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: Алгоритмы поиска пути в графах

Студент гр. 8383 _____ Федоров И.А

Преподаватель _____ Ерёменко А.А.

Санкт-Петербург

2020

Цель работы.

Ознакомиться с жадным алгоритмом поиска пути в графе и алгоритмом A^*. Реализовать алгоритм Дейкстры поиска пути в графе на основе кода 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
```

Вариант №5: Реализовать алгоритм Дейкстры поиска пути в графе (на основе кода A^*).

Описание структур данных хранения графа (для жадного алгоритма)

```
struct Edge{                        // ребро
  char startVertex;                 // начальная и
  char endVertex;                   // конечная вершины
  double weight;                    // вес
  bool notTouch = true;             // проходили ли через
  Edge(char,char, double);          // это ребро
};
```

Структура `Edge` - структура для хранения ребра графа. Хранит в себе начальную и конечные вершины, вес, а так же логическую переменную, показывающую, проходил ли алгоритм через эту вершину.

```
class EdgeList{                     //"список ребер"
  friend void dialog();
  private:
    vector<Edge*> list;
    int countEdge;                  // количество ребер графа
  public:
```

```
    EdgeList();
    void addEdge(Edge*);              // методы добавления
    void addEdge(char, char, double); //ребра
};
```

Граф хранится с помощью списка ребер. Структура `EdgeList` хранит в себе соответственно вектор ребер графа, количество ребер, а так же имеет методы добавления нового ребра в список.

Описание жадного алгоритма поиска.

Жадный алгоритм на каждом шаге делает локально наилучший выбор в надежде, что итоговое решение будет оптимальным. В данном случае перемещение происходит в ту вершину, путь до которой является самым дешёвым из последней посещённой вершины.

В реализации алгоритма вначале заводится стек пройденных вершин, который необходим для возврата к предыдущей вершине. Главный цикл алгоритма работает до тех пор, пока не будет достигнута конечная вершина.

Для текущей вершины просматриваются ее соседи, среди которых выбирается тот, до которого ребро из текущей вершины короче. Уже просмотренные ребра игнорируются. Если же не найдена смежная вершина (либо из текущей вершины больше нет ребер или все они просмотрены), то из стека достается предыдущая вершина (происходит откат назад) с которой будет продолжена работа. Если же стек пуст, то значит пути нет и алгоритм прекращает работу.

Сложность алгоритма в данной реализации будет $O(V^2)$, $|V|$ - число ребер, т.к. главный цикл в худшем случае проходит все ребра, а для просмотра соседей алгоритм так же проходится по списку ребер.

Граф хранится как список ребер, соответственно сложность по памяти будет $O(|V|)$.

Описание структур данных хранения графа.

```
struct Edge{                        //структура "ребра"
  char startVertex;                 //начало и конец ребра
  char endVertex;
  double distant;                   //вес ребра
  Edge(char,char, double);
};
```

Структура `Edge` - структура для хранения одного ребра ориентированного графа. Хранит в себе следующие поля:

- `startVertex` - начальная вершина ребра.
- `endVertex` - конечная вершина ребра.
- `distant` - вес ребра.

```
using EdgeList = std::vector <Edge>;   //вектор смежных ребер

struct AdjacencyList{          //структура "Список инцидентности"
    std::map<char, EdgeList> adjList;
    int countEdge;                      //число ребер в списке
    Adjacencylist();
    int size();
    void addEdge(char, char, double);  //метод добавления ребра в список
};
```

Сам граф хранится с помощью списка смежности (инцидентности). Структура `AdjacencyList` хранится в себе следующие поля и методы:

- `adjList` - список смежности, заданный с помощью `std::map`. Первый элемент - вершина графа, а второй соответственно вектор всех смежных с этой вершиной ребер.
- `countEdge` - количество ребер в графе.
- `size()` - размер списка смежности.
- метод `addEdge()` - метод добавления ребра в список смежности. Получает на вход начальную, конечную вершины ребра и его вес. После этого либо добавляет ребро к уже существующему списку смежных ребер для начальной вершины, либо создает новую пару вершина-вектор.

  Описание алгоритма A*.

  `bool Dialog::algorithm_A_star(char start, char goal);`

Метод, реализующий алгоритм A* (класс `Dialog` рассмотрен ниже, необходим лишь для диалога с пользователем), принимает на вход начальную и конечную вершину пути, а так же исходный граф (хранится в поле класса `Dialog`). Возвращаемое логическое значение зависит от того, найден ли путь или нет.

В начале создаются несколько необходимых для работы множеств:

1) `openSet` - множество вершин, которые предстоит обработать (открытый список). В начале здесь находится только стартовая вершина.
2) `finishedSet` - множество вершин, которые уже обработаны (закрытый список).
3) `fromStart` - список значений стоимости пути от начальной вершины, до текущей. Реализован с помощью `std::map` (вершина - значение).
4) `function` - список значений стоимости вместе с эвристической функцией. Реализован с помощью `std::map` (вершина - значение).

Главный цикл алгоритма работает до тех пор, пока открытый список не станет пустым. По условию в качестве эвристической функции взята близость символов, обозначающих вершины графа, в таблице ASCII.

На каждом шаге алгоритм достает из открытого списка вершину, имеющую самую низкую оценку стоимости, и добавляет ее список обработанных вершин. Затем рассматриваются все смежные с текущей вершины, и выполняется следующий ряд действий:

- Если вершина (сосед) находится в закрытом списке, то она пропускает.
- Вычисляется стоимость пути от начальной до текущей.
- Если вершина не содержится в открытом списке, то она добавляется туда, и в `fromStart` и `function` записываются свойства этой вершины.

- Если же вершина уже находится в открытом списке (а это значит что для данного соседа уже известны стоимость пути и значение эвристической функции), то, если вычисленная стоимость оказалась меньше известной (содержится в `fromStart`), то обновляются значения для данной вершины. Иначе существует более дешевый маршрут через этого соседа и он игнорируется.

Алгоритм работает до тех пор, пока открытый список не будет пустым (то есть рассматривать больше нечего, в таком случае путь не найден), либо же пока не будет достигнута целевая вершина.

Для восстановления пути используется вспомогательная функция

