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

Студент гр. 8303 _____ Логинов Е.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Изучить и реализовать на языке программирования C++ жадный алгоритм поиска пути в графе и алгоритм A* поиска кратчайшего пути в графе между двумя заданными вершинами.

Формулировка задания.

Вар. 4. Модификация A* с двумя финишами (требуется найти путь до любого из двух).

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.

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

Соответствующие выходные данные для Жадного алгоритма

abcde

Соответствующие выходные данные для A*

ade

Соответствующие входные и выходные данные для модифицированного A* для двух финишей

a e f

a b 3.0

b c 1.0

c d 1.0

a d 4.0

d e 1.0

_____

way 0) ade, 1

way 1) abf, 4

Описание алгоритмов.

Жадный алгоритм:

Для начала работы жадного алгоритма список ребер сортируется по не убыванию их весов, сам поиск начинается из заданной вершины.

Текущая просматриваемая вершина добавляется в список просмотренных. В отсортированном списке рёбер выбирается первое (сортировка гарантирует, что это будет минимальное), которое начинается в просматриваемой вершине, если эта вершина не просмотрена, то текущей вершиной становится та, в которой заканчивается это ребро, если она уже просмотрена, то выбирается следующее ребро. В случае, если в какой-то момент из текущей вершины нет путей, то происходит откат на шаг назад, и в предыдущей вершине

выбирается другое ребро, если это возможно. Алгоритм заканчивает свою работу, когда текущей вершиной становится искомая, или когда были просмотрены все рёбра, которые начинаются из исходной вершины.

A*:

Поиск начинается из исходной вершины. В текущие возможные пути добавляются все рёбра из начальной вершины. Происходит выбор минимального пути, где учитывается эвристическая близость вершины к искомой (близость в таблице ASCII), если в выбранном пути последняя вершина уже была просмотрена, то этот путь удаляется из списка, и снова происходит выбор минимального пути. Выбираются из всех рёбер графа те, которые начинаются из последней вершины в этом пути. Эта вершина добавляется к этому пути, и новый путь заносится в список возможных путей, с увеличением стоимости, равной переходу по этому ребру.

Когда были выбраны все рёбра, которые начинаются из последней вершины в этом пути, то эта вершина добавляется в список просмотренных, а сам путь удаляется из списка возможных путей. Дальше снова происходит выбор минимального пути. Алгоритм заканчивает свою работу, когда достигается искомая вершина.

Алгоритм A* для двух финишей:

Принцип работы схож с обычной версией. В отличии от обычного A*, модифицированная версия не завершает свою работу сразу после того, как мы дошли до конечной вершины, а продолжает искать кратчайший путь для второй вершины, если такая была подана на вход.

N – количество ребер + количество вершин.

Сложность Жадного алгоритма по операциям: O(N logN + N)

Сложность алгоритмов A* по операциям: O (N^2)

Сложность алгоритмов по памяти: O (N)

Описание структур данных.

1.

```
struct edge //ребра графа
{
   char first; //первая вершина
   char second; //вторая вершина
   float weight; //вес ребра
};
```

Структура предназначена для хранения ребер графа с вершинами first, second и весом weight. Используется в Жадном алгоритме и алгоритме A*.

2.

```
class GraphG
{
private:
   vector <edge> graph; //контейнер для ребер
   vector <char> solutions; //контейнер для текущего пути
   vector <char> checkedpoint; //контейнер для просмотренных вершин
   char start; //начальная вершина
   char destination; //вершина назначения
};
```

Класс для осуществления работы Жадного алгоритма. Содержит контейнеры для хранения ребер, вершин и текущего пути, а также начальный и конечный символ.

3.

```
struct steps
{
        string path; // символьная строка, выражающая путь
        double length; // длина пути
};
```

Структура для хранения различных путей и их длин. Используется в алгоритме A*.

