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 НА ГРАФАХ

Студентка гр. 8383 _____ Максимова А.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Разработка программы, решающей задачу построения пути в ориентированном графе при помощи жадного алгоритма и программы, решающей задачу построения кратчайшего пути методом A*.

Постановка задач.

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`

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

Вар. 6. Реализация очереди с приоритетами, используемой в A*, через двоичную кучу.

1. Жадный алгоритм:

Основные теоретические положения.

Жадный алгоритм — алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.

Жадный алгоритм строит решения посредством последовательности шагов, на каждом из которых получается частичное решение поставленной задачи, пока не будет получено полное решение.

На каждом шаге выбор должен быть:

- Допустимым, то есть удовлетворять условиям задачи;

- Локально оптимальным, то есть наилучшим локальным выбором среди всех допустимых вариантов, доступных на каждом шаге;
- Окончательным, то есть, будучи сделанным, он не может быть изменен последующими шагами алгоритма.

Описание структур данных.

1. Класс, необходимый для хранения информации о смежных, выходящих из одной вершины, ребрах.

```
class Edge {                                //РЕБРО
private:
    char name;                              //вершина ребра
    float weight;                           //вес ребра
    bool visit;
public:
    Edge(char name, float weight);
    void setName(char name);                //сеттеры
    void setWeight(float weight);
    void setVisit(bool visit);
    char getName();                         //геттеры
    float getWeight();
    bool getVisit();
    };
```

Поля:

- `char name;` - имя вершины, на другом конце ребра
- `float weight;` - вес ребра
- `bool visit;` - булевская переменная, для проверки посещения вершины

Методы:

- `Edge(char name, float weight);` - конструктор класса, инициализирует поля `name` и `weight` значениями, полученными при считывании. Ничего не возвращает.
- `void setName(char name);` - метод, принимающий переменную типа

char, необходимый для обновления значения поля name. Ничего не возвращает.

- void setWeight(float weight); - метод, принимающий переменную типа float, необходимый для обновления значения поля weight. Ничего не возвращает.
- void setVisit(bool visit); - метод, принимающий булевскую переменную, для задания значения поля visit. Вызывается при посещении вершины. Ничего не возвращает.
- char getName(); – метод, ничего не принимает. Используется для получения значения приватного поля name. Возвращает значение типа char.
- float getWeight(); - метод, ничего не принимает. Используется для получения значения приватного поля weight. Возвращает значение типа float.
- bool getVisit(); - метод, ничего не принимает. Используется для получения значения приватного поля visit. Возвращает значение типа bool.

2. Класс, используемый для хранения информации о вершинах графа (только тех, из которых выходят ребра).

```
class Vertex {                              //ВЕРШИНА
private:
    char name;
    std::vector <Edge> edge;                //вектор смежных ребер
public:
    Vertex(char name);
    void setEdge(Edge edge);                //сеттеры
    char getName();                         //геттеры
    std::vector <Edge>& getEdge();
};
```

Поля:

- char name; - имя вершины

- `std::vector <Edge> edge;` – вектор смежных ребер

Методы:

- `Vertex(char name);` - конструктор класса, используется для инициализации поля name, по полученным из считывания данных.
- `void setEdge(Edge edge);` - метод, принимающий на вход ребро, используется для добавления ребер в вектор. Ничего не возвращает.
- `char getName();` - метод, ничего не принимает. Необходим для получения значения приватного поля `name`. Возвращает значение типа `char`.
- `std::vector <Edge>& getEdge();` - метод, ничего не принимает. Возвращает вектор ребер.

Описание алгоритма.

Для реализации жадного алгоритма с целью построения пути в ориентированном графе была написана рекурсивная функция `bool findPath(char start, char finish, std::vector <Vertex>& vertex, std::stack <char>& path)`.

Функция принимает переменные типа char - start и finish, определяющие начало работы алгоритма и конец; вектор `vertex`, содержащий в себе список всех вершин, из которых выходят ребра, то есть те, по которым можно передвигаться; стек `path`, в который записывается имена вершин, формирующих путь. Построив путь, функция возвращает булевскую переменную true.

Алгоритм можно представить в виде последовательности шагов:

Шаг 0. Кладем "стартовую" вершину в стек Path.

Шаг 1. Проверяем не совпадает ли старт с финишей.

Шаг 1.1. Если да, заканчиваем работу алгоритма.

Шаг 1.2. Иначе, продолжаем обработку старта. Переходим к шагу 2.

Шаг 2. Проверяем не является ли стартовая вершина "тупиком".

Шаг 2.1. Если да, откатываем рекурсию назад, возвращаясь к вершине, из которой попали в текущую. Повторяем алгоритм, начиная с шага 0.

Шаг 2.2. Иначе, продолжаем обработку вершины. Переходим к шагу 3.

Шаг 3. Рассматриваем смежные ребра.

Шаг 3.1. Если вершина имеет только одно смежное ребро, сразу переходим по нему (то есть рекурсивно вызываем функцию, передав в качестве старта, вершину, в которую переходим).

Шаг 3.2. Иначе, вершина имеет более одного смежного ребра. Выбираем ребро с минимальным весом и переходим по нему. Возвращаемся к шагу 0.

*Ноль смежных ребер вершина иметь не может, так как этот вариант мы уже исключили при проверке "тупика".

Иллюстрация работы алгоритма:

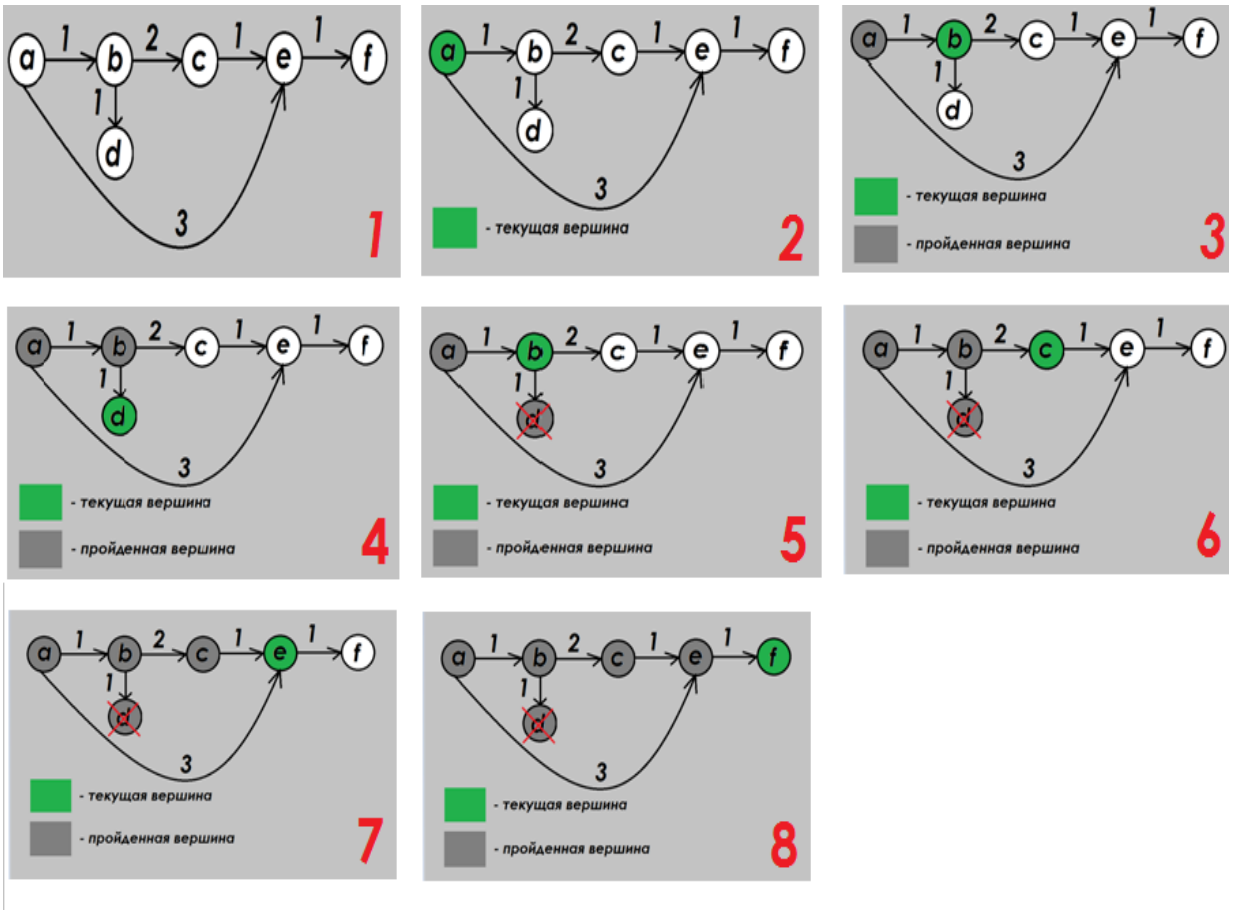


Рисунок 1 - Жадный алгоритм

Описание дополнительных функций.

