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 _____ Субботин М. О.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург
2021

Цель работы.

Познакомиться с одним из часто используемых на практике алгоритмом, поиска пути в графах. Получить навыки решения задач на этот алгоритм.

Задание.

Вар. 1. В A* вершины именуются целыми числами (в т. ч. отрицательными).

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

Задача на алгоритм A*:

Разработайте программу, которая решает задачу построения кратчайшего пути в *ориентированном* графе **методом A***. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес. В качестве эвристической функции следует взять близость символов, обозначающих вершины графа, в таблице ASCII.
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

ade

Описание алгоритмов.

Жадный алгоритм.

Начиная со стартовой вершины смотрятся смежные вершины к текущей. Среди этих смежных вершин выбирается та, вес ребра, которое соединяет текущую со смежной, наименьший среди остальных. Эта новая вершина прибавляется в текущий путь, а вершина, для которой смотрели смежные считается пройденной. На следующей итерации проделывается то же самое для вершины, которая была выбрана на предыдущем шаге. Если же случилось так, что все смежные вершины с текущей пройдены, то следует вернуться в пути на одну вершину назад. Алгоритм считается завершенным, как только будет рассматриваться конечная вершина (в плане рассмотрения смежных ей вершин).

На каждом шаге алгоритма перебираются смежные ребра, следовательно сложность по времени работы $O(V * E)$, где V – количество вершин, E – количество ребер.

Для хранения графа используется список смежности т.е. в этом случае $O(E)$, где E – количество ребер в графе. Также используется стек с вершинами. Тогда сложность будет $O(V + E)$, где V – количество вершин в графе.

Алгоритм A*.

На каждом шаге выбирается вершина с наименьшим приоритетом. Функция для оценки приоритета состоит из двух слагаемых: текущего расстояния от начальной вершины и эвристической функции(в данной задаче это разница между ASCII кодами символов или в случае целых чисел просто их разница по модулю). Затем для выбранной вершины рассматриваются смежные ей вершины. Для каждой смежной вершины проверяется ее кратчайший путь до начальной вершины, и если текущий путь короче, то заменяется на него. После этого эта смежная вершина помещается в очередь с приоритетом, где значение приоритетности — это текущий кратчайший путь до начальной вершины и эвристика. Алгоритм заканчивает работу, как только из очереди вытащится конечная вершина.

В лучшем случае, когда имеем наиболее подходящую эвристическую функцию, которая позволяет делать каждый шаг в верном направлении получается сложность по времени $O(V + E)$, где V – количество вершин и E – количество рёбер в графе.

В худшем случае, когда эвристическая функция очень плоха и угадывает правильное направление в последний момент, то тогда надо проходить всевозможные пути. Таким образом время работы будет расти экспоненциально по сравнению с длиной оптимального пути.

Т.к. в худшем случае все пути будут храниться в очереди, то и сложность по памяти будет экспоненциальная. В лучшем же случае для каждой вершины будет храниться путь от начала до нее. Поэтому оценка сложности по памяти будет $O(V * (V + E))$, где V – количество вершин, E – количество рёбер в графе.

Описание функций и структур данных.

std::vector<T> greedySearch() – функция, отвечающая за жадный алгоритм. Возвращаемое значение: вектор, состоящий из элементов, которые входят в кратчайший путь.

std::vector<T> aStarSearch() – функция, отвечающая за A* алгоритм. Возвращаемое значение: вектор, состоящий из элементов, которые входят в кратчайший путь.

std::map<T, std::vector<std::pair<T, double>>> graph – структура данных, отвечающая за хранения графа. Ключом является вершина графа и значение – вектор пар, где пара — это ребро (смежная вершина и вес ребра).

std::map<T, bool> visited – структура данных для отслеживания пройденных вершин в жадном алгоритме. Ключ – вершина, значение – пройден или нет.