4.

```
class GraphA
{
   struct edge
   {
      char first; //первая вершина
      char second; //вторая вершина
      float weight; //вес ребра
   };

   struct steps
   {
      string path; //строка, содержащая путь
      float length; //длина пути
   };

private:
   vector <edge> graph; //контейнер для ребер
   vector <steps> solutions; //контейнер для текущего пути
   vector <char> checkedpoint; //контейнер для просмотренных вершин
   char start; //начальная вершина
   char destination; //вершина назначения
        };
```

Класс реализации алгоритма A*.

5.

```
class GraphAModify
{
   struct edge
   {
      char first; //первая вершина
      char second; //вторая вершина
      float weight; //вес ребра
   };

   struct steps
   {
      string path; //строка, содержащая путь
```

```
        double length; //длина пути
    };
private:
    vector <edge> graph; //контейнер для ребер
    vector <steps> solutions[2]; //контейнер для текущего пути
    vector <char> checkedpoint[2]; //контейнер для просмотренных вершин
    char start; //начальная вершина
    char destination[2]; //вершина назначения
    int found[2]; //массив, позволящий отслеживать кол-ко уже найденных вершин
```

Класс реализации модифицированного алгоритма A*. Отличается от обычного A* тем, что контейнеры solutions, checkedpoint и массив destionation хранят комбинации путей сразу для двух вершин. Массив found предназначен для отслеживания количества уже найденных вершин.

Описание методов Жадного алгоритма.

1.

GraphG()

Конструктор класса GraphG. Запускает процедуру считывания информации о графе от пользователя. Поступают данные о все ребрах графа. Каждое ребро добавляется в вектор graph. После считывания запускается сортировка контейнера ребер и запускается функция поиска пути.

2.

bool search(char elem)

Метод класса GraphG для поиска пути в графе. На вход получает исходную вершину. Функция рекурсивно проходит по графу на каждом шаге выбирая ребро с минимальным весом. Возвращает true если находит конечную вершину, возвращает false, если такая вершина недостижима. Информация о полученном пути хранится в векторе solutions.

3.

bool isCheck(char elem)

Функция класса GraphG для проверки является ли вершина просмотренной. Получает на вход вершину. Проводится поиск в векторе

checkedpoint и возвращается резудьтат.

4.

void result()

Функция класса GraphG для вывода результата поиска на экран.

Описание функций алгоритма A*.

1.

GraphA()

Конструктор класса GraphA. Запускает процедуру считывания информации о графе от пользователя. Поступают данные о все ребрах графа. Каждое ребро добавляется в вектор graph. После считывания в вектор solutions добавляется первая вершина.

2.

void search()

Функция класса GraphA для поиска пути в графе. Функция итеративно проходит по графу на каждом шаге выбирая ребро с минимальной эвристической функцией. Выводит на экран путь, если находит конечную вершину.

3.

int minInd()

Функция класса GraphA для нахождения ребра с минимальной эвристической функцией. Для выполнения своей задачи функция проходит по всем текущим путям в векторе solutions и возвращает индекс минимального элемента.

Описание функций модифицированного алгоритма A*.

1.

GraphAModify, ()

Метод работает аналогично с конструктором GraphA обычного A*.

Две вершины назначения записываются в массив destination[]. В случае, если на вход подается только одна вершина назначения, то второму элементу массива detination[] присваивается значение первой вершины. Это необходимо для корректного поиска даже с одной конечной вершиной.

2.

void search()

Метод работает аналогично с методом search () из GraphA алгоритма A*. В самом начале массив found[] из двух элементов инициализируются -1 и будут выполнять роль «флага». Далее, цикл while() будет рабоать до тех пор, пока не найдутся кратчайшие пути для обеих вершин.

3.

int minInd(int k)

Метод работает аналогично с методом minInd() из GraphA алгоритма A*. На вход принимает индекс текущей вершины, для которой ищется минимальный путь.

