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*

Студентка гр. 9383 ___ Сергиенкова А.А.

Преподаватель ___ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Изучить алгоритмы поиска пути в ориентированном графе (A* и жадный). Написать программы, реализующие алгоритмы.

Задание.

1) 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.

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.

2) Алгоритм A*. (Вариант 4)

Модификация A* с двумя финишами: требуется найти путь до каждого, а затем найти мин. путь между ними, при этом начальная вершина не должна находиться в окончательном ответе.

Основные теоретические положения.

Жадный алгоритм – это алгоритм, который, на каждом шагу принимает локально оптимальное решение, не заботясь о том, что будет дальше. Он не всегда верен, но есть задачи, где жадный алгоритм работает правильно.

Описание:

В начале работы алгоритма сравниваются смежные вершины графа. После, выбирается вершина, у которой ребро имеет наименьший вес. Пройденная вершина записывается в ответ. Далее, рекурсивно делаем то же самое из новой вершины. Концом алгоритма считается просмотр конечной вершины.

Сложность:

Временная сложность модифицированного алгоритма A* - O(V*E).

Алгоритм A* - алгоритм поиска по первому наилучшему совпадению на графе, который находит маршрут с наименьшей стоимостью от одной вершины (начальной) к другой (целевой, конечной).

Описание:

Стартовой вершине присваивается метка равная 0. Выбирается вершина, которая имеет наивысший приоритет. Далее происходит переход по всем соседним вершинам, также вычисляется стоимость перехода по ним, выбирается наилучший и происходит переход. Та вершина, в которую мы перешли, добавляется в очередь обработки с её приоритетом. . Концом алгоритма считается просмотр конечной вершины.

Сложность:

Временная сложность модифицированного алгоритма A* - O(V+E).

Описание функций и структур данных:

- class Graph – граф, вектор вершин.
- class Edge Link – ребро графа.
- class Resolver – решает задачу с помощью жадного алгоритма.
- void Resolver() – ввод данных.
- bool Go – посещает указанную вершину.
- double Weight – вес указанного пути.
- std::vector<Edge> – выводит результат.
- class Resolver2 – решает задачу с помощью модифицированного алгоритма A*.
- std::priority_queue<QueueData, std::vector<QueueData>, std::less<QueueData>> _Queue – очередь посещения вершин.
- std::map<char, char> _CameFrom – из какой вершины пришли в каждую вершину.
- std::map<char, int> _Cost – все известные стоимости вершин.
- char _Start – начало движения.
- char _End – куда пришли.
- int Heuristic – эвристическая функция.
- std::vector<char> Result() – вектор результата.
- struct QueueData – данные для приоритетной очереди.
- class Resolver3 – решает задачу с помощью вариативного задания.

Тестирование жадного алгоритма.

Рисунок 1 – тест жадного алгоритма с входными данными №1.

Рисунок 2 – Тест жадного алгоритма с входными данными №2.

Тестирование модифицированного A*.

Рисунок 3 – Тест алгоритма A с входными данными №1.*

Рисунок 4 – Тест алгоритма A с входными данными №2.*

Тестирование задания по варианту.

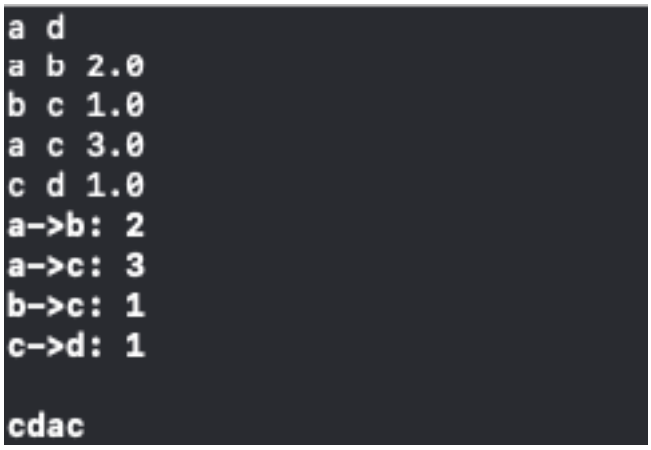

Рисунок 5 – Тест вариативного задания с входными данными №1.

