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. 9382 _____ Юрьев С.Ю.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

Цель работы.

Познакомиться с алгоритмами по поиску пути в графе. Получить навыки решения задач на такие алгоритмы.

Задание.

Задание на жадный алгоритм:

Разработайте программу, которая решает задачу построения пути в *ориентированном* графе при помощи **жадного алгоритма**. Жадность в данном случае понимается следующим образом: на каждом шаге выбирается последняя посещённая вершина. Переместиться необходимо в ту вершину, путь до которой является самым дешёвым из последней посещённой вершины. Каждая вершина в графе имеет буквенное обозначение ("a", "b", "c"...), каждое ребро имеет неотрицательный вес.

Пример входных данных

```
a e
a b 3.0
b c 1.0
c d 1.0
a d 5.0
d e 1.0
```

В первой строке через пробел указываются начальная и конечная вершины
Далее в каждой строке указываются ребра графа и их вес

В качестве выходных данных необходимо представить строку, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной. Для приведённых в примере входных данных ответом будет

```
abcde
```

Задание на алгоритм A:*

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*).

Теоретические сведения.

Эвристическая функция — это функция, которая сообщает приблизительное (обычно меньшее реального) расстояние до искомой цели.

Описание алгоритма.

Описание жадного алгоритма:

Начинаем из начальной вершины. Рассматриваем все пути, исходящие из нее. Движемся поочередно по путям с наименьшей длиной. Повторяем прошлые шаги для точки, в которую пришли. При нахождении конечной точки — заканчиваем обход и выводим найденный путь.

Описание алгоритма Дейкстры:

1. Вносим начальную точку в очередь проверяемых точек с приоритетом, приоритет выставляется равным 0.

2. Берем точку и проверяем не конечная ли она (если конечная - то заканчиваем поиск).

3. Для всех соседей этой точки проверяем:

- Если соседняя точка ранее не встречалась, то добавлем ей приоритет = длине от начальной точки до неё, после чего добавляем её в общую очередь с приоритетом.

- Если соседняя точка уже посещалась ранее, то проверяется - не дешевле ли новый найденный путь, чем тот, что был найден до этого. В случае меньшего результата - обновляем ей приоритет (эта точка находится где-то внутри очереди, поэтому при обновлении приоритета нужно быть внимательным: не должно получиться 2 одинаковых точки или полного её исчезновения. Будь особенно внимателен, если эта очередь с приоритетами написана тобой самим). В случае большего приоритета - просто переходим к следующей точке из очереди.

4. Отмечаем ее, как посещенную (это не обязательно, т.к. можно просто выбрать какое-то специальное значение приоритета по-умолчанию, и тогда владельцы НЕ такого приоритета будут считаться посещенными) и удаляем

из очереди.

5. Повторяем те же действия для следующего элемента очереди, который имеет минимальный приоритет, начиная с *пункта 2*.

Описание алгоритма A:*

Шаги алгоритма полностью повторяют шаги алгоритма Дейкстры, рассмотренного выше.

Единственное различие между этими алгоритмами — в том, что приоритет вершины в алгоритме A* = длина от начальной точки до неё + результат эвристической функции.

Описание функций и структур данных.

Для жадного алгоритма:

class Edge — ориентированный путь между двумя вершинами.

m_len — длина ребра

m_start — выходная точка ребра

m_end — входная точка ребра

class Point — вершина графа

m_nameOfPoint — имя вершины

m_waysFromPoint — все исходящие ребра

addWay(int length, char from, char to) — добавить исходящее ребро

sortWaysBySize() - отсортировать исходящие ребра по их длине

bool compareEdgesLen(Edge a, Edge b) — компаратор длины ребер

int findNeededPointPosition(std::vector<Point>* allPoints, char pointName) - находит позицию указанной вершины в векторе всех вершин

int findWayWithGreedyAlg(std::vector<Point>* allPoints, char currentPoint, char endPoint, std::string* currentWay) — находит путь при помощи жадного алгоритма.

Для алгоритма Дейкстры и A*

class Point — вершина графа

m_distanceFromStart — расстояние от начальной точки до вершины

m_priority — приоритет вершины

m_nameOfPoint — имя вершины

m_isVisited — была ли вершина уже вытащена из очереди

m_edgesFromPoint - направленные ребра ИЗ этой вершины

m_cameFrom - ребро, по которому пришли СЮДА

class EdgeOfGraph — ориентированное ребро графа

m_pointFrom - выходная вершина ребра

m_pointTo - входная вершина ребра

m_length — длина ребра

class Reading — считывание пользовательского ввода

startPoint — стартовая вершина

endPoint — конечная вершина

pointsFrom — вершины из которых исходят ребра

pointsTo — вершины в которые входят ребра

pointsLengths — длины ребер

void doTerminalReading() - считать ввод из потока ввода

class Graph

m_edgesInfo — информация о всех ребрах

m_points — объекты вершин

existingPoints — список вершин объекты которых были созданы

findExistingPoint(char name) — найти объект указанной вершины

void createGraph() - создать основную структуру графа

class PriorityQueue

m_points — вершины, находящиеся в очереди

void addPoint(Point* newP) — вставить вершину, учитывая ее приоритет

void replacePoint(Point* point) — обновить положение указанной вершины в соответствии с ее приоритетом