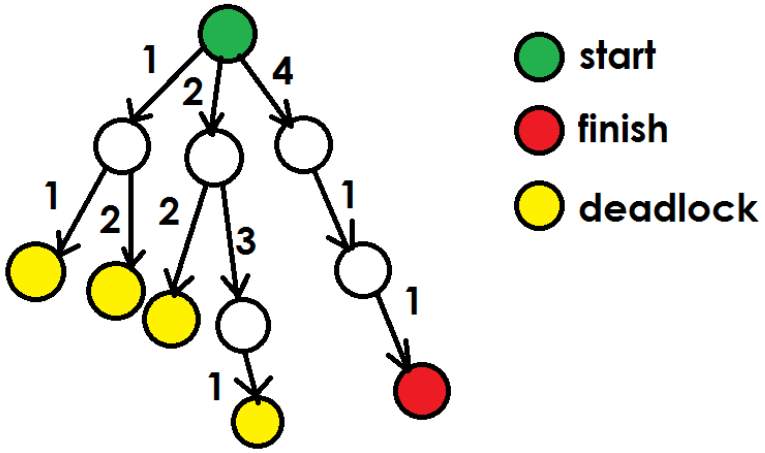
- Функция `bool readingGraph(std::vector <Vertex>& vertexGraph, char& finish);` - принимает на вход вектор, для хранения графа и переменную типа char, в которой хранится значение финиша. Функция предназначена для считывания графа: по данным из файла (для считывания) создаются вершины и смежные к ним ребра. Также в функции производится проверка на то, что во входных данных есть хотя бы одно ребро, ведущее к финишу, то есть проверяется правильность введенного графа. Функция возвращает true, если по полученным данным невозможно построить путь, иначе возвращает false.
- Функция `bool checkDeadlock(std::vector <Vertex>& vertex, char dealock);` - принимает на вход граф и имя вершины, которую нужно проверить - является ли она "тупиком" или нет. Функция возвращает true, если вершина имеет смежные ребра, и false в обратном случае.
- Функция `void minEdge(Edge& min, Vertex& curVertex);` - принимает на вход "минимальное ребро" (изначально вес этого ребра равен 999) и вершину, из смежных ребер которой нужно выбрать минимальное. Ребра вершины сравниваются с "минимальным ребром", до тех пор, пока не будет отобрано минимальное. Выбранное ребро помечается как посещенное, а его значение помещается в переменную min. Функция вызывается при работе жадного алгоритма с целью выбора вершины, переход в которую по ребру будет самым оптимальным. Функция ничего не возвращает.
- Функция `int check(std::vector <Vertex>& vertex, char name);` - принимает граф и имя вершины, которая проверяется на то, что она является родителем вершины "тупика" и при этом других детей у нее нет. Вызов данной функции используется при откатывание рекурсии назад. Возвращает ноль, если правда, иначе единицу.
- Функция `std::stack <char> coupStack(std::stack <char>& path);` - принимает на вход стек, в котором хранятся имена вершин, вошедших в

построенный путь. Функция необходима для переворачивания стека вверх дном. Возвращает измененный стек.

- Функция `void printPath(std::stack <char> path);` - принимает на вход стек и распечатывает его содержимое - то есть построенный путь. Функция ничего не возвращает.

Оценка сложности алгоритма.

По времени жадный алгоритм в худшем случае можно оценить как $O(|V|)$, так как будут пройдены все вершины.



По памяти жадный алгоритм в худшем случае можно оценить как $O(|V|)$, так как в процессе работы алгоритм хранит путь, который в худшем случае будет состоять из всех вершин, входящих в граф.

Тестирование.

Подробный тест.

Входные данные.

```
a i
a e 3
a b 1
b c 4
a d 2
e f 5
f g 1
h i 3
f h 2
g i 1
```

Выходные данные.

```
Запуск Жадного Алгоритма
--------------------------------------------------

Переход на (a)

Рассматриваемые ребра:
a --3--> e
a --1--> b
a --2--> d

Ребро с минимальным весом: a --1--> b
--------------------------------------------------

Переход на (b)

Рассматриваемые ребра:
b --4--> c

Ребро с минимальным весом: b --4--> c
--------------------------------------------------

Переход на (c)

Тупик!
Возвращаемся назад!
Возвращаемся назад!
--------------------------------------------------
```

```
Переход на (a)

Рассматриваемые ребра:
a --3--> e
a --1--> b
a --2--> d

Ребро с минимальным весом: a --2--> d
--------------------------------------------------

Переход на (d)

Тупик!
Возвращаемся назад!
--------------------------------------------------

Переход на (a)

Рассматриваемые ребра:
a --3--> e
a --1--> b
a --2--> d

Ребро с минимальным весом: a --3--> e
--------------------------------------------------

Переход на (e)

Рассматриваемые ребра:
e --5--> f

Ребро с минимальным весом: e --5--> f
--------------------------------------------------

Переход на (f)

Рассматриваемые ребра:
f --1--> g
f --2--> h

Ребро с минимальным весом: f --1--> g
--------------------------------------------------

Переход на (g)

Рассматриваемые ребра:
g --1--> i

Ребро с минимальным весом: g --1--> i
--------------------------------------------------

Переход на (i)

Достигнут финиш!
--------------------------------------------------

Оптимальный путь:
a --> e --> f --> g --> i
```

Таблица тестирования.

Входные данные	***Выходные данные***
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	
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 e f 2.0 a g 8.0 f g 1.0	
a i a b 2 c d 1 c f 2 f g 2 f h 3 d e 2 e e 1 b c 1 h i 4	

a a *a a 1*	Оптимальный путь: a
a h a b 1 a d 2 b c 1 c d 1 d e 1 e f 1 g h 1 a h 3 f g 1 d h 2	Оптимальный путь: a --> b --> c --> d --> e --> f --> g --> h
a k a b 1 a j 2 b c 1 c d 1 d e 1 e f 1 f g 1 g h 1 h i 1 i j 1 j k 1	Оптимальный путь: a --> b --> c --> d --> e --> f --> g --> h --> i --> j --> k

2. Алгоритм A*:

Основные теоретические положения.

Алгоритм A* - алгоритм поиска, который находит во взвешенном графе маршрут наименьшей стоимости от начальной вершины до выбранной конечной.

Порядок обхода вершин определяется эвристической функцией $f(x) = g(x) + h(x)$, где $g(x)$ - наименьшая стоимость пути из стартовой вершины в x, а $h(x)$ - эвристическое приближение стоимости пути от x до конечной цели. Таким образом, $f(x)$ - это длина пути до цели, которая складывается из пройденного расстояния $g(x)$ и оставшегося $h(x)$.

Описание структур данных.

1. Класс, необходимый для хранения информации о смежных, выходящих из одной вершины, ребрах.

```
class Edge {                                    //ребро
private:
     char name;
     float weight;
public:
     Edge(char name, float weight);
     char getName();
     float getWeight();
    };
```

Поля:

- `char name;` - имя вершины, на другом конце ребра
- `float weight;` - вес ребра

Методы:

- `Edge(char name, float weight);` - конструктор класса, инициализирует поля `name` и `weight` значениями, полученными при считывании. Ничего не возвращает.

- `char getName();` – метод, ничего не принимает. Используется для получения значения приватного поля `name`. Возвращает значение типа `char`.
- `float getWeight();` – метод, ничего не принимает. Используется для получения значения приватного поля `weight`. Возвращает значение типа `float`.

2. Класс, используемый для хранения информации о вершинах графа (только тех, из которых выходят ребра).

```
class Vertex {                                  //вершина
private:
      char name;
      char parent;
      float funcG;
      float funcH;                              //h(x) - эвристическое
приближение стоимости пути от x до конечной цели
      float funcF;                              //f(x) = g(x) + h(x) - длина
пути до цели
      std::vector <Edge> edges;        //смежные ребра
public:
      Vertex(char name);
      void setParent(char parent);
      void setG(float g);
      void setH(char fnsh);
      void setF();
      void setEdge(Edge edge);
      char getName();
      char getParent();
      float getG();
      float getH();
      float getF();
      std::vector <Edge>& getEdge();
};
```

Поля:

- `char name;` - имя вершины
- `char parent;` - имя родителя
- `float funcG;` - численное значение наименьшей стоимости пути в вершину из стартовой
- `float funcH;` - численное значение эвристического приближения стоимости пути от вершины до конечной цели
- `float funcF;` - численное значение длины пути до цели
- `std::vector <Edge> edge;` – вектор смежных ребер

Методы:

- `Vertex(char name);` - конструктор класса, используется для инициализации поля name, по полученным из считывания данных.
- `void setParent(char parent);` - метод, принимающий на вход имя родителя. Используется для инициализации поля `parent`. Метод ничего не возвращает.
- `void setG(float g);` - метод, принимающий на вход значение наименьшей стоимости пути в вершину из стартовой и устанавливающий это значение в поле `funcG`. Метод ничего не возвращает.
- `void setH(char fnsh);` – метод, принимающий значение имени конечной вершины и вычисляющий численное значение эвристического приближения стоимости пути от вершины до этой конечной вершины, которое устанавливается в поле `funcH`. Метод ничего не возвращает.
- `void setF();` - метод, вычисляющий значение длины пути до конечной вершины. Ничего не принимает и не возвращает, устанавливает вычисленное значение в поле `funcF`.
- `void setEdge(Edge edge);` - метод, принимающий на вход ребро, используется для добавления ребер в вектор. Ничего не возвращает.
- `char getName();` - метод, ничего не принимает. Необходим для

получения значения приватного поля `name`. Возвращает значение типа `char`.