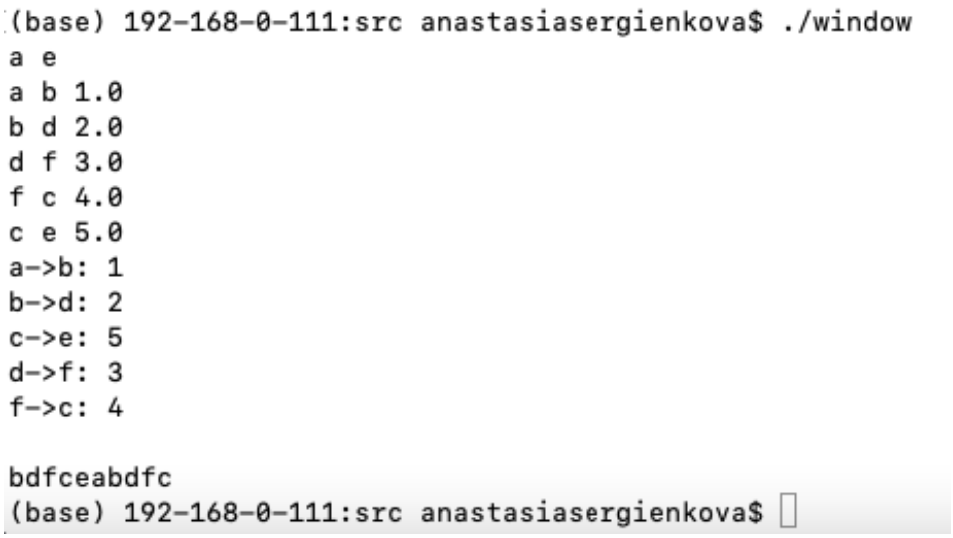

Рисунок 6 Тест вариативного задания с входными данными №2.

Вывод.

Изучены алгоритмы поиска пути в ориентированном графе (A* и жадный). Написаны программы, реализующие алгоритмы.

Приложение А
Исходный код программы

Edge.cpp

```
#include "Edge.h"


char Edge::From() {
    return _From;
}
char Edge::To() {
    return _To;
}
double Edge::Weight() {
    return _Weight;
}

std::istream& operator>>(std::istream& is, Edge& e) {
    return is >> e._From >> e._To >> e._Weight;
}
std::ostream& operator<<(std::ostream& os, Edge& e) {
    return os << e._From << "->" << e._To << ": " << e._Weight;
}
bool operator<(const Edge& a, const Edge& b) {
    return a._Weight < b._Weight;
}
bool operator>(const Edge& a, const Edge& b) {
    return a._Weight > b._Weight;
}
Edge::Edge() {
    _From = 0;
    _To = 0;
    _Weight = 0;
}
Edge::Edge(char from, char to, double weight) {
    _From = from;
    _To = to;
    _Weight = weight;
}
```

Edge.h

```
#pragma once
#include <iostream>

// ребро графа
class Edge
{
    char _From;
    char _To;
    double _Weight;
```

```
public:

    Edge();
    Edge(char from, char to, double weight);

    char From();          // откуда
    char To();            // куда
    double Weight(); // с каким весом

    friend std::istream& operator>>(std::istream& is, Edge& e);
    friend std::ostream& operator<<(std::ostream& os, Edge& e);
    friend bool operator<(const Edge& a, const Edge& b);
    friend bool operator>(const Edge& a, const Edge& b);
};
```

Graph.cpp

```
#include "Graph.h"
#include <algorithm>

std::istream& operator>>(std::istream& is, Graph& g) {
    // ввод начала и конца
    is >> g._Start >> g._End;

    // ввод всех ребер, засовывая их в нужные вектора
    Edge e;
    do {
     is >> e;
     g._Data[e.From()].push_back(e);
    } while (e.To() != g._End);

    // сортировка всех векторов
    for (auto i : g._Data) {
     std::sort(i.second.begin(), i.second.end());
     g._Data[i.first] = i.second;
    }

    return is;
}
std::ostream& operator<<(std::ostream& os, Graph& g) {
    for (auto i : g._Data)
     for (auto j : i.second)
          os << j << std::endl;
    return os;
}
char Graph::Start() {
    return _Start;
}
char Graph::End() {
    return _End;
}
std::vector<Edge> Graph::operator[](char node){ // возвращает сортированный
по весам вектор ребер из указанной вершины
```

```
    return _Data[node];
}
```