4.

bool isCheck(int k, char value)

Функция получает на вход одну из двух вершин назначения и текущею рассматриваемую вершину, после чего происходит проверка, не является ли данная вершина уже просмотренной. Если это так, функция возвращает true, вершина не рассматривается .

Способ хранения частичных решений.

Частичные решения, т.е. различные варианты путей в графе хранятся в векторе типа vector<steps> (оба варианта A*) и в векторе типа vector<char>(Жадный алгоритм).

Тестирование.

Жадный алгоритм	**A***	Модифицированный A*
a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0 ! abcde a d a b 1.0 b c 1.0 c a 1.0 a d 5.0 ! ad a l a b 1 a f 3 b c 5 b g 3 f g 4 c d 6 d m 1 g e 4 e h 1 e n 1 n m 2 g i 5 i j 6 i k 1 j l 5 m j 3 ! abgenmjl	a e a b 3.0 b c 1.0 c d 1.0 a d 5.0 d e 1.0 ! ade a d a b 1.0 b c 1.0 c a 1.0 a d 5.0 ! ad a l a b 1 a f 3 b c 5 b g 3 f g 4 c d 6 d m 1 g e 4 e h 1 e n 1 n m 2 g i 5 i j 6 i k 1 j l 5 m j 3 ! abgenmjl	a e f a b 3.0 b c 1.0 b f 1.0 c d 1.0 a d 5.0 d e 1.0 ! 0) ade, 6 1) abf, 4 a l f a b 1 a f 3 b c 5 b g 3 f g 4 c d 6 d m 1 g e 4 e h 1 e n 1 n m 2 g i 5 i j 6 i k 1 j l 5 m j 3 ! 0) abgenmjl, 19 1) af, 3

Выводы.

В ходе выполнения лабораторной работы были изучены и реализованы на языке программирования C++ жадный алгоритм поиска пути в графе и алгоритм A* поиска кратчайшего пути в графе между двумя заданными вершинами. Также алгоритм A* был изменен для двух финишных вершин. Можно сделать вывод, что жадный алгоритм выгодно использовать, когда важно время работы программы, а не минимальность найденного пути. A* наоборот стоит использовать для нахождения кратчайшей траектории. Также была создана модифицированная версия A* для нахождения пути для двух финальных вершин.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД

Жадный алгоритм

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

struct edge //ребра графа
{
   char first; //первая вершина
   char second; //вторая вершина
   float weight; //вес ребра
};

bool compare(edge first, edge second)
{
   return first.weight < second.weight;
}

class GraphG
{
private:
   vector <edge> graph; //контейнер для ребер
   vector <char> solutions; //контейнер для текущего пути
   vector <char> checkedpoint; //контейнер для просмотренных вершин
   char start; //начальная вершина
   char destination; //вершина назначения
public:
   int getStart() { return start; }

   GraphG()
   {
      cin >> start;
      cin >> destination;
      char symb;
      while(cin >> symb) //посимвольное считывание строки, элементы заносятся в
структуру edge
      {
         if (symb == '!') break;
         edge cell;
         cell.first = symb;
         if(!(cin >> cell.second)) //нет второй вершины
            break;
         if(!(cin >> cell.weight)) // у ребра нет веса
```

```
            break;

        graph.push_back(cell);
    }
    sort(graph.begin(), graph.end(), compare); //сортировка вершин графа по возрасттанию. 1 < 2
}

void print()
{
    for(int i = 0; i < solutions.size(); i++)
        cout << solutions[i];
}


bool search(char elem) //Жадный алгоритм, принимает на одну вершину
{
    if(start != destination) { //если источник не равен назначению
        if (elem == destination) // если источник == назначению, значит рещультат получен
        {
            solutions.push_back(elem);
            return true;
        }
        checkedpoint.push_back(elem); //кладем рассматриваемую вершину в массив
        for (int i = 0; i < graph.size(); i++) {
            if (elem == graph[i].first) //перебор вершин
            {
                if (isCheck(graph[i].second)) //если вершина уже просмотрена пропускаем ее
                    continue;
                solutions.push_back(graph[i].first); //добавляем вершину в массив с текущем путем
                if (search(graph[i].second)) //продолжаеи поиск до тех пор, пока не найдем финальную вершину
                    return true;
                solutions.pop_back();
            }
        }
        return false;
    }
}

bool isCheck(char elem)
{
    for(int i = 0; i < checkedpoint.size(); i++)
        if(checkedpoint[i] == elem)
            return true;
    return false;
}