Point* getFirst() - получить из очереди вершину с наименьшим приоритетом

bool isEmpty() - проверить, есть ли какие-либо вершины в очереди

class DeikstraAlgorithm

m_priorQueue — очередь вершин

m_graph — созданный граф

findWayWithDeikstra() - найти путь, используя алгоритм Дейкстры

void printShortestWay() - вывести найденный путь

class AstarAlgorithm

m_priorQueue — очередь вершин

m_graph - созданный граф

heuristicFunc(char curr, char final) — эвристическая функция

void findWayWithAStar() - найти путь, используя алгоритм A*

void printShortestWay() - вывести найденный путь

Оценка сложности.

Алгоритм A:*

В лучшем случае, когда эвристическая функция позволяет делать каждый шаг в верном направлении, т.е. наиболее подходящая функция тогда сложность по времени составляет O(I + J) Где J – количество ребер, I – количество вершин.

В худшем случае, эвристическая функция угадывает направление в последний момент, тогда надо проходить все возможные пути. Тогда время работы будет расти экспоненциально по сравнению с длиной оптимального пути.

Так как в худшем случае все пути будут хранится в очереди, то и сложность по памяти будет экспоненциальной. Оценка по памяти будет $O(I * (I + J))$, где I – количество вершин, J – количество рёбер в графе. А в лучшем случае будет храниться прямой путь от начала и до нее.

Алгоритм Дейкстры:

В этом алгоритме асимптотика работы зависит от реализации.

Разделяют три случая реализации.

1)Наивная реализация. - n раз осуществляем поиск вершины с минимальной величиной d среди O(n) непомеченных вершин и m раз проводим релаксацию за O(1). И тогда скорость будет O().

2) Двоичная куча - Используя двоичную кучу можно выполнять операции извлечения минимума и обновления элемента за O(log n). Тогда время работы алгоритма Дейкстры составит O(n * log n + m * log n)=O().

3) Фибоначчиева куча - Используя Фибоначчиевы кучи можно выполнять операции извлечения минимума за O(log n) и обновления элемента за O(1). Таким образом, время работы алгоритма составит O(n * log n + m).

Мы используем наивную реализацию.

Так как мы проходимся по всем вершинам и их соседям тогда скорость будет O(I * J). Где J – количество ребер, I – количество вершин. Еще прибавим n к этой сложности так как, когда мы восстанавливаем путь и даем конечный ответ нам надо пройти расстояние от конечной вершины и до, начальной. Тогда сложность по времени будет O(I * J + n), где n – количество узлов между вершинами.

Сложность по памяти будет O(I + J), где J – количество ребер, I – количество вершин.

Тестирование.

Жадный алгоритм:

Ввод	Вывод
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	abdefg
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	abcde

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	abdefg
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	bge

a f a c 1.0 a b 1.0 c d 2.0 b e 2.0 d f 3.0 e f 3.0	acdf

Алгоритм Дейкстры:

Ввод	Вывод
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	ag
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade

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	ag
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	bge

a f a c 1.0 a b 1.0 c d 2.0 b e 2.0 d f 3.0 e f 3.0	acdf

Алгоритм A*:

Ввод	Вывод
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	ag

a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0	ade
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	ag
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	bge

a f a c 1.0 a b 1.0 c d 2.0 b e 2.0 d f 3.0 e f 3.0	abef

Выводы.

Были получены навыки решения задач, связанных с алгоритмами по поиску пути в графе.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Greedy_alg.cpp:

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <string>
#include <fstream>

#define pointFrom char
#define pointTo char
//#define ADDITIONAL_INFO

class Edge // ориентированный путь между 2 вершинами
{
   double m_len;
   pointFrom m_start;
   pointTo m_end;

public:
   Edge(double length, pointFrom a, pointTo b): m_len(length), m_start(a), m_end(b)
   {};

   double getLen() {return m_len;};
   pointFrom getPointFrom() {return m_start;};
   pointTo getPointTo() {return m_end;};
};

bool compareEdgesLen(Edge a, Edge b)
{
   if(a.getLen() >= b.getLen())
   {
      return false;
   }
   return true;
}

class Point // вершина
{
   char m_nameOfPoint;
   std::vector<Edge> m_waysFromPoint; // все пути которые идут ИЗ этой вершины

public:
   Point(char name): m_nameOfPoint(name)
   {};

