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. 9383 ___ Рыбников Р.А.

Преподаватель ___ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Изучить алгоритмы поиска пути в ориентированном графе. Реализовать соответствующие программы

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 – алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным.

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

Описание алгоритмов

Жадный алгоритм:

1. В самом начале работы алгоритма идет сравнение смежных вершин графа. Для перехода к следующей вершины, выбирается та, у который стоимость(вес) ребра наименьший. После сделанного шага, данная вершина записывается в пройденный путь, для дальнейшего получения ответа.
2. Рекурсивно проделываем то же самое из новой вершины, сканируя на обнаружения наименьшего веса ребер во всех возможных дорогах графа.
3. Если получилось так, что текущий путь лучше (короче по длине) чем результат, то перезаписываем результат.
4. Конец алгоритма – это просмотр вершины, которая считается конечной.

Алгоритм A*:

1. Стартовая точка имеет стоимость перехода, равную 0.
2. Выбираем вершину, с высшим приоритетом.
3. Происходит переход по всем соседним вершинам.
4. Вычисляется стоимость перехода по ним, выбирается наилучший и совершается переход.
5. Получаем приоритет обработки вершины, в которую только что перешли.
6. Вершина добавляется в очередь обработки с этим приоритетом.
7. Конец алгоритма – просмотр конечной вершины.

Сложность

Сложность жадного алгоритма O(V*E); V – количество вершин, E – количество рёбер.

Сложность алгоритма A* O(V+E); V – количество вершин, E – количество рёбер, если используется эвристическая функция

Описание основных функций и структур данных

- Orgraph – класс графа, на базе вектора, который хранит вершины.
- Edge – класс клетки, которая имеет поля стартовой,конечной клетки, вес ребра, вектор клеток.
- Edge& operator[](int index) – перегруженный оператор, который позволяет вернуть ребро по его индексу.
- std::vector<Edge> operator[](char node) – позволяет вернуть ребро из указанной вершины. Все исходящие вершины сортируются.
- GreedyAlg::Resolve – метод поиска решений с помощью жадного алгоритма. Происходит очистка решений, затем с помощью метода Scan происходит перемещение по вершинам графа.
- GreedyAlg::Scan – рекурсивный метод, в котором реализованы ограничители на: посещенные вершины, проверка конца пути, проверка на длину пути. Исходя из всех этих проверок реализован жадный алгоритм.
- AStar – класс, который решает задачу методом A*. В этом классе реализована структура, которая представляет собой очередь с приоритетом. В конструкторе задаются текущие значение вершины и её приоритета.
- int Heuristic – эвристическая оценка для каждой вершины, которая возвращает расстояние от данной вершины до конечной вершины.
- Option – класс, который позволяет решить вариативное задание, с

помощью класса AStar. Сначала ищется первое решение от начальной вершины до первого финиша. Ответ выводится без самого первого элемента. Затем ищется второе решение от начала до второго финиша. После этого ищется решение от первого до второго финиша.

Тестирование.

Жадный алгоритм:

```
a b 3
b c 1
c d 1
a d 5
d e 1

abcde
Program ended with exit code: 0
```

Рисунок 1 – Тестирование жадного алгоритма

A*:

```
a b 3
b c 1
c d 1
a d 5
d e 1

ade
Program ended with exit code: 0
```

Рисунок 2 – Тестирование алгоритма A*

A*(2 финиша):

```
a b 3
b c 1
a d 100
a g 40
e c 40
c d 1
a d 5
d e 1

start to end1: de
start to end2: adec
end1 to end2: ec

Program ended with exit code: 0
```

Рисунок 3 – Тестирование алгоритма A* с двумя финишами.

Вывод.

Были получены навыки работы с жадным алгоритмом и алгоритмом A*.

Реализованы программы, которые демонстрируют работу этих алгоритмов.

Приложение А

Файл main.cpp

```
#include <iostream>
#include <sstream>
#include "Orgraph.h"
#include "GreedyAlg.h"
#include "AStar.h"
#include "Option.h"

int main() {
	Orgraph g;
	char end2;
	//std::stringstream ss("a e\n a b 3.0\nb c 1.0\nc d 1.0\na d 5.0\nd e 1.0");
   //std::stringstream ss("a e\n a b 3.0\nb c 1.0\na d 100\na g 40\ne c 40\nc d 1.0\a d 5.0\nd e 1.0 c");

	//ss >> g;
	//ss >> end2;
	std::cin >> g;
	std::cin >> end2;

   std::cout << g << std::endl;


	//GreedyAlg resolver;
	//resolver.Resolve(g, g.Start(), g.End());

   //AStar resolver;
	//resolver.Resolve(g, g.Start(), g.End());

	Option resolver;
	resolver.Resolve(g, g.Start(), g.End(), end2);

   std::cout << resolver << std::endl;


   return 0;
}
```

