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: Кратчайшие пути. Жадный алгоритм и A.

Студент гр. 1384 _____ Тапеха В.А.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург
2023

Цель работы.

Изучить принцип работы алгоритмов поиска кратчайшего пути. Написать две программы, которая решает задачу построения пути в ориентированном графе при помощи жадного алгоритма и алгоритма A*.

Задание 1.

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

Задание 2.

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.

При выполнение работы была создана структура *Node*, хранящая точку и стоимость пути до нее. Для нее был перегружен оператор равенства, который необходим для корректной работы *std::unordered_map*. Еще был определен *std::hash* для *Node*, который также нужен для корректной работы с *std::unordered_map*. При реализации последнего, была использована функция *std::hash* для типов *char* и *float*.

Была написана функция *graph_type read_graph*, считывающая граф. Этот метод считывает данные со стандартного ввода и создает структуру данных графа. Граф представляет *std::unordered_map*, где ключом является некоторой вершина, а значением список смежных вершин и стоимость пути до них от искомой вершины.

Для решения задач были написаны классы *Greedy* (класс, реализующий решение жадного алгоритма) и *AStar* (класс, реализующий алгоритм A*).

Greedy:

В нем хранятся конечная вершина, структура, хранящая граф, и вектор, который хранит ответ.

Внутри класса были созданы методы:

void sort_graph() – метод, который сортирует рёбра графа по возрастанию их длины.

void greedy_algorithm(char current) – жадный алгоритм, решающий задачу.

void print_answer() – метод, печатающий ответ.

Подробнее рассмотрим метод *void greedy_algorithm(char current)*.

Этот метод реализует рекурсивный алгоритм, решающий задачу.

По достижении конечной вершины алгоритм завершает работу. Заранее все вершины графа были отсортиованы по возрастанию. Если текущая вершина была найдена в графе, то просматриваются последовательно по воз-

растанию длины, смежные вершины. Иначе ветка решений обрывается и последняя вершина удаляется из массива, содержащего решение задачи.

AStar:

В нем хранятся начальная и конечная вершины, структура, хранящая граф, и список родителей для каждой вершины, включенной в итоговое решение.

У него были созданы методы:

char min_heuristic_plus_cost(const std::vector<char>& array) – метод, находящий минимальное значение эвристической функции "расстояние + стоимость" для некоторой вершины;

[[nodiscard]] inline float heuristic(char current) const – метод, реализующий эвристическую функцию, оценивающую расстояние от некоторой вершины до конечной вершины;

void print_answer() – метод, печатающий ответ;

bool A_star() – метод реализующий алгоритм A*.

Подробнее рассмотрим метод *bool A_star()*.

Этот метод реализует алгоритм A* для поиска самого короткого пути до заданной вершины.

Алгоритм A* находит самый короткий путь до некоторой вершины, используя эвристическую функцию, которая оценивает оставшееся расстояние от каждой вершины до конечной вершины. Он поддерживает два списка вершин: список обработанных вершин и список вершин, которые нужно обработать. Алгоритм выбирает следующую вершину для обработки из списка вершин, которые еще необходимо обработать, на основе суммы стоимости достижения этой вершины из начальной вершины и предполагаемого оставшегося расстояния до конечной вершины, которое было оценено с помощью эвристической функции. Он продолжается до тех пор, пока не будет достигнута конечная вершина или не останется вершин, которые нужно обра-

ботать. Возвращает буллевое значение, которая показывает была ли найдена нужная вершина.

Выводы.

В ходе данной лабораторной работы были изучены принципы алгоритмов поиска кратчайшего пути в графе. По и

Были реализованы жадный алгоритм, жадность которого заключается в том, что на каждом шаге выбирается последняя посещенная вершина, и алгоритм A* для поиска наименьшего по стоимости пути в ориентированном графе. Жадный алгоритм был реализован рекурсивно, а алгоритм A* был стандартно реализован с использованием в качестве эвристической функции расстояние между символами в таблице ASCII. Обе программы успешно прошли все тесты на платформе «Stepik».