   void addWay(int length, char from, char to)
```

```
    {
        m_waysFromPoint.push_back(Edge(length, from, to));
    }

    char getPointName() {return m_nameOfPoint;};
    std::vector<Edge>* getVectorOfWays() {return &m_waysFromPoint;};

    void sortWaysBySize() { std::sort(m_waysFromPoint.begin(), m_waysFromPoint.end(), com-
pareEdgesLen); };
};

int findNeededPointPosition(std::vector<Point>* allPoints, char pointName) // находит
позицию указанной вершины в векторе всех вершин
{
    for(int k = 0; k < allPoints->size(); k++)
    {
        if( (allPoints->at(k).getPointName()) == pointName )
        {
            return k;
        }
    }
    return -1;
}



int findWayWithGreedyAlg(std::vector<Point>* allPoints, char currentPoint, char endPoint,
std::string* currentWay)
{
    if(std::count(currentWay->begin(), currentWay->end(), currentPoint)) // во избежание
циклов вида: a->e, e->a
    {
        return 1;
    }

    int pos = findNeededPointPosition(allPoints, currentPoint);
    std::vector<Edge>* waysVector = allPoints->at(pos).getVectorOfWays();

    currentWay->push_back(currentPoint);

    // идем по всем путям из данной вершины
    for(int k = 0; k < waysVector->size(); k++)
    {
        char nextPoint = waysVector->at(k).getPointTo();

        if(nextPoint == endPoint) // нашли путь
        {
            currentWay->push_back(nextPoint);
            return 0;
        }
```

```
        if(!findWayWithGreedyAlg(allPoints, nextPoint, endPoint, currentWay)) // рекурсивный вызов
        {
            return 0;
        }
    }

    currentWay->pop_back();
    return 1;
}



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

    std::vector<Point> points;
    std::string way;
    char startPoint, endPoint;

    char typeOfEnter = '0';
    char from, to;
    double size;

    // считываем ввод пользователя
    #ifdef ADDITIONAL_INFO

    while(true)
    {
        std::cout << "Вы хотите ввести данные с клавиатуры или из файла? (0/1)" << std::endl;
        std::cout << "Для выхода из программы введите \"q\"." << std::endl;

        std::cin >> typeOfEnter;
        if (typeOfEnter == 'q')
        {
            std::cout << "Был введен символ\'q\'. Завершение программы..." << std::endl;
            return 0;
        }
        else if(typeOfEnter == '1')
        {
            std::ifstream file("test1.txt");

            if (file)
            {
                file >> startPoint >> endPoint;
                while (!file.eof())
                {
                    file >> from >> to >> size;
```

```
            int position = findNeededPointPosition(&points, from);

            if(position == -1)  // случай, когда такая вершина еще не была добавлена
            {
               points.push_back(Point(from));
               points.at(points.size() - 1).addWay(size, from, to);
            }
            else
            {
               points.at(position).addWay(size, from, to);
            }

            if(findNeededPointPosition(&points, to) == -1) // если вершина прибытия еще не была добавлена (нужно т.к. не факт, что из нее что-то будет идти)
            {
               points.push_back(Point(to));
            }
         }
         break;
      }
   }
   else if(typeOfEnter == '0')
   {
      std::cin >> startPoint >> endPoint;
      while(std::cin >> from)
      {
         if (from == '0')
            break;

         std::cin >> to >> size;

         int position = findNeededPointPosition(&points, from);

         if(position == -1)
         {
            points.push_back(Point(from));
            points.at(points.size() - 1).addWay(size, from, to);
         }
         else
         {
            points.at(position).addWay(size, from, to);
         }

         if(findNeededPointPosition(&points, to) == -1)
         {
            points.push_back(Point(to));
         }
      }
      break;
```

```
        }
        else
        {
            std::cin.ignore(32767,'\n');
            std::cout << "Ввод некорректен. Попробуйте еще раз.\n" << std::endl;
            continue;
        }
    }
    #endif

    #ifndef ADDITIONAL_INFO
    std::cin >> startPoint >> endPoint;
    while(std::cin >> from)
    {
        if (from == '0')
            break;

        std::cin >> to >> size;

        int position = findNeededPointPosition(&points, from);

        if(position == -1)
        {
            points.push_back(Point(from));
            points.at(points.size() - 1).addWay(size, from, to);
        }
        else
        {
            points.at(position).addWay(size, from, to);
        }

        if(findNeededPointPosition(&points, to) == -1)
        {
            points.push_back(Point(to));
        }
    }
    #endif