```
void Dialog::reconstructPath(char start, char goal);
```

которая добавляет в результирующий список все вершины от конечной до начальной, после чего данный список инвертируется и выводится.

От "жадного алгоритма" алгоритм A* отличается тем, что при выборе вершины он учитывает, помимо прочего, весь пройденный до неё путь (берется стоимости пути от начальной, а не от предыдущей, как в "жадном"). Кроме того приоритет пути определяется по значению f(x) = g(x) + h(x), где h(x) - эвристическая оценка расстояния от вершины *x*, до конечной. Благодаря этому сначала просматривается меньше вершин и дуг.

Оценка сложности алгоритма A*.

Сложность работы функции нахождения вершины из открытого списка с наименьшей оценкой стоимости *O(V)*. Сложность проверки наличия вершины в закрытом/открытом списке так же *O(V)*. Обработка смежных вершин выполняется со сложностью $O(E)\cdot(O(V)+O(V))$, т.к. $|E|$ - количество смежных вершин, внутри цикла проводятся проверка на наличие в закрытом и открытом списках. Алгоритм посещает в худшем случае $|V|$ вершин, поэтому общая сложность алгоритма будет $O(|V|^2 + 2\cdot|V|\cdot|E|)$.

Т.к. для хранения графа используется список инцидентности, который хранит ровно $|E|$ ребер и $|V|$, то сложность по памяти для хранения графа $O(|V| + |E|)$.

Описание алгоритма Дейкстры на основе кода A*.

Так как при эвристической оценке $h(x) = 0$ алгоритм A* полностью совпадает с алгоритмом Дейкстры поиска пути, то все основные этапы работы будут совпадать с описанными выше.

В данном случае алгоритму достаточно иметь только список значений стоимости пути от начальной вершины, до текущей - `fromStart`, т.к. значение эвристической оценки берется равной 0 и поэтому $f(x) = g(x)$. Алгоритм так же работает до тех пор, пока не будет пустым список открытых вершин или не будет достигнута целевая вершина, из открытого списка на каждом шаге выбирается вершина с минимальной стоимостью пути до нее.

Сложность алгоритма Дейкстры поиска пути будет равна $O(|V|^2 + 2 \cdot |V| \cdot |E|)$, по памяти - $O(|V| + |E|)$.

Описание функций и структур данных.

```
struct Edge{                        //структура "ребра"
  char startVertex;                 //начало и конец ребра
  char endVertex;
  double distant;                   //вес ребра
  Edge(char,char, double);
};
```

Структура `Edge` - структура для хранения одного ребра ориентированного графа была описана выше.

```
using EdgeList = std::vector <Edge>;   //вектор смежных ребер
struct AdjacencyList{                  //структура "Список инцидентности"
    std::map<char, EdgeList> adjList;
    int countEdge;                     //число ребер в списке
    Adjacencylist();
    int size();
    void addEdge(char, char, double);  //метод добавления ребра в список
};
```

Структура `AdjacencyList` для хранения графа так же описана выше.

```
std::ostream &operator<<(std::ostream& stream, std::vector<char>& vect);
```

Был перегружен оператор вывод в поток для более удобного вывода вектора вершин.