- `char getParent();` - метод, необходимый для получения значения приватного поля `parent`. Метод ничего не принимает, возвращает значение типа `char`.
- `float getG();` - метод, используемый для получения значения приватного поля `funcG`. Метод ничего не принимает, возвращает значение типа `float`.
- `float getH();` - метод, используемый для получения значения приватного поля `funcH`. Метод ничего не принимает, возвращает значение типа `float`.
- `float getF();` - метод, используемый для получения значения приватного поля `funcF`. Метод ничего не принимает, возвращает значение типа `float`.
- `std::vector <Edge>& getEdge();` - метод, ничего не принимает. Возвращает вектор ребер.

***** Также реализованы два класса: `BinaryHeap и PriorityQueue`, которые описаны отдельно в разделе "Очередь с приоритетами".

Описание алгоритма.

Для реализации алгоритма A* была написана итеративная функция `bool findPathAstar(std::vector <Vertex> graph, std::map <char, char>& minPath, char finish, char start, int & weightPath).`

Функция принимает переменные типа char - start и finish, определяющие начало работы алгоритма и конец; вектор `graph`, содержащий в себе список всех вершин, из которых выходят ребра, то есть те, по которым можно передвигаться; карту для хранения ребер, входящих в кратчайший путь, и целочисленную переменную `weightPath`, используемую для хранения общей стоимости пути.

Алгоритм можно представить в виде последовательности шагов:

Шаг 0. Кладем в очередь с приоритетом стартовую вершину (значения funcG, funcH, funcF задаются заранее).

Шаг 1. Кладем в вершину для обработки верхний элемент очереди. Удаляем этот элемент из очереди и кладем его в вектор обработанных вершин (то есть отмечаем как просмотренный). Записываем в карту значения ребра, по которому перешли в текущую вершину (кроме стартовой).

Шаг 2. Выполняем обработку вершины.

Шаг 2.1. Если имя вершины совпадает с именем конечной вершины, то путь построен. Вес построенного пути фиксируется. Функция возвращает true.

Шаг 2.2. Иначе продолжаем обработку.

Шаг 2.2.1. Проверяем есть ли у вершины смежные ребра, если да, переходим к шагу 3.

Шаг 2.2.2. Иначе прекращаем работу алгоритма, так как граф неправильный. *См. пояснение ниже.

Шаг 3. Рассматриваем **соседей.

Шаг 3.1. Проверяем не является ли сосед "тупиком". Если да, пропускаем его. Возвращаемся к шагу 3.

Шаг 4. Проверяем, был ли сосед обработан ранее (из других вершин). Если да, пропускаем его. Возвращаемся к шагу 3.

Шаг 5. Вычисляем $g(x)$ для соседа. Проверяем находится ли сосед в очереди.

Шаг 5.1. Если нет, добавляем в очередь и записываем значения полей (funcG, funcH, funcF).

Шаг 5.2. Иначе, сравниваем новое $g(x)$ со старым.

Шаг 5.2.1 Если новое значение $g(x)$ оказалось меньше, то обновляем значения полей этого соседа.

Шаг 5.2.2 Иначе, переход из текущей вершины в соседа является более дорогим, поэтому сосед пропускается.

Переходим к шагу 3, до тех пор пока не кончались соседи.

Возвращаемся к шагу 1, до тех пор пока есть элементы в очереди.

Пояснения:

***** Шаг 2.2.2 сигнализирует о неправильности графа, так как когда в очереди находится только стартовая вершина, и при этом из нее нет смежных ребер, это означает, что невозможно построить путь от старта до финиша. В других случаях, "тупики" отлавливаются на шаге 3.1 и игнорируются алгоритмом.

** Соседи - две концевые вершины одного и того же ребра. Рассматриваем соседей текущей вершины.

Описание дополнительных функций.

- Функция `bool readingGraph(std::vector <Vertex>& graph, char finish, int & weightPath);` - принимает на вход вектор, предназначенный для хранения графа, имя конечной вершины и переменную для хранения суммарного веса пути. Функция используется для считывания вершин и смежных им ребер (с их весами). Также в функции проверяется наличие хотя бы одного ребра, ведущего в конечную вершину, в соответствии с этим функция возвращает true, если такого ребра нет, или false.
- Функция `bool isProcessed(char name, std::vector <char> processed);` - принимает на вход вектор обработанных вершин и имя проверяемой вершины. Используется для проверки вхождения вершины в данный вектор. Возвращает `true,` в случае вхождения, иначе `false.`
- Функция `bool isDeadlock(char name, std::vector <Vertex> graph`); - принимающая на вход вектор вершин и имя вершины, которую нужно проверить. Если вершина является тупиком, функция возвращает `false`, иначе `true.`
- Функция `Vertex find(char name, std::vector <Vertex> graph);` - принимает граф и имя вершины, которую нужно найти. Возвращает найденную вершину.

- Функция `void newValueFunc(Vertex & vertex, char parent, float g, char finish);` – принимает вершину, ее родителя, численное значение $g(x)$ и конечную вершину. Используется для обновления значения полей `(funcG, funcH, funcF, parent)` принимаемой вершины. Ничего не возвращает.
- Функция `void printInfo(Vertex vertex);` - принимает вершину. Предназначена для печати полей вершины `(funcG, funcH, funcF)` в качестве промежуточных выходных данных. Ничего не возвращает.
- Функция `void printInfo(PriorityQueue queue, std::vector <char> processed, int &iter);` - принимает "списки" открытых и закрытых вершин, а также числовую переменную - количество итераций. Используется для печати промежуточных данных. Ничего не возвращает.
- Функция `void printPath(std::stack <char>& answer);` - принимает имена вершин, вошедших в построенный кратчайший путь, и распечатывает их как результат. Ничего не возвращает.
- Функция `void buildPath(char start, char finish, std::stack <char>& answer, std::map <char, char> minPath);` - принимает имена стартовой и конечной вершин, стек, для хранения имен вершин, вошедших в построенный кратчайший путь и карту, в которой хранятся имена ребер. По карте восстанавливает путь. Результат хранится в `answer`. Ничего не возвращает.

Иллюстрация работы алгоритма:

