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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Построение и анализ алгоритмов»

Тема: Жадный алгоритм и A*

Студент гр. 1304 _____ Стародубов М.В.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург

2023

Цель работы.

Реализовать программы, находящие путь в ориентированом графе с помощью жадного алгоритма и алгоритма A*.

Задание.

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
```

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
```

Выполнение работы.

1. Жадный алгоритм.

Исходный код программы находится в приложении А, название файла: *greedy.cpp*.

Для хранения графа используется ассоциативный массив, ключами данного массива являются символы, обозначающие вершины графа, значениями каждого ключа является набор вершин, в которые можно попасть из вершины, являющейся ключем, вместе с каждой вершиной хранится вес ребра, соединяющего данные вершины. Данный набор хранится как очередь с приоритетом, в начале очереди находится вершина с наименьшей стоимостью дуги, соединяющей вершины.

Класс *Solution* инкапсулирует методы, решающие задачу.

Метод *get_graph* производит считывание графа из входного потока программы и возвращает его в виде описанной выше структуры данных.

Метод *present_result* получает на вход стек и возвращает его содеримое в виде строки так, что самый верхний символ, хранящийся в стеке, находится в конце строки.

Метод *find_path* производит построение пути от начальной до конечной вершины в графе с помощью жадного алгоритма. В качестве входных параметров метод принимает начальную и конечную вершины, а также граф. Метод возвращает построенный путь в виде строки.

Построение пути происходит с помощью поиска в ширину. В начале выполнения метода в стек заносится начальная вершина. Далее в цикле просматривается вершина на верху стека, если данная вершина является конечной вершиной, то поиск прекращается. Если из даной вершины невозможно пройти ни в одну другую вершину, или в ходе работы алгоритма был совершен проход по всем дугам, исходящим из данной вершины, то вершина извлекается из стека и начинается следующая итерация. Во всех остальных случаях в стек заносится вершина с наименьшей стоимостью дуги, дуга помечается как пройденная.

В итоге работы цикла все вершины, составляющие искомый путь (если он существует) будут находиться в стеке. Данный путь конвертируется в строку с помощью метода *present_result*.

2. Алгоритм A*.

Исходный код программы находится в приложении А, название файла: *a_star.cpp*.

Для хранения графа используется ассоциативный массив, ключами данного массива являются символы, обозначающие вершины графа, значениями каждого ключа является набор вершин, в которые можно попасть из вершины, являющейся ключем, вместе с каждой вершиной хранится вес ребра, соединяющего данные вершины.

Класс *Solution* инкапсулирует методы, решающие задачу.

Метод *get_graph* производит считывание графа из входного потока программы и возвращает его в виде описанной выше структуры данных.

Метод *present_result* получает на вход конечную вершину и ассоциативный массив, ключами являются вершины, а значениями вершины, из которых был совершен переход в вершину, являющуюся ключем. Данный метод представляет путь до конечной вершины в виде строки.

Метод *find_path* производит построение пути от начальной до конечной вершины в графе с помощью алгоритма A*. Обход графа происходит с помощью очереди с приоритетом, в начале очереди находится вершина с минимальной оценкой пути через нее, оценка вычисляется по следующей формуле:

$f = g + h$,

где g — длина пройденного пути для достижения данной вершины, h — эвристическая оценка пути до конечной вершины, которя по заданию вычисляется как разница между численным значением символов, обозначающих конечную и данную вершину. В очереди вершины хранятся как кортежи из трех значений: текущая вершина, расстояние, пройденное до данной вершины, вершина, из которой был совершен переход в данную вершину. Для определения пути после работы алгоритма используется ассоциативный массив *path*, ключами которого являются вершины, а значениями вершины, из которых был совершен переход в вершину, являющуюся ключем. Для определения минимального расстояния до вершины из начальной вершины используется ассоциативный массив *best_distance*. В начале выполнения алгоритма в очередь заносится начальная вершина. Далее происходит выполнение цикла, на каждой итерации которого из очереди извлекается и просматривается вершина. Если вершина еще не была обработана или расстояние до нее меньше, чем расстояние, записанное в ассоциативном массиве *best_distance*, то происходит обработка вершины. Обновляются значения в ассоциативных массивах *path* и *best_distance*, если вершина является конечной, то цикл завершает работу, иначе в очередь заносятся все верины, соседствующие с текущей.

