) Двоичная куча - Используя двоичную кучу можно выполнять операции извлечения минимума и обновления элемента за O(log n). Тогда время работы алгоритма Дейкстры составит O(n * log n + m * log n)=O(m $*$ log n).

3) Фибоначчиева куча - Используя Фибоначчиевы кучи можно выполнять операции извлечения минимума за O(log n) и обновления элемента за O(1). Таким образом, время работы алгоритма составит O(n * log n + m).

Мы используем наивную реализацию.

Так как для поиска минимально не используемой вершины мы проходимся по всему графу, а потом еще проходимся по всем соседним вершинам чтобы расставить минимальную сумму скорость будет O(I * J). Где J – количество ребер, I – количество вершин. Еще прибавим n к этой сложности так как, когда мы восстанавливаем путь и даем конечный ответ нам надо пройти расстояние от конечной вершины и до, начальной. Тогда сложность по времени будет O(I * J + n), где n – количество узлов между вершинами.

Мы создаем новый вектор структур в который вносим наш граф, т.е. сложность по памяти будет O(*2I* + *J*) Где *J* – количество ребер, *I* – количество вершин.

Описание функций и структур данных.

struct Edge – Структура ребер, хранит длину и имя к какому узлу подключена.

struct Vertex – Структура вершин, хранит имя, использована ли вершина и вектор ребер.

class Graph – класс графов, отвечает за правильное построение графа.

void AddVertex(char name) – Добавляет вершину в граф. Аргументом принимает имя вершины

void AddEdge(char vertex1, char vertex2, int mass) – Добавляет ребро. Первый аргумент имя вершины отправки, второй имя вершины получателя, третий вес ребра

Vertex *FindVertex(char name) – Находит вершину. Аргумент - имя искомой вершины. Возвращает указатель на вершину.

class GreedyAlgorithm – Класс жадного алгоритма.

GreedyAlgorithm(Graph *a) – Конструктор. Аргумент указатель на граф.

void getShortestPath(char vertex1, char vertex2) – Решение жадного алгоритма. Аргументы, соответственно: начальная вершина, конечная.

class Dijkstra – Класс по решение с помощью Дейкстры

struct NodeInfo – Структура графа для алгоритма. Содержит: имя вершины, сумму за которую можно до нее добраться, предыдущая вершина и

использована ли эта вершина.

Dijkstra(Graph *a) - Конструктор. Аргумент указатель на граф.

string getShortestPath(char vertex1, char vertex2) - Решение алгоритма Дейкстры. Аргументы, соответственно: начальная вершина, конечная. Возвращает строку с ответом

string RestorePath(char vertex1, char vertex2) – Составление ответа. Аргументы, соответственно: откуда, куда.
Возвращает строку с ответом

char FindUnusedMinimalNode() – Находит минимальную не используемую вершину. Возвращает имя вершины.

void SetSumToNextNodes(char curr) – Устанавливает самую маленькую сумму за которую можно добраться до соседей.
Аргумент – вершина, у соседей которых, надо установить сумму.

void Init() – Инициализирует структуру NodeInfo.

NodeInfo *GetNodeInfo(char vertex)- Возвращает указатель на искомую вершину. Аргумент – имя искомой вершины

Тестирование.

Тестирование жадного алгоритма.

№	Входные данные	Выходные данные
1	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 1.0 e f 2.0 a g 8.0 f g 1.0	abdefg
2	a e a b 7.0 a c 3.0 b c 1.0 c d 8.0 b e 4.0	abe
3	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 f g 1.0	abdefg

4	a d a b 1.0 b c 9.0 c d 3.0 a d 9.0 a e 1.0 e d 3.0	abcd
5	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0 f g 1.0 c m 1.0 m n 1.0	abdefg
6	a d a b 1.0 b c 1.0 c a 1.0 a d 8.0	ad
7	a d a b 1.0 b c 9.0 c d 3.0 a d 9.0 a e 1.0	abcd

	e d 3.0	
8	a f a c 1.0 a b 1.0 c d 2.0 b e 2.0 d f 3.0 e f 3.0	acdf
9	b e a b 1.0 a c 2.0 b d 7.0 b e 8.0 a g 2.0 b g 6.0 c e 4.0 d e 4.0 g e 1.0	bge

Тестирование алгоритма Дейкстры.

