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: Жадный алгоритм и A*.

Студент гр. 8382 _____ Черницын П.А.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2020

Цель работы

Ознакомиться с алгоритмом A* и научиться применять его на практике. Написать программу реализовывающую поиск пути в графе.

Постановка задачи

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

Индивидуальное задание

Вариант 1.

В A* вершины именуются целыми числами (в т. ч. отрицательными).

Описание алгоритма

В первой строке на вход программе подаются начальная и конечная вершины. Далее в каждой строчке указываются рёбра графа и их вес.

Связи графа хранятся в контейнере STL map, в цикле while происходит заполнение point. Цикл прекратит работу в тот момент, когда будет считана пустая строка.

Инициализируется экземпляр класса Graph, после чего вызывается метод Graph::init(). Метод читает информацию о начальной точке и о точке окончания. После происходит считывание зависимостей графа и сохранение их в контейнер map point. Описание структур приведено ниже.

Далее вызывается метод класса Graph::greedySearch(). Метод ищет путь от start до end и возвращает стек результата.

Также был реализован скрипт, собирающий и тестирующий программу.

Сложность алгоритма

На каждой итерации выполняется поиск не просмотренного минимального ребра, выходящего из текущей вершины. Если такого ребра не нашлось, то происходит откат (текущая вершина принимает своё предыдущее значение и удаляется из результата). Если путь нашелся, то за вершину, из которой требуется найти путь, принимается конец ребра, предыдущая добавляется в результат. Продолжаем, пока текущая вершина не станет конечной.

Алгоритм является модификацией алгоритма поиска в глубину. Его сложность - O(N + M*LOGN), где N – количество вершин, а M – кол-во ребер. Поиск не просмотренного минимального значения занимает log(N), так как значения хранятся в ассоциативном контейнере.

Так как на каждом шаге хранится массив смежных ребер, а их количество не превышает число вершин в графе, то получаем сложность по памяти O(N^2), где N - число вершин в графе, M – число ребер в графе.

Память. При более точной эвристической функции сложность по памяти может составлять O(|E|+|V|), т. к. есть возможность сразу построить правильный путь без возвращения к предыдущим вершинам.

В ином случае, каждый шаг будет неверным и придётся просматривать каждое ребро графа. Тогда в худшем случае сложность может оказаться экспоненциальной.

Когда эвристическая функция идеальная и на каждом шаге указано верное направление, получаем сложность O(N+M), где N – количество вершин, M – количество ребер. Так, максимальная длина пути – N и на каждом шаге требуется пройтись по всем ребрам, выходящим из текущей вершины для того, чтобы найти путь с наименьшим приоритетом.

В худшем случае, когда вершины расположены случайным образом и эвристическая функция не идеальна, будут рассмотрены все пути. Отсюда сложность по времени $O(N^M)$, где N – кол-во вершин, а M – кол-во ребер в графе.

Так как промежуточные пути хранятся в контейнере, и длина пути не может превосходить число вершин в графе, то получаем сложность по памяти O(N!).

Описание структур

struct Triple – хранит информацию о ребре. Имеет три поля: first, second, third – имя вершины, вес ребра и флаг (проходили по ней или нет) соответственно.

Class Graph – хранит стартовую точку, точку окончания и информацию о связях.

map<int, set<Triple, SetCompare>> point – контейнер, хранящий связи вершин.

std::stack<int> res – стек, хранящий результат. Заполняется и возвращается функцией greedySearch

Описание функций

void expand_stack(std::stack<int>& res) – принимает стек и «переворачивает» его.

std::stack<int> Graph::greedySearch() – функция поиска пути в графе. Работает по принципу поиска в глубину. Идем по графу пока не достигнем конца (по условию), либо пока не окажемся в тупике. Если дальше пути нет (за этим

следит флаг can_go), откатываемся на вершину назад. В итоге получаем либо стек с результатом, либо пустой стек, что означает, что требуемого пути нет.

void Graph::print_graph() – печатает список зависимостей point

void Graph::init() - Метод читает информацию о начальной точке и о точке окончания. После происходит считывание зависимостей графа и сохранение их в контейнер map point. Описание структур приведено ниже.