Путь, построенный в итоге работы цикла можно восстановить с помощью ассоциативного массива *path*. Чтобы представить данный путь в виде строки используется метод *present_result.*

Выводы.

В ходе выполнения работы реализованы программы, реализующие алгоритмы нахождения пути в ориентированном графе. Для выполнения данной задачи программы используют жадный алгоритм и алгоритм A*. Жадный алгоритм построен с использованием обхода графа в глубину. Для реализации алгоритма A* использована очередь с приоритетом. В обоих алгоритмах хранение графа происходит с помощью ассоциативного массива.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: greedy.cpp

```
#include <iostream>
#include <functional>
#include <vector>
#include <queue>
#include <map>
#include <string>
#include <stack>


/*
 * A priority queue is used to store nodes that can be jumped to from the current node,
 * at the top is the element with the lowest edge weight.
 * The queue stores elements as value pairs node name - edge weight.
 */
typedef std::priority_queue<std::pair<char, float>, std::vector<std::pair<char, float>>,
                            std::function<bool(std::pair<char, float> &, std::pair<char, float> &)>> nodes_queue;
/*
 * The graph is represented as an associative array, the keys are the names of the nodes, the values are the
 * queues of the nodes in which you can go to from key node.
 */
typedef std::unordered_map<char, nodes_queue> graph_t;


/*
 * Class Solver is a class that encapsulates the methods needed to solve the problem.
 */
class Solution
{
public:
    graph_t get_graph();
        std::string find_path(char start_node, char finish_node, graph_t &graph);
private:
    std::string present_result(std::stack<char> &stack);
};


/*
 * Reads a graph from the input stream and returns it.
 */
graph_t Solution::get_graph()
{
      auto cmp = [](std::pair<char, float> &left, std::pair<char, float> &right)
```

```
        { return left.second > right.second; };
        graph_t graph;
        char source_node, destination_node;
        float weight;

        while (std::cin >> source_node >> destination_node >> weight)
        {
            if (graph.find(source_node) == graph.end())
            {
                graph[source_node] = nodes_queue(cmp);
            }
            graph[source_node].emplace(destination_node, weight);
        }

        return graph;
    }

    /*
     * Finds a path using greedy algorithm.
     */
    std::string Solution::find_path(char start_node, char finish_node,
graph_t &graph)
    {
        std::stack<char> stack;
        stack.push(start_node);

        while (!stack.empty())
        {
            char current_node = stack.top();

            if (current_node == finish_node)
            {
                break;
            }

            if (graph[current_node].empty())
            {
                stack.pop();
                continue;
            }

            stack.push(graph[current_node].top().first);
            graph[current_node].pop();
        }

        return present_result(stack);
    }

    /*
     * Represents the contents of the passed stack as a string,
     * with the top element of the stack at the end of the string.
     */
    std::string Solution::present_result(std::stack<char> &stack)
    {
        std::string result;

        while (!stack.empty())
        {
```

```
            result.insert(0, 1, stack.top());
            stack.pop();
        }

        return result;
    }


    int main()
    {
        char start_node, finish_node;
        std::cin >> start_node >> finish_node;

        Solution s;
        graph_t graph = s.get_graph();
         std::cout << s.find_path(start_node, finish_node, graph) << '\
n';

        return 0;
    }
```