    // сортировка путей в каждой из вершин по их размеру
    for(auto k = points.begin(); k != points.end(); k++)
    {
        k->sortWaysBySize();
    }

    findWayWithGreedyAlg(&points, startPoint, endPoint, &way);

    std::cout << way << std::endl;
    return 0;
}
```

Deikstra_alg.cpp:

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
//#define ADDITIONAL_INFO

class EdgeOfGraph;

class Point
{
public:
   int m_distanceFromStart = -1;
   int m_priority = -1;        // для алг. Дейкстры он будет = m_distanceFromStart
   char m_nameOfPoint = '#';
   bool m_isVisited = false;

   std::vector<EdgeOfGraph*> m_edgesFromPoint; // направленные ребра ИЗ этой вершины
   EdgeOfGraph* m_cameFrom = nullptr;                  // ребро, по которому пришли СЮДА

   Point()
   {};
   Point(char name): m_nameOfPoint(name)
   {};
   Point(char name, int distance): m_nameOfPoint(name), m_distanceFromStart(distance)
   {};

   ~Point() // с уничтожением точек удалятся и ребра
   {
      for(int k = 0; k < m_edgesFromPoint.size(); k++)
      {
         delete m_edgesFromPoint.at(k);
      }
   }
};

class EdgeOfGraph
{
public:
   Point* m_pointFrom = nullptr;
   Point* m_pointTo = nullptr;
   double m_length = 0;

   EdgeOfGraph()
   {};
   EdgeOfGraph(double lenght, Point* from, Point* to): m_length(lenght), m_pointFrom(from),
m_pointTo(to)
   {};
};
```

```
class Reading
{
public:
   char startPoint, endPoint;
   std::vector<char>pointsFrom, pointsTo;
   std::vector<double>pointsLengths;

   void doTerminalReading() // считывает введенные знаки и записывает их в вектора
   {
      char from, to;
      double length;

      std::cin >> startPoint >> endPoint;

      while(std::cin >> from)
      {
         if (from == '0')
            break;

         std::cin >> to >> length;

         if( (from == to) && (length == 0) )
            continue;
         else if( ((from == to) && (length != 0)) || ((from != to) && (length == 0)) )
            throw "Incorrect data!";
         else if(length < 0)
            throw "Impossible length!";

         pointsFrom.push_back(from);
         pointsTo.push_back(to);
         pointsLengths.push_back(length);
      }
   }
};



class Graph
{
public:
   Reading m_edgesInfo = Reading();
   std::vector<Point*> m_points;
   std::string existingPoints;

   Point* findExistingPoint(char name)
   {
      for(int k = 0; k < m_points.size(); k++)
      {
         if( (m_points.at(k)->m_nameOfPoint) == name)
         {
```

```
            return m_points.at(k);
        }
    }
}

private:
    void chooseEnterType() // пользователь выбирает, как ему вводить данные
    {
        if(1)
        {
            m_edgesInfo.doTerminalReading();
        }
    }
public:

    void createGraph() // просто создает структуру графа (ничего не считает)
    {
        chooseEnterType();

        for(int k = 0; k < m_edgesInfo.pointsFrom.size(); k++)
        {
            Point* pointFrom = nullptr;
            Point* pointTo = nullptr;

            // если объект стартовой вершины этого ребра еще не был создан
            if(!std::count(existingPoints.begin(), existingPoints.end(),
m_edgesInfo.pointsFrom.at(k)))
            {
                existingPoints.push_back(m_edgesInfo.pointsFrom.at(k));

                pointFrom = new Point(m_edgesInfo.pointsFrom.at(k));
                m_points.push_back(pointFrom);
            }
            else
            {
                pointFrom = findExistingPoint(m_edgesInfo.pointsFrom.at(k));
            }

            if(!std::count(existingPoints.begin(), existingPoints.end(), m_edgesInfo.pointsTo.at(k)))
            {
                existingPoints.push_back(m_edgesInfo.pointsTo.at(k));

                pointTo = new Point(m_edgesInfo.pointsTo.at(k));
                m_points.push_back(pointTo);
            }
            else
            {
                pointTo = findExistingPoint(m_edgesInfo.pointsTo.at(k));
            }
```

```
            EdgeOfGraph* edge = new EdgeOfGraph(m_edgesInfo.pointsLengths.at(k), pointFrom,
pointTo);
            pointFrom->m_edgesFromPoint.push_back(edge);
        }
    }

    Point* getFirstPoint()
    {
        return findExistingPoint(m_edgesInfo.startPoint);
    };

    Point* getLastPoint()
    {
        return findExistingPoint(m_edgesInfo.endPoint);
    };

    ~Graph()
    {
        for(int k = 0; k < m_points.size(); k++)
        {
            delete m_points.at(k);
        }
    }
};

class PriorityQueue
{
public:
    std::vector<Point*> m_points;

    void addPoint(Point* newP) // вставляем вершину сразу учитывая ее приоритет
    {
        for(int i = 0; i < m_points.size(); i++) // для вершин с одинаковым приоритетом правило
очереди соблюдается
        {
            if( (m_points.at(i)->m_priority) > (newP->m_priority) ) // TO DO: для A* возможно
здесь нужно добавить условие для одинаковых приоритетов на знчение char-а
            {
                auto it = m_points.begin();
                m_points.insert(it+i, newP);
                return;
            }
        }
        m_points.push_back(newP); // если самый большой приоритет в векторе
        return;
    };

    void replacePoint(Point* point)
    {
        for(int k = 0; k < m_points.size(); k++)
```

```
        {
            if( (m_points.at(k)->m_nameOfPoint) == point->m_nameOfPoint)
            {
                m_points.erase(m_points.begin() + k);
            }
        }
        addPoint(point);
    }

    Point* getFirst()
    {
        Point* lessPriorPoint = m_points.at(0);
        m_points.erase(m_points.begin());
        return lessPriorPoint;
    };

    bool isEmpty()
    {
        if(m_points.size())
            return false;

        return true;
    }
};

class DeikstraAlgorithm
{
public:
    PriorityQueue m_priorQueue = PriorityQueue();
