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

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Построение и анализ алгоритмов»

Тема: Алгоритм A*

Студентка гр. 7383 _____ Чемова К.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

Цель работы.

Исследовать и реализовывать задачу построения кратчайшего пути в ориентированном графе, используя алгоритм A*.

Формулировка задачи.

Необходимо разработать программу, которая решает задачу построения кратчайшего пути в ориентированном графе методом A* до любой из представленных вершин. Каждая вершина в графе имеет буквенное обозначение («a», «b», «c» …), каждое ребро имеет неотрицательный вес.

Вариант 2м: граф представлен в виде матрицы смежности, эвристическая функция для каждой вершины задаётся положительным числом во входных данных.

Входные данные: в первой строке указываются начальная и конечная вершины. Затем количество вершин и их эвристические значения. Далее идут данные о рёбрах графа и их весе.

Выходные данные: кратчайший путь из стартовой вершины в конечную.

Выполнение работы.

В данной работе используются главная функция `main()` и структуры данных `class A_star_algorithm` и `class A_star`, `struct Priority`. `A_star` отвечает за хранение графа в виде матрицы смежности, а так же содержит функцию `algorithm`, которая осуществляет поиск кратчайшего пути из начальной вершины в конечную, используя A*. Структура используется для хранения данных в очереди с приоритетами.

В функции `main()` считываются начальная и конечная вершины, между которыми нужно найти кратчайший путь. Затем записываются в массив значения эвристики для каждой из вершин графа. Далее в цикле начинается считывание рёбер графа и их вес. Запускается алгоритм A*, который рассматривает всех соседей текущей вершины, помещает пути до них в очередь, в зависимости от значения эвристической функции, и находит

кратчайший путь для вершины, которая на данном шаге находится ближе. Пока очередь не пуста алгоритм выполняет свою работу.

Исходный код программы представлен в приложении Б.

Тестирование.

Программа собрана в операционной системе Ubuntu 16.04 с использованием компилятора g++. В других ОС и компиляторах тестирование не проводилось.

Результаты тестирования показали, что поставленная цель выполнена. Результаты тестирования представлены в Приложении А.

Исследование алгоритма.

Сложность алгоритма составляет O(|V|•|E|), где |V| – количество вершин в графе, |E| – количество ребер в графе. На каждом шаге работы программы просматриваются всевозможные пути из искомой вершины. В худшем случае могут быть просмотрены все пути данного графа. Тогда сложность зависит от количества вершин.

Выводы.

В ходе лабораторной работы был изучен алгоритм поиска кратчайшего пути A*. Был написан код на языке программирования C++, который применял этот метод для поставленной задачи. Сложность реализованного алгоритма составляет O(|V|•|E|).

ПРИЛОЖЕНИЕ А
ТЕСТОВЫЕ СЛУЧАИ

Входные данные	Результат
b g 5 b 3.0 c 1.0 d 7.0 e 2.0 g 8.0 b c 3.0 c d 1.0 d e 1.0 b e 5.0 e g 1.0	beg
a e 5 a 3 b 1 c 9 d 1 e 4 a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade

ПРИЛОЖЕНИЕ Б
ИСХОДНЫЙ КОД ПОИСКА ПОДСТРОКИ

```
#include <iostream>
#include <cstring>
#include <climits>
#include <queue>

#define N 26

typedef struct Priority {
  std::vector <int> path;
  double prior_vertex;
  double character;
} Priority;

bool operator < (const Priority &comp_var_1, const Priority
&comp_var_2) {
  return comp_var_1.prior_vertex > comp_var_2.prior_vertex;
}

class A_star{
private:
  double graph[N][N];
  std::vector <int> result;
  std::vector <int> str;
public:
  A_star(int start) {
    result.push_back(start);
    for (int i = 0; i < N; i++)
      memset(graph[i], 0.0, N * sizeof(double));
  }

  ~A_star() {
    result.clear();
    str.clear();
  }

  void insert_paths(int i, int j, double path_length) { //заполнение
длин путей графа
    graph[i][j] = path_length;
  }

  void print() { //вывод результата
    if (result.size() - 1) {
      for (int i = 0; i < result.size(); i++)
        std::cout << (char)(result[i] + 'a');
        std::cout << std:: endl;
      }
```

```
    else std::cout << "Пути не существует!" << std::endl;
  }

  void algorithm(int first, int finish, std:: priority_queue
<Priority> &queue, double common_way, std:: vector <double> &evristic)
{
    std::vector <int> check(N, 0);
    int run = 1;
    while (run) {
      for (int j = 0; j < N; j++) //проходим по всем вершинам
            if (graph[first][j] != 0 && check[j] == 0){    //если
соседи, т.е существует ребро
                  check[j] = 1;
            Priority new_elem;
            new_elem.character = evristic[j];
            new_elem.prior_vertex = graph[first][j] + common_way +
new_elem.character;
            for (int i = 0; i < str.size(); i++)
              new_elem.path.push_back(str[i]);
            new_elem.path.push_back(j);
            queue.push(new_elem);
          }
      if (queue.empty())
        run = 0;
      if (!queue.empty()) {
        Priority temp;
        temp = queue.top();
        queue.pop();
        first = temp.path[temp.path.size()-1];
        str = temp.path;
        common_way = temp.prior_vertex - temp.character;
      }
      if (str[str.size() - 1] == finish) {
        for (int i = 0; i < str.size(); i++)
          result.push_back(str[i]);
        run = 0;
      }
    }
  }
};

int main() {
  char start, finish, from, to, curr;
  double path_length, evristic;
  int count = 0;
  std::vector <double> character(N, 0);
  std::cin >> start >> finish;
  std::cin >> count;
  for (int i = 0; i < count; i++) {
    std::cin >> curr >> evristic;
    if (evristic >= 0.0)
```

```
      character[curr - 'a'] = evristic;
    else {
      std::cout << "Введены неверные значения!" << std::endl;
      return 0;
    }
  }
  std::priority_queue <Priority> queue;
  A_star graph(start - 'a');
  while (std::cin >> from >> to >> path_length)
    graph.insert_paths(from - 'a', to - 'a', path_length);
  graph.algorithm(start - 'a', finish - 'a', queue, 0.0, character);
  graph.print();
  character.clear();
  return 0;
}
```