Graph.h

```
#pragma once
#include <iostream>
#include <map>
#include <vector>
#include "Edge.h"

// граф
class Graph
{
    std::map<char, std::vector<Edge>> _Data; // набор ребер на каждый узел
    char _Start;
    char _End;
public:

    char Start();
    char End();

    std::vector<Edge> operator[](char node); // возвращает сортированный по весам вектор ребер из указанной вершины

    friend std::istream& operator>>(std::istream& is, Graph& g);
    friend std::ostream& operator<<(std::ostream& os, Graph& g);
};
```

Main.cpp

```
#include <iostream>
#include <sstream>
#include "Graph.h"
#include "Resolver.h"
#include "Resolver2.h"
#include "Resolver3.h"

int main() {
    // ввод данных
    Graph g;

    std::cin >> g;

    // вывод графа
    std::cout << g << std::endl;

    // производим решение
    //Resolver r;
```

```
    //Resolver2 r;
    //r.Resolve(g, g.Start(), g.End());

    Resolver3 r;
    r.Resolve(g, g.Start(), g.End(), 'c');

    // вывод результата
    std::cout << r << std::endl;

    return 0;
}
```

Resolver.cpp

```
#include "Resolver.h"

void Resolver::Resolve(Graph& g, char a, char b) {
    // очистка предыдущего результата
    _Result.clear();

    // создаем нулевой путь
    std::vector<Edge> path;

    // посещаем первую вершину
    Go(Edge(0, a, 0), g, path, b);
}

std::ostream& operator<<(std::ostream& os, Resolver r) {
    if (r._Result.size() == 0) os << "Has no result";
    for (auto i : r._Result) os << i.To();
    return os;
}
bool Resolver::Go(Edge edge, Graph& g, std::vector<Edge>& path, char end)
    // посещает указанную вершину
{
    // если вес текущего пути больше чем лучший найденый то не посещаем
вершину
    if (_Result.size() > 0 && Weight(path) > Weight(_Result)) return false;

    // добавляем ребро в путь
    path.push_back(edge);

    // если дошли до конца, то проверяем получившийся путь
    if (edge.To() == end) {
     if (_Result.size() == 0 || Weight(path) < Weight(_Result))
           _Result = path;
    }
    else { // в противном случае посещаем все остальные вершины
     // посещаем все вершины из этого пути
     for (auto i : g[edge.To()])
           if (!Go(i, g, path, end)) break;
    }
```

```
    // удаляем ребро из пути
    path.pop_back();

    // сообщает что посетили успешно
    return true;
}
double Resolver::Weight(std::vector<Edge>& path) {    // вес указанного пути
    double weight = 0;
    for (auto i : path) weight += i.Weight();
    return weight;
}
```

Resolver.h

```
#pragma once
#include <iostream>
#include <vector>
#include "Graph.h"

// производит решение задачи
class Resolver
{
    std::vector<Edge> _Result;  // результат
    bool Go(Edge edge, Graph& g, std::vector<Edge>& path, char end);
    // посещает указанную вершину
    double Weight(std::vector<Edge>& path);// вес указанного пути
public:
    void Resolve(Graph& g, char a, char b);

    friend std::ostream& operator<<(std::ostream& os, Resolver r);
};
```