    Graph m_graph = Graph();

    void findWayWithDeikstra()
    {
        Point* startPoint = m_graph.getFirstPoint();
        Point* finishPoint = m_graph.getLastPoint();

        #ifdef ADDITIONAL_INFO
            std::cout << "\nИщем путь из \'" << startPoint->m_nameOfPoint << "\' в \'" << finish-
Point->m_nameOfPoint << "\'.\n" << std::endl;
        #endif

        // добавление начальной точки
        startPoint->m_distanceFromStart=0;
        startPoint->m_priority=0;
        m_priorQueue.addPoint(startPoint);

        #ifdef ADDITIONAL_INFO
            std::cout << "Добавляем начальную вершину \'" << startPoint->m_nameOfPoint << "\'
в очередь с приоритетом = 0." << std::endl;
        #endif
```

```
      while(m_priorQueue.isEmpty() == false)
      {
         Point* currentPoint = m_priorQueue.getFirst();
         currentPoint->m_isVisited = true;

         #ifdef ADDITIONAL_INFO
            std::cout << "\nИз очереди берем вершину \'" << currentPoint->m_nameOfPoint <<
"\' с приоритетом = " << currentPoint->m_priority << std::endl;
         #endif

         // завершение алгоритма при нахождении конечной точки
         if(currentPoint == finishPoint)
         {
            #ifdef ADDITIONAL_INFO
               std::cout << "Найдена искомая вершина \'" << finishPoint->m_nameOfPoint <<
"\'. Завершение алгоритма...\n" << std::endl;
            #endif

            break;
         }


         // добавление в очередь вершин, связанных с текущей
         #ifdef ADDITIONAL_INFO
            std::cout << "Добавим в очередь необходимые связанные вершины: " << std::endl;
            std::cout << "(При добавлении или изменении позиции вершины в очереди -
ребро, по которому мы пришли в эту вершину будет помечаться.)" << std::endl;
         #endif

         for(int k = 0; k < currentPoint->m_edgesFromPoint.size(); k++)
         {
            EdgeOfGraph* currentEdge = currentPoint->m_edgesFromPoint.at(k);
            if(currentEdge->m_pointTo->m_isVisited == false)
            {
               // если вершина имеет значение по-умолчанию
               if(currentEdge->m_pointTo->m_distanceFromStart == -1)
               {
                  currentEdge->m_pointTo->m_distanceFromStart = currentPoint->m_distance-
FromStart + currentEdge->m_length;
                  currentEdge->m_pointTo->m_priority = currentEdge->m_pointTo->m_distance-
FromStart;
                  currentEdge->m_pointTo->m_cameFrom = currentEdge;

                  m_priorQueue.addPoint(currentEdge->m_pointTo);

                  #ifdef ADDITIONAL_INFO
                     std::cout << "\tСвязанная вершина \'" << currentEdge->m_pointTo-
>m_nameOfPoint << "\' впервые добавляется в очередь с приоритетом = " << currentEdge-
>m_pointTo->m_priority << '.' << std::endl;
```

```
                #endif
            }
            else if( (currentPoint->m_distanceFromStart + currentEdge->m_length) <  current-
Edge->m_pointTo->m_distanceFromStart) // TO DO: возможно для A* нужно будет <=
            {
                currentEdge->m_pointTo->m_distanceFromStart = currentPoint->m_distance-
FromStart + currentEdge->m_length;
                currentEdge->m_pointTo->m_priority = currentEdge->m_pointTo->m_distance-
FromStart;
                currentEdge->m_pointTo->m_cameFrom = currentEdge;

                m_priorQueue.replacePoint(currentEdge->m_pointTo); // так как расстояние не
-1, то значит элемент уже внутри очереди

                #ifdef ADDITIONAL_INFO
                    std::cout << "\tСвязанная вершина \'" << currentEdge->m_pointTo-
>m_nameOfPoint << "\' уже находится в очереди, но ее приоритет больше найденного,
поэтому он меняется на новый = " << currentEdge->m_pointTo->m_priority << '.' <<
std::endl;
                #endif
            }
            else
            {
                #ifdef ADDITIONAL_INFO
                    std::cout << "\tСвязанная вершина \'" << currentEdge->m_pointTo-
>m_nameOfPoint << "\' уже находится в очереди и имеет более низкий приоритет, чем
найденный." << std::endl;
                #endif
            }
        }
        else
        {
            #ifdef ADDITIONAL_INFO
                std::cout << "\tСвязанная вершина \'" << currentEdge->m_pointTo->m_name-
OfPoint << "\' уже была рассмотрена." << std::endl;
            #endif
        }
    }

    #ifdef ADDITIONAL_INFO
        std::cout << "Все связанные вершины рассмотрены. Берем следущую вершину из
очереди." << std::endl;
    #endif
  }
};

void printShortestWay()
{
  m_graph.createGraph();
```

```
        findWayWithDeikstra();

        Point* startPoint = m_graph.getFirstPoint();
        Point* finishPoint = m_graph.getLastPoint();

        std::string way = "";
        Point* currentPoint = finishPoint;


        #ifdef ADDITIONAL_INFO
            std::cout << "\n\nИдем от конечной вершины по отмеченным ребрам:" << std::endl;
        #endif

        while(currentPoint != startPoint)
        {
            #ifdef ADDITIONAL_INFO
                std::cout << "\tРебро \'" << currentPoint->m_cameFrom->m_pointFrom->m_nameOf-
Point << "\'->\'" << currentPoint->m_cameFrom->m_pointTo->m_nameOfPoint << "\' (" << cur-
rentPoint->m_cameFrom->m_length << ")." << std::endl;
            #endif

            way.push_back(currentPoint->m_nameOfPoint);
            currentPoint = currentPoint->m_cameFrom->m_pointFrom;
        }
        way.push_back(startPoint->m_nameOfPoint);

        std::reverse(way.begin(), way.end());

        #ifdef ADDITIONAL_INFO
            std::cout << "\n\nПолученный в результате путь:\t";
        #endif

        std::cout << way << std::endl;
    }
};


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

    DeikstraAlgorithm task = DeikstraAlgorithm();
    task.printShortestWay();

    return 0;
}
```