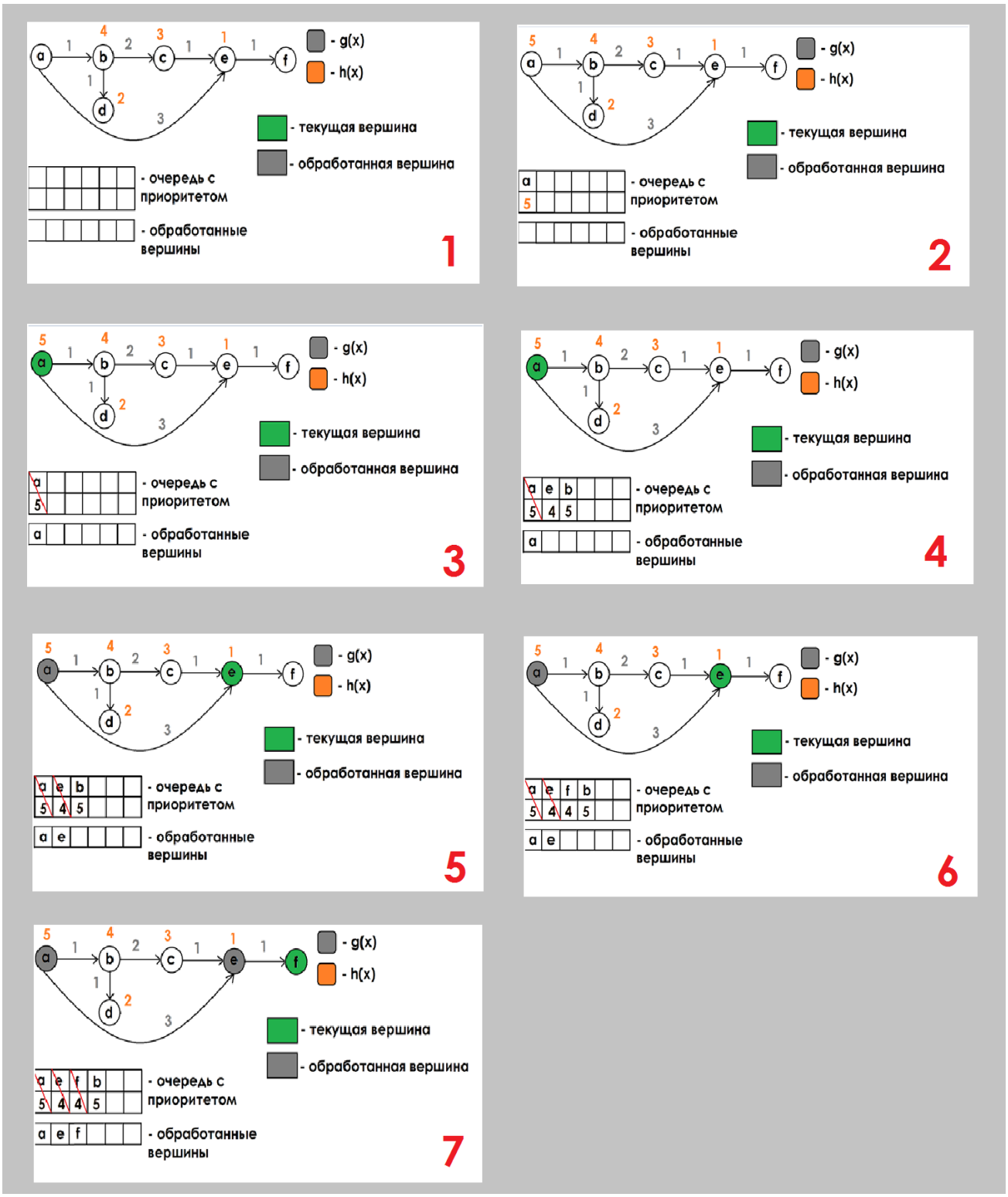


Рисунок 2 - Алгоритм A*

Оценка сложности алгоритма.

Сложность по времени алгоритма A* зависит от эвристики. В худшем случае, число вершин, исследуемых алгоритмом, растёт экспоненциально по сравнению с длиной оптимального пути, но сложность становится полиномиальной, когда эвристика удовлетворяет следующему условию:

$$|h(x) - h^*(x)| \leq O(\log h^*(x));$$

где h* — оптимальная эвристика, то есть точная оценка расстояния из вершины x к цели. Другими словами, ошибка *h(x)* не должна расти быстрее, чем логарифм от оптимальной эвристики.

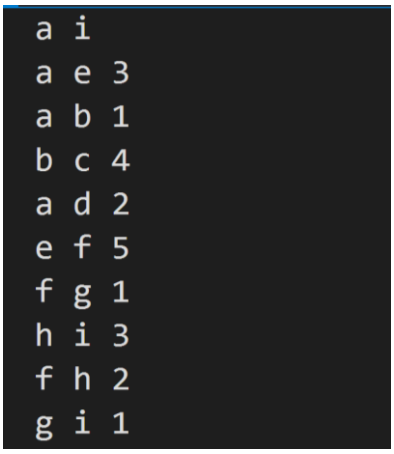
Сложность алгоритма A* в данной программе можно оценить как $O(E \hat{} \log E)$, где E - количество ребер в графе, $O(log\, E)$ - сложность операций приоритетной очереди (на базе бинарной кучи).

По памяти алгоритм A* в худшем случае можно оценить как $O(2(|V| + |E|))$, так как в программе хранится граф, из $|V|$ вершин и $|E|$ ребер, используется map, в котором максимально могут лежать все $|E|$ ребер, а также используются "списки" закрытых и открытых вершин, которые в сумме хранят $|V|$ вершин. (вершина не может находиться сразу в обоих)

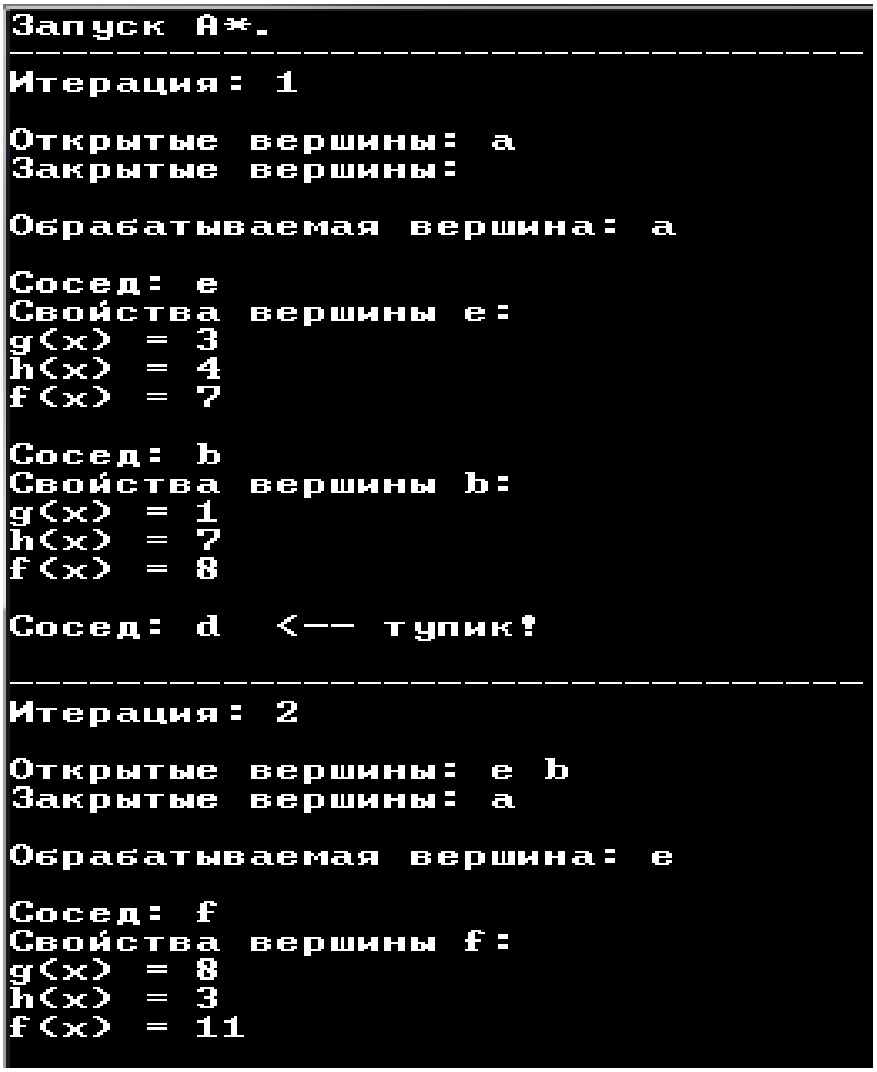
Тестирование.

Подробный тест.

Входные данные.

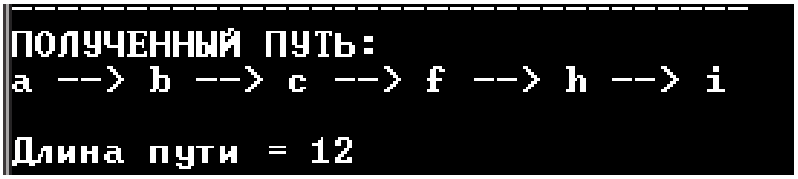


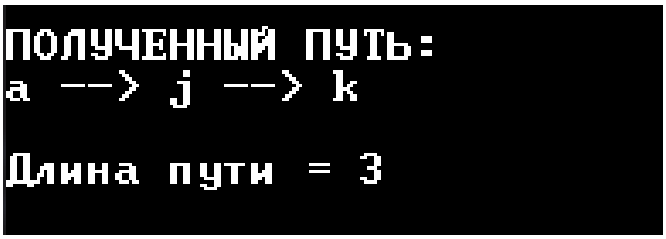
Выходные данные.



```
------------------------------------
Итерация: 3

Открытые вершины: b f
Закрытые вершины: a e

Обрабатываемая вершина: b

Сосед: c  <-- тупик!

------------------------------------
Итерация: 4

Открытые вершины: f
Закрытые вершины: a e b

Обрабатываемая вершина: f

Сосед: g
Свойства вершины g:
g(x) = 9
h(x) = 2
f(x) = 11

Сосед: h
Свойства вершины h:
g(x) = 10
h(x) = 1
f(x) = 11

------------------------------------
Итерация: 5

Открытые вершины: g h
Закрытые вершины: a e b f

Обрабатываемая вершина: g

Сосед: i
Свойства вершины i:
g(x) = 10
h(x) = 0
f(x) = 10

------------------------------------
Итерация: 6

Открытые вершины: i h
Закрытые вершины: a e b f g

Обрабатываемая вершина: i

Кратчайший путь построен.
------------------------------------
ПОЛУЧЕННЫЙ ПУТЬ:
a --> e --> f --> g --> i

Длина пути = 10
```

Таблица тестирования.