Resolver2.cpp

```
#include "Resolver2.h"
#include <list>

bool operator<(QueueData d1, QueueData d2) {
    return d1.Weight < d2.Weight;
}
bool operator>(QueueData d1, QueueData d2) {
    return d1.Weight > d2.Weight;
}

std::ostream& operator<<(std::ostream& os, Resolver2 r)
{
    auto res = r.Result();
    if (res.empty()) return os << "A* has no result!";
```

```
    for (auto i : res) os << i;
    return os;
}
void Resolver2::Resolve(Graph& g, char a, char b)
{
    // принимаем данные
    _Start = a;
    _End = b;
    // запуск расчетов
    _Queue.push(QueueData(_Start, 0));
    _CameFrom[_Start] = 0;
    _Cost[_Start] = 0;

    // цикл обхода
    while (!_Queue.empty()) {
     // берем наилучший вариант
     auto current = _Queue.top();
     _Queue.pop();
     // если этот вариант уже вконце то расчеты закончены
     if (current.Point == _End) break;
     // переходим из наилучшего варианта во все возможные новые смежные
вершины
     for (auto e : g[current.Point]) {
          // расчет стоимости перехода в новую вершину
          auto newCost = _Cost[current.Point] + e.Weight();
          // если новая вершина содержится в списке переходов
          // и новая стоимость перехода больше или равна уже имеющейся
стоимости
          // то пропускаем новый переход
          if (_Cost.count(e.To()) > 0 && _Cost[e.To()] >= newCost)
continue;
          // производим переход в вершину
          _Cost[e.To()] = newCost; // пишем стоимость перехода
          _CameFrom[e.To()] = current.Point; // пишем откуда пришли
          // добавляем в приоритетную очередь новую вершину с приоритетом,
          // учитывающим эвристическую функцию
          _Queue.push(QueueData(e.To(), newCost + Heuristic(e.To())));
     }
    }
}

std::vector<char> Resolver2::Result() // вектор результата
{
    std::vector<char> vector;
    // ограничитель
    if (_CameFrom.count(_End) == 0)
     return vector;

    // формируем список
    std::list<char> list;
    auto cur = _End;
    while (cur != 0) {
     list.push_front(cur);
     cur = _CameFrom[cur];
    }
```

```
    // конвертируем в вектор
    for (auto i : list) vector.push_back(i);

    // вывод результата
    return vector;
}
```

Resolver2.h

```
#pragma once
#include <queue>
#include <vector>
#include <map>
#include "Graph.h"

// данные для приоритетной очереди
struct QueueData {
    char Point;
    int Weight;

    QueueData(char point, int weight) {
     this->Point = point;
     this->Weight = weight;
    }

    friend bool operator<(QueueData d1, QueueData d2);
    friend bool operator>(QueueData d1, QueueData d2);
};
// решает вторую задачу
class Resolver2
{
    std::priority_queue<QueueData, std::vector<QueueData>,
std::less<QueueData>> _Queue; // очередь посещения вершин
    std::map<char, char> _CameFrom; // откуда пришли в каждую вершину
    std::map<char, int> _Cost;        // все известные стоимости вершин
    char _Start;      // откуда
    char _End;  // куда
    int Heuristic(char point) { return abs(_End - point); } // эвристическая
функция
public:
    void Resolve(Graph& g, char a, char b);
    std::vector<char> Result(); // вектор результата

    friend std::ostream& operator<<(std::ostream& os, Resolver2 r);
};
```

Resolver3.cpp

```
#include "Resolver3.h"
```

```
std::ostream& operator<<(std::ostream& os, Resolver3 r)
{
    if (r._Result.empty()) return os << "resolver3 has no result!";
    for (auto i : r._Result) os << i;
    return os;
}

void Resolver3::Resolve(Graph& g, char start, char end1, char end2)
{
    // создаем ресолверы
    Resolver2 r1;
    Resolver2 r2;
    Resolver2 r3;

    // ищем пути до 2 концов
    r1.Resolve(g, start, end1);
    r2.Resolve(g, start, end2);

    // ищем путь между 2 концами
    r3.Resolve(g, end1, end2);

    // формируем результат
    int n = 0;
    for (auto i : r1.Result()) {
     if (n++ == 0) continue; // пропуск стартовой вершины
     _Result.push_back(i);
    }
    for (auto i : r2.Result()) _Result.push_back(i);
    for (auto i : r3.Result()) _Result.push_back(i);
}
```

Resolver3.h

```
#pragma once
#include "Resolver2.h"
#include <vector>

// решает 3 задачу
class Resolver3
{
    std::vector<char> _Result;
public:
    void Resolve(Graph& g, char start, char end1, char end2);

    friend std::ostream& operator<<(std::ostream& os, Resolver3 r);
};
```