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: Алгоритм A*

Студентка гр. 7383 _____ Иолшина В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2019

Содержание

Цель работы

познакомиться с алгоритмом поиска A*, создать программу, осуществляющую поиск кратчайшего пути в графе с помощью алгоритма A*.

Формулировка задачи: 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.

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

Выходные данные: строка, в которой перечислены вершины, по которым необходимо пройти от начальной вершины до конечной

Реализация задачи

В данной работе используются главная функция main() и дополнительные функции.

Была реализована структура

```
struct Elem
{
   double key;   //приоритет
   vector <char> path;  //путь по вершинам графа до текущей
   double heur;
   friend bool operator < (const Elem& v1, const Elem& v2)
```

```
    {
        return v1.key > v2.key;
    }
}Elem;
```

И реализован следующий класс:

```
class Adj_list
    {
    private:
        vector <tuple <char, char, double> > edge;
        int count;
        double way_to_finish; //длина пути до конечной вершины
       double way_to_ver[N];  //длина пути до вершины
    public:
    double heuristic(char ver1, char ver2);
    bool is_admissible(double way_to, char ver, char finish);
    bool is_monotonic(int i, char finish);
    void make_path(char start, char finish, vector <char>& ans_path);
    void print_result(vector <char>& ans_path, char finish);
```

- double heuristic(char ver1, char ver2) находит разницу между символами, обозначающими вершины графа, в таблице ASCII.
- bool is_admissible(double way_to, char ver, char finish) проверяет, является ли эвристика допустимой. Для этого смотрит, переоценивает ли эвристика путь до достижения цели.
- bool is_monotonic(int i, char finish) проверяет, является ли эвристика монотонной. Для этого сравнивает путь между двумя вершинами и разницу значения эвристики для этих вершин.
- void make_path(char start, char finish, vector <char>& ans_path)

алгоритм A*. В процессе работы алгоритма, мы идем по графу, образуя очередь с приоритетами.

- void print_result(vector <char>& ans_path, char finish) выводит результат работы программы – кратчайший путь и сведения о том, является ли функция монотонной и допустимой.

Тестирование

Программа собрана в операционной системе Ubuntu 16.04.2 LTS", с использованием компилятора g++ (Ubuntu 5.4.0-6ubuntu1~16.04.5). В других ОС и компиляторах тестирование не проводилось.

Тестовые случаи представлены в Приложении А.

Исследование

Сложность алгоритма составляет O(|V|·|E|), где |V| – количество вершин в графе, |E| - количество ребер в графе. При обработке каждое вершины алгоритм обрабатывает все ребра графа, в наихудшем случае алгоритм обработает каждую вершину

Выводы

В ходе выполнения данной работы был изучен метод поиска кратчайшего пути A*. Была написана программа, применяющая метод A* для поиска кратчайшего пути в графе. А также эвристика исследована на допустимость и монотонность.

ПРИЛОЖЕНИЕ А

ТЕСТОВЫЕ СЛУЧАИ

Ввод	Вывод	Верно?
a e a b 3.0 3.0 b c 1.0 2.0 c d 1.0 1.0 a d 5.0 1.0 d e 1.0 0	ade	Да
a x f a b 3 b d 5 a e 4 a c 3.5 e f 7 e d 7 d f 3	aef	Да

ПРИЛОЖЕНИЕ Б

Исходный код программы

```
#include <iostream>
#include <vector>
#include <tuple>
#include <queue>
#include <cmath>
#define N 26


using namespace std;

typedef struct Elem
{
   double key;   //приоритет
   vector <char> path;  //путь по вершинам графа до текущей
   double heur;
   friend bool operator < (const Elem& v1, const Elem& v2)
   {
      return v1.key > v2.key;
   }
}Elem;

class Adj_list
{
private:
   vector <tuple <char, char, double> > edge;
   int count;
   double way_to_finish; //длина пути до конечной вершины
   double way_to_ver[N];  //длина пути до вершины
public:
   void add_edge(char v1, char v2, double length)
   {
      edge.push_back(make_tuple(v1, v2, length));
```

```
}

double heuristic(char ver1, char ver2)
{
   return abs(ver1 - ver2);
}

bool is_admissible(double way_to, char ver, char finish)
{
   return  way_to_finish - way_to >= heuristic(finish, ver);
}

bool is_monotonic(int i, char finish)
{
   return  get<2>(edge[i]) >= heuristic(get<0>(edge[i]), finish) - heuristic(get<1>(edge[i]),
finish);
}

void make_path(char start, char finish, vector <char>& ans_path)   //алгоритм
{
  vector<char> tmp_path;
  priority_queue <Elem> queue1;
  double way_len = 0;   //длина пути
  count = 0;  //количество вершин
  way_to_ver[count] = 0;  //путь до вершины
  while(true)
  {
      for(int i = 0; i < edge.size(); i++)
         if(get<0>(edge[i]) == start)
         {
            Elem elem1;
            elem1.heur = heuristic(finish, get<1>(edge[i]));
            elem1.key = get<2>(edge[i]) + way_len + elem1.heur;  //эвристика
            for(auto j : tmp_path)
               elem1.path.push_back(j);
            elem1.path.push_back(get<1>(edge[i]));
            queue1.push(elem1);
```

```
            }
        if(!queue1.empty())
        {
            Elem popped = queue1.top();
            queue1.pop();
            start = popped.path[popped.path.size() - 1];
            tmp_path = popped.path;
            way_len = popped.key - popped.heur;
            count++;
            way_to_ver[count] = way_len;
        }
        else
          break;

          if(tmp_path[tmp_path.size() - 1] == finish)
          {
            for (auto j : tmp_path)
              ans_path.push_back(j);
            way_to_finish = way_to_ver[count];
            break;
          }
  }

}
void print_result(vector <char>& ans_path, char finish)
{
  bool admissible = 1;
  bool monotonic = 1;
  int i=0;
  for(auto j : ans_path)
  {
    admissible = is_admissible(way_to_ver[i], j, finish)  && admissible ? 1 : 0;
    monotonic = is_monotonic(i, finish) && monotonic ? 1 : 0;
    i++;
    if(i>=count) break;
  }
  for(auto j : ans_path)
```

```
      cout<<j;
    if(admissible)
      cout << endl << "Эвристика допустима" << endl;
    else
      cout << endl << "Эвристика не допустима" << endl;
    if(monotonic)
      cout << "Эвристика монотонна" << endl;
    else
      cout << "Эвристика не монотонна" << endl;
  }
};


int main()
{
   char start, finish;
   char first, second;
   double length;
   cin >> start >> finish;
   Adj_list head;
   bool f=1;
   while(cin >> first && isalpha(first))
   {
      cin >> second;
      cin >> length;
      head.add_edge(first, second, length);
   }
   vector <char> ans_path;
   ans_path.push_back(start);
   head.make_path(start, finish, ans_path);
   cout << endl;
   head.print_result(ans_path, finish);
   return 0;
}
```