```
class Dialog{
 private:
    Adjacencylist graph;
    std::map<char,char> resultPath;
    void reconstructPath(char,char);
    void toLower(std::string &str);
    void clear();
 public:
    void runDialog();
    int  heuristic(char, char);
    char min_F(std::vector<char>& openSet, std::map<char, double>&
function);
    bool algorithm_A_star(char, char);
    bool Dijkstra(char, char);
};
```

Класс Dialog - класс реализующий примитивный диалог с пользователем. Хранит в себе следующие поля и методы:

- `graph` - соответственно граф, который введет пользователь.
- `resultPath` - результирующий путь, хранит в себе пары вершина-источник (то есть из какой вершины попали в текущую), используется методом `reconstructPath()` для вывода результирующего пути.
- `toLower()` - вспомогательная функция для диалога с пользователем, приводит строку к нижнему регистру.
- `clear()` - вспомогательная функция, очищает хранящийся граф graph.
- `runDialog()` - главный диалоговый метод. Считывает введенный пользователем граф, обрабатывает неверный ввод. После считывания вызывает метод `algorithm_A_star()` или `Dijkstra()`, и в зависимости от результата выводи ошибку (путь не найден) либо же результирующий путь. После окончания работы предлагает пользователю продолжить или выйти.
- `heuristic()` - метод эвристической оценки. По условию возвращает близость символов, обозначающих вершины графа, в таблице ASCII. Используется функцией `algorithm_A_star()`.

- `algorithm_A_star()` и `Dijkstra()` - соответственно методы реализованных алгоритмов A* и Дейкстры, принимают на вход стартовую и конечную вершины.
- `min_F()` - метод поиска вершины из открытого списка с минимальной стоимостью пути до нее. Используется в алгоритме A* и Дейкстры (в случае A* принимает список `function` значений стоимости вместе с эвристической функцией, Дейкстры - список `fromStart` значений стоимости пути от начальной вершины, до текущей).
- `reconstructPath()` - метод восстановление полученного пути. Принимает на вход стартовую и конечную вершины, так же оперирует с вектором вершин `resultPath`. Путь прослеживается от конечной к стартовой: сначала в качестве текущей будет конечная вершина (добавляется в вывод), после чего до достижения начальной вершины в вывод добавляется вершина, из которой пришли в текущую. После того, как получен путь от конечной до начальной, список инвертируется и выводится.

Выводы.

В ходе работы были изучены и реализованы жадный алгоритм, алгоритм A* и реализованный на основе его кода алгоритм Дейкстры нахождения пути в ориентированном графе.

Тестирование.

Таблица 1 - тестирование жадного алгоритма.

Входные данные	Результат
a g a b 1.0 a c 2.0 a d 3.0	Полученный путь: a c f g

c e 3.0 c f 1.0 e g 1.0 f g 10.0	