Файл Orgraph.h

```
#pragma once
#include <iostream>
#include <vector>

class Orgraph
{
public:
	class Edge;
private:
	char m_Start = 0;
	char m_End = 0;
	std::vector<Edge> m_Edges;
public:
```

```
	char Start() { return m_Start; }
	char End() { return m_End; }

	Edge& operator[](int index) { return m_Edges[index]; } // возвращает ребро по его индексу
	std::vector<Edge> operator[](char node);		// возвращает все исходящие ребра из указанной вершины, причем все исходящие вершины сортируются

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

class Orgraph::Edge {
	char m_Start;	// откуда
	char m_End;		// куда
	float m_W;		// вес
public:
	Edge() {
		m_Start = 0;
		m_End = 0;
		m_W = 0;
	}
	Edge(char start, char end, float w) {
		this->m_Start = start;
		this->m_End = end;
		this->m_W = w;
	}

	const char Start() { return m_Start; }
	const char End() { return m_End; }
	const float W() { return m_W; }

	friend std::istream& operator>>(std::istream& is, Edge& e);
	friend std::ostream& operator<<(std::ostream& os, Edge& e);
	friend bool operator<(const Orgraph::Edge& e1, const Orgraph::Edge& e2);
	friend bool operator>(const Orgraph::Edge& e1, const Orgraph::Edge& e2);
};

inline bool operator<(const Orgraph::Edge& e1, const Orgraph::Edge& e2) {
	return e1.m_W < e2.m_W;
}
inline bool operator>(const Orgraph::Edge& e1, const Orgraph::Edge& e2) {
	return e1.m_W > e2.m_W;
}
```

Файл Orgraph.cpp

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

std::istream& operator>>(std::istream& is, Orgraph& g)
{
	Orgraph::Edge e;
	is >> g.m_Start;
	is >> g.m_End;
	do {
```

```
		is >> e;
		g.m_Edges.push_back(e);
	} while (e.End() != g.m_End);
	return is;
}
std::ostream& operator<<(std::ostream& os, Orgraph& g) {
	for (auto i = g.m_Edges.begin(); i < g.m_Edges.end(); ++i)
		os << *i << std::endl;
	return os;
}
std::istream& operator>>(std::istream& is, Orgraph::Edge& e) {
	return is >> e.m_Start >> e.m_End >> e.m_W;
}
std::ostream& operator<<(std::ostream& os, Orgraph::Edge& e) {
	return os << e.m_Start << ' ' << e.m_End << ' ' << e.m_W;
}
std::vector<Orgraph::Edge> Orgraph::operator[](char node)	// возвращает все исходящие ребра из указанной вершины, причем все исходящие вершины сортируются
{
	std::vector<Edge> res;
	for (auto i = m_Edges.begin(); i < m_Edges.end(); ++i) {
		if (i->Start() == node) res.push_back(*i);
	}
	std::sort(res.begin(), res.end(), std::less<Edge>());
	return res;
}
```

Файл GreedyAlg.h

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

class GreedyAlg
{
	std::vector<Orgraph::Edge> m_Result;
	std::vector<Orgraph::Edge> m_Current;
	Orgraph m_Graph;
	char m_Start;
	char m_End;
	void Scan(Orgraph::Edge edge);
	float Weight(std::vector<Orgraph::Edge> path);
public:

	void Resolve(Orgraph g, char start, char end);

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

Файл GreedyAlg.cpp

```
#include "GreedyAlg.h"

void GreedyAlg::Resolve(Orgraph g, char start, char end)
{
	// очистка
	m_Result.clear();
	m_Current.clear();

	m_Graph = g;
	m_Start = start;
	m_End = end;

	Scan(Orgraph::Edge(start, start, 0));
}
std::ostream& operator<<(std::ostream& os, GreedyAlg r) {
	if (r.m_Result.size() == 0) os << "has no result!";
	for (auto i : r.m_Result) os << i.End();
	return os;
}

void GreedyAlg::Scan(Orgraph::Edge edge) {

	for (auto i : m_Current)
		if (i.End() == edge.End())
			return;

	m_Current.push_back(edge);

	// если дошли до конца то проверка текущего пути
	if (m_Current[m_Current.size() - 1].End() == m_End) {
		// если текущий путь короче чем результат то его запоминаем
		if (m_Result.size() == 0 || Weight(m_Current) < Weight(m_Result))
			m_Result = m_Current;
		m_Current.pop_back();
		return;
	}
	// если длина пути стала больше чем лучший результат то ничего не делаем
	if (m_Result.size() > 0 && Weight(m_Current) >= Weight(m_Result)) {
		m_Current.pop_back();
		return;
	}

	for (auto i : m_Graph[edge.End()])
		Scan(i);

	m_Current.pop_back();
}
float GreedyAlg::Weight(std::vector<Orgraph::Edge> path)
{
	float w = 0;
	for (auto i : path)
		w += i.W();
	return w;
}
```

Файл AStar.h