№	Входные данные	Выходные данные
1	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade
2	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 1.0 e f 2.0 a g 8.0 f g 1.0	ag
3	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0 f g 1.0	ag

4	a e a b 7.0 a c 3.0 b c 1.0 c d 8.0 b e 4.0	abe
5	b e a b 1.0 a c 2.0 b d 7.0 b e 8.0 a g 2.0 b g 6.0 c e 4.0 d e 4.0 g e 1.0	bge
6	a f a c 1.0 a b 1.0 c d 2.0 b e 2.0 d f 3.0 e f 3.0	acdf

	a d a b 1.0 b c 9.0 c d 3.0 a d 9.0 a e 1.0 e d 3.0	aed
8	a g a b 3.0 a c 1.0 b d 2.0 b e 3.0 d e 4.0 e a 3.0 e f 2.0 a g 8.0 f g 1.0 c m 1.0 m n 1.0	ag

Выводы.

Были получены навыки решения задач связанные с алгоритмами по поиску пути в графе.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <string>

int PRINT = 1;

using namespace std;

struct Edge {
    char name;
    int mass;

    Edge(char name, int mass) {
        this->name = name;
        this->mass = mass;
    }
};

struct Vertex {
    Vertex(char name) {
        this->name = name;
        used = false;
        //edge = new Edge();
    }

    char name;
    bool used;
    vector<Edge *> edge;

};

class Graph {
public:
    Graph() {

    }

    ~Graph() {
```

```
    }

    void AddVertex(char name) {
        if (FindVertex(name) == nullptr) {
            Vertex currVertextex(name);
            graph.push_back(currVertextex);
        }

    }

    void AddEdge(char vertex1, char vertex2, int mass) {
        FindVertex(vertex1)->edge.push_back(new Edge(vertex2, mass));
        // FindVertex(vertex2)->edge.push_back(new Edge(vertex1, mass));
    }

    Vertex *FindVertex(char name) {
        for (int i = 0; i < graph.size(); ++i) {
            if (graph[i].name == name) {
                return &graph[i];
            }
        }
        return nullptr;
    }

    vector<Vertex> graph;
};


class GreedyAlgorithm {
private:
    Graph *solve;
    vector<Vertex *> result;

public:

    GreedyAlgorithm(Graph *a) {
        solve = a;
    }

    void getShortestPath(char vertex1, char vertex2) {
        if (PRINT) {
            cout << "\t\t\tGreedy Algorithm Start!";
```

```
        }
        int minSize;
        char cur = vertex1;
        Vertex *currVertex;
        result.push_back(FindVertex(vertex1));
        currVertex = FindVertex(cur);
        int i = 0;
        while (cur != vertex2) {
            if (PRINT) {
                cout << "\n\n_____" << i
                     << "_____\n\t\tCurrent Vertex
!= answer. \n\t\t\t" << cur << " != "
                     << vertex2 << "\n\t\t\tcontinue...\n\n";
            }

            ++i;
            bool found = false;
            //result.push_back(FindVertex(cur));
            Vertex *nextVert;
            minSize = INT32_MAX;
            if (PRINT) {
                cout << "\tWe look at the neighboring vertices at the current
node - " << cur << "\n";
                cout << "\tAnd choose the smallest path.\n";
            }


            for (int i = 0; i < currVertex->edge.size(); ++i) {
                if (PRINT) {
                    cout << "\tAdjacent vertex: " << cur << " -> " << currVertex-
>edge[i]->name << "\n";

                }
                if (!FindVertex(currVertex->edge[i]->name)->used && currVertex-
>edge[i]->mass < minSize) {

                    minSize = currVertex->edge[i]->mass;
                    nextVert = FindVertex(currVertex->edge[i]->name);
                    found = true;
                    if (PRINT) {
                        cout << "\t\tThe edge is minimal " << minSize << ", we
keep the path.";
                        cout << "\t\tTemp answer is: " << cur << " -> " <<
currVertex->edge[i]->name << "\n";
                    }
```