std::map<T, std::pair<std::vector<T>, double>> shortestPaths – структура данных, которая отвечает за текущие кратчайшие пути вершин от начальной вершины. Она используется в A* алгоритме. Ключ – вершина, значение – пара: вектор вершин (сам путь) и длина этого пути.

std::priority_queue<std::pair<T, double>, std::vector<std::pair<T, double>>, CustomCompare<T>> priorityQueue – очередь в A* алгоритме. Состоит из пар: вершина и ее оценочная функция f = g + h, где g кратчайшее расстояние до этой вершины и h – эвристика. Также для нее был написан специальный компаратор, который справляется с правильным определением приоритета.

Тестирование.

№	Входные данные	Выходные данные	Результат
1	1 7 1 2 3.0 1 3 1.0 2 4 2.0 2 5 3.0 4 5 4.0 5 1 3.0 5 6 2.0 1 7 8.0 6 7 1.0 3 13 1.0 13 14 1.0	1 7	Правильно
2	1 7 1 2 3.0 1 3 1.0 2 4 2.0 2 5 3.0 4 5 4.0 5 1 1.0 5 6 2.0 1 7 8.0 6 7 1.0	1 7	Правильно
3	1 7 1 2 3.0 1 3 1.0 2 4 2.0 2 5 3.0	1 7	Правильно

	4 5 4.0 5 1 3.0 5 6 2.0 1 7 8.0 6 7 1.0		
4	1 5 1 2 7.0 1 3 3.0 2 3 1.0 3 4 8.0 2 5 4.0	1 2 5	Правильно
5	1 4 1 2 1.0 2 3 1.0 3 1 1.0 1 4 8.0	1 4	Правильно
6	1 4 1 2 1.0 2 3 9.0 3 4 3.0 1 4 9.0 1 5 1.0 5 4 3.0	1 5 4	Правильно
7	1 6 1 3 1.0 1 2 1.0 3 4 2.0 2 5 2.0 4 6 3.0	1 3 4 6	Правильно

	5 6 3.0		
8	1 2 1 2 1.0 1 3 1.0	1 2	Правильно
9	2 5 1 2 1.0 1 3 2.0 2 4 7.0 2 5 8.0 1 7 2.0 2 7 6.0 3 5 4.0 4 5 4.0 7 5 1.0	2 7 5	Правильно
10	-2 4 -2 -1 3.0 -2 0 1.0 -1 1 2.0 -1 2 3.0 1 2 4.0 2 -2 3.0 2 3 2.0 -2 4 8.0 3 4 1.0 0 10 1.0 10 11 1.0	-2 4	Правильно
11	-9 -6 -9 -8 1.0 -8 -7 9.0	-9 -5 -6	Правильно

	-7 -6 3.0 -9 -6 9.0 -9 -5 1.0 -5 -6 3.0		

Выводы.

Был исследован часто используемый на практике алгоритм - поиск пути в графах. Также были получены навыки решения задач на этот алгоритм.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <map>
#include <limits>
#include <queue>
#include <fstream>

//couts
#define DEBUG
//astar algorithm if 1, if 0 -- greedy
#define ASTAR 1
//int -- main task, char -- stepik
#define TYPE int
//1 -- input from file, 0 - input from console
#define FILE_INPUT 1
//for ints
#define TAB



//структура-компаратор, для определения приоритета в очереди
//если 'стоимость' двух вершин одна и та же, то возвращается меньшая в алфавитном
порядке
//если же разная, то большая
template <typename T>
struct CustomCompare{
    bool operator() (std::pair<T, double> a, std::pair<T, double> b){
        return (a.second == b.second) ? (a.first < b.first) : (a.second >
b.second);
    }
};

//класс поиска
template <typename T>
class ShortestPathFinder{
public:
    std::vector<T> greedySearch();
    std::vector<T> aStarSearch();
```