```
#pragma once
#include <list>
#include "Orgraph.h"

class AStar
{
private:
	struct QueueData {
	public:
		char Node;
		int Priority;

		QueueData() {}
		QueueData(char node, char priority) {
			this->Node = node;
			this->Priority = priority;
		}

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

	std::list<Orgraph::Edge> m_Result;
	Orgraph m_Graph;
	char m_Start;
	char m_End;
	float Weight(std::vector<Orgraph::Edge> path);

	int Heuristic(char point) { return abs(point - m_End); }
public:
	void Resolve(Orgraph g, char start, char end);
	friend std::ostream& operator<<(std::ostream& os, AStar r);

	std::list<Orgraph::Edge>::iterator begin() { return m_Result.begin(); }
	std::list<Orgraph::Edge>::iterator end() { return m_Result.end(); }
};
```

Файл AStar.cpp

```
#include "AStar.h"
#include <queue>
#include <map>

std::ostream& operator<<(std::ostream& os, AStar r) {
	if (r.m_Result.size() == 0) os << "has no result!";
	for (auto i : r.m_Result) os << i.End();
	return os;
}
float AStar::Weight(std::vector<Orgraph::Edge> path)
{
	float w = 0;
	for (auto i : path)
		w += i.W();
```

```
	return w;
}
void AStar::Resolve(Orgraph g, char start, char end)
{
	m_Result.clear();
	std::priority_queue<AStar::QueueData, std::vector<AStar::QueueData>,

   std::less<AStar::QueueData>> queue;
	std::map<char, int> cost;
	std::map<char, char> cameFrom;

	// задаем значения
	m_Graph = g;
	m_Start = start;
	m_End = end;

	// пишем начальные данные
	queue.push(AStar::QueueData(m_Start, 0));
	cost[m_Start] = 0; // стоимость перехода в начальную точку 0
	cameFrom[m_Start] = 0; // вначале какбы переход фиктивный

	while (!queue.empty()) {
		// берем элемент с высшим приоритетом
		auto current = queue.top();
		queue.pop();
		if (current.Node == m_End) break;
		// переходим по всем соседним узлам
		for (auto edge : m_Graph[current.Node]) {
			auto newCost = cost[current.Node] + edge.W();
			// ограничитель, если переходить ненужно (по стоимости)
			if (cost.find(edge.End()) != cost.end() && // если стоимость есть
				newCost >= cost[edge.End()]) // и эта стоимость меньше или равна новой
				continue;
			// совершаем переход
			cost[edge.End()] = newCost;
			cameFrom[edge.End()] = edge.Start();
			// вычисляем приоритет обработки вершины в которую перешли
			auto priority = newCost + Heuristic(edge.End());
			// добавляем эту вершину в очередь обработки с этим приоритетом
			queue.push(AStar::QueueData(edge.End(), priority));
		}
	}

	m_Result.clear();
	auto current = m_End;
	while (cameFrom.find(current) != cameFrom.end() && cameFrom[current]) {
		m_Result.push_front(Orgraph::Edge(cameFrom[current], current, cost[current]));
		current = cameFrom[current];
	}
	if(cameFrom[current]==0)m_Result.push_front(Orgraph::Edge(current, current, 0));
}
```

Файл Option.h

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

class Option
{
	std::vector<char> m_Result1;
	std::vector<char> m_Result2;
	std::vector<char> m_Result3;
public:
	void Resolve(Orgraph &g, char start, char end1, char end2);
	friend std::ostream& operator<<(std::ostream& os, Option& resolver);
};
```

Файл Option.cpp

```
#include "Option.h"
#include "AStar.h"

std::ostream& operator<<(std::ostream& os, Option& resolver)
{
	if (resolver.m_Result1.size() == 0 &&
		resolver.m_Result2.size() == 0 &&
		resolver.m_Result3.size() == 0)
		return os << "Option has no result";

	os << "start to end1: ";
	for (auto i : resolver.m_Result1) os << i;
	os << std::endl;
	os << "start to end2: ";
	for (auto i : resolver.m_Result2) os << i;
	os << std::endl;
	os << "end1 to end2: ";
	for (auto i : resolver.m_Result3) os << i;
	os << std::endl;
	return os;
}

void Option::Resolve(Orgraph& g, char start, char end1, char end2)
{
   AStar res;

	m_Result1.clear();
	m_Result2.clear();
	m_Result3.clear();

	// ищем решение от начала до первого конца
	res.Resolve(g, start, end1);
	// пишем решение без 1 элемента
	int n = 0;
	for (auto i : res) {
```

```
            if (n++ == 0) continue;
            m_Result1.push_back(i.End());
        }

        res.Resolve(g, start, end2);
        for (auto i : res) m_Result2.push_back(i.End());

        res.Resolve(g, end1, end2);
        for (auto i : res) m_Result3.push_back(i.End());
}
```