a z a b 1.0 a c 2.0 a d 3.0 c h 2.0 c y 3.0 h z 1.0 y z 3.0	Полученный путь: a c h z
a f a b 2.0 c d 4.0 f e 3.0	Ошибка! Путь не найден!
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	Полученный путь: b g e

Таблица 2 - тестирование алгоритма A*.

Входные данные	Результат
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	Полученный путь: a g

e f 2.0 a g 8.0 f g 1.0	
a z a b 1.0 a c 2.0 a d 3.0 c h 2.0 c y 3.0 h z 1.0 y z 3.0	Полученный путь: a c y g
a z a b 1.0 a c 2.0 b c 1.0 b d 1.0 c b 1.0 c d 1.0	Ошибка! Путь не найден!
a z ----	Ошибка! Путь не найден!
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	Полученный путь: a d e

Таблица 3 - тестирование алгоритма Дейкстры.

Входные данные	Результат
a f a b 1 a c 2 a d 1	Полученный путь: a d e f

b d 3 c e 2 d e 2 e f 2 d f 6	
a g a b 2 a c 4 a h 3 d g 3	Ошибка! Путь не найден!
ae a b 10 a c 8 c b 1 b d 3 d e 7 b e 9 c e 11	Полученный путь: a c b e
a b b c 3 a b 3 a c 1 c b 1	Полученный путь: a c b

Пример работы программы без вывода промежуточных данных (рис. 1):

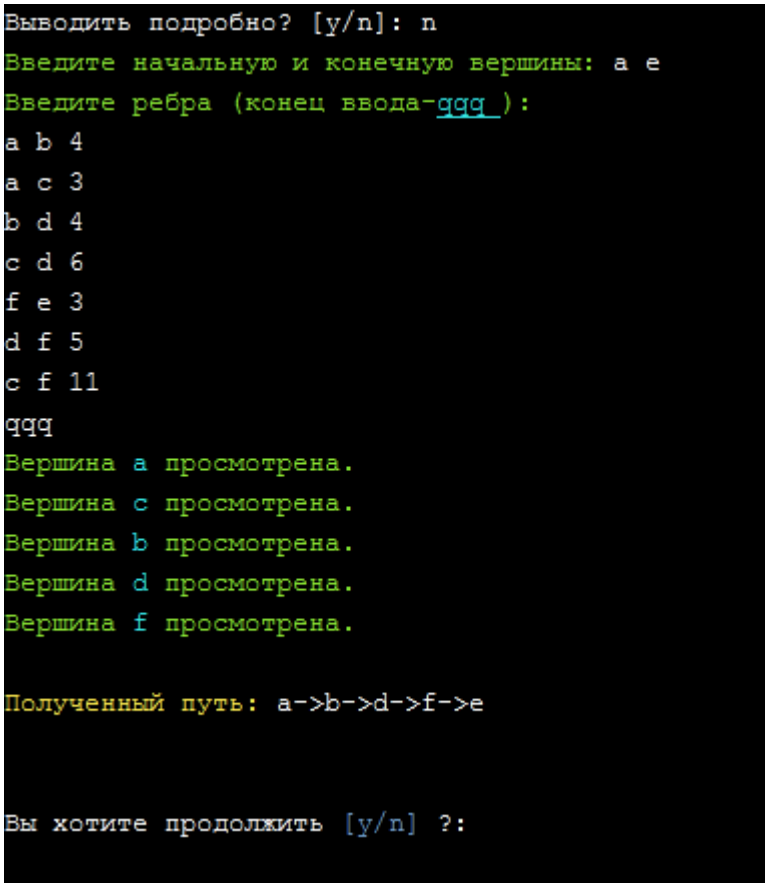

Рисунок 1 - Вывод результата (без промежуточных)

Пример работы программы с выводом промежуточных данных (рис. 2):

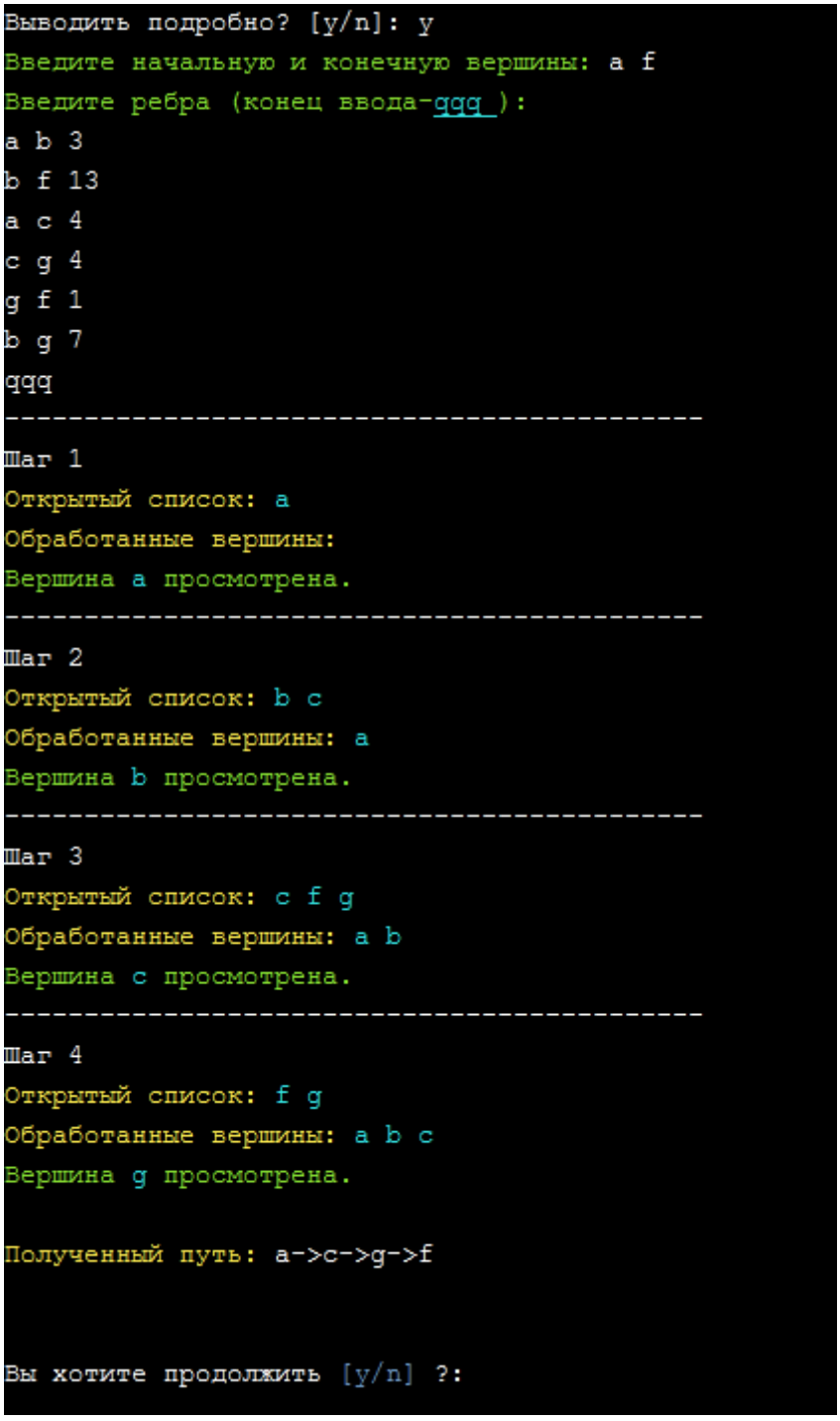

Рисунок 2 - Работа с выводом промежуточных данных

Пример работы программы при ошибке (отсутствие пути) см. рис. 3.

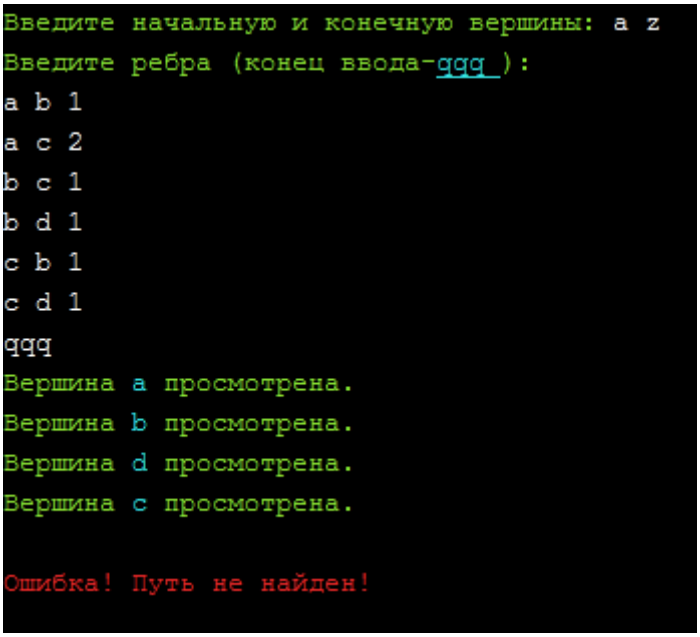

Рисунок 3 - Не найден путь.

Приложение

Код "жадного" алгоритма

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

using namespace std;

const std::string UNDERLINE = "\x1b[1;4;36m";
const std::string CYAN = "\x1b[1;36m";
const std::string RED = "\x1b[1;31m";
const std::string GREEN = "\x1b[1;32m";
const std::string YELLOW = "\x1b[1;33m";
const std::string NORMAL = "\x1b[0m";



struct Edge{                        // ребро
  char startVertex;                 // начальная и
  char endVertex;                  // конечная вершины
  double weight;                    // вес
  bool notTouch = true;             // проходили ли через
  Edge(char,char, double);          // это ребро
};


Edge::Edge(char start, char end, double weight_)
: startVertex(start), endVertex(end), weight(weight_) {}
```

```
void dialog();

class EdgeList{                    //"список ребер"
  friend void dialog();
  private:
    vector<Edge*> list;
    int countEdge;                // количество ребер графа
  public:
    EdgeList();
    void addEdge(Edge*);              // методы добавления
    void addEdge(char, char, double); //ребра
};


EdgeList::EdgeList() : countEdge(0) {}


void EdgeList::addEdge(Edge* edge){
    list.push_back(edge);
}

void EdgeList::addEdge(char start, char end, double weight_)
{