};

int main()
```

```
{
   GraphG cell;
   cell.search(cell.getStart());
   cell.print();

   return 0;
}
```

Алгоритм A*

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

using namespace std;

//#define SHOW_DEBUG

class GraphA
{
   struct edge
   {
      char first; //первая вершина
      char second; //вторая вершина
      float weight; //вес ребра
   };

   struct steps
   {
      string path; //строка, содержащая путь
      float length; //длина пути
   };

private:
   vector <edge> graph; //контейнер для ребер
   vector <steps> solutions; //контейнер для текущего пути
   vector <char> checkedpoint; //контейнер для просмотренных вершин
   char start; //начальная вершина
   char destination; //вершина назначения
public:
   GraphA()
   {
      cin >> start;
      cin >> destination;
      char symb;
      while(cin >> symb)
      {
         if (symb == '!') break;
```

```
            edge cell;
            cell.first = symb;
            cin >> cell.second;
            cin >> cell.weight;
            graph.push_back(cell);
        }
        string buf = "";
        buf += start;
        for(int  i = 0; i < graph.size(); i++)
        {
            if(graph[i].first == start)
            {
                buf += graph[i].second;
                solutions.push_back( {buf, graph[i].weight} );
                buf = start;
            }
        }
        checkedpoint.push_back(start);
    }

    int dist(int i, char destination) {
        return solutions[i].length + abs(destination - solutions[i].path.back());
    }

    int minInd() //поиск индекса минимального элемента из непросмотренных
    {
        float min = 111;
        int ind; //индекс минимального элемента
        for(int i=0; i < solutions.size(); i++)
        {
            if(dist(i, destination) < min)
            {
                if(isCheck(solutions.at(i).path.back())) // если i-ый путь ведет в просмотренный
узел
                {
                    solutions.erase(solutions.begin() + i); // то удаляем i-ый путь из solutions
                }
                else
                {
                    min = dist(i, destination);
                    ind = i;
                }
            }
        }
        return ind;
    }

    bool isCheck(char value)
    {
        for(size_t i = 0; i < checkedpoint.size(); i++)
            if(checkedpoint[i] == value)
                return true;
```

```
        return false;
    }

    void search()
    {
        while(true)
        {
#ifdef SHOW_DEBUG
            // debug
            cout << "checkedpoint: ";
            for(auto c : checkedpoint) {
                cout << c;
            }
            cout << endl;

            cout << "solutions: ";
            for(auto r : solutions) {
                cout << r.path << "|" << r.length << ", ";
            }
            cout << endl;
#endif
            size_t min = minInd();
            if(solutions[min].path.back() == destination)
            {
                cout << solutions[min].path;
                return;
            }
//          for(int i=0; i < graph.size(); i++)
            for(auto v : graph)
            {
                if(v.first == solutions[min].path.back())
                {
                    string buf = solutions[min].path;
                    buf += v.second;
                    solutions.push_back({buf, v.weight + solutions[min].length});
                }
            }
            checkedpoint.push_back(solutions[min].path.back());
            solutions.erase(solutions.begin() + min);
        }
    }
};

int main()
{
    GraphA cell;
    cell.search();
    return 0;
}
```