Входные данные	***Выходные данные***
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	
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 e f 2.0 a g 8.0 f g 1.0	
a i a b 2 c d 1 c f 2 f g 2 f h 3 d e 2 e e 1 b c 1 h i 4	

a a *a a 1*	
a h a b 1 a d 2 b c 1 c d 1 d e 1 e f 1 g h 1 a h 3 f g 1 d h 2	
a k a b 1 a j 2 b c 1 c d 1 d e 1 e f 1 f g 1 g h 1 h i 1 i j 1 j k 1	
a e a e 2.40 e b 1.5 a d 1.0 b c 1.0	

c f 2.0 f g 1.0 a b 1.0 d e 1.30 e g 2.0	

Очередь с приоритетами:

Основные теоретические положения.

Очередь с приоритетами - это абстрактная структура данных; это очередь, в которой все элементы имеют приоритет и упорядочены в соответствии с ним. В данной работе, очередь с приоритетами, используемая в методе A*, реализована на базе двоичной кучи.

Двоичная куча.

Двоичная куча (пирамида, сортирующее дерево) - это полной бинарное дерево, в котором поддерживается свойство порядка размещения вершин; это структура, позволяющая хранить объекты с приоритетами, извлекать самый приоритетный объект, добавлять новые объекты, быстро обновлять их приоритеты. Любая куча является очередью с приоритетами, обратное верно не всегда.

Двоичная куча может быть максимальной (max-heap) или минимальной (min-heap). В данной реализации используется min-heap, из чего следует, что значение в любой вершине не больше, чем значение ее потомков (см. рис. 1).

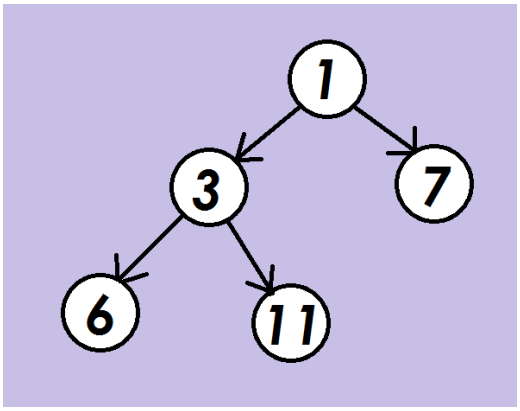


Рисунок 2 - Min-heap

Так как это полное бинарное дерево, то глубина всех листьев - расстояние до корня, отличается не более чем на один слой и последний слой заполняется строго слева направо, без пропусков. Пример см. на рис. 2.

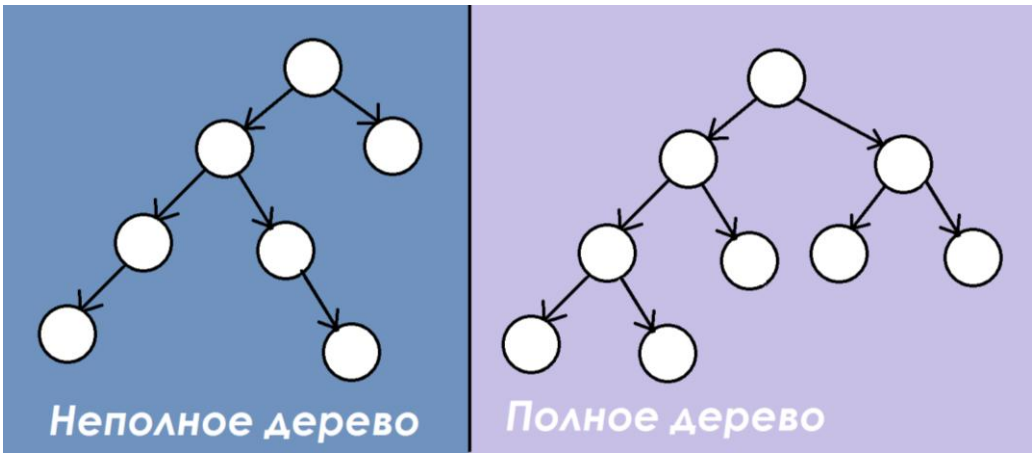


Рисунок 3 - Деревья

Описание структур данных.

1. Класс, реализующий бинарную кучу.

```
class BinaryHeap {                          //бинарная куча
private:
     std::vector <Vertex> arr;              //на базе вектора
     bool cmp(Vertex a, Vertex b);
     void swap(int a, int b);
     int parent(int i);
     int leftChild(int i);
     int rightChild(int i);
     void heapify(int i);
public:
     void addHeap(Vertex elem);
     void heapDelete();
     int find(char elem);
     Vertex find_(char elem);
     void outHeap();
     void outArr();
     Vertex getMin();
     void set(int pos, Vertex vertex);
```

```
    int sizeHeap();
    bool isEmpty();
   };
```

Поля:

- `std::vector <Vertex> arr;` - вектор для хранения элементов двоичной кучи. См. рис. 4.

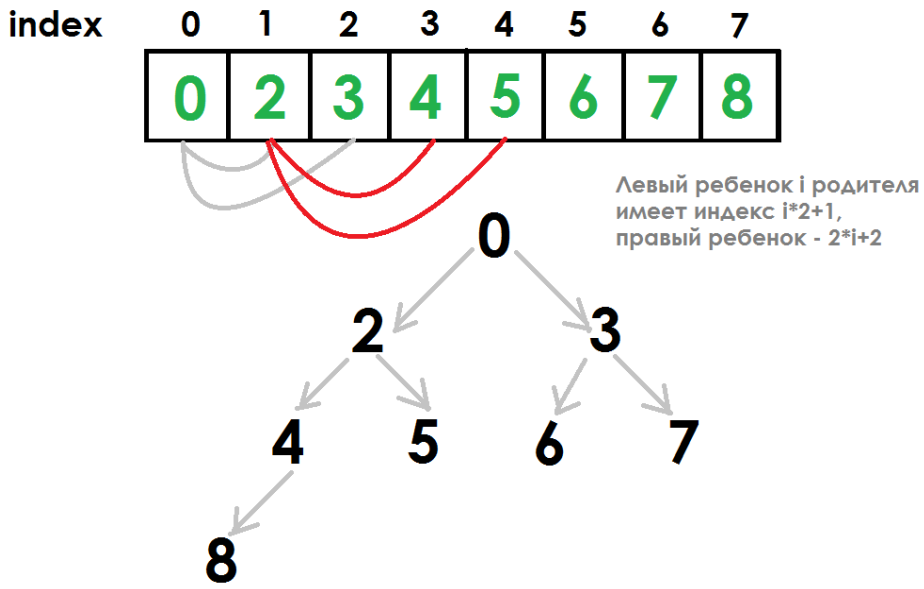


Рисунок 4 - Хранение бинарной кучи

Методы:

- `bool cmp(Vertex a, Vertex b);` - компаратор, сравнивающий две вершины по полю `funcF`, в случае равенства, по полю name. Используется для сортировки элементов бинарной кучи `(min-heap)`. Возвращает `true`, если первый элемент имеет меньшее значение, или `false`.
- `void swap(int a, int b);` - метод, принимающий позиции элементов, которые нужно поменять местами. Используется в методе `heapify(int i)`. Метод ничего не возвращает.
- `int parent(int i);` - метод, принимающий индекс элемента. Вычисляет и возвращает индекс его родителя.

- `int leftChild(int i);` - метод, принимающий индекс элемента. Вычисляет и возвращает индекс его левого ребенка.
- `int rightChild(int i);` - метод, принимающий индекс элемента. Вычисляет и возвращает индекс его равого ребенка.
- `void heapify(int i);` - метод, принимающий индекс элемента, начиная с которого производится восстановление свойств кучи, которые могут быть нарушены операцией удаления. Метод ничего не возвращает.
- `void addHeap(Vertex elem);` - метод, принимающий элемент, который нужно вставить в кучу. Используется для добавления элемента и его выдвигания вверх по дереву, пока значение в родителе не станет меньше, чем значение, которое было добавлено. Метод ничего не возвращает.
- `void heapDelete();` - метод, необходимый для удаления элемента (минимального). Осуществляется заменой значения в корне на значение последнего узла в куче с последующим удалением данного узла и восстановлением свойств кучи.
- `int find(char elem);` - метод, принимающий имя элемента, который нужно найти. Возвращает индекс найденного элемента или -1, в случае его отсутствия.
- `Vertex find_(char elem);` - метод, принимающий имя элемента, который нужно найти. Возвращает найденный элемент.
- `void outHeap();` - метод, используемый для печати содержимого кучи в виде кучи. Метод ничего не принимает и ничего не возвращает.
- `void outArr();` - метод, используемый для печати содержимого кучи в виде массива. Метод ничего не принимает и ничего не возвращает.
- `Vertex getMin();` - метод, возвращающий минимальный элемент кучи, который всегда лежит в корне. Метод ничего не принимает.
- `void set(int pos, Vertex vertex);` - метод, принимающий позицию и имя элемента, который нужно установить в данном месте. Метод ничего не возвращает.