Название файла: a_star.cpp

```
    #include <iostream>
    #include <functional>
    #include <vector>
    #include <queue>
    #include <map>
    #include <string>
    #include <cmath>


    /*
     * A priority queue is used to traverse the graph,
     * at the top is the element with the smallest path estimate f = g
+ h,
     * where g -- actual distance traveled to reach a given node,
      *  h  --  heuristic  evaluation,  equal  to  the  difference  in  the
numerical
     * values of the symbols denoting the nodes.
      * Queue elements are tuples consisting of the name of the node,
the distance traveled to this node,
     * the name of the node from which the transition to this node was
made.
     */
    typedef    std::priority_queue<std::tuple<char,    float,    char>,
std::vector<std::tuple<char, float, char>>,
                    std::function<bool(std::tuple<char,  float,  char>  &,
std::tuple<char, float, char> &)>> queue_t;
    /*
     * Stores pairs of elements -- the name of the node and the weight
of the edge to the this node.
     */
    typedef std::vector<std::pair<char, float>> nodes_container;
    /*
     * The graph is represented as an associative array, the keys are
the names of the nodes, the values are the
     * containers of the nodes in which you can go to from key node.
```

```
     */
    typedef std::unordered_map<char, nodes_container> graph_t;


    /*
     * Class Solver is a class that encapsulates the methods needed to
solve the problem.
     */
    class Solution
    {
    public:
        graph_t get_graph();
            std::string  find_path(char  start_node,  char  finish_node,
graph_t &graph);
    private:
                        std::string    present_result(char    finish_node,
std::unordered_map<char, char> &path);
    };


    /*
     * Reads a graph from the input stream and returns it.
     */
    graph_t Solution::get_graph()
    {
        graph_t graph;
        char source_node, destination_node;
        float weight;

        while (std::cin >> source_node >> destination_node >> weight)
        {
            if (graph.find(source_node) == graph.end())
            {
                graph[source_node] = nodes_container();
            }
            graph[source_node].emplace_back(destination_node, weight);
        }

        return graph;
    }

    /*
     * Finds a path using A* algorithm.
     */
    std::string Solution::find_path(char start_node, char finish_node,
graph_t &graph)
    {
         auto cmp = [&finish_node](std::tuple<char, float, char> &left,
std::tuple<char, float, char> &right)
        {
                 float  left_heuristics  =  std::get<1>(left)  +  (float)
std::abs(finish_node - std::get<0>(left));
                float  right_heuristics  =  std::get<1>(right)  +  (float)
std::abs(finish_node - std::get<0>(right));
            return left_heuristics > right_heuristics;
        };
        queue_t queue(cmp);
        std::unordered_map<char, char> path;
```

```
        std::unordered_map<char, float> best_distance;

        queue.emplace(start_node, 0, 0);

        while (!queue.empty())
        {
            auto current_item = queue.top();
            char current_node = std::get<0>(current_item);
            float distance = std::get<1>(current_item);
            char previous_node = std::get<2>(current_item);
            queue.pop();

                        if (path.find(current_node) != path.end() &&
best_distance[current_node] <= distance)
            {
                continue;
            }

            path[current_node] = previous_node;
            best_distance[current_node] = distance;

            if (current_node == finish_node)
            {
                break;
            }

            for (auto neighbor_node: graph[current_node])
            {
                char next_node = neighbor_node.first;
                                float next_node_distance = distance +
neighbor_node.second;
                            queue.emplace(next_node, next_node_distance,
current_node);
            }

        }

        return present_result(finish_node, path);
    }

    /*
     * Represents the path to the finish node as a string, using an
associative array,
     * the keys are the names of nodes, values are the nodes from which
the transition to key node was made.
     */
    std::string       Solution::present_result(char       finish_node,
std::unordered_map<char, char> &path)
    {
        std::string result;
        char current_node = finish_node;

        while (current_node)
        {
            result.insert(0, 1, current_node);
            current_node = path[current_node];
        }
```

```
        return result;
    }


    int main()
    {
        char start_node, finish_node;
        std::cin >> start_node >> finish_node;

        Solution s;
        graph_t graph = s.get_graph();
         std::cout << s.find_path(start_node, finish_node, graph) << '\
n';

        return 0;
    }
```