Тестирование

```
***Compilation completed***
INPUT:
0 5
0 1 3.0
0 2 1.0
1 3 2.0
1 4 3.0
3 4 4.0
4 0 1.0
4 6 2.0
0 5 8.0
6 5 1.0

ANSWER: 013405
OUTPUT:
013405
========================
INPUT:
-5 7
-5 -3 3.0
-5 -1 1.0
-3 0 2.0
-3 3 3.0
0 3 4.0
3 -5 3.0
3 5 2.0
-5 7 8.0
5 7 1.0

ANSWER: -5-30357
OUTPUT:
-5-30357
========================
INPUT:
-5 7
-5 -3 3.0
-5 -1 1.0
-3 0 2.0
-3 3 3.0
0 3 4.0
3 -5 3.0
3 5 2.0
-5 7 8.0
5 7 1.0
-1 8 1.0
8 9 1.0

ANSWER: -5-30357
OUTPUT:
-5-30357
```

```
========================
INPUT:
-5 3
-5 -3 7.0
-5 -1 3.0
-3 -1 1.0
-1 0 8.0
-3 3 4.0

ANSWER: -5-33
OUTPUT:
-5-33
========================
INPUT:
-5 0
-5 -3 1.0
-3 -1 1.0
-1 -5 1.0
-5 0 8.0

ANSWER: -5-3-1-50
OUTPUT:
-5-3-1-50
========================
INPUT:
-5 0
-5 -3 1.0
-3 -1 9.0
-1 0 3.0
-5 0 9.0
-5 3 1.0
3 0 3.0

ANSWER: -530
OUTPUT:
-530
```

```
INPUT:
-5 5
-5 -1 1.0
-5 -3 1.0
-1 0 2.0
-3 3 2.0
0 5 3.0
3 5 3.0

ANSWER: -5-105
OUTPUT:
-5-105
========================
INPUT:
-5 -3
-5 -3 1.0
-5 -1 1.0

ANSWER: -5-3
OUTPUT:
-5-3
========================
INPUT:
-3 3
-5 -3 1.0
-5 -1 2.0
-3 0 7.0
-3 3 8.0
-5 7 2.0
-3 7 6.0
-1 3 4.0
0 3 4.0
7 3 1.0

START NOT ZERO. ANSWER: -373
OUTPUT:
-373
```

Выводы.

В ходе выполнения лабораторной работы были изучены и применены на практике жадный алгоритм и алгоритм A*.

ПРИЛОЖЕНИЕ А. Исходный код программы.

```
#include <iostream>
#include <stack>
#include <string>
#include <map>
#include <set>

using std::map;
using std::set;
using std::pair;
using std::string;
using std::cout;
using std::cin;
using std::endl;

typedef struct Triple
{
   // структура, хранящая информацию о ребре и было ли оно пройдено
   int first;
   double second;
   mutable bool third;
   Triple() {}
   Triple(int name, double weight, bool flag=false) : first(name), second(weight), third(flag) {}
} Triple;

struct SetCompare
{
   bool operator()(Triple v1, Triple v2)
   {
      if (v1.second == v2.second)
         return v1.first < v2.first;
      return  v1.second < v2.second;
   }
};

class Graph
{
public:
   // point хранит зависимость вершин в виде: вершина - массив смежных вершин
   // Массив смежных вершин отсортирован по возрастанию веса ребра (SetCompare)
   map<int, set<Triple, SetCompare>> point;
   int start, end;
public:
   void init();
   void print_graph();
   std::stack<int> greedySearch();

};
```

```
void Graph::init()
/* Читаем start, end. После заполняем массив зависимостей */
{
   string input;
   //cout << "Enter start and end point: ";
   getline(cin, input);
   start = std::stoi(input.substr(0));
   end   = std::stoi(input.substr(2));

   //cout << "Enter adjacency list:" << endl;
   while (getline(cin, input))
   {
      if (input.empty()) break;
      point[std::stoi(input.substr(0))].emplace(std::stoi(input.substr(2)), std::stod(input.substr(4)));
   }
}

void Graph::print_graph()
{
   for (auto var : point)
   {
      cout << var.first << ": ";
      for (auto var2 : var.second)
         cout << var2.first << " " << var2.second << " " << var2.third << "; ";
      cout << std::endl;
   }
}

std::stack<int> Graph::greedySearch()
{
   // В стеке храним результат. Сразу записываем первую вершину
   // curr хранит массив смежных вершин к текущей вершине
   std::stack<int> res;
   res.push(start);

   set<Triple, SetCompare> curr = point[res.top()];

   while (!res.empty() && res.top() != end)
   {
      bool can_go = false;
      int tmp;
      if (!curr.empty())
      {
         for (auto &var : point[res.top()]) //point[res.top()] == curr. Сделано для того, чтобы флаг изменялся
         // Ищем следующую непосещённую вершину
         {
            if (!var.third)
            {
               can_go = true;
```

```
                var.third = true;
                tmp = var.first;
                break;
            }
        }
    }

    if (can_go)
    {
        res.push(tmp);
        curr = point[tmp];
    } else {
        res.pop();
        if (!res.empty()) curr = point[res.top()];
    }
  }

  return res;
}

void expand_stack(std::stack<int>& res)
{
  std::stack<int> tmp;
  tmp.swap(res);
  while (!tmp.empty())
  {
    res.push(tmp.top());
    tmp.pop();
  }
}

int main()
{
  Graph one;
  one.init();
  //one.print_graph();

  auto res = one.greedySearch();
  string res_str;
  expand_stack(res);

  while (!res.empty())
  {
    cout << res.top();
    res.pop();
  }
  cout << endl;
  return 0;
}
```