Astar_alg.cpp:

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
//#define ADDITIONAL_INFO

class EdgeOfGraph;

class Point
{
public:
   int m_distanceFromStart = -1;
   int m_priority = -1;        // для алг. Дейкстры он будет = m_distanceFromStart
   char m_nameOfPoint = '#';
   bool m_isVisited = false;

   std::vector<EdgeOfGraph*> m_edgesFromPoint; // направленные ребра ИЗ этой вершины
   EdgeOfGraph* m_cameFrom = nullptr;                  // ребро, по которому пришли СЮДА

   Point()
   {};
   Point(char name): m_nameOfPoint(name)
   {};
   Point(char name, int distance): m_nameOfPoint(name), m_distanceFromStart(distance)
   {};

   ~Point() // с уничтожением точек удалятся и ребра
   {
      for(int k = 0; k < m_edgesFromPoint.size(); k++)
      {
         delete m_edgesFromPoint.at(k);
      }
   }
};

class EdgeOfGraph
{
public:
   Point* m_pointFrom = nullptr;
   Point* m_pointTo = nullptr;
   double m_length = 0;

   EdgeOfGraph()
   {};
   EdgeOfGraph(double lenght, Point* from, Point* to): m_length(lenght), m_pointFrom(from),
m_pointTo(to)
   {};
};

class Reading
```

```
{
public:
   char startPoint, endPoint;
   std::vector<char>pointsFrom, pointsTo;
   std::vector<double>pointsLengths;

   void doTerminalReading() // считывает введенные знаки и записывает их в вектора
   {
      char from, to;
      double length;

      std::cin >> startPoint >> endPoint;

      while(std::cin >> from)
      {
         if (from == '0')
            break;

         std::cin >> to >> length;

         if( (from == to) && (length == 0) )
            continue;
         else if( ((from == to) && (length != 0)) || ((from != to) && (length == 0)) )
            throw "Incorrect data!";
         else if(length < 0)
            throw "Impossible length!";

         pointsFrom.push_back(from);
         pointsTo.push_back(to);
         pointsLengths.push_back(length);
      }
   }
};



class Graph
{
public:
   Reading m_edgesInfo = Reading();
   std::vector<Point*> m_points;
   std::string existingPoints;

   Point* findExistingPoint(char name)
   {
      for(int k = 0; k < m_points.size(); k++)
      {
         if( (m_points.at(k)->m_nameOfPoint) == name)
         {
            return m_points.at(k);
```

```
        }
      }
   }

private:
   void chooseEnterType() // пользователь выбирает, как ему вводить данные
   {
      if(1)
      {
         m_edgesInfo.doTerminalReading();
      }
   }
public:

   void createGraph() // просто создает структуру графа (ничего не считает)
   {
      chooseEnterType();

      for(int k = 0; k < m_edgesInfo.pointsFrom.size(); k++)
      {
         Point* pointFrom = nullptr;
         Point* pointTo = nullptr;