Модифицированный алгоритм A*

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

```
#include <string>
#include <cmath>
#include <fstream>

using namespace std;

//#define SHOW_DEBUG

class GraphAModify
{
   struct edge
   {
      char first; //первая вершина
      char second; //вторая вершина
      float weight; //вес ребра
   };

   struct steps
   {
      string path; //строка, содержащая путь
      double length; //длина пути
   };
private:
   vector <edge> graph; //контейнер для ребер
   vector <steps> solutions[2]; //контейнер для текущего пути
   vector <char> checkedpoint[2]; //контейнер для просмотренных вершин
   char start; //начальная вершина
   char destination[2]; //вершина назначения
   int found[2]; //массив, позволящий отслеживать кол-ко уже найденных вершин

public:
   GraphAModify()
   {
      //ifstream fin("input.txt");
      cin >> start >> destination[0] >> destination[1];
      char symb;
      while(cin >> symb)
      {
         if (symb == '!') break;
         edge cell;
         cell.first = symb;
         cin >> cell.second;
         cin >> cell.weight;
         graph.push_back(cell);
      }
      string buf = "";
      buf += start;
      for(int  i = 0; i < graph.size(); i++)
      {
         if(graph[i].first == start)
         {
            buf += graph[i].second;
```

```
            for(int k=0;k<2;k++)
               solutions[k].push_back( { buf, graph[i].weight} );

            buf.resize(1);
         }
      }

      for(int k=0;k<2;k++)
         checkedpoint[k].push_back(start);
   }



   int minInd(int k) //поиск вершины с наименьшим весом, используется эвристика
   {
      float min = 111;
      int ind;
      for(int i = 0; i < solutions[k].size(); i++)
      {
         if(solutions[k][i].length + abs(destination[k] - solutions[k][i].path.back()) < min) //если
длина новой вершины хуже предыдущей,
                                                                  // то не рассматриваем этот
вариант
         {
            if(isCheck(k, solutions[k][i].path.back()))
            {
               solutions[k].erase(solutions[k].begin() + i);
            }
            else
            {
               min = solutions[k][i].length + abs(destination[k] - solutions[k][i].path.back());
               ind = i;
            }
         }
      }
      return ind; //позция наименьшего элемента
   }

   bool isCheck(int k, char value) //проверка, не просмотрен ли путь
   {
      for(int i = 0; i < checkedpoint[k].size(); i++)
         if(checkedpoint[k][i] == value)
            return true;
      return false;
   }

   void search()
   {
      found[0] = -1; // путь до 1-ой цели еще не найден
      found[1] = -1; // путь до 2-ой цели еще не найден
      while(found[0]<0 || found[1]<0) //поиск продолжается до тех пор, пока существует
```

```
хоть одна найденная вершина
        {
            for(int k=0;k<2;k++) {

                if(found[k]!=-1)
                    continue;
#ifdef SHOW_DEBUG
                // debug
                cout << "checkedpoint " << k << ": ";
                for(auto c : checkedpoint[k]) {
                    cout << c;
                }
                cout << endl;

                cout << "solutions " << k << ": ";
                for(auto r : solutions[k]) {
                    cout << r.path << "|" << r.length << ", ";
                }
                cout << endl;
#endif
                int min = minInd(k);
                if(solutions[k].at(min).path.back() == destination[k])
                {
                    found[k] = min;
                    continue;
                }
                //for(int i = 0; i < graph.size(); i++)
                for(auto v : graph)
                {
                    if(v.first == solutions[k].at(min).path.back())
                    {
                        string buf = solutions[k].at(min).path;
                        buf += v.second;
                        solutions[k].push_back({buf, v.weight + solutions[k].at(min).length});
                    }
                }
                checkedpoint[k].push_back(solutions[k][min].path.back());
                solutions[k].erase(solutions[k].begin() + min);
            }
        }
        for(int k=0;k<2;k++) {
            cout << k << ") " << solutions[k].at(found[k]).path << ", " <<
solutions[k].at(found[k]).length << endl;
        }
    }
};

int main()
{
    GraphAModify element;
    element.search();
```

```
    return 0;
}
```