```
                } else if (FindVertex(currVertex->edge[i]->name)->used) {
                    if (PRINT) {
                        cout << "\t\tThe neighbor has already been visited.\n";
                    }
                } else {
                    if (PRINT) {
                        cout << "\t\tCurrent weight is less than new " << minSize
<< " < " << currVertex->edge[i]->mass
                                << "\n";
                    }
                }

            }


            if (!(currVertex->name == vertex1)) {
                currVertex->used = true;
            }


            if (!found) {

                if (!result.empty()) {
                    result.pop_back();
                    currVertex = result.back();
                    cur = currVertex->name;
                }
                if (PRINT) {
                    cout << "\nAll neighboring vertices have been visited, we
return to the previous vertex - "
                         << currVertex->name << ".\n";
                }
                continue;
            }
            if (PRINT) {
                cout << "\nAdd vertex \"" << nextVert->name << "\" to the
response stack with minimum weight \""
                     << minSize
                     << "\"\n";
            }

            currVertex = nextVert;
            cur = currVertex->name;
```

```
            result.push_back(currVertex);
            if (PRINT) {
                cout << "Answer in the curren elment is : ";


                for (int i = 0; i < result.size(); ++i) {
                    cout << result[i]->name << "->";
                }
            }
        }


        if (PRINT) {
            cout
                    << "\nWe reached the final vertices. \nCompleting the
algorithm.\n\n//////////////////////////////////////////////////////////////////////
///////////\n";
        }
        cout << "Shortest path using greedy algorithm:\n";


        for (int i = 0; i < result.size(); ++i) {
            cout << result[i]->name;
        }
        return;

    }


    Vertex *FindVertex(char nameVer) {
        for (int i = 0; i < solve->graph.size(); ++i) {
            if (nameVer == solve->graph[i].name) {
                return &solve->graph[i];
            }
        }
        cout << "Can't find vertex - " << nameVer << "\n";
        exit(-1);
    }


};

class Dijkstra {
```

```
private:

    class NodeInfo {
    public:
        char vertex;
        char prev;
        int sum;
        bool used;

        NodeInfo(char vertex) {
            this->vertex = vertex;
            used = false;
            sum = INT32_MAX;
            prev = vertex;
        }
    };

    Graph *solve;
    vector<NodeInfo *> info;
    vector<char> result;

public:

    Dijkstra(Graph *a) {
        solve = a;
    }

    string getShortestPath(char vertex1, char vertex2) {
        Init();
        if (PRINT) {
            cout << "\t\t\tDijkstra Algorithm Start!\nInitialization new struct
data...\n";
            cout << "The sum of the starting vertex \'" << vertex1 << "\' is
equal to zero.\n"
                 << GetNodeInfo(vertex1)->sum
                 << " = 0\n";
        }


        GetNodeInfo(vertex1)->sum = 0;
        char curr;

        if (PRINT) {
```

```
            cout << "\tLooking for the minimum unused vertex\n";
        }

        while ((curr = FindUnusedMinimalNode()) != '\0') {
            if (PRINT) {
                cout << "\t_____________" << curr << "_____________________\n\t\tSet
the sum to the all next neighbors.\n";
            }

            SetSumToNextNodes(curr);
            if (PRINT) {
                cout << "\t___________________________________\n\n";
                cout << "\tLooking for the minimum unused vertex\n";
            }

        }
        if (PRINT) {
            cout
                    <<
"\n\n||||||||||||||||||||||||||||||||||||||||||||||||||||||||||\n\tArranged paths,
let's walk along them\n";
        }

        return RestorePath(vertex1, vertex2);
    }

    string RestorePath(char vertex1, char vertex2) {
        string path(1, vertex2);
        if (PRINT) {
            cout << "We write at the end the required vertex \'" << path <<
"\'\n";
        }

        while (vertex2 != vertex1) {
            if (PRINT) {
                cout << "Previous vertex is \'" << GetNodeInfo(vertex2)->prev <<
"\' From: \'" << vertex2 << "\'\n";