```
    void inputGraph(std::istream& stream);
    void          printQueue(std::priority_queue<std::pair<T,          double>,
std::vector<std::pair<T, double>>, CustomCompare<T>> queue);

private:
    std::map<T, std::vector<std::pair<T, double>>> graph;
    std::map<T, bool> visited;
    T start;
    T end;
    int numberOfEdges;
};


template <typename T>
void ShortestPathFinder<T>::inputGraph(std::istream& stream) {
    T start,end;
    stream >> start >> end;
    this->start = start;
    this->end = end;
    int counter = 0;

    while(stream >> start && start != '#'){
        double weight;
        stream >> end >> weight;
        graph[start].push_back({end,weight});
        visited[start] = false;
        visited[end] = false;
        counter++;
    }
    this->numberOfEdges = counter;
}

template <typename T>
void ShortestPathFinder<T>::printQueue(std::priority_queue<std::pair<T, double>,
std::vector<std::pair<T, double>>, CustomCompare<T>> queue) {
    while(!queue.empty()){
        std::cout << "{ " << queue.top().first << ", " << queue.top().second <<
" } ";
        queue.pop();
    }
}
```

```
template <typename T>
std::vector<T> ShortestPathFinder<T>::aStarSearch() {
    std::map<T, std::pair<std::vector<T>, double>> shortestPaths;
    std::priority_queue<std::pair<T, double>, std::vector<std::pair<T, double>>,
CustomCompare<T>> priorityQueue;

    priorityQueue.push({start, 0});
    std::vector<T> vec;
    vec.push_back(start);
    shortestPaths[start].first = vec;

    while(!priorityQueue.empty()){
#ifdef DEBUG
        std::cout << std::endl << "Состояние очереди: " << std::endl;
        printQueue(priorityQueue);
        std::cout << std::endl;

        std::cout << "Проверяем, является ли приоритетный элемент в очереди
конечным пунктом: ";
#endif

        if(priorityQueue.top().first == end){
#ifdef DEBUG
            std::cout << "Да, является "<< end <<" Заканчиваем поиск." <<
std::endl;
#endif
            return shortestPaths[end].first;
        }
#ifdef DEBUG
        std::cout << "Нет, приоритетный " << priorityQueue.top().first << ", а
конечный " << end << std::endl;
#endif

        auto tempVertex = priorityQueue.top();
#ifdef DEBUG
        std::cout << "Достаем приоритетный элемент из очереди: {" <<
tempVertex.first << ", " << tempVertex.second << "}" << std::endl;
#endif
        priorityQueue.pop();
```

```
#ifdef DEBUG
        std::cout << "Начинаем рассматривать соседей этой вершины: " << std::endl;
#endif
        //проходимся по всем вершинам, которые соединяются с текущей вершиной
        for(auto &item : graph[tempVertex.first]){
            double currentPathLength = shortestPaths[tempVertex.first].second + item.second;
#ifdef DEBUG
            std::cout << "Вершина: {" << item.first << " ," << item.second << " }" << std::endl;
            std::cout << "Текущее расстояние: " << currentPathLength << std::endl;
#endif

            //если раньше пути не существовало или найденный путь лучше, меняем на него
            if(shortestPaths[item.first].second == 0 || shortestPaths[item.first].second > currentPathLength){
                //добавляем в путь родительской вершины еще и текущую вершину, для которой расстояние оказалось меньше
                std::vector<T> path = shortestPaths[tempVertex.first].first;
                path.push_back(item.first);

                //меняем путь и расстояние на новые
                shortestPaths[item.first] = {path,currentPathLength};
#ifdef DEBUG
                std::cout<<"Текущее расстояние либо меньше предыдущего, либо предыдущего не было , записываем его:";
                for(auto &el : path){
                    std::cout << el << " ";
                }
                std::cout << std::endl;
#endif
                //эвристическая функция, для целых чисел и букв одинакова
                double euristic = abs(end - item.first);
                //записываем в очередь текущую вершину и соответствующую ей функцию f = g + h
                priorityQueue.push({item.first, euristic + shortestPaths[item.first].second});