    Edge* tmp =  new Edge(start, end, weight_);
    list.push_back(tmp);
}




void dialog(){
    EdgeList workList;
    char globalStart;
    char globalEnd;
    char start, goal;
    double weight;

    std::cout << GREEN << "Введите начальную и конечную вершины: " <<
NORMAL;
    while (!(std::cin >> globalStart >> globalEnd)){
        std::cout << RED << "Ошибка ввода. Попробуйте снова: " <<NORMAL;
    }

    std::cout << GREEN << "Введите ребра (конец ввода-"<< UNDERLINE <<
"qqq "<< NORMAL << GREEN <<"):" << NORMAL<<std::endl;

    while(cin >> start >> goal >> weight){
        workList.addEdge(start,goal, weight);
        workList.countEdge++;
    }
```

```
  char Current = globalStart;
  double currentMin; int index;
  vector<char> stack;                    //стек просмотренных вершин для
возврата из тупика
                                         //он же результирующий список
вершин
  while (Current != globalEnd){          // главный цикл, пока не
достигнем конечной вершины
      index = 0;
                                         // игнорируем несмежные ребра и
просмотренные
      while(workList.list[index]->startVertex != Current ||
workList.list[index]->notTouch==false){
          index++;
          if (index == workList.countEdge) break;
//тупик
      }

      if (Current == globalEnd)  continue;

      if (index >= workList.countEdge){            //т.к. тупик,
"откатываем" назад
         if (stack.empty()){                  // если откатывать не куда,
то пути нет
              std::cout << RED << "\nОшибка! Путь не найден!" << NORMAL
<< std::endl;
              return;
          }



          Current = stack.back();
          stack.pop_back();
          continue;
      }


      currentMin = workList.list[index]->weight;

      for (int i = index; i < workList.list.size(); i++)       //ищем
смежные ребра
      {
          if (workList.list[i]->startVertex != Current ||
workList.list[i]->notTouch==false)
               continue;
          else
           {
              if (workList.list[i]->weight < currentMin)       //"жадный
выбор"
                  {
                     index = i;
```

```
                        currentMin = workList.list[i]->weight;
                    }
                }
            }

            stack.push_back(Current);
            Current = workList.list[index]->endVertex;
            workList.list[index]->notTouch = false;             //пометели как
"просмотренную"
        }


        std::cout << YELLOW << "Полученный путь: " << NORMAL;
        for (int i = 0; i < stack.size(); i++){       //выводим результат
            cout << stack[i] << "->";
        }
         cout << globalEnd;
}




int main()
{
    dialog();

    return 0;
}
```