- `int sizeHeap();` - метод, возвращающий количество узлов дерева. Метод ничего не принимает.
- `bool isEmpty();` - метод, проверяющий кучу на пустоту, возвращает true, если куча пустая. Метод ничего не принимает.

2. Класс, реализующий интерфейс очереди с приоритетом, основанный на использовании класса бинарной кучи.

```
class PriorityQueue {                          //очередь с
приоритетом
private:
     BinaryHeap binHeap;                       //на базе бинарной
кучи
public:                                             //интерфейс
очереди
     void push(Vertex elem);
     void pop();
     void set(int pos, Vertex a);
     void out();
     Vertex find_(char value);
     Vertex top();
     int find(char value);
     int size();
     bool empty();
};
```

Все методы вызывают одноименные методы класса бинарной кучи (кроме push - addHeap(elem), pop - heapDelete()). С точки зрения выполнения операций, данный класс не является необходимым, но наличие отдельного класса со своим интерфейсом имеет преимущества: для использования очереди пользователю не нужно вдаваться в особенности реализации кучи.

Оценка сложности очереди с приоритетами (кучи).

Сложность добавления элемента в кучу равняется $O(\log n)$, так как в худшем случая добавленный в конец элемент нужно будет поднять в корень, то есть пройти по всему дереву, высота которого равна $O(\log n)$, где n - количество узлов дерева.

Сложность восстановления свойств кучи также равняется $O(\log n)$, так как в худшем случая мы пройдем по всему дереву.

Сложность удаления элемента равняется $O(\log n)$, так как элемент перемещенный с конца на вершину, нужно будет спустить вниз, используя `heapify`.

Вывод.

Были разработаны программы, решающие задачу построения пути в ориентированном графе Жадным алгоритмом и методом A*.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ЖАДНОГО АЛГОРИТМА

```
#include <algorithm>
#include <iostream>
#include <fstream>
#include <vector>
#include <stack>
std::ifstream infile("test.txt");

class Edge {                                //РЕБРО
private:
    char name;                              //вершина ребра
    float weight;                           //вес ребра
    bool visit;

public:
    Edge(char name, float weight);

    void setName(char name);                //сеттеры
    void setWeight(float weight);
    void setVisit(bool visit);

    char getName();                         //геттеры
    float getWeight();
    bool getVisit();
};

Edge::Edge(char name, float weight) {
    this->name = name;
    this->weight = weight;
```

```
    visit = false;
}
                                        //сеттеры
void Edge::setName(char name) {
    this->name = name;
}

void Edge::setWeight(float weight) {
    this->weight = weight;
}

void Edge::setVisit(bool visit) {
    this->visit = visit;
}
                                        //геттеры
char Edge::getName() {
    return name;
}

float Edge::getWeight() {
    return weight;
}

bool Edge::getVisit() {
    return visit;
}
//---------------------------------

class Vertex {                          //ВЕРШИНА
private:
    char name;
```

```
        std::vector <Edge> edge;            //вектор смежных ребер

    public:
        Vertex(char name);

        void setEdge(Edge edge);            //сеттеры

        char getName();                     //геттеры
        std::vector <Edge>& getEdge();
    };

    Vertex::Vertex(char name) {
        this->name = name;
    }
                                            //сеттеры
    void Vertex::setEdge(Edge edge) {
        this->edge.push_back(edge);
    }
                                            //геттеры
    char Vertex::getName() {
        return name;
    }

    std::vector <Edge>& Vertex::getEdge() {
        return edge;
    }
    //---------------------------------

    bool readingGraph(std::vector <Vertex>& vertexGraph, char& finish) {
//считывание графа
        char vertex1, vertex2;
```

```
        float weightEdge;

        bool newVertex;

        bool finishNotExist = true;



        while (infile >> vertex1 >> vertex2 >> weightEdge && vertex1 != '!') {
//для окончания считывания вводим !!!

            if (vertex1 == vertex2) {

                continue;

            }

            if (vertex2 == finish) finishNotExist = false;
//проверка что ребро до финишной вершины существует

            Edge e(vertex2, weightEdge);

            newVertex = true;



            for (int i = 0; i < vertexGraph.size(); i++) {

                if (vertexGraph[i].getName() == vertex1) {
// если уже встречали вершину - добавляем новое ребро

                    vertexGraph[i].setEdge(e);

                    newVertex = false;

                }

            }



            if (newVertex) {
//если новая вершина

                Vertex v(vertex1);

                v.setEdge(e);

                vertexGraph.push_back(v);

            }

        }

        return finishNotExist;

    }



    bool checkDeadlock(std::vector <Vertex>& vertex, char dealock) {
//проверка не является ли вершина тупиком
```

```
        bool flag = false;

        for (int i = 0; i < vertex.size(); i++) {

            if (vertex[i].getName() == dealock) {                          //если
нет в векторе вершин, имеющих наборы смежных ребер

                flag = true;                                               //не
тупик

            }

        }

        return flag;                                                       //тупик

    }



    void minEdge(Edge& min, Vertex& curVertex) {                           //для
выбора ребра с минимальным весом

        int index = 0;

        for (int j = 0; j < curVertex.getEdge().size(); j++) {
//пробегаемся по ребрам

            if (min.getWeight() > curVertex.getEdge()[j].getWeight() &&
curVertex.getEdge()[j].getVisit() == false) { //нашли меньше

                min.setName(curVertex.getEdge()[j].getName());

                min.setWeight(curVertex.getEdge()[j].getWeight());

                index = j;

            }

        }

        curVertex.getEdge()[index].setVisit(true);
//отмечаем ребро как посещенное

    }



    int check(std::vector <Vertex>& vertex, char name) {                   //на
случаи когда из родителя вершины-тупика, нет других ребер

        bool flag = false;

        for (int i = 0; i < vertex.size(); i++) {

            if (vertex[i].getName() == name) {

                if (vertex[i].getEdge().size() > 1) {

                    flag = true;

                }
```

```
            }

        }

        return flag;

    }


    bool findPath(char start, char finish, std::vector <Vertex>& vertex,
std::stack <char>& path) { //построение оптимального пути - рекурсивная функция



        path.push(start);
//записываем текущую стартовую вершину в текущий путь

        std::cout << "-----------------------------------------------\n";

        std::cout << "\nПереход на (" << start << ")\n\n";



        if (start == finish) {
//оптимальный путь построен

            std::cout << "Достигнут финиш!\n";

            return true;

        }



        else {
//продолжаем построение оптимального пути

            if (checkDeadlock(vertex, start)) {                          //если
не тупик

                for (int i = 0; i < vertex.size(); i++) {

                    if (vertex[i].getName() == start) {

                        std::cout << "Рассматриваемые ребра: \n";

                        for (int j = 0; j < vertex[i].getEdge().size(); j++) {

                            std::cout << start << " --"<<
vertex[i].getEdge()[j].getWeight() << "--> " << vertex[i].getEdge()[j].getName()
<< "\n";

                        }std::cout << "\n";

                        if (vertex[i].getEdge().size() == 1) {          //если
у текущей вершины только ОДНО смежное ребро

                            std::cout << "Ребро с минимальным весом: " << start
<< " --" << vertex[i].getEdge()[0].getWeight() << "--> " <<
vertex[i].getEdge()[0].getName() << "\n";
```

```
                                vertex[i].getEdge()[0].setVisit(true);      //то
переходим по нему

                                return findPath(vertex[i].getEdge()[0].getName(),
finish, vertex, path/*, weightPath*/);

                            }



                            else {

                                Edge min('0', 999);

                                minEdge(min, vertex[i]);                    //нужно
выбрать самое дешевое ребро

                                std::cout << "Ребро с минимальным весом: " << start
<< " --" << min.getWeight() << "--> " << min.getName() << "\n";

                                return findPath(min.getName(), finish, vertex,
path);

                            }

                        }

                    }

                }

                else {                                                      //если
тупик

                    std::cout << "Тупик!\n";

                    do {

                        std::cout << "Возвращаемся назад!\n";

                        path.pop();
//убираем из пути

                        start = path.top();
//стартом становится предыдущая вершина

                    } while (!check(vertex, start));
//проверка можно ли из нового старта пойти еще куда-то кроме тупика

                    path.pop();
//которая будет добавлена в путь еще раз - поэтому убираем повтор

                    return findPath(start, finish, vertex, path);

                }

            }

        }
```

```
    std::stack <char> coupStack(std::stack <char>& path) {
//переворачивание стека (cba-->abc)

        std::stack <char> path2;;

        while (!path.empty()) {

            path2.push(path.top());

            path.pop();

        }

        return path2;

    }


    void printPath(std::stack <char> path) {                              //вывод
содержимого стека

        std::cout << "-----------------------------------------------\n";

        std::cout << "\nОптимальный путь:\n";

        do {

            if (path.size() != 1) std::cout << path.top() << " --> ";

            else std::cout << path.top();

            path.pop();

        } while (!path.empty());

        std::cout << "\n";

    }


    int main() {

        setlocale(LC_ALL, "ru");


        std::vector <Vertex> vertexGraph;
//вектор вершин - граф

        std::stack <char> path;                                           //для
хранения пути


        char start, finish;                                               //старт
и финиш

        infile >> start >> finish;

        std::cout << "\n\nПостроения оптимального пути из " << start << " в "
```

```
<< finish << ".\n";


        if (start == finish) std::cout << "\nОптимальный путь:\n" << start;

        else

            if (!readingGraph(vertexGraph, finish)) {
//считывание графа

                std::cout << "\nЗапуск Жадного Алгоритма\n\n";

                findPath(start, finish, vertexGraph, path);
//построение пути - жадный алгорит

                path = coupStack(path);
//восстанавление пути

                printPath(path);
//печать пути

            }

            else {

                std::cout << "Граф введен неправильно!\n";

            }

        return 0;

    }
```