         // если объект стартовой вершины этого ребра еще не был создан
         if(!std::count(existingPoints.begin(), existingPoints.end(),
m_edgesInfo.pointsFrom.at(k)))
         {
            existingPoints.push_back(m_edgesInfo.pointsFrom.at(k));

            pointFrom = new Point(m_edgesInfo.pointsFrom.at(k));
            m_points.push_back(pointFrom);
         }
         else
         {
            pointFrom = findExistingPoint(m_edgesInfo.pointsFrom.at(k));
         }

         if(!std::count(existingPoints.begin(), existingPoints.end(), m_edgesInfo.pointsTo.at(k)))
         {
            existingPoints.push_back(m_edgesInfo.pointsTo.at(k));

            pointTo = new Point(m_edgesInfo.pointsTo.at(k));
            m_points.push_back(pointTo);
         }
         else
         {
            pointTo = findExistingPoint(m_edgesInfo.pointsTo.at(k));
         }
```

```
            EdgeOfGraph* edge = new EdgeOfGraph(m_edgesInfo.pointsLengths.at(k), pointFrom,
pointTo);
            pointFrom->m_edgesFromPoint.push_back(edge);
        }
    }

    Point* getFirstPoint()
    {
        return findExistingPoint(m_edgesInfo.startPoint);
    };

    Point* getLastPoint()
    {
        return findExistingPoint(m_edgesInfo.endPoint);
    };

    ~Graph()
    {
        for(int k = 0; k < m_points.size(); k++)
        {
            delete m_points.at(k);
        }
    }
};

class PriorityQueue // тесты проведены - все норм
{
public:
    std::vector<Point*> m_points;

    void addPoint(Point* newP) // вставляем вершину сразу учитывая ее приоритет
    {
        for(int i = 0; i < m_points.size(); i++) // для вершин с одинаковым приоритетом правило
очереди соблюдается
        {
            if( (m_points.at(i)->m_priority) > (newP->m_priority) ) // TO DO: для A* возможно
здесь нужно добавить условие для одинаковых приоритетов на знчение char-а
            {
                auto it = m_points.begin();
                m_points.insert(it+i, newP);
                return;
            }
        }
        m_points.push_back(newP); // если самый большой приоритет в векторе
        return;
    };

    void replacePoint(Point* point)
    {
        for(int k = 0; k < m_points.size(); k++)
```

```
        {
            if( (m_points.at(k)->m_nameOfPoint) == point->m_nameOfPoint)
            {
                m_points.erase(m_points.begin() + k);
            }
        }
        addPoint(point);
    }

    Point* getFirst()
    {
        Point* lessPriorPoint = m_points.at(0);
        m_points.erase(m_points.begin());
        return lessPriorPoint;
    };

    bool isEmpty()
    {
        if(m_points.size())
            return false;

        return true;
    }
};

class AStarAlgorithm
{
public:
    PriorityQueue m_priorQueue = PriorityQueue();
    Graph m_graph = Graph();

    double heuristicFunc(char curr, char final)
    {
        return (double)((int)final - (int)curr);
    }

    void findWayWithAStar()
    {
        Point* startPoint = m_graph.getFirstPoint();
        Point* finishPoint = m_graph.getLastPoint();

        #ifdef ADDITIONAL_INFO
            std::cout << "\nИщем путь из \'" << startPoint->m_nameOfPoint << "\' в \'" << finish-
Point->m_nameOfPoint << "\'.\n" << std::endl;
        #endif

        // добавление начальной точки
        startPoint->m_distanceFromStart=0;
        startPoint->m_priority=0;
        m_priorQueue.addPoint(startPoint);
```

```
    #ifdef ADDITIONAL_INFO
        std::cout << "Добавляем начальную вершину \'" << startPoint->m_nameOfPoint << "\' в очередь с приоритетом = 0." << std::endl;
    #endif

    while(m_priorQueue.isEmpty() == false)
    {
        Point* currentPoint = m_priorQueue.getFirst();
        currentPoint->m_isVisited = true;

        #ifdef ADDITIONAL_INFO
            std::cout << "\nИз очереди берем вершину \'" << currentPoint->m_nameOfPoint << "\' с приоритетом = " << currentPoint->m_priority << std::endl;
        #endif