            }
            vertex2 = GetNodeInfo(vertex2)->prev;

            path = string(1, vertex2) + path;
            if (PRINT) {
                cout << "Writing in the start: " << path << "\n";
            }
```

```
        }
        return path;
    }

    void SetSumToNextNodes(char curr) {
        NodeInfo *currInfo = GetNodeInfo(curr);
        currInfo->used = true;

        for (int i = 0; i < solve->FindVertex(curr)->edge.size(); ++i) {
            NodeInfo *prevInfo = GetNodeInfo(solve->FindVertex(curr)->edge[i]-
>name);
            int newSum = currInfo->sum + solve->FindVertex(curr)->edge[i]->mass;
            if (PRINT) {
                cout << "\t\t\tFor vertex \'" << prevInfo->vertex << "\' set a
new amount. Was - " << prevInfo->sum
                     << " now - " << newSum << "\n";

            }

            if (newSum < prevInfo->sum) {
                prevInfo->sum = newSum;
                prevInfo->prev = curr;
                if (PRINT) {
                    cout << "\t\t\tRetained a pointer to the previous vertex.
\tCurrent vertex: " << prevInfo->vertex
                         << " Previous vertex: " << prevInfo->prev << "\n";

                }

            }

        }
        if (PRINT) {
            cout << "\t\tAll neighboring vertices have been visited.\n";

        }

    }

    char FindUnusedMinimalNode() {
        int minSum = INT32_MAX;
```

```
        char minVertex = '\0';
        for (int i = 0; i < solve->graph.size(); ++i) {
            NodeInfo *tempInfo = GetNodeInfo(solve->graph[i].name);
            if (tempInfo->used) continue;
            if (tempInfo->sum < minSum) {
                if (PRINT) {
                    cout << "\t\tFound an unused minimum node vertex: " <<
tempInfo->vertex << "\n";
                    cout << "\t\tPrevious minimum size is - " << minSum << ". New
is - " << tempInfo->sum << "\n\n";
                }
                minSum = tempInfo->sum;
                minVertex = solve->graph[i].name;

            }
        }
        if (PRINT) {
            cout << "\n\tThe minimum unused vertex turned out to be : " <<
minVertex << ". Size - " << minSum << "\n";
        }
        return minVertex;
    }

    void Init() {
        for (int i = 0; i < solve->graph.size(); ++i) {
            info.push_back(new NodeInfo(solve->graph[i].name));
        }
    }

    NodeInfo *GetNodeInfo(char vertex) {
        for (int i = 0; i < info.size(); ++i) {
            if (info[i]->vertex == vertex) {
                return info[i];
            }
        }
        cout << "Can't find Node info about vertex -  " << vertex;
        exit(-1);

    }


};
```

```
int main() {
    Graph a;
    string length;
    char start, end;
    char mainVertex, secondVertex;

    int i = 0;
    int choise;
    int b;
    cout << "enable Intermediate data? 1 - Yes 0 - No\n";
    cin >> b;
    PRINT = b;
    if (PRINT) {
        cout << "Gready Algoritm - 0, Dijkstra - 1\n";
    }
    cin >> choise;
    if (PRINT) {
        cout << "Input data with ')' on end: \n";
    }
    cin >> start >> end;
    while (mainVertex != ')' && cin >> mainVertex) {

        if (mainVertex != ')') {
            cin >> secondVertex >> length;
            a.AddVertex(mainVertex);
            a.AddVertex(secondVertex);
            a.AddEdge(mainVertex, secondVertex, stoi(length));
        }

    }

    if (choise == 0) {
        GreedyAlgorithm *some = new GreedyAlgorithm(&a);

        some->getShortestPath(start, end);
    } else {
        Dijkstra *some = new Dijkstra(&a);
        //cout << "Shortest Path using Dijkstra algorithm:\n";
        cout << "\nAnswer is: " + some->getShortestPath(start, end) + '\n';
    }
    system("pause>nul");
    return 0;
}
```