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: greedy.cpp

```
#include <iostream>
#include <utility>
#include <vector>
#include <unordered_map>

// Предварительно объявлен класс Node, чтобы использовать его в краткой записи с помощью using
class Node;

using Nodes = std::vector<Node>;
using graph_type = std::unordered_map<char, Nodes>;


// Структура, хранящая точку и стоимость пути до нее
struct Node {
    char point; // Точка
    float cost; // Стоимость пути до нее

    // Конструкторы
       explicit Node(char point = 'a', float cost = 0.0) : point(point), cost(cost) {}
        explicit Node(const std::pair<char, float>& node) : point(node.first), cost(node.second) {}

     /* Перегружен оператор равенства, который необходим для корректной работы
      * std::unordered_map с пользовательской структурой */
      bool operator==(const Node& other) const { return point == other.point && cost == other.cost; }
};

// Определен std::hash для Node, который нужен для корректной работы с std::unordered_map
template <>
struct std::hash<Node>
{
    std::size_t operator()(const Node& node) const
    {
        // Хэш-функция, преобразовывающая поля структуры
                    return ((std::hash<char>()(node.point) ^ (std::hash<float>()(node.cost) << 1)) >> 1);
    }
};


/* Функция, считывающая граф.
 * Переменная from показывает из какой вершины идет путь.
 * Переменная to показывает в какую вершину направлен путь. */
graph_type read_graph() {
```

```
        graph_type graph;
        char from, to;
        float length;

        while (std::cin >> from >> to >> length) {
            graph[from].emplace_back(to, length);
        }

        return graph;
    }


    // Класс, реализующий решение жадного алгоритма
    class Greedy {
    public:
        // Конструктор
        Greedy(char start, char goal, graph_type graph)
                                :  goal(goal),  graph(std::move(graph))
{ answer.push_back(start); }

        // Сортирует считанный граф
        void sort_graph();
        // Жадный алгоритм, решающий задачу
        void greedy_algorithm(char current);
        // Печатает ответ
        void print_answer();
    private:
        char goal; // конечная вершина
        graph_type graph; // структура, хранящая граф
        std::vector<char> answer; // вектор, хранящий ответ
    };

    // Сортирует рёбра графа по возрастанию их длины
    void Greedy::sort_graph() {
        for(auto& i : graph) {
            std::sort(i.second.begin(), i.second.end(),
                      [](const Node& edge1, const Node& edge2) {
                          return edge1.cost < edge2.cost;
                      });
        }
    }

    /**
     * Рекурсивный алгоритм, решающий задачу.
     * По достижении конечной вершины алгоритм завершает работу.
     * Заранее все вершины графа были отсортиованы по возрастанию.
     * Если текущая вершина была найдена в графе, то просматриваются
последовательно по возрастанию длины,
     * смежные вершины. Иначе ветка решений обрывается и последняя
вершина удаляется из массива,
     * содержащего решение задачи.
     */
    void Greedy::greedy_algorithm(char current) {
        if (current == goal) {
            return;
```

```
        }

        auto iter = graph.find(current);
        if (iter == graph.end()) {
            answer.pop_back();
            return;
        }

        for(const auto& elem : iter->second) {
            answer.push_back(elem.point);
            iter->second.pop_back();
            greedy_algorithm(elem.point);

            if (answer.back() == goal) {
                return;
            }
        }
        answer.pop_back();
    }

    // Печатает ответ на задачу
    void Greedy::print_answer() {
        for (char i : answer) {
            std::cout << i;
        }
    }


    int main() {
        char start, final;
        std::cin >> start >> final;

        Greedy solver(start, final, read_graph());
        solver.sort_graph();
        solver.greedy_algorithm(start);

        solver.print_answer();


        return 0;
    }
```