#ifdef DEBUG
```

```
                std::cout << "Записываем в очередь вершину " << item.first << "
с оценочной функцией равной f = g + h = ";
                std::cout << shortestPaths[item.first].second << " + " << euristic
<< " = " << euristic + shortestPaths[item.first].second;
                std::cout << std::endl;
#endif
            }

        }

    }
#ifdef DEBUG
    std::cout << "Очередь опустела, заканчиваем поиск." << std::endl;
#endif

    return shortestPaths[end].first;
}

template <typename T>
std::vector<T> ShortestPathFinder<T>::greedySearch() {
    double min;
    std::vector<T> result;
    result.reserve(this->numberOfEdges);
    //добавляем начальную вершину в стек
    result.push_back(this->start);

    T currentVertex = this->start;

    //проходимся по графу, пока не дойдем до конечной вершины
    while(currentVertex != this->end){
        T nextVertex;
        min = std::numeric_limits<double>::max();
        bool found = false;
#ifdef DEBUG
        std::cout << std::endl << "Состояние стека: ";
        for(auto &el : result){
            std::cout << el << " ";
        }
        std::cout << std::endl;
        std::cout << "Текущая вершина: " << currentVertex << std::endl;
        std::cout << "Начинаем рассматривать ее соседей: " << std::endl;
```

```
#endif

        //рассматриваем соседей текущей вершины
        for(auto &item : this->graph[currentVertex]){
#ifdef  DEBUG
            std::cout << "Смежная вершина: " << item.first << std::endl;
#endif
            //если сосед раньше не посещался и у него минимальный вес среди всех
остальных соседей
            if(!visited[item.first] && item.second < min){
                min = item.second;
                nextVertex = item.first;
                found = true;
#ifdef DEBUG
                std::cout << "Ребро "<<currentVertex << "-" << item.first <<"
имеет пока минимальный вес " << item.second << std::endl;
#endif
            }
#ifdef DEBUG
            else if(visited[item.first]){
                std::cout << "Вершина " << item.first << " уже посещалось" <<
std::endl;
            }
            else {
                std::cout << "Текущий минимальный вес " << min << " меньше веса
" << item.second << " ребра " << currentVertex << "-" << item.first << std::endl;
            }
#endif
        }

        //посетили вершину
        visited[currentVertex] = true;

        //если нет смежных вершин или все они уже были посещены
        if(!found){
            //если стек не пуст, то берем следующий элемент
            if(!result.empty()){
                result.pop_back();
                currentVertex = result.back();
            }
#ifdef DEBUG
```

```
            std::cout << "Не посещенных смежных вершин не было найдено, возвращаемся к прошлой вершине" << std::endl;
#endif
            continue;
        }
#ifdef DEBUG
        std::cout << "Добавляем вершину " << nextVertex << " в стек с минимальным весом " << min << " и рассматриваем в следующий раз" << std::endl;
#endif
        //если же все таки смежную вершину нашли, то в следующий раз начинаем с неё
        currentVertex = nextVertex;
        result.push_back(currentVertex);
    }
#ifdef DEBUG
    std::cout << "Мы дошли до конечной вершины, заканчиваем поиск." << std::endl;
#endif
    return result;
}


int main() {
#ifdef DEBUG
    std::cout << "Введите данные, для окончания ввода #: " << std::endl;
#endif

    std::ifstream file("../test.txt");
    ShortestPathFinder<TYPE> finder;
    finder.inputGraph(FILE_INPUT ? file : std::cin);
    std::vector<TYPE> out = ASTAR ? finder.aStarSearch() : finder.greedySearch();
    for(auto &item : out){
        std::cout << item;
#ifdef TAB
        std::cout << " ";
#endif
    }

    return 0;
}
```