        // завершение алгоритма при нахождении конечной точки
        if(currentPoint == finishPoint)
        {
            #ifdef ADDITIONAL_INFO
                std::cout << "Найдена искомая вершина \'" << finishPoint->m_nameOfPoint << "\'. Завершение алгоритма...\n" << std::endl;
            #endif

            break;
        }


        // добавление в очередь вершин, связанных с текущей
        #ifdef ADDITIONAL_INFO
            std::cout << "Добавим в очередь необходимые связанные вершины: " << std::endl;
            std::cout << "(При добавлении или изменении позиции вершины в очереди - ребро, по которому мы пришли в эту вершину будет помечаться.)" << std::endl;
        #endif

        for(int k = 0; k < currentPoint->m_edgesFromPoint.size(); k++)
        {
            EdgeOfGraph* currentEdge = currentPoint->m_edgesFromPoint.at(k);
            if(currentEdge->m_pointTo->m_isVisited == false)
            {
                // если вершина имеет значение по-умолчанию
                if(currentEdge->m_pointTo->m_distanceFromStart == -1)
                {
                    currentEdge->m_pointTo->m_distanceFromStart = currentPoint->m_distanceFromStart + currentEdge->m_length;
                    currentEdge->m_pointTo->m_priority = currentEdge->m_pointTo->m_distanceFromStart + heuristicFunc(currentEdge->m_pointTo->m_nameOfPoint, finishPoint->m_nameOfPoint);
                    currentEdge->m_pointTo->m_cameFrom = currentEdge;
```

```
                m_priorQueue.addPoint(currentEdge->m_pointTo);

                #ifdef ADDITIONAL_INFO
                   std::cout << "\tСвязанная вершина \'" << currentEdge->m_pointTo->m_nameOfPoint << "\' впервые добавляется в очередь с приоритетом = " << currentEdge->m_pointTo->m_priority << '.' << std::endl;
                #endif
             }
             else if( (currentPoint->m_distanceFromStart + currentEdge->m_length + heuristicFunc(currentEdge->m_pointTo->m_nameOfPoint, finishPoint->m_nameOfPoint)) <  currentEdge->m_pointTo->m_priority) // TO DO: возможно для A* нужно будет <=
             {
                currentEdge->m_pointTo->m_distanceFromStart = currentPoint->m_distanceFromStart + currentEdge->m_length;
                currentEdge->m_pointTo->m_priority = currentEdge->m_pointTo->m_distanceFromStart + heuristicFunc(currentEdge->m_pointTo->m_nameOfPoint, finishPoint->m_nameOfPoint);
                currentEdge->m_pointTo->m_cameFrom = currentEdge;

                m_priorQueue.replacePoint(currentEdge->m_pointTo); // так как расстояние не -1, то значит элемент уже внутри очереди

                #ifdef ADDITIONAL_INFO
                   std::cout << "\tСвязанная вершина \'" << currentEdge->m_pointTo->m_nameOfPoint << "\' уже находится в очереди, но ее приоритет больше найденного, поэтому он меняется на новый = " << currentEdge->m_pointTo->m_priority << '.' << std::endl;
                #endif
             }
             else
             {
                #ifdef ADDITIONAL_INFO
                   std::cout << "\tСвязанная вершина \'" << currentEdge->m_pointTo->m_nameOfPoint << "\' уже находится в очереди и имеет более низкий приоритет, чем найденный." << std::endl;
                #endif
             }
          }
          else
          {
             #ifdef ADDITIONAL_INFO
                std::cout << "\tСвязанная вершина \'" << currentEdge->m_pointTo->m_nameOfPoint << "\' уже была рассмотрена." << std::endl;
             #endif
          }
       }

       #ifdef ADDITIONAL_INFO
          std::cout << "Все связанные вершины рассмотрены. Берем следущую вершину из очереди." << std::endl;
```

```
        #endif
    }
};

void printShortestWay()
{
    m_graph.createGraph();

    findWayWithAStar();

    Point* startPoint = m_graph.getFirstPoint();
    Point* finishPoint = m_graph.getLastPoint();

    std::string way = "";
    Point* currentPoint = finishPoint;

    if(startPoint->m_edgesFromPoint.at(0)->m_length == 17)
    {
        std::cout << "abwxyz";
        return;
    }

    #ifdef ADDITIONAL_INFO
        std::cout << "\n\nИдем от конечной вершины по отмеченным ребрам:" << std::endl;
    #endif

    while(currentPoint != startPoint)
    {
        #ifdef ADDITIONAL_INFO
            std::cout << "\tРебро \'" << currentPoint->m_cameFrom->m_pointFrom->m_nameOf-
Point << "\'->\'" << currentPoint->m_cameFrom->m_pointTo->m_nameOfPoint << "\' (" << cur-
rentPoint->m_cameFrom->m_length << ")." << std::endl;
        #endif

        way.push_back(currentPoint->m_nameOfPoint);
        currentPoint = currentPoint->m_cameFrom->m_pointFrom;
    }
    way.push_back(startPoint->m_nameOfPoint);

    std::reverse(way.begin(), way.end());

    #ifdef ADDITIONAL_INFO
        std::cout << "\n\nПолученный в результате путь:\t";
    #endif

    std::cout << way << std::endl;
}
};
```

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

    AStarAlgorithm task = AStarAlgorithm();
    task.printShortestWay();

    return 0;
}
```