ПРИЛОЖЕНИЕ Б
ИСХОДНЫЙ КОД АЛГОРИТМА A*

```
#include <iostream>
#include <fstream>
#include <algorithm>
#include <vector>
#include <stack>
#include <map>

std::ifstream infile("test.txt");

class Edge {                                    //ребро
private:
 char name;
 float weight;

public:
 Edge(char name, float weight);
 char getName();
 float getWeight();
};

Edge::Edge(char name, float weight) {
 this->name = name;
 this->weight = weight;
}

char Edge::getName() {
 return name;
}

float Edge::getWeight() {                          //вес ребра
 return weight;
}
//_____________________________________________________________________
class Vertex {                                     //вершина
private:
 char name;
 char parent;
```

```
      float funcG;                              //g(x) - наименьшая стоимость
пути в x из стартовой вершины
      float funcH;                              //h(x) - эвристическое
приближение стоимости пути от x до конечной цели
      float funcF;                              //f(x) = g(x) + h(x) - длина пути
до цели
      std::vector <Edge> edges;           //смежные ребра

     public:
      Vertex(char name);

      void setParent(char parent);
      void setG(float g);
      void setH(char fnsh);
      void setF();
      void setEdge(Edge edge);

      char getName();
      char getParent();
      float getG();
      float getH();
      float getF();
      std::vector <Edge>& getEdge();
     };

     Vertex::Vertex(char name) {
      this->name = name;
      parent = '0';
      funcG = funcH = funcF = -1;
     }

     void Vertex::setParent(char parent) {      //сеттеры
      this->parent = parent;
     }
     void Vertex::setG(float g) {
      funcG = g;
     }
     void Vertex::setH(char fnsh) {
      funcH = abs((int)fnsh - (int)name);
     }
     void Vertex::setF() {
      funcF = funcG + funcH;
```

```
}
void Vertex::setEdge(Edge edge) {
 edges.push_back(edge);
}

char Vertex::getName() {                    //геттеры
 return name;
}
char Vertex::getParent() {
 return parent;
}
float Vertex::getG() {
 return funcG;
}
float Vertex::getH() {
 return funcH;
}
float Vertex::getF() {
 return funcF;
}
std::vector <Edge>& Vertex::getEdge() {
 return edges;
}
//_____________________________________________________________________________
class BinaryHeap {                          //бинарная куча
private:
 std::vector <Vertex> arr;           //на базе вектора
 bool cmp(Vertex a, Vertex b);
 void swap(int a, int b);
 int parent(int i);
 int leftChild(int i);
 int rightChild(int i);
 void heapify(int i);

public:

 void addHeap(Vertex elem);
 void heapDelete();
 int find(char elem);
 Vertex find_(char elem);
 void outHeap();
 void outArr();
```

```
 Vertex getMin();
 void set(int pos, Vertex vertex);
 int sizeHeap();
 bool isEmpty();
};

bool  BinaryHeap::cmp(Vertex a, Vertex b) {
 if (a.getF() == b.getF()) return a.getName() < b.getName() ? true : false;
       //если приоритеты равны
 else return a.getF() < b.getF() ? true : false;
}

void  BinaryHeap::swap(int a, int b) {          //используется в heapify
 Vertex temp = arr[a];
 arr[a] = arr[b];
 arr[b] = temp;
}

int  BinaryHeap::parent(int i) {                      //вычисление индексов
родителей и потомков
 return (i - 1) / 2;
}

int  BinaryHeap::leftChild(int i) {
 return 2 * i + 1;
}

int  BinaryHeap::rightChild(int i) {
 return 2 * i + 2;
}

void  BinaryHeap::heapify(int i) {              //для восстановления
свойств кучи
 int cur = i;
 int left = leftChild(i);
 int right = rightChild(i);

       //проверка
 if (left < arr.size() && cmp(arr[left], arr[cur])) {
       cur = left;
 }
```

```
      if (right < arr.size() && cmp(arr[right], arr[cur])) {
            cur = right;
      }

      if (cur != i) {                    //если свойство нарушено
            swap(i, cur);
            heapify(cur);
      }
     }

     void  BinaryHeap::addHeap(Vertex elem) {         //добавление элемента в
конец и перемещение наверх, в соответствии с приоритетом
      arr.push_back(elem);
      int i = arr.size() - 1;
      while (i > 0 && cmp(arr[i], arr[parent(i)])) {
            swap(i, parent(i));
            i = parent(i);
      }
     }

     void  BinaryHeap::heapDelete() {                      //удаление элемента с
минимальным приоритетом (так как min-heap)
      if (arr.size() < 1) {
            std::cout << "Ошибка: нет элемента для удаления!\n";
            exit(1);
      }

      arr[0] = arr[arr.size() - 1]; //на его место ставится элемент с конца
      arr.pop_back();
      heapify(0);                              //который потом переходит на
нужную позицию
     }

     int  BinaryHeap::find(char elem) {                    //возвращает индекс
искомого элементв
      for (int i = 0; i < arr.size(); i++) {
            if (arr[i].getName() == elem) {
                  return i;
            }
      }
      return -1;
     }
```

```
    Vertex  BinaryHeap::find_(char elem) {                 //возвращает сам
элемент
     for (int i = 0; i < arr.size(); i++) {
           if (arr[i].getName() == elem) {
                 return arr[i];
           }
     }
    }

    void  BinaryHeap::outHeap() {                          //для печати в виде
кучи
     int i = 0;
     int j = 1;

     while (i < arr.size()) {
           while ((i < j) && (i < arr.size())) {
                 std::cout << arr[i].getName() << " ";
                 i++;
           }
           std::cout << "\n";
           j = j * 2 + 1;
     }
    }

    void  BinaryHeap::outArr() {                                  //для печати в
виде массива
     for (int i = 0; i < arr.size(); i++) {
           std::cout << arr[i].getName() << " ";
     }
     std::cout << "\n";
    }

    Vertex  BinaryHeap::getMin() {                         //минимум всегда в
корне
     if (arr.size() > 0) return arr[0];
     else std::cout << "Ошибка: пустая куча"; exit(1);
    }

    void  BinaryHeap::set(int pos, Vertex vertex) {
     arr[pos] = vertex;
    }
```

```
int  BinaryHeap::sizeHeap() {
 return arr.size();
}

bool  BinaryHeap::isEmpty() {
 return (arr.size() == 0);
}
//_____________________________________________________________________
class PriorityQueue {                              //очередь с приоритетом
private:
 BinaryHeap binHeap;                               //на базе бинарной кучи
public:                                                  //интерфейс очереди
 void push(Vertex elem);
 void pop();
 void set(int pos, Vertex a);
 void out();
 Vertex find_(char value);
 Vertex top();
 int find(char value);
 int size();
 bool empty();
};

                                          //интерфейс очереди
void PriorityQueue::push(Vertex elem) {
 binHeap.addHeap(elem);
}
void PriorityQueue::pop() {
 binHeap.heapDelete();
}
void PriorityQueue::set(int pos, Vertex a) {
 binHeap.set(pos, a);
}
void PriorityQueue::out() {
 binHeap.outArr();
}
Vertex PriorityQueue::find_(char value) {
 return binHeap.find_(value);
}
Vertex PriorityQueue::top() {
 return binHeap.getMin();
```

```
}
int PriorityQueue::find(char value) {
 return binHeap.find(value);
}
int PriorityQueue::size() {
 return binHeap.sizeHeap();
}
bool PriorityQueue::empty() {
 return binHeap.isEmpty();
}
//_____________________________________________________________________________

bool readingGraph(std::vector <Vertex>& graph, char finish, int &
weightPath) {                        //считывание ориентированного графа

 char vertex1, vertex2;
 float weightEdge;
 bool newVertex;
 bool finishNotExist = true;
                  //обработка ошибок пользователя

 while (infile >> vertex1 >> vertex2 >> weightEdge) {
       if (vertex2 == finish) finishNotExist = false;
       if (graph[0].getName() == finish) {
             if (vertex1 == graph[0].getName() && vertex2 == finish) {
                   weightPath = weightEdge;
             }
       }
       Edge edge(vertex2, weightEdge);
       newVertex = true;

       for (int i = 0; i < graph.size(); i++) {
             if (graph[i].getName() == vertex1) {
             //если встречали вершину - добавляем новое ребро
                   graph[i].setEdge(edge);
                   newVertex = false;
             }
       }

       if (newVertex) {
                   //если новая вершина
             Vertex vertex(vertex1);
```

```
                vertex.setEdge(edge);
                graph.push_back(vertex);
            }
        }

        graph[0].setG(0);
                        //заполняем св-ва вершины
        graph[0].setH(finish);
        graph[0].setF();

        return finishNotExist;
    }