КОД АЛГОРИТМА A*

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
#include <ostream>


const std::string UNDERLINE = "\x1b[1;4;36m";
const std::string CYAN = "\x1b[1;36m";
const std::string RED = "\x1b[1;31m";
const std::string GREEN = "\x1b[1;32m";
const std::string YELLOW = "\x1b[1;33m";
const std::string NORMAL = "\x1b[0m";



//перегруженный оператор вывода в поток
//для полученного пути
std::ostream &operator<<(std::ostream& stream, std::vector<char>& vect)
{
    for (const auto &elem: vect){
```

```
        stream << elem;
    }
    return stream;
}

void fixed(){
    std::cin.clear();
    while (std::cin.get() != '\n');
}


struct Edge{                        //структура "ребра"
  char startVertex;                 //начало и конец ребра
  char endVertex;
  double distant;                   //вес ребра
  Edge(char,char, double);
};


Edge::Edge(char start, char end, double distant_)
: startVertex(start), endVertex(end), distant(distant_) {}


using EdgeList = std::vector <Edge>;      //вектор смежных ребер


struct Adjacencylist{                     //структура "Список
инциндентности"
    std::map<char, EdgeList> adjList;
    int countEdge;                        //число ребер в списке
    Adjacencylist();
    int size();
    void addEdge(char, char, double);       //метод добавления ребра в
список
};



Adjacencylist::Adjacencylist() : countEdge(0) {}


int Adjacencylist::size(){
    return adjList.size();
}


void Adjacencylist::addEdge(char start, char end, double weight_)
{
    Edge newEdge(start, end, weight_);
    EdgeList newList;
    if (adjList.find(start) == adjList.end())
        adjList[start] = newList;
```

```
        //adjList.insert(std::pair <char, EdgeList>(firstNode,
emptyVector));
    adjList[start].push_back(newEdge);
    countEdge++;
}



class Dialog{
 private:
    Adjacencylist graph;
    std::map<char,char> resultPath;
    void reconstructPath(char,char);
    void toLower(std::string &str);
    void clear();
 public:
    void runDialog();
    int heuristic(char, char);
    char min_F(std::vector<char>& openSet, std::map<char, double>&
function);
    bool algorithm_A_star(char, char);
};


void Dialog::toLower(std::string &str){
    int i = 0;
    while (str[i]){
      str[i] = tolower(str[i++]);
    }
}

void Dialog::clear(){
  for(auto& item : graph.adjList)
   {
     item.second.clear();
     item.second.shrink_to_fit();
   }
   graph.adjList.clear();
}



void Dialog::runDialog(){                                    //общая
функция, поддерживающая диалог
    std::string str;
    char start, goal;
    char startVertex, endVertex;
    double distant;
 do{
    std::cout << GREEN << "введите начальную и конечную вершины: " <<
NORMAL;
    while (!(std::cin >> start >> goal)){
```

```
        std::cout << RED << "Ошибка ввода. Попробуйте снова: " <<NORMAL;
    }

    std::cout << GREEN << "введите ребра (конец ввода-"<< UNDERLINE <<
"qqq "<< NORMAL << GREEN <<"):" << NORMAL<<std::endl;
    while(std::cin >> startVertex >> endVertex >> distant){
        graph.addEdge(startVertex,endVertex, distant);
    }

    bool check = algorithm_A_star(start, goal);
    if (!check){
        std::cout << RED << "\nОшибка! Путь не найден!" << NORMAL <<
std::endl;
        return;
    }

    std::cout << YELLOW << "Полученный путь: " << NORMAL;
    reconstructPath(start, goal);

    fixed();
    std::cout << "\n\n" << "вы хотите продолжить \033[1;34m[y/n]\033[0m
?: ";
    std::cin >> str;
    toLower(str);

    while (str!="y" && str!="n"){
          std::cout << "\nОшибка, введите [y/n] :";
          std::cin >> str; toLower(str);
      }

    clear();
 }while(str != "n");
}



int Dialog::heuristic(char current, char goal){
//эвристическая функция, по условию - близость
    return abs((int)current - (int)goal);                    //символов
по таблице ASCII
}



char Dialog::min_F(std::vector<char>& openSet, std::map<char, double>&
function){   //возвращает вершину с наименьшим
    int index = 0; double min = function[openSet[0]]; int i = 0;
    for (const auto &vertex: openSet){
        if (function[vertex] < min) index = i;
        i++;
    }
   return openSet[index];
```

```
}


bool Dialog::algorithm_A_star(char start, char goal){
    std::vector<char> openSet;                                  // множество
вершин, которые требуется рассмотреть
    std::vector<char> finishedSet;                              // множество
рассмотренных вершин
    openSet.push_back(start);                                   //
изначально здесь присутствует только начальная вершина start
    std::map<char, double> function;                            // значение
стоимости вместе с эвристической функцией
    std::map<char, double> fromStart;                           // стоимость
пути от начальной вершины до текущей
    fromStart[start] = 0;
    function[start] = fromStart[start] + heuristic(start, goal);
    char current; int tentative; char neighbour;
    int resDistant = 0;
while(openSet.size() != 0){
        current = min_F(openSet, function);
// вершина из openset имеющая самую низкую оценку f(x)
        if (current == goal){
// достигли целевой вершины
             return true;              //reconstructPath
        }
        auto it = remove(openSet.begin(), openSet.end(), current);
        openSet.erase(it);
        std::cout << GREEN << "вершина "<< CYAN << current <<  GREEN << "
просмотрена."<<NORMAL<<std::endl;
        finishedSet.push_back(current);
        for (auto currEdge = graph.adjList[current].begin(); currEdge !=
graph.adjList[current].end(); currEdge++)
        {
// Проверяем каждого соседа текущей вершины
               neighbour = currEdge->endVertex;
               auto st = find(finishedSet.begin(), finishedSet.end(),
neighbour);
               if (st != finishedSet.end())
// пропускаем соседей из "закрытого" списка
                     continue;
               tentative = fromStart[current] + currEdge->distant;
// вычисляем текущее расстояние от начальной
               bool better;
               auto bt = find(openSet.begin(), openSet.end(), neighbour);
               if( bt == openSet.end()){
// если сосед не в "открытом" списке - добавляем
                  openSet.push_back(neighbour);
                  better = true;
// вводим признак того, что нужно обновить свойства для соседей
               }
               else{
                   if (tentative < fromStart[neighbour])
```