Название файла: astar.cpp

```
    #include <iostream>
    #include <utility>
    #include <vector>
    #include <unordered_map>
    #include <numeric>


    // Предварительно объявлен класс Node, чтобы использовать его в
краткой записи с помощью using
    class Node;
```

```
using Nodes = std::vector<Node>;
using graph_type = std::unordered_map<char, Nodes>;


// Структура, хранящая точку и стоимость пути до нее
struct Node {
    char point; // Точка
    float cost; // Стоимость пути до нее

    // Конструкторы
       explicit Node(char point = 'a', float cost = 0.0) :
point(point), cost(cost) {}
        explicit Node(const std::pair<char, float>& node) :
point(node.first), cost(node.second) {}

     /* Перегружен оператор равенства, который необходим для кор-
ректной работы
      * std::unordered_map с пользовательской структурой */
      bool operator==(const Node& other) const { return point ==
other.point && cost == other.cost; }
};

// Определен std::hash для Node, который нужен для корректной ра-
боты с std::unordered_map
template <>
struct std::hash<Node>
{
    std::size_t operator()(const Node& node) const
    {
        // Хэш-функция, преобразовывающая поля структуры
                    return   ((std::hash<char>()(node.point)   ^
(std::hash<float>()(node.cost) << 1)) >> 1);
    }
};


/* Функция, считывающая граф.
 * Переменная from показывает из какой вершины идет путь.
 * Переменная to показывает в какую вершину направлен путь. */
graph_type read_graph() {
    graph_type graph;
    char from, to;
    float length;

    while (std::cin >> from >> to >> length) {
        graph[from].emplace_back(to, length);
    }

    return graph;
}

// Класс, реализующий алгоритм A*
class AStar {
public:
```

```
        // Конструктор
        explicit AStar(char start, char goal, graph_type graph = {})
        : goal(goal), start(start), graph(std::move(graph)) {}

        // Метод, реализующий алгоритм A*
        bool A_star();
        // Печатает ответ
        void print_answer();
    private:
        // Эвристическая функция, оценивающая расстояние от вершины x до конечной вершины
        [[nodiscard]] inline float heuristic(char current) const;
            // Находит минимальное значение эвристической функции "расстояние + стоимость" для вершины x
        char min_heuristic_plus_cost(const std::vector<char>& array);

         std::unordered_map<char, float> heuristic_plus_cost; // хранит значения функции "расстояние + стоимость" для всех x
        char start, goal; // начальная и конечные вершины
        graph_type graph; // структура, хранящая граф
         std::unordered_map<char, char> parents; // хранит родителей вершин, которые включены в итоговое решение
    };


    /** Этот метод реализует алгоритм A* для поиска кратчайшего пути до заданной вершины.
       * Алгоритм A* находит кратчайший путь до нужно вершины, используя эвристическую функцию,
       * которая оценивает оставшееся расстояние от каждой вершины до конечной вершины.
       * Алгоритм поддерживает два списка вершин: список обработанных вершин
       * и список вершин, которые нужно обработать. Алгоритм выбирает следующую вершину
       * для обработки из списка вершин, которые еще необходимо обработать,
       * на основе суммы стоимости достижения этой вершины из начальной вершины
       * и предполагаемого оставшегося расстояния до конечной вершины,
       * которое было оценено с помощью эвристической функции. Алгоритм продолжается
       * до тех пор, пока не будет достигнута конечная вершина или не останется вершин,
       * которые нужно обработать.
       * @return буллевое значение, которая показывает была ли найдена нужная вершина
       */
    bool AStar::A_star() {
        std::vector<char> used, need_to_check;
        need_to_check.push_back(start);

        std::unordered_map<char, float> cost_to_final;
        cost_to_final[start] = 0;
```

```
                heuristic_plus_cost[start]   =   cost_to_final[start]   +
heuristic(start);

          while (!need_to_check.empty()) {
              char current = min_heuristic_plus_cost(need_to_check);
              if (current == goal) {
                  return true;
              }

                 need_to_check.erase(std::remove(need_to_check.begin(),
need_to_check.end(), current),
                                     need_to_check.end());
              used.push_back(current);

              for (auto v : graph[current]) {
                      float tentative_score = cost_to_final[current] +
v.cost;
                   if ((std::find(used.begin(), used.end(), v.point) ==
used.end() && *used.end() != v.point)
                      || tentative_score < cost_to_final[v.point])  {
                      used.push_back(v.point);
                      parents[v.point] = current;
                      cost_to_final[v.point] = tentative_score;
                                          heuristic_plus_cost[v.point]  =
cost_to_final[v.point] + heuristic(v.point);
                                    if  (std::find(need_to_check.begin(),
need_to_check.end(), v.point) == need_to_check.end()
                          && *need_to_check.end() != v.point) {
                          need_to_check.push_back(v.point);
                      }
                  }
              }
          }
          return false;
      }

      // Выводит значения из структуры, хранящей родителей вершин,
которые включены в итоговое решение.
      void AStar::print_answer() {
          std::vector<char> result;
          char current = goal;

          while (parents[current] != current) {
              result.insert(result.begin(), current);
              current = parents[current];
          }

          for (auto element : result) {
              std::cout << element;
          }
      }

      // Эвристическая функция
      float AStar::heuristic(char current) const {
          return static_cast<float>(goal - current);
```

```
    }

    // Нахождение минимальной вершины по оценке стоимость + эвристика
    char    AStar::min_heuristic_plus_cost(const    std::vector<char>
&array) {
        float min = std::numeric_limits<float>::infinity();
        char result;

        for (auto element : array) {
            min = std::min(min, heuristic_plus_cost[element]);
             result = heuristic_plus_cost[element] == min ? element :
result;
        }

        return result;
    }


    int main() {
        char start, final;

        std::cin >> start >> final;

        AStar a_star(start, final, read_graph());

        a_star.A_star();
        a_star.print_answer();


        return 0;
    }
```