    bool isProcessed(char name, std::vector <char> processed) {
                                                    //обработана ли вершина? в
векторе нет find
        for (int i = 0; i < processed.size(); i++) {
            if (processed[i] == name) {
                return true;
            }
        }
        return false;
    }

    bool isDeadlock(char name, std::vector <Vertex> graph) {
                                                    //проверка на тупик
        for (int i = 0; i < graph.size(); i++) {
            if (graph[i].getName() == name) {
                                                        //если есть в векторе
вершин, имеющих ребра - то не тупик
                return true;
            }
        }
        return false;
    }

    Vertex find(char name, std::vector <Vertex> graph){
                                                //получение соседней вершины
        for (int i = 0; i < graph.size(); i++) {
            if (graph[i].getName() == name) {
                return graph[i];
            }
```

```
      }
     }

     void newValueFunc(Vertex & vertex, char parent, float g, char finish) {
                                          //обновляем значения функций - свойства
вершины
      vertex.setG(g);
      vertex.setH(finish);
      vertex.setF();
      vertex.setParent(parent);
     }

     void printInfo(Vertex vertex) {
                                                                          //для
вывода промежуточных данных
      std::cout << "\nСвойства вершины " << vertex.getName() << ": \n";
      std::cout << "g(x) = " << vertex.getG() << " \nh(x) = " << vertex.getH()
<< " \nf(x) = " << vertex.getF() << " \n\n";
     }

     void printInfo(PriorityQueue queue, std::vector <char> processed, int
&iter) {
      std::cout << "--------------------------------";
      std::cout << "\nИтерация: " << ++iter << "\n\n";
      std::cout << "Открытые вершины: ";
      queue.out();
      std::cout << "Закрытые вершины: ";
      for (int i = 0; i < processed.size(); i++) {
            std::cout << processed[i] << " ";
      } std::cout << "\n\n";
     }

     bool findPathAstar(std::vector <Vertex> graph, std::map <char, char>&
minPath, char finish, char start, int & weightPath) {            //поиск
кратчайшего пути - итеративная функция
      int iter = 0;
      float gFunc;

      //минимальный путь до текущей вершины
```

```
	std::vector <char> processed;
                                                        //массив обработанных
вершин (хранит имена вершин)

		//очередь с приоритетом, хранящая вершины, которые нужно
обработать
	PriorityQueue queue;
                            //добавляем стартовую вершину в очередь
	queue.push(graph[0]);

	while (!queue.empty()) {
		printInfo(queue, processed, iter);

			//берем из отсортированной очереди вершину с наим приоритетом
		Vertex curVertex = queue.top();
		std::cout << "Обрабатываемая вершина: " << curVertex.getName() <<
"\n\n";

		if (curVertex.getName() != graph[0].getName()) {
			minPath.insert(std::make_pair(curVertex.getName(),
curVertex.getParent()));
		}

			//убираем из очереди
		queue.pop();

		processed.push_back(curVertex.getName());
                                                        //отмечаем как
просмотренную

		if (curVertex.getName() == finish) {	//дошли до финиша
			weightPath = curVertex.getF();
			std::cout << "Кратчайший путь построен.\n";
			return true;
		}
		else {


			if (!curVertex.getEdge().empty()) {
                                                        //если есть соседи -
просматриваем
```

```
                        for (int i = 0; i < curVertex.getEdge().size(); i++) {
                              std::cout << "Сосед: " <<
curVertex.getEdge()[i].getName() << " ";
                              if (isDeadlock(curVertex.getEdge()[i].getName(),
graph)) {                                          //если сосед не является тупиком
или это финишная вершина(!)


                                    if
(!isProcessed(curVertex.getEdge()[i].getName(), processed)) {
      //сосед не был обработан - работаем с соседом
                                          Vertex neighbour =
find(curVertex.getEdge()[i].getName(), graph);
                                          gFunc = curVertex.getG() +
curVertex.getEdge()[i].getWeight();                  //минимальный путь от
старта до рассматриваемого соседа

                                                            //не записан в
очередь

      if(queue.find(curVertex.getEdge()[i].getName()) == -1){
                                                                        //то
нужно присвоить значение и записать в очередь
                                          newValueFunc(neighbour,
curVertex.getName(), gFunc, finish);                 //то обновляем параметры

      //добавляем в очередь
                                          queue.push(neighbour);
                                          printInfo(neighbour);


                                          }
                                          else {

      //записан


      //проверяем, не нашли ли новый минимальный путь?
                                          if (gFunc <
      queue.find_(neighbour.getName()).getG()) {           //нашли новый
минимальный путь
                                                newValueFunc(neighbour,
curVertex.getName(), gFunc, finish);           //то обновляем параметры
```

```
                                                int index =
queue.find(neighbour.getName());
                                                queue.set(index,
neighbour);
                                                printInfo(neighbour);
                                          }
                                          else {

      printInfo(queue.find_(neighbour.getName()));
                                          }

                                    }


                              }
                              else {

      //уже обрабатывали соседа
                                    std::cout << " <-- уже
обработана!\n\n";
                                    continue;
                              }
                        }
                        else {      //тупик
                              std::cout << " <-- тупик!\n\n";
                              continue;
                        }
                  }
            }
            else {

      //проблемы с графом
                  return false;
            }
      }
     }

     void printPath(std::stack <char>& answer) {
      std::cout << "--------------------------------";
      std::cout << "\nПОЛУЧЕННЫЙ ПУТЬ:\n";
```

```
	do {
		if(answer.size() != 1) std::cout << answer.top() << " --> ";
		else std::cout << answer.top();
		answer.pop();
	} while (!answer.empty());

}

void buildPath(char start, char finish, std::stack <char>& answer, std::map
<char, char> minPath) {		//восстанавливаем путь
	char curNode = finish;
	answer.push(curNode);

	while (curNode != start) {
		curNode = minPath[curNode];
		answer.push(curNode);
	}
	printPath(answer);
                                                                    //вызов
печати пути
}

int main() {
	setlocale(LC_ALL, "rus");

	std::vector <Vertex> graph;
                                                                    //считанный
ориентированный граф
	std::map <char, char> minPath;
                                                                    //кратчайший
путь
	std::stack <char> answer;
                                                                    //результат
	int weightPath = 0;

	char start, finish;
	infile >> start >> finish;

	Vertex vertexSt(start);
                                                                    //чтобы старт -
как вершина, всегда был в начале массива граф
	graph.push_back(vertexSt);
```

```
	Vertex vertexFn(finish);
	graph.push_back(vertexFn);

	if (!readingGraph(graph, finish, weightPath)) {

	//считывание
		if (weightPath) {
			std::cout << "ПОЛУЧЕННЫЙ ПУТЬ: " << start << "\nДлина пути = "
<< weightPath << "\n";
		}

		else {
			std::cout << "Запуск A*.\n";
			if (findPathAstar(graph, minPath, finish, start, weightPath))
{
	//алгоритм построения кратчайшего пути
				buildPath(vertexSt.getName(), finish, answer, minPath);
												//строим путь по
карте и печатаем
				std::cout << "\n\nДлина пути = " << weightPath << "\n";
			}
			else {
				std::cout << "Граф введен неправильно.\n";

	//предупреждение ошибок
			}
		}

	}
	else {
		std::cout << "Граф введен неправильно.\n";
												//если переход к
финишу не возможен ни через какое ребро
	}

	return 0;
     }
```