```
                    better = true;
                else
                    better = false;
// путь до этой вершины дороже (есть
            }


            if (better == true)
            {
                resultPath[neighbour] = current;
// обновляем значения
                fromStart[neighbour] = tentative;
                function[neighbour] = fromStart[neighbour] +
heuristic(neighbour, goal);
                resDistant += currEdge->distant;
            }
        }//for

    }
                            //путь не был найден
    return false;

}

void Dialog::reconstructPath(char start, char goal){          //функция
реконструкции пути
    std::vector<char> result;
    char current = goal;        // "обавляем goal в результирующий путь
    result.push_back(current);
    while (current != start){
        current = resultPath[current];
        result.push_back(current);
    }

    std::reverse(result.begin(), result.end());              // т.к.
конечная->..->начальная
    std::cout << YELLOW << "Путь: "<< UNDERLINE;
    std::cout << result << NORMAL << std::endl;
}




int main()
{
    Dialog go;
    go.runDialog();

    return 0;
}
```

КОД АЛГОРИТМА ДЕЙКСТРЫ

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
#include <ostream>



bool PRINT_INTEMEDIA = false;

const std::string UNDERLINE = "\x1b[1;4;36m";
const std::string CYAN = "\x1b[1;36m";
const std::string RED = "\x1b[1;31m";
const std::string GREEN = "\x1b[1;32m";
const std::string YELLOW = "\x1b[1;33m";
const std::string NORMAL = "\x1b[0m";
const std::string LINE = "-------------------------------------------";


//перегруженный оператор вывода в поток
//для полученного пути
std::ostream &operator<<(std::ostream& stream, std::vector<char>& vect)
{
    for (const auto &elem: vect){
        stream << elem;
        if (elem != vect.back())
            stream << "->";
    }
    return stream;
}

void fixed(){
    std::cin.clear();
    while (std::cin.get() != '\n');
}


struct Edge{                        //структура "ребра"
  char startVertex;                 //начало и конец ребра
  char endVertex;
  double distant;                   //вес ребра
  Edge(char,char, double);
};


Edge::Edge(char start, char end, double distant_)
: startVertex(start), endVertex(end), distant(distant_) {}


using EdgeList = std::vector <Edge>;      //вектор смежных ребер
```

```
struct Adjacencylist{                       //структура "Список
инциндентности"
    std::map<char, EdgeList> adjList;
    int countEdge;                          //число ребер в списке
    Adjacencylist();
    int size();
    void addEdge(char, char, double);       //метод добавления ребра в
список
};


Adjacencylist::Adjacencylist() : countEdge(0) {}


int Adjacencylist::size(){
    return adjList.size();
}


void Adjacencylist::addEdge(char start, char end, double weight_)
{
    Edge newEdge(start, end, weight_);
    EdgeList newList;
    if (adjList.find(start) == adjList.end())
        adjList[start] = newList;
        //adjList.insert(std::pair <char, EdgeList>(firstNode,
emptyVector));
    adjList[start].push_back(newEdge);
    countEdge++;
}



class Dialog{
 private:
    Adjacencylist graph;
    std::map<char,char> resultPath;
    void reconstructPath(char,char);
    void toLower(std::string &str);
    void clear();
 public:
    void runDialog();
    void printInter(std::vector<char>& openSet, std::vector<char>&
finishedSet);
    int heuristic(char, char);
    char min_FD(std::vector<char>& openSet, std::map<char, double>&
function);
    bool Dijkstra(char start, char goal);
};
```

```
void Dialog::printInter(std::vector<char>& openSet, std::vector<char>&
finishedSet)
{
    std::cout << "\n" << YELLOW << "Открытый список:" << CYAN;
    for (int i = 0; i < openSet.size(); i++)
        std::cout << " " << openSet[i];
    std::cout << "\n" << YELLOW << "Обработанные вершины:" << CYAN;
    for (int i = 0; i <  finishedSet.size(); i++)
        std::cout << " " << finishedSet[i];
    std::cout << NORMAL << "\n";
}



void Dialog::toLower(std::string &str){
    int i = 0;
    while (str[i]){
      str[i] = tolower(str[i++]);
    }
}

void Dialog::clear(){
  for(auto& item : graph.adjList)
   {
     item.second.clear();
     item.second.shrink_to_fit();
   }
   graph.adjList.clear();
}



void Dialog::runDialog(){                                          //общая
функция, поддерживающая диалог
    std::string str;
    char start, goal;
    char startVertex, endVertex;
    double distant;
 do{
    std::cout << "выводить подробно? [y/n]: ";
    std::cin >> str;
    while (str!="y" && str!="n"){
          std::cout << "\nОшибка, введите [y/n] :";
          std::cin >> str; toLower(str);
      }
    if (str == "y") PRINT_INTEMEDIA = true;
    std::cout << GREEN << "введите начальную и конечную вершины: " <<
NORMAL;
    while (!(std::cin >> start >> goal)){
```

```
        std::cout << RED << "Ошибка ввода. Попробуйте снова: " <<NORMAL;
    }

    std::cout << GREEN << "введите ребра (конец ввода-"<< UNDERLINE <<
"qqq "<< NORMAL << GREEN <<"):" << NORMAL<<std::endl;
    while(std::cin >> startVertex >> endVertex >> distant){
        graph.addEdge(startVertex,endVertex, distant);
    }

    bool check = Dijkstra(start, goal);
    if (!check){
        std::cout << RED << "\nОшибка! Путь не найден!" << NORMAL <<
std::endl;
        return;
    }

    reconstructPath(start, goal);

    fixed();
    std::cout << "\n\n" << "вы хотите продолжить \033[1;34m[y/n]\033[0m
?: ";
    std::cin >> str;
    toLower(str);

    while (str!="y" && str!="n"){
          std::cout << "\nОшибка, введите [y/n] :";
          std::cin >> str; toLower(str);
      }
    PRINT_INTEMEDIA = false;
    clear();
 }while(str != "n");
}



int Dialog::heuristic(char current, char goal){
//эвристическая функция, по условию - близость
    return abs((int)current - (int)goal);                    //символов
по таблице ASCII
}


char Dialog::min_FD(std::vector<char>& openSet, std::map<char, double>&
function){   //возвращает вершину с наименьшим
    int index = 0; double min = function[openSet[0]]; int i = 0;
    char chmin = openSet[0];
    for (const auto &vertex: openSet){
        if (function[vertex] < min) index = i;
        else if (function[vertex] == min) {
            if (vertex > chmin) index = i;
        }
         i++;
```

```
    }
   return openSet[index];
}
```

```
bool Dialog::Dijkstra(char start, char goal){
    std::vector<char> openSet;                                  // множество
вершин, которые требуется рассмотреть
    std::vector<char> finishedSet;                              // множество
рассмотренных вершин
    openSet.push_back(start);                                   //
изначально здесь присутствует только начальная вершина start
    std::map<char, double> fromStart;                           // стоимость
пути от начальной вершины до текущей
    fromStart[start] = 0;
    char current; int tentative; char neighbour; int step = 0;
    int resDistant = 0;
    while(openSet.size() != 0){
        current = min_FD(openSet, fromStart);
// вершина из openset имеющая самую низкую оценку f(x)
        if (current == goal){
// достигли целевой вершины
             return true;              //reconstructPath
        }

        step++;

        if (PRINT_INTEMEDIA){
            std::cout << LINE;
            std::cout << "\nШаг "<< step;
            printInter(openSet, finishedSet);
        }

        auto it = remove(openSet.begin(), openSet.end(), current);
        openSet.erase(it);
        std::cout << GREEN << "вершина "<< CYAN << current <<  GREEN << "
просмотрена."<<NORMAL<<std::endl;
        finishedSet.push_back(current);
        for (auto currEdge = graph.adjList[current].begin(); currEdge !=
graph.adjList[current].end(); currEdge++)
        {
// Проверяем каждого соседа текущей вершины
            neighbour = currEdge->endVertex;
            auto st = find(finishedSet.begin(), finishedSet.end(),
neighbour);
            if (st != finishedSet.end())
// пропускаем соседей из "закрытого" списка
                continue;
```

```
                tentative = fromStart[current] + currEdge->distant;
// вычисляем текущее расстояние от начальной
                bool better;
                auto bt = find(openSet.begin(), openSet.end(), neighbour);
                if( bt == openSet.end()){
// если сосед не в "открытом" списке - добавляем
                   openSet.push_back(neighbour);
                   better = true;
// вводим признак того, что нужно обновить свойства для соседей
                }
                else{
                    if (tentative < fromStart[neighbour])
                        better = true;
                    else
                        better = false;
// путь до этой вершины дороже (есть
                }


                if (better == true)
                {
                   resultPath[neighbour] = current;
// обновляем значения
                   fromStart[neighbour] = tentative;
                   resDistant += currEdge->distant;
                }
        }//for

    }
                            //путь не был найден
    return false;

}
```

```
void Dialog::reconstructPath(char start, char goal){          //функция
реконструкции пути
    std::vector<char> result;
    char current = goal;                                       //
"обавляем goal в результирующий путь
    result.push_back(current);
    while (current != start){
        current = resultPath[current];
        result.push_back(current);
    }
```

```
    std::reverse(result.begin(), result.end());                // т.к.
конечная->..->начальная
    std::cout << YELLOW << "Полученный путь: " << NORMAL;
    std::cout << result << NORMAL << std::endl;
}




int main()
{
    Dialog go;
    go.runDialog